

ILOG CPLEX and Concert C++

API 9.1

Reference Manual

April 2005

Copyright © 1987-2005, by ILOG SA. All rights reserved. Legal terms. Privacy policy.

ILOG, the ILOG design, CPLEX, and all other logos and product and service names of ILOG are registered trademarks or trademarks of ILOG in France, the U.S. and/or other countries.

Java™ and all Java-based marks are either trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Microsoft, Windows, and Windows NT are either trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries.

All other brand, product and company names are trademarks or registered trademarks of their respective holders.

Printed in France

About This Manual	14
Concepts	16
Group optim.concert	33
ILOANYBINARYPREDICATE0	40
ILOANYTERNARYPREDICATE0	41
ILOINTBINARYPREDICATE0	42
ILOINTTERNARYPREDICATE0	43
ILO_NO_MEMORY_MANAGER	44
IloAbs	45
IloAbstraction	46
IloAdd	48
IloAlgorithm	49
IloAlgorithm::CannotExtractException	56
IloAlgorithm::CannotRemoveException	58
IloAlgorithm::Exception	60
IloAlgorithm::NotExtractedException	61
IloAlgorithm::Status	63
IloAllDiff	65
IloAllMinDistance	68
IloAllNullIntersect	71
IloAnd	72
IloAny	76
IloAnyArray	77
IloAnyBinaryPredicate	79
IloAnySet	81
IloAnySet::Iterator	85
IloAnySetVar	86
IloAnySetVarArray	90
IloAnyTernaryPredicate	93
IloAnyTupleSet	95
IloAnyTupleSetIterator	98

IloAnyVar	99
IloAnyVarArray	101
IloArcCos	103
IloArray	104
IloBarrier	108
IloBaseEnvMutex	111
IloBool	112
IloBoolAbstraction	113
IloBoolArray	115
IloBoolVar	118
IloBoolVarArray	121
IloBox	124
IloCard	128
IloCeil	129
IloCondition	130
IloConstraint	133
IloConstraintArray	136
IloConversion	139
IloCplex::Status	142
IloDeleterMode	143
IloDiff	144
IloDisableNANDetection	147
IloDistribute	148
IloDiv	152
IloEmptyHandleException	153
IloEnableNANDetection	154
IloEndMT	155
IloEnv	156
IloEnvironmentMismatch	161
IloEqIntersection	162
IloEqPartition	163

IloEqUnion	164
IloException	165
IloExponent	167
IloExpr	168
IloExprArray	175
IloExpr::LinearIterator	177
IloExtractable	179
IloExtractableArray	182
IloFastMutex	184
IloFloatArray	188
IloFloatVar	189
IloFloatVarArray	190
IloFloor	191
IloFunction	192
IloGetClone	193
IloHalfPi	194
IloIfThen	195
IloInitMT	198
IloInt	199
IloIntArray	200
IloIntBinaryPredicate	204
IloIntExpr	206
IloIntExprArg	209
IloIntExprArray	211
IloIntSet	214
IloIntSet::Iterator	218
IloIntSetVar	219
IloIntSetVarArray	224
IloIntTernaryPredicate	226
IloIntTupleSet	228
IloIntTupleSetIterator	231

IloIntVar	232
IloIntArray	238
IloInverse	242
IloNaN	245
IloIterator	246
IloLinearTimeConverter	248
IloLog	249
IloMax	250
IloMaximize	251
IloMember	252
IloMin	253
IloMinimize	254
IloModel	255
IloModel::Iterator	258
IloMonotonicDecreasingNumExpr	260
IloMonotonicIncreasingNumExpr	261
IloMutexDeadlock	262
IloMutexNotOwner	263
IloMutexProblem	264
IloNot	266
IloNotMember	268
IloNullIntersect	269
IloNum	270
IloNumArray	271
IloNumColumn	274
IloNumColumnArray	277
IloNumExpr	279
IloNumExprArg	282
IloNumExprArray	284
IloNumExpr::NonLinearExpression	287
IloNumFunction	288

IloNumSet	289
IloNumVar	293
IloNumVarArray	299
IloNumVar::Type	304
IloObjective	305
IloObjective::Sense	311
IloOr	312
IloPack	316
IloPartition	321
IloPathLength	322
IloPathTransitFunction	327
IloPathTransitI	328
IloPi	330
IloPiecewiseLinear	331
IloPower	332
IloQuarterPi	333
IloRandom	334
IloRange	337
IloRangeArray	344
IloRound	351
IloSOS1	352
IloSOS1Array	355
IloSOS2	358
IloSOS2Array	361
IloScalProd	364
IloScalProd	365
IloScalProd	366
IloScalProd	367
IloSemaphore	368
IloSemiContVar	371
IloSemiContVarArray	375

IloSequence	.378
IloSolution	.381
IloSolutionArray	.392
IloSolution::Iterator	.393
IloSolutionIterator	.395
IloSolutionManip.	.397
IloSquare	.398
IloSubset	.399
IloSubsetEq	.400
IloSum	.401
IloTableConstraint	.402
IloThreeHalfPi	.404
IloTimer	.405
IloTwoPi	.407
IloTypedList	.408
IloTypedListManagerI.	.409
operator &&	.410
operator *	.411
operator new	.412
operator!.	.413
operator!=.	.414
operator%.	.415
operator%.	.416
operator%.	.417
operator+	.418
operator-.	.419
operator/.	.420
operator<	.421
operator<<	.422
operator<=	.423
operator==	.424

operator>	.425
operator>=	.426
operator>>	.427
operator	.428
Group optim.concert.xml	.428
IloXmlContext.	.428
IloXmlInfo	.435
IloXmlReader	.446
IloXmlWriter	.451
Group optim.concert.extensions	.454
IloCsvLine	.456
IloCsvReader	.460
IloCsvReader::IloColumnHeaderNotFoundException	.469
IloCsvReader::IloCsvReaderParameterException	.470
IloCsvReader::IloDuplicatedTableException	.471
IloCsvReader::IloFieldNotFoundException	.472
IloCsvReader::IloFileNotFoundException	.473
IloCsvReader::IloIncorrectCsvReaderUseException	.474
IloCsvReader::IloLineNotFoundException	.475
IloCsvReader::IloTableNotFoundException	.476
IloCsvReader::LineIterator	.477
IloCsvReader::TableIterator	.479
IloCsvTableReader	.481
IloCsvTableReader::LineIterator	.485
IloDifference.	.487
IloDifference.	.488
IloIntersection.	.489
IloIntervalList	.490
IloIntervalListCursor	.495
IloMax.	.498
IloMin	.499

IloNumToAnySetStepFunction	500
IloNumToAnySetStepFunctionCursor	509
IloNumToNumSegmentFunction	512
IloNumToNumSegmentFunctionCursor	519
IloNumToNumStepFunction	522
IloNumToNumStepFunctionCursor	528
IloUnion	530
IloUnion	531
operator *	532
operator+	533
operator-	534
operator<<	535
operator==	536
operator==	537
operator==	538
Group optim.cplex.cpp	538
ILOBARRIERCALLBACK0	545
ILOBRANCHCALLBACK0	546
ILOCONTINUOUSCALLBACK0	547
ILOCPLEXGOAL0	548
ILOCROSSOVERCALLBACK0	550
ILOCUTCALLBACK0	551
ILODISJUNCTIVECUTCALLBACK0	552
ILOFRACTIONALCUTCALLBACK0	553
ILOHEURISTICCALLBACK0	554
ILOINCUMBENTCALLBACK0	555
ILOLAZYCONSTRAINTCALLBACK0	556
ILOMIPCALLBACK0	557
ILONETWORKCALLBACK0	558
ILONODECALLBACK0	559
ILOPRESOLVECALLBACK0	560

ILOPROBINGCALLBACK0	561
ILOSIMPLEXCALLBACK0	562
ILOSOLVECALLBACK0	563
ILOUSERCUTCALLBACK0	564
IloCplex	565
IloCplex::Algorithm	606
IloCplex::BarrierCallbackI	608
IloCplex::BasisStatus	610
IloCplex::BasisStatusArray	611
IloCplex::BoolParam	612
IloCplex::BranchCallbackI	614
BranchCallbackI::BranchType	622
IloCplex::BranchDirection	623
IloCplex::BranchDirectionArray	624
IloCplex::Callback	625
IloCplex::CallbackI	627
Callback::Type	632
IloCplex::ContinuousCallbackI	633
IloCplex::ControlCallbackI	636
ControlCallbackI::IntegerFeasibility	644
IloCplex::CplexStatus	645
IloCplex::CrossoverCallbackI	648
IloCplex::CutCallbackI	650
IloCplex::DeleteMode	653
IloCplex::DisjunctiveCutCallbackI	654
IloCplex::DualPricing	656
IloCplex::Exception	657
IloCplex::FlowMIRCutCallbackI	659
IloCplex::FractionalCutCallbackI	661
IloCplex::Goal	663
IloCplex::GoalI	665

Goall::BranchType	.679
Goall::IntegerFeasibility	.680
Goall::IntegerFeasibilityArray	.681
IloCplex::HeuristicCallbackI	.682
IloCplex::IISStatus	.689
IloCplex::IISStatusArray	.690
IloCplex::IncumbentCallbackI	.691
IloCplex::IntParam	.694
IloCplex::InvalidCutException	.702
IloCplex::LazyConstraintCallbackI	.703
IloCplex::MIPCallbackI	.705
MIPCallbackI::NodeData	.712
IloCplex::MIPEmphasisType	.713
IloCplex::MultipleConversionException	.715
IloCplex::MultipleObjException	.717
IloCplex::NetworkCallbackI	.718
IloCplex::NodeCallbackI	.720
IloCplex::NodeEvaluator	.725
IloCplex::NodeEvaluatorI	.727
IloCplex::NodeSelect	.731
IloCplex::NumParam	.732
IloCplex::PWLFormulation	.735
IloCplex::PresolveCallbackI	.736
IloCplex::PrimalPricing	.738
IloCplex::ProbingCallbackI	.739
IloCplex::Quality	.742
IloCplex::SearchLimit	.746
IloCplex::SearchLimitI	.748
IloCplex::SimplexCallbackI	.750
IloCplex::SolveCallbackI	.752
IloCplex::StringParam	.757

IloCplex::UnknownExtractableException	758
IloCplex::UserCutCallbackI	759
IloCplex::VariableSelect	761
Group optim.cplex.cpp.piecelwise linear	762
IloCplex::PWLFormulation	764
Group optim.cplex.cpp.logical	764
Group optim.cplex.advanced.cpp	765
IloCplex::BranchCallbackI	767
IloCplex::ControlCallbackI	775
IloCplex::CutCallbackI	783
IloCplex::Goal	786
IloCplex::GoalI	788
IloCplex::HeuristicCallbackI	802
IloCplex::IncumbentCallbackI	809
IloCplex::LazyConstraintCallbackI	812
IloCplex::NodeCallbackI	814
IloCplex::SolveCallbackI	819
IloCplex::UserCutCallbackI	824

About This Manual

This reference manual documents the C++ application programming interface (API) of ILOG CPLEX. There are separate reference manuals for the Callable Library, Java, and C#.NET APIs. Following this table summarizing the groups in this manual, you will find more information:

- ◆ What Are the ILOG CPLEX Component Libraries?
- ◆ What You Need to Know
- ◆ Notation and Naming Conventions
- ◆ Related Documentation

What Are the ILOG CPLEX Component Libraries?

The ILOG CPLEX Component Libraries are designed to facilitate the development of applications to solve, modify, and interpret the results of linear, mixed integer, continuous convex quadratic, quadratically constrained, and mixed integer quadratic or quadratically constrained programming.

The ILOG CPLEX Component Libraries consist of:

- ◆ the CPLEX Callable Library, a C application programming interface (API), and
- ◆ ILOG Concert Technology, an object-oriented API for C++, Java, and C#.NET users.

ILOG Concert Technology is also part of ILOG Solver, enabling cooperative strategies using CPLEX and Solver together for solving difficult optimization problems.

What You Need to Know

This manual assumes that you are familiar with the operating system on which you are using ILOG CPLEX.

This Concert Technology library is written in the C++ programming language. If you use this product, this manual assumes you can write code in C++ and that you have a

working knowledge of a supported integrated development environment (IDE) for that language.

Notation and Naming Conventions

Throughout this manual:

- ◆ The names of classes and global functions defined in Concert Technology C++ API begin with `Ilo`. This convention helps prevent name space conflicts with user-written routines and other code libraries.
- ◆ The names of classes, member functions or methods, and arguments appear in `this typeface` (example: `IloCplex::solveFixed`).

Related Documentation

In addition to this *Reference Manual* documenting the C++ API, ILOG CPLEX also comes with these resources:

- ◆ *Getting Started with ILOG CPLEX* introduces you to ways of specifying models and solving problems with ILOG CPLEX.
- ◆ The *ILOG CPLEX User's Manual* explores programming with ILOG CPLEX in greater depth. It provides practical ideas about how to use CPLEX in your own applications and shows how and why design and implementation decisions in the examples were made.
- ◆ The *ILOG CPLEX Release Notes* highlight the new features and important changes in this version.
- ◆ The *ILOG CPLEX Callable Library Reference Manual* documents the C API of CPLEX.
- ◆ The *ILOG CPLEX Java Reference Manual* supplies detailed definitions of the Concert Technology Java interfaces and ILOG CPLEX Java classes.
- ◆ The *ILOG CPLEX C#.NET Reference Manual* supplies detailed definitions of the Concert Technology C#.NET interfaces and ILOG CPLEX C#.NET classes.
- ◆ Source code for examples is delivered in the standard distribution.
- ◆ A file named `readme.html` is delivered in the standard distribution. This file contains the most current information about platform prerequisites for ILOG CPLEX.

All of the manuals and Release Notes are available in online versions. The online documentation, in HTML format, can be accessed through standard HTML browsers.

Arrays

For most basic classes (such as `IloNumVar` or `IloConstraint`) in Concert Technology, there is also a corresponding class of arrays where the elements of the array are instances of that basic class. For example, elements of an instance of `IloConstraintArray` are instances of the class `IloConstraint`.

Arrays in an Environment

Every array must belong to an environment (an instance of `IloEnv`). In other words, when you create a Concert Technology array, you pass an instance of `IloEnv` as a parameter to the constructor. All the elements of a given array must belong to the same environment.

Extensible Arrays

Concert Technology arrays are extensible. That is, you can add elements to the array dynamically. You add elements by means of the `add` member function of the array class.

You can also remove elements from an array by means of its `remove` member function.

References to an array change whenever an element is added to or removed from the array.

Arrays as Handles

Like other Concert Technology objects, arrays are implemented by means of two classes: a handle class corresponding to an implementation class. An object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Concert Technology user, you will be working primarily with handles.

Copying Arrays

Many handles may point to the same implementation object. This principle holds true for arrays as well as other handle classes in Concert Technology. When you want to create more than one handle for the same implementation object, you should use either the copy constructor or the assignment operator of the array class. For example,


```

new impl      IloNumArray array(env);      // creates a handle pointing to
same impl      IloNumArray array1(array); // creates a handle pointing to
of array      IloNumArray array2;          // creates an empty handle
               array2 = array;             // sets impl of handle array2 to impl

```

Programming Hint: Using Arrays Efficiently

If your application only reads an array (that is, if your function does not modify an element of the array), then we recommend that you pass the array to your function as a `const` parameter. This practice forces Concert Technology to access the `const` conversion of the index operator (that is, `operator[]`), which is faster.

Assert and NDEBUG

Most member functions of classes in Concert Technology are inline functions that contain an `assert` statement. This statement asserts that the invoking object and the member function parameters are consistent; in some member functions, the `assert` statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

Compilation with `assert` statements will not prevent core dumps by incorrect code. Instead, compilation with `assert` statements moves the execution of the incorrect code (the core dump, for example) to a place where you can see what is causing the problem in a source code debugger. Correctly written code will never cause one of these Concert Technology `assert` statements to fail.

Branch & Cut

CPLEX uses *branch & cut search* when solving mixed integer programming (MIP) models. The branch & cut search procedure manages a search tree consisting of *nodes*. Every node represents an LP or QP subproblem to be processed; that is, to be solved, to be checked for integrality, and perhaps to be analyzed further. Nodes are called *active* if they have not yet been processed. After a node has been processed, it is no longer active. Cplex processes active nodes in the tree until either no more active nodes are available or some limit has been reached.

A *branch* is the creation of two new nodes from a parent node. Typically, a branch occurs when the bounds on a single variable are modified, with the new bounds remaining in effect for that new node and for any of its descendants. For example, if a branch occurs on a binary variable, that is, one with a lower bound of 0 (zero) and an

upper bound of 1 (one), then the result will be two new nodes, one node with a modified upper bound of 0 (the downward branch, in effect requiring this variable to take only the value 0), and the other node with a modified lower bound of 1 (the upward branch, placing the variable at 1). The two new nodes will thus have completely distinct solution domains.

A *cut* is a constraint added to the model. The purpose of adding any cut is to limit the size of the solution domain for the continuous LP or QP problems represented at the nodes, while not eliminating legal integer solutions. The outcome is thus to reduce the number of branches required to solve the MIP.

As an example of a cut, first consider the following constraint involving three binary (0-1) variables:

$$20x + 25y + 30z \leq 40$$

That sample constraint can be strengthened by adding the following cut to the model:

$$1x + 1y + 1z \leq 1$$

No feasible integer solutions are ruled out by the cut, but some fractional solutions, for example (0.0, 0.4, 1.0), can no longer be obtained in any LP or QP subproblems at the nodes, possibly reducing the amount of searching needed.

The branch & cut method, then, consists of performing branches and applying cuts at the nodes of the tree. Here is a more detailed outline of the steps involved.

First, the branch & cut tree is initialized to contain the root node as the only active node. The root node of the tree represents the entire problem, ignoring all of the explicit integrality requirements. Potential cuts are generated for the root node but, in the interest of keeping the problem size reasonable, not all such cuts are applied to the model immediately. If possible, an incumbent solution (that is, the best known solution that satisfies all the integrality requirements) is established at this point for later use in the algorithm. Such a solution may be established either by CPLEX or by a user who specifies a starting solution by means of the Callable Library routine `CPXcopymipstart` or the Concert Technology method `IloCplex::setVectors`. Alternatively, when solving a sequence of modified problems, the user can set the mip start parameter (`CPX_PARAM_MIPSTART` in the Callable Library or `MIPStart` in Concert Technology) to the value 1 (one) to indicate that the solution of the previous problem should be used as a starting incumbent for the present problem.

When processing a node, CPLEX starts by solving the continuous relaxation of its subproblem. that is, the subproblem without integrality constraints. If the solution violates any cuts, CPLEX may add some or all of them to the node problem and may

resolve it, if CPLEX has added cuts. This procedure is iterated until no more violated cuts are detected (or deemed worth adding at this time) by the algorithm. If at any point in the addition of cuts the node becomes infeasible, the node is pruned (that is, it is removed from the tree).

Otherwise, CPLEX checks whether the solution of the node-problem satisfies the integrality constraints. If so, and if its objective value is better than that of the current incumbent, the solution of the node-problem is used as the new incumbent. If not, branching will occur, but first a heuristic method may be tried at this point to see if a new incumbent can be inferred from the LP-QP solution at this node, and other methods of analysis may be performed on this node. The branch, when it occurs, is performed on a variable where the value of the present solution violates its integrality requirement. This practice results in two new nodes being added to the tree for later processing.

Each node, after its relaxation is solved, possesses an optimal objective function value Z . At any given point in the algorithm, there is a node whose Z value is better (less, in the case of a minimization problem, or greater for a maximization problem) than all the others. This Best Node value can be compared to the objective function value of the incumbent solution. The resulting MIP Gap, expressed as a percentage of the incumbent solution, serves as a measure of progress toward finding and proving optimality. When active nodes no longer exist, then these two values will have converged toward each other, and the MIP Gap will thus be zero, signifying that optimality of the incumbent has been proven.

It is possible to tell CPLEX to terminate the branch & cut procedure sooner than a completed proof of optimality. For example, a user can set a time limit or a limit on the number of nodes to be processed. Indeed, with default settings, CPLEX will terminate the search when the MIP Gap has been brought lower than 0.0001 (0.01%), because it is often the case that much computation is invested in moving the Best Node value after the eventual optimal incumbent has been located. This termination criterion for the MIP Gap can be changed by the user, of course.

Callbacks in Concert Technology

A callback is an object with a `main` method implemented by a user. This user-written `main` method is called by the `IloCplex` algorithm at specific points during optimization.

Callbacks may be called repeatedly at various points during optimization; for each place a callback is called, ILOG CPLEX provides a separate callback class (derived from the class `IloCplex::CallbackI`). Such a callback class provides the specific API as a protected method to use for the particular implementation.

There are two varieties of callbacks:

- ◆ Query callbacks enable your application to retrieve information about the current solution in an instance of `IloCplex`. The information available depends on the

algorithm (primal simplex, dual simplex, barrier, mixed integer, or network) that you are using. For example, a query callback can return the current objective value, the number of simplex iterations that have been completed, and other details. Query callbacks can also be called from presolve, probing, fractional cuts, and disjunctive cuts.

- ◆ Control callbacks enable you to direct the search when you are solving a MIP in an instance of `IloCplex`. For example, control callbacks enable you to select the next node to process or to control the creation of subnodes (among other possibilities). Control callbacks are an advanced feature of ILOG CPLEX, and as such, they require a greater degree of familiarity with CPLEX algorithms.

You do not create instances of the class `IloCplex::CallbackI`; rather, you use one of its child classes to implement your own callback. In order to implement your user-written callbacks with an instance of `IloCplex`, you should follow these steps:

- ◆ Determine which kind of callback you want to write, and choose the appropriate class for it. The class hierarchy in Tree may give you some ideas here.
- ◆ Derive your own subclass, `MyCallbackI`, say, from the appropriate predefined callback class.
- ◆ In your subclass of the callback class, use the protected API defined in the base class to implement the `main` routine of your user-written callback. (All constructors of predefined callback classes are protected; they can be called only from user-written derived subclasses.)
- ◆ In your subclass, implement the method `duplicateCallback`.
- ◆ Write a function `myCallback`, say, that creates an instance of your implementation class in the Concert Technology environment and returns it as an `IloCplex::Callback` handle.
- ◆ Create an instance of your callback class and pass it to the member function `IloCplex::use`.

There are macros of the form `ILOXXXCALLBACKn` (for `n` from 0 to 7) available to facilitate steps 2 through 5, where `XXX` stands for the particular callback under construction and `n` stands for the number of arguments that the function written in step 5 is to receive in addition to the environment argument.

You can use one instance of a callback with only one instance of `IloCplex`. When you use a callback with a second instance of `IloCplex`, a copy will be automatically created using the method `duplicateCallback`, and that copy will be used instead.

Also, an instance of `IloCplex` takes account of only one instance of a particular callback at any given time. That is, if you call `IloCplex::use` more than once with the same class of callback, the last call overrides any previous one. For example, you can

use only one primal simplex callback at a time, or you can use only one network callback at a time; and so forth.

Existing extractables should never be modified within a callback. Temporary extractables, such as arrays, expressions, and range constraints, can be created and modified. Temporary extractables are often useful, for example, for computing cuts.

Compound Termination Criteria in Callbacks

Here is an example showing you how to terminate optimization after a given period of time if the solution is good enough. It uses one of the predefined macros to facilitate writing a control callback with a timer, a time limit, and a way to recognize a good enough solution.

```
ILOMIPCALLBACK3(MyCallback, IloTimer, myTimer,
                 IloNum, timeLimit,
                 IloNum, acceptableGap) {
    if ( hasIncumbent() ) {
        IloNum objval = getIncumbentObjValue();
        IloNum bound  = getBestObjValue();
        IloNum gap = fabs(objval - bound) / (1.0 + fabs(bound));
        if ( myTimer.getTime() > timeLimit    &&
             gap < acceptableGap ) {
            cout << "Good enough solution at = "
                  << myTimer.getTime() << " sec., gap = "
                  << gap * 100.00 << ", quitting" << endl;
            abort();
        }
    }
}
```

Column-Wise Modeling

Concert Technology supports column-wise modeling, a technique widely used in the math programming and operations research communities to build a model column by column. In Concert Technology, creating a new column is comparable to creating a new variable and adding it to a set of constraints. You use an instance of `IloNumColumn` to do so. An instance of `IloNumColumn` allows you to specify to which constraints or other extractable objects Concert Technology should add the new variable along with its data. For example, in a linear programming problem (an LP), if the new variable will appear in some linear constraints as ranges (instances of `IloRange`), you need to specify the list of such constraints along with the non zero coefficients (a value of `IloNum`) for each of them.

You then create a new column in your model when you create a new variable with an instance of `IloNumColumn` as its parameter. When you create the new variable, Concert Technology will add it along with appropriate parameters to all the extractable objects you have specified in the instance of `IloNumColumn`.

Instead of building an instance of `IloNumColumn`, as an alternative, you can use a column expression directly in the constructor of the `variable`. You can also use instances of `IloNumColumn` within column expressions.

The following undocumented classes provide the underlying mechanism for column-wise modeling:

- ◆ `IloAddValueToObj`

- ◆ `IloAddValueToRange`

The following operators are useful in column-wise modeling:

- ◆ in the class `IloRange`,

`IloAddValueToRange operator() (IloNum value);`

- ◆ in the class `IloObjective`,

`IloAddValueToObj operator () (IloNum value);`

That is, the `operator ()` in extractable classes, such as `IloRange` or `IloObjective`, creates descriptors of how Concert Technology should add the new, yet-to-be-created variable to the invoking extractable object.

You can use the `operator +` to link together the objects returned by `operator ()` to form a column. You can then use an instance of `IloNumColumn` to build up column expressions within a programming loop and thus save them for later use or to pass them to functions.

Here is how to use an instance of `IloNumColumn` with operators from `IloRange` and `IloObjective` to create a column with a coefficient of 2 in the objective, with 10 in `range1`, and with 3 in `range2`. The example then uses that column when it creates the new variable `newvar1`, and it uses column expressions when it creates `newvar2` and `newvar3`.

```
IloNumColumn col = obj(2) + range1(10) + range2(3);
IloNumVar newvar1(col);
IloNumVar newvar2(col + range3(17));
IloNumVar newvar3(range1(1) + range3(3));
```

In other words, given an instance `obj` of `IloObjective` and the instances `range1`, `range2`, and `range3` of `IloRange`, those lines create the new variables `newvar1`, `newvar2`, and `newvar3` and add them as linear terms to `obj`, `range1`, and `range3` in the following way:

```
obj: + 2*newvar1 + 2*newvar2
range1: +10*newvar1 + 10*newvar2 + 1*newvar3
range2: + 3*newvar1 + 3*newvar2
```

```
range3: + 17*newvar2 +3*newvar3
```

For more information, refer to the documentation of [IloNumColumn](#), [IloObjective](#), and [IloRange](#).

Exceptions, Errors

An exception is thrown; it is not allocated in a Concert Technology environment; it is not allocated on the C++ heap. It is not necessary for you as a programmer to delete an exception explicitly. Instead, the system calls the constructor of the exception to create it, and the system calls the destructor of the exception to delete it.

When exceptions are enabled on a platform that supports C++ exceptions, an instance of a class of Concert Technology is able to throw an exception in case of error. On platforms that do not support C++ exceptions, it is possible for Concert Technology to exit in case of error.

Programming Hint: Throwing and Catching Exceptions

Exceptions are thrown by value. They are not allocated on the C++ heap, nor in a Concert Technology environment. The correct way to catch an exception is to catch a reference to the error (indicated by the ampersand &), like this:

```
catch(IloException& oops);
```

Extraction

Concert Technology offers classes for you to design a *model* of your problem. You can then invoke an algorithm to extract information from your model to solve the problem. In this context, an algorithm is an instance of a class such as `IloCplex`, documented in the *ILOG CPLEX Reference Manual*, or `IloSolver`, documented in the *ILOG Solver Reference Manual*.

For details about what each algorithm extracts from a model, see the reference manual documenting that algorithm. For example, the *ILOG CPLEX Reference Manual* lists precisely which classes of Concert Technology are extracted by an instance of `IloCplex`. In general terms, an instance of `IloCplex` extracts a model as rows and columns, where the columns indicate decision variables of the model. Also in general terms, an instance of `IloSolver` extracts an instance of a class whose name begins `Ilo` to a corresponding instance of a class whose name begins `Ilc`. For example, an instance of `IloAllDiff` is extracted by `IloSolver` as an instance of `IlcAllDiff`.

Goals

Goals can be used to control the branch and cut search in `IloCplex`. Goals are implemented in the class `IloCplex::GoalI`. The class `IloCplex::Goal` is the handle class. That is, it contains a pointer to an instance of `IloCplex::GoalI` along with accessors of objects in the implementation class.

To implement your own goal, you need to subclass `IloCplex::GoalI` and implement its virtual member functions `execute` and `duplicateGoal`. The method `execute` controls the branch & cut search. The method `duplicateGoal` creates a copy of the invoking goal object to be used for parallel branch & cut search. Implementing your goal can be greatly simplified if you use one of the macros `ILOCPLEXGOALn`.

Every branch & cut node maintains a goal stack, possibly containing `IloCplex::GoalI` objects. After `IloCplex` solves the relaxation of a node, it pops the top goal from the goal stack and calls its method `execute`. There are several types of goals:

- ◆ If `OrGoal` is executed, `IloCplex` will create child nodes. Each of the child nodes will be initialized with a copy of the goal stack of the current node. Then, for each child node, the specified goal in the `OrGoal` is pushed onto the corresponding goal stack of the child node. Finally, the current node is deleted. (See `IloCplex#GoalI::OrGoal` for a more detailed discussion.)
- ◆ If a cut goal is executed, the constraint will be added to the current node relaxation. Constraint goals are provided to represent both local and global cuts. Local cut goals are conventionally used to express branching.
- ◆ If `AndGoal` is executed, its subgoals are simply pushed onto the stack. (See `IloCplex::GoalI::AndGoal` for a more detailed discussion.)
- ◆ If `IloCplex::GoalI::FailGoal` is executed, `IloCplex` will prune the current node; that is, it will discontinue the search at the current node. `IloCplex` will continue with another node if there is one still available in the tree.
- ◆ If `IloCplex::GoalI::SolutionGoal` is executed, `IloCplex` will attempt to inject a user-provided solution as a new incumbent. Before `ILOG CPLEX` accepts the injected solution, it first tests whether the injected solution is feasible with respect to the model and goals.
- ◆ When `ILOG CPLEX` executes any other goal, the returned goal is simply pushed onto the stack.

`IloCplex` continues popping goals from the goal stack until `OrGoal` or `FailGoal` is executed, or until the stack becomes empty; in the case of an empty stack, it will continue with a built-in search strategy.

The predefined goals `OrGoal` and `AndGoal` allow you to combine goals. `AndGoal` allows you to execute different goals at one node, while `OrGoal` allows you to execute different goals on different, newly created nodes. A conventional use of these two goals in a return statement of a user-written goal looks like this:

```
return AndGoal ( OrGoal (branch1, branch2), this);
```

This `AndGoal` first pushes `this` (the goal currently being executed) onto the goal stack, and then it pushes the `OrGoal`. Thus the `OrGoal` is on top of the stack and will be executed next. When the `OrGoal` executes, it creates two new nodes and copies the remaining goal stack to both of them. Thus both new nodes will have `this` goal on top of the goal stack at this point. Then the `OrGoal` proceeds to push `branch1` onto the goal stack of the first child node and `branch2` onto the goal stack of the second child node. Conventionally, `branch1` and `branch2` contain cut goals, so by executing `branch1` and `branch2` at the respective child nodes, the child nodes will be restricted to represent smaller subproblems than their parent. After `branch1` and `branch2` have been executed, `this` is on top of the node stack of both child nodes; that is, both child nodes will continue branching according to the same rule. In summary, this example creates the branches `branch1` and `branch2` and continues in both branches to control the same search strategy `this`.

To perform a search using a goal, you need to solve the extracted model by calling the method `IloCplex::solve(goal)` with the goal to use as an argument instead of the standard `IloCplex::solve`. The method `solve(goal)` simply pushes the `goal` onto the goal stack of the root node before calling the standard `solve`.

See Also

`IloCplex::Goal` and `IloCplex::GoalI`

Handle Class

Most Concert Technology entities are implemented by means of two classes: a handle class and an implementation class, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Concert Technology user, you will be working primarily with handles.

As handles, these objects should be passed in either of these ways:

- ◆ as `const` by value (when no change is involved);
- ◆ by reference (when the function to which the handle is passed changes the implementation of that handle).

They should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Lazy Copy

Concert Technology makes a lazy copy when you use any of the following objects inside a predefined Concert Technology object:

- ◆ an expression (an instance of `IloExpr` or one of its subclasses),
- ◆ a column (an instance of `IloNumColumn`),
- ◆ or a set (such as an instance of `IloIntSet`).

That is, a physical copy of those objects is created only when needed.

In Concert Technology, expressions, columns, and sets are implemented by handle classes and corresponding implementation classes. One or more handles may point to the same implementation object. For example, many instances of the handle class `IloNumColumn` may point to the same implementation object.

A handle may be empty; that is, it may point to 0 (zero). You can test whether a handle is empty by means of the member function `handle.getImpl`. If that member function returns 0, the handle points to a null implementation.

When you modify an expression, a column, or a set that has been used in a Concert Technology object, Concert Technology considers whether the handle you are modifying is the sole reference to the corresponding implementation object. If so, Concert Technology simply makes the modification.

In contrast, if the handle you are modifying points to an implementation object that is used by other objects predefined in Concert Technology, Concert Technology first copies the implementation object for the handle you are modifying and then makes the modification. The other handles pointing to the original implementation object remain unchanged and your modification has no impact on them.

Examples:

Here is an example illustrating lazy copy of variables:

```
IloEnv env;  
IloIntVar a1(env, 0, 10);  
IloIntVar a2(env, 0, 10);  
IloIntVar a3(env, 0, 10);  
IloExpr e = a1 + a2;  
IloConstraint ct = e <= 10;  
e += a3;  
return 0;
```

Because of the lazy copy, even though `a3` was added to `A`, `ct` uses only `a1` and `a2`.

Normalization: Reducing Linear Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, $x + 3y$ is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, $x + 3y + 2x$. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)`.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse)`.

In classes such as `IloExpr` or `IloRange`, there are member functions that check the setting of the member function `IloEnv::setNormalizer` in the environment and behave accordingly. The documentation of those member functions indicates how they behave with respect to normalization.

When you set `IloEnv::setNormalizer(IloFalse)`, those member functions assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of a class to fail if you do not compile with the flag `-DNDEBUG`.

By default, those member functions attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of a failed assertion in case of duplicates.

For more information, refer to the documentation of `IloEnv`, `IloExpr`, and `IloRange`.

Notification

You may modify the elements of a model in Concert Technology. For example, you may add or remove constraints, change the objective, add or remove columns, add or remove rows, and so forth.

In order to maintain consistency between a model and the algorithms that may use it, Concert Technology notifies algorithms about changes in the objects that the algorithms have extracted. In this manual, member functions that are part of this notification system are indicated like this:

Note: *This member function notifies Concert Technology algorithms about this change of this invoking object.*

Deletion of Extractables

As a modeling layer, Concert allows the creation and destruction of extractables. This is accessible through the method `IloExtractable::end()` and `IloExtractableArray::endElements()` method. The goal of these methods is to reclaim memory associated with the deleted objects while maintaining the safest possible Concert environment. In this context, a safe Concert environment is defined by the property that no object points to a deleted object; this is referred to as a dangling pointer in C++.

There exist two paradigms to ensure the safeness of the delete operation. The first, linear mode, comes from math programming and is possible only on extractables and objects used in linear programming. The second, safe generic mode, is more strict and is valid on all Concert extractables.

You can access both paradigms by calling `IloEnv::setDeleter(IloDeleterMode mode)`, where mode may be `IloLinearDeleterMode` or `IloSafeDeleterMode`.

Linear Mode

To use linear mode, you must either

- ◆ call `IloEnv::setDeleter(IloLinearDeleterMode)`, or
- ◆ refrain from calling `IloEnv::setDeleter()`, as it is the default mode.

In linear mode, the following behavior is implemented:

- ◆ If a range constraint is deleted, it is removed from the models that contain it.

- ◆ If a variable is deleted, its coefficient is set to 0 in the ranges, expressions, and objectives where it appears. The variable is removed from the SOS1, SOS2, and IloConversion where it appears.

Example

This example tests the linear mode deletion of a variable x.

```
void TestLinearDeleter() {
    IloEnv env;
    env.out() << "TestLinearDeleter" << endl;
    try {
        IloModel model(env);
        IloNumVar x(env, 0, 10, "x");
        IloNumVar y(env, 0, 10, "y");
        IloConstraint con = (x + y <= 0);
        IloConstraint con2 = y >= 6;
        IloNumVarArray ar(env, 2, x, y);
        IloSOS1 sos(env, ar, "sos");
        model.add(con);
        model.add(con2);
        model.add(sos);
        env.out() << "Before Delete" << endl;
        env.out() << model << endl;
        x.end();
        con2.end();
        env.out() << "After Delete" << endl;
        env.out() << model << endl;
    } catch (IloException& e) {
        cout << "Error : " << e << endl;
    }
    env.end();
}
```

The example produces the following output:

```
TestLinearDeleter
Before Delete
IloModel model0 = {
IloRange rng3(    1 * x + 1 * y ) <= 0

IloRange rng46 <=(    1 * y )

IloSOS1I (sos)
  _varArray [x(F)[0..10], y(F)[0..10]]
  _valArray []
}

After Delete
```

```

IloModel model0 = {
  IloRange rng3(    1 * y ) <= 0

  IloSOS1I (sos)
    _varArray [y(F)[0..10]]
    _valArray []
}

```

Safe Generic Mode

To use safe generic mode, you must:

- ◆ call `IloEnv::setDeleter(IloSafeDeleterMode)`, and
- ◆ add `#include <ilconcert/ilodeleter.h>` to your program.

In this mode, the environment builds a dependency graph between all extractables. This graph contains all extractables created

- ◆ after a call to `IloEnv::setDeleter(IloSafeDeleterMode)` and
- ◆ before a call to `IloEnv::unsetDeleter()`.

Objects not managed by this dependency graph are referred to here as "nondeletable". An attempt to delete a nondeletable object will throw an exception.

We recommended that you create this graph just after the creation of the environment and that you refrain from using `IloEnv::unsetDeleter`. We make these recommendations because building an incomplete dependency graph is very error prone and should only be attempted by advanced users. A good example of this incomplete graph is the separation of a model between a nondeletable base model and deletable extensions of this base model.

Calling `IloExtractable::end()` on extractable `xi` will succeed only if no other extractable uses extractable `xi`. If this is not the case, a call to `IloExtractable::end()` will throw an exception `IloDeleter::RequiresAnotherDeletionException` indicating which extractable uses the extractable that you want to delete.

Example

This example shows an attempt to delete one extractable that is used by another.

```

void TestSafeDeleter() {
  IloEnv env;
  env.out() << "TestSafeDeleter" << endl;
  env.setDeleter(IloSafeDeleterMode);
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10);

```

```

        IloNumVar y(env, 0, 10);
        IloConstraint con = (x + y <= 0);
        try {
            x.end();
        } catch (IloDeleter::RequiresAnotherDeletionException &e) {
            cout << "Caught " << e << endl;
            e.getUsers()[0].end();
            e.end();
        }
        x.end();
    } catch (IloException& e) {
        cout << "Error : " << e << endl;
    }
    env.unsetDeleter();
    env.end();
}

```

The example produces the following output:

```

TestSafeDeleter
Caught You cannot end x1(F)[0..10] before IloRange rng3( 1 * x1 + 1 * x2
) <= 0

```

To address this, you should use the [IloExtractableArray::endElements\(\)](#) method. With this method, all extractables appearing in the array are deleted one after another. Thus, if an extractable is used by another extractable and this other extractable is deleted before the first one, the system will not complain and will not throw an exception.

Example

This example illustrates the use of the `endElements()` method

```

void TestSafeDeleterWithArray() {
    IloEnv env;
    env.out() << "TestSafeDeleterWithArray" << endl;
    env.setDeleter(IloSafeDeleterMode);
    try {
        IloModel model(env);
        IloNumVar x(env, 0, 10);
        IloNumVar y(env, 0, 10);
        IloConstraint con = (x + y <= 0);
        IloExtractableArray ar(env, 2, con, x);
        ar.endElements();
    } catch (IloException& e) {
        cout << "Error : " << e << endl;
    }
}

```

```
env.unsetDeleter();  
env.end();  
}
```

The example will not throw an exception.

Note: *Please note that in this last example, the constraint `con` must appear before the variable `x` as it will be deleted before the variable `x`.*

Group optim.concert

The ILOG Concert API.

Classes Summary	
IloAlgorithm	The base class of algorithms in Concert Technology.
IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.
IloAllDiff	A constraint that forces constrained integer variables to assume different values in a model.
IloAllMinDistance	A constraint on the minimum absolute distance between a pair of variables in an array.
IloAnd	A conjunctive constraint that defines a logical conjunctive-AND among other constraints.
IloAnyArray	The array class of the enumerated type definition <code>IloAny</code> .
IloAnyBinaryPredicate	A class to define binary predicates on objects in a model.
IloAnySet	A class whose instances represent a set of enumeration values.
IloAnySet::Iterator	An iterator to traverse the elements of <code>IloAnySet</code> .
IloAnySetVar	Class to represent a set of enumerated values as a constrained variable.

IloAnySetVarArray	The array class of the set variable class <code>IloAnySetVar</code> .
IloAnyTernaryPredicate	A class to define ternary predicates on objects in a model.
IloAnyTupleSet	
IloAnyTupleSetIterator	
IloAnyVar	A class to represent an enumerated variable.
IloAnyVarArray	A class to represent an array of enumerated variables.
IloArray	A template to create classes of arrays for elements of a given class.
IloBarrier	A system class to synchronize threads at a specified number.
IloBaseEnvMutex	A class to initialize multithreading in an application.
IloBoolArray	
IloBoolVar	
IloBoolVarArray	
IloBox	A class of multidimensional boxes for multidimensional placement problems.
IloCondition	
IloConstraint	
IloConstraintArray	
IloConversion	
IloDiff	
IloDistribute	
IloEmptyHandleException	The class of exceptions thrown if an empty handle is passed.
IloEnv	The class of environments for models or algorithms in Concert Technology.
IloEnvironmentMismatch	
IloException	
IloExpr	
IloExprArray	
IloExpr::LinearIterator	An iterator over the linear part of an expression.
IloExtractable	

IloExtractableArray	
IloFastMutex	
IloFunction	
IloIfThen	
IloIntArray	
IloIntBinaryPredicate	
IloIntExpr	The class of integer expressions in Concert Technology 2.0.
IloIntExprArg	A class used internally in Concert Technology.
IloIntExprArray	The array class of <code>IloIntExpr</code> .
IloIntSet	
IloIntSet::Iterator	
IloIntSetVar	The class <code>IloIntSetVar</code> .
IloIntSetVarArray	The array class of the set variable class for integer values.
IloIntTernaryPredicate	
IloIntTupleSet	
IloIntTupleSetIterator	
IloIntVar	
IloIntVarArray	The array class of <code>IloIntVar</code> .
IloInverse	
IloIterator	A template to create iterators for a class of extractable objects.
IloLinearTimeConverter	
IloModel	
IloModel::Iterator	
IloMutexDeadlock	The class of exceptions thrown due to mutex deadlock.
IloMutexNotOwner	The class of exceptions thrown.
IloMutexProblem	
IloNot	
IloNumArray	
IloNumColumn	
IloNumColumnArray	
IloNumExpr	The class of numeric expressions in a Concert model.
IloNumExprArg	A class used internally in Concert Technology.
IloNumExprArray	The class <code>IloNumExprArray</code> .

IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numerical constant of a non-linear expression is set or queried.
IloNumSet	The class <code>IloNumSet</code> .
IloNumVar	
IloNumVarArray	The array class of <code>IloNumVar</code> .
IloObjective	
IloOr	
IloPack	
IloPathLength	
IloPathTransitI	
IloRandom	
IloRange	
IloRangeArray	
IloSOS1	
IloSOS1Array	
IloSOS2	
IloSOS2Array	
IloSemaphore	
IloSemiContVar	
IloSemiContVarArray	
IloSequence	
IloSolution	
IloSolution::Iterator	
IloSolutionIterator	
IloSolutionManip	
IloTimer	
IloTypedList	
IloTypedListManagerI	

Macros Summary	
ILOANYBINARYPREDICATE0	
ILOANYTERNARYPREDICATE0	
IloFloatArray	
IloFloatVar	
IloFloatVarArray	The array class of <code>IloFloatVar</code> .
IloHalfPi	
ILOINTBINARYPREDICATE0	

ILOINTTERNARYPREDICATE0	
IloPi	
IloQuarterPi	
IloThreeHalfPi	
IloTwoPi	

Enumerations Summary	
IloAlgorithm::Status	An enumeration for the class IloAlgorithm .
IloDeleterMode	An enumeration to set the mode of an IloDeleter .
IloNumVar::Type	
IloObjective::Sense	

Type Definitions Summary	
IloAny	
IloBool	
IloCplex::Status	An enumeration for the class IloAlgorithm .
IloInt	
IloNum	
IloNumFunction	
IloPathTransitFunction	
IloSolutionArray	

Global Functions Summary	
IloAbs	
IloAbstraction	Returns a constraint that abstracts the values of one array into the abstract value of another array.
IloAdd	
IloAllNullIntersect	

IloArcCos	
IloBoolAbstraction	A function to create a <i>constraint</i> that abstracts an array of Boolean variables.
IloCard	
IloCeil	
IloDisableNANDetection	
IloDiv	
IloEnableNANDetection	
IloEndMT	
IloEqIntersection	
IloEqPartition	
IloEqUnion	
IloExponent	
IloFloor	
IloGetClone	
IloInitMT	
IloIsNAN	
IloLog	
IloMax	
IloMaximize	
IloMember	
IloMin	
IloMinimize	
IloMonotonicDecreasingNumExpr	
IloMonotonicIncreasingNumExpr	
IloNotMember	
IloNullIntersect	
IloPartition	
IloPiecewiseLinear	
IloPower	
IloRound	
IloScalProd	
IloScalProd	
IloScalProd	
IloScalProd	
IloSquare	
IloSubset	
IloSubsetEq	
IloSum	
IloTableConstraint	

operator &&	
operator *	
operator new	
operator!	
operator!=	
operator%	
operator%	
operator%	
operator+	
operator-	
operator/	
operator<>	
operator	
operator	
operator==	
operator>	
operator>=	
operator>>	
operator	

Global Variables Summary	
ILO_NO_MEMORY_MANAGER	

Description

Concert Technology offers a C++ library of classes and functions that enable you to design models of problems for both math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) and constraint programming solutions.

ILOANYBINARYPREDICATE0

Category

Macro

Synopsis

```
ILOANYBINARYPREDICATE0(name)
ILOANYBINARYPREDICATE1(name, type1, nameArg1)
ILOANYBINARYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILOANYBINARYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3,
nameArg3)
ILOANYBINARYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3,
nameArg3, type4, nameArg4)
ILOANYBINARYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3,
nameArg3, type4, nameArg4, type5, nameArg5)
ILOANYBINARYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3,
nameArg3, type4, nameArg4, type5, nameArg5, type6, nameArg6)
```

Description

This macro defines a predicate class named `nameI` with n data members for use in a model. When n is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type T_i and a name `datai`. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class [IloAnyBinaryPredicate](#) that points to it.

You are not obliged to use this macro to define binary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcAnyPredicateI`, documented in the *ILOG Solver Reference Manual*.

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also

[IloAnyBinaryPredicate](#)

ILOANYTERNARYPREDICATE0

Category Macro

Synopsis **ILOANYTERNARYPREDICATE0**(name)

Description This macro defines a predicate class named `nameI` with n data members for use in a model. When n is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type T_i and a name `datai`. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class [IloAnyTernaryPredicate](#) that points to it.

You are not obliged to use this macro to define ternary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcAnyPredicateI` (documented in the *ILOG Solver Reference Manual*).

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also [IloAnyTernaryPredicate](#)

ILOINTBINARYPREDICATE0

Category	Macro
Synopsis	ILOINTBINARYPREDICATE0 (name)
Description	This macro defines a predicate class named <code>nameI</code> with n data members for use in a model. When n is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type T_i and a name <code>data_i</code>. The call to the macro must be followed immediately by the body of the <code>isTrue</code> member function of the predicate class being defined. Besides the definition of the class <code>nameI</code>, this macro also defines a function named <code>name</code> that creates an instance of the class <code>nameI</code> and that returns an instance of the class IloIntBinaryPredicate that points to it. You are not obliged to use this macro to define binary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing <code>IlcIntPredicateI</code>, documented in the <i>ILOG Solver Reference Manual</i>. Since the argument <code>name</code> is used to name the predicate class, it is not possible to use the same name for several predicate definitions.
See Also	IloIntBinaryPredicate

ILOINTTERNARYPREDICATE0

Category Macro

Synopsis **ILOINTTERNARYPREDICATE0**(name)

Description This macro defines a predicate class named `nameI` with n data members for use in a model. When n is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type T_i and a name `datai`. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class [IloIntTernaryPredicate](#) that points to it.

You are not obliged to use this macro to define ternary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcIntPredicateI` (documented in the *ILOG Solver Reference Manual*).

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also [IloIntTernaryPredicate](#)

ILO_NO_MEMORY_MANAGER

Category Global Variable

Definition File ilconcert/ilosys.h

This operating-system environment variable enables you to control the memory manager of Concert Technology.

Concert Technology uses its own memory manager to provide faster memory allocation for certain Concert Technology objects. The use of this memory manager can hide memory problems normally detected by memory usage applications (such as Rational Purify, for example). If you are working in a software development environment capable of detecting bad memory access, you can use this operating-system environment variable to turn off the Concert Technology memory manager in order to detect such anomalies during software development.

For example, if you are working in such a development environment on a personal computer running Microsoft NT, use this statement:

```
set ILO_NO_MEMORY_MANAGER=1
```

If you are working on a UNIX platform, using a C-shell, here is one way of setting this environment variable:

```
setenv ILO_NO_MEMORY_MANAGER
```

IloAbs

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloAbs(const IloNumExprArg arg)
public IloNum IloAbs(IloNum val)
public IloNum IloPower(IloNum val1,
                     IloNum val2)
public IloIntExprArg IloAbs(const IloIntExprArg arg)
```

Description Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

IloAbs returns the absolute value of its argument.

What Is Extracted

IloAbs is extracted by an instance of IloSolver (documented in the *ILOG Solver Reference Manual*) as an instance of IlcAbs.

IloAbstraction

Category Global Function

Definition File ilconcert/iloany.h

Synopsis

```
public IloConstraint IloAbstraction(const IloEnv env,
    const IloAnyVarArray y,
    const IloAnyVarArray x,
    const IloAnyArray values,
    IloAny abstractValue)
```

Summary Returns a constraint that abstracts the values of one array into the abstract value of another array.

Description This function returns a constraint that abstracts the values of the elements of one array of constrained variables (called *x*) in a model into the abstract value of another array of constrained variables (called *y*). In other words, for each element *x[i]*, there is a variable *y[i]* corresponding to the abstraction of *x[i]* with respect to an array of numeric *values*. That is,

$x[i] = v$ with *v* in *values* if and only if $y[i] = v$;

$x[i] = v$ with *v* not in *values* if and only if $y[i] = \text{abstractValue}$.

This constraint maintains a many-to-one mapping that makes it possible to define constraints that impinge only on a particular set of values from the domains of constrained variables. The abstract value (indicated by *abstractValue*) must not be in the domain of *x[i]*.

Adding a Constraint to a Model, Extracting a Model for an Algorithm

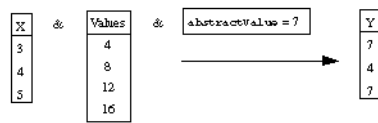
In order for the constraint returned by *IloAbstraction* to take effect, you must add it to a model with the template *IloAdd* or the member function *IloModel::add* and extract the model for an algorithm with the member function *IloAlgorithm::extract*.

Exceptions

If the arrays *x* and *y* are not the same size, this function throws the exception *IloAbstraction::InvalidArraysException*.

Example

For simplicity, let's assume that our array *x* consists of three elements with the domains {3}, {4}, and {5}. We will also assume that our abstract value is 7, and the *values* we are interested in are {4, 8, 12, 16}. Then *IloAbstraction* produces these elements in the array *y*:



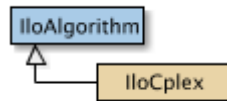
IloAdd

Category	Global Function
Definition File	ilconcert/ilomodel.h
Synopsis	<pre>public X IloAdd(IloModel & mdl, X x)</pre>
Description	This C++ template helps when you want to add elements to a model. In those synopses, <i>X</i> represents a class, <i>x</i> is an instance of the class <i>X</i>. The class <i>X</i> must be <code>IloExtractable</code>, <code>IloExtractableArray</code>, or one of their subclasses. If <code>model</code> is an instance of <code>IloModel</code>, derived from <code>IloExtractable</code>, then <i>x</i> will be added to the top level of that model. As an alternative to this way of adding extractable objects to a model, you may also use <code>IloModel::add</code>. This template preserves the original type of its argument <i>x</i> when it returns <i>x</i>. This feature of the template may be useful, for example, in cases like this: <pre>IloRange rng = IloAdd(model, 3 * x + y == 17);</pre> For a comparison of these two ways of adding extractable objects to a model, see Adding Extractable Objects in the documentation of <code>IloExtractable</code>.
See Also	<code>IloAnd</code> , <code>IloExtractable</code> , <code>IloExtractableArray</code> , <code>IloModel</code> , <code>IloOr</code>

IloAlgorithm

Category Class

InheritancePath



Definition File `ilconcert/iloalg.h`

Summary The base class of algorithms in Concert Technology.

Constructor Summary

public	IloAlgorithm (IloAlgorithmI * impl=0)

Method Summary

public void	clear() const
public void	end()
public ostream &	error() const
public void	IloAlgorithm::extract (const IloModel) const
public IloEnv	getEnv() const
public IloInt	getIntValue (const IloIntVar) const
public void	getIntValues (const IloIntVarArray , IloIntArray) const
public IloNum	getObjValue() const
public IloAlgorithm::Status	getStatus() const
public IloNum	getTime() const
public IloNum	getValue (const IloNumExprArg) const
public IloNum	getValue (const IloObjective) const
public IloNum	getValue (const IloIntVar) const
public IloNum	getValue (const IloNumVar) const
public void	getValues (const IloIntVarArray , IloNumArray) const
public void	getValues (const IloNumVarArray , IloNumArray) const

public IloBool	isExtracted(const IloExtractable) const
public ostream &	out() const
public void	printTime() const
public void	resetTime() const
public void	setError(ostream &)
public void	setOut(ostream &)
public void	setWarning(ostream &)
public IloBool	solve() const
public ostream &	warning() const

Inner Enumeration

IloAlgorithm::Status	An enumeration for the class IloAlgorithm.

Inner Class

IloAlgorithm::IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.

Description

IloAlgorithm is the base class of algorithms in Concert Technology. An instance of this class represents an algorithm in Concert Technology.

In general terms, you define a model, and Concert Technology extracts objects from it for your target algorithm and then solves for solutions.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

Status

The member function `getStatus` returns a status indicating information about the currently extracted model and the solution (if there is one). For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

Exceptions

The class `IloAlgorithm::Exception`, derived from the class `IloException`, is the base class of exceptions thrown by classes derived from `IloAlgorithm`. For an explanation of exceptions thrown by instances of `IloAlgorithm`, see `IloAlgorithm::Exception`.

Streams and Output

The class `IloAlgorithm` supports these communication streams:

- ◆ `ostream& error()` const; for error messages.
- ◆ `ostream& out()` const; for general output.
- ◆ `ostream& warning()` const; for warning messages about non-fatal conditions.

See Also

`IloEnv`, `IloModel`, `IloAlgorithm::Status`,
`IloAlgorithm::Exception`

Constructors

```
public IloAlgorithm(IloAlgorithmI * impl=0)
```

This constructor creates an algorithm in Concert Technology from its implementation object. This is the default constructor.

Methods

```
public void clear() const
```

This member function clears the current model from the algorithm.

```
public void end()
```

This member function deletes the invoking algorithm. That is, it frees memory associated with the invoking algorithm.

```
public ostream & error() const
```

This member function returns a reference to the stream currently used for error messages from the invoking algorithm. `IloAlgorithm::error` is initialized with the value of `IloEnv::error`.

```
public void extract(const IloModel) const
```

This member function extracts the extractable objects from a model into the invoking algorithm if a member function exists to extract the objects from the model for the invoking algorithm. Not all extractable objects can be extracted by all algorithms; see the documentation of the algorithm class you are using for a list of extractable classes it supports.

When you use this member function to extract extractable objects from a model, it extracts all the elements of that model for which Concert Technology creates the representation of the extractable object suitable for the invoking algorithm.

The attempt to extract may fail. In case such a failure occurs, Concert Technology throws the exception `CannotExtractException` on platforms that support C++ exceptions when exceptions are enabled.

For example, a failure will occur if you attempt to extract more than one objective for an invoking algorithm that accepts only one objective, and Concert Technology will throw the exception `MultipleObjException` (documented in the ILOG CPLEX Reference Manual).

```
public IloEnv getEnv() const
```

This member function returns the environment of the invoking algorithm.

```
public IloInt getIntValue(const IloIntVar) const
```

This member function returns the integer value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getIntValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public void getIntValues(const IloIntVarArray,
                        IloIntArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public IloNum getObjValue() const
```

This member function returns the numeric value of the objective function associated with the invoking algorithm.

```
public IloAlgorithm::Status getStatus() const
```

This member function returns a status indicating information about the current model and the solution. For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

```
public IloNum getTime() const
```

This member function returns the amount of time elapsed in seconds since the most recent reset of the invoking algorithm. (The member function `printTime` directs the output of `getTime` to the output channel of the invoking algorithm.)

The type of time returned is platform dependent. On Windows systems, the time returned is elapsed wall clock time. On UNIX systems, the time returned is CPU time.

```
public IloNum getValue(const IloNumExprArg) const
```

This member function returns the value of an expression in the current solution of the invoking algorithm. For example, to access the expression, use the member function `getValue(expr)` where `expr` is an instance of the class `IloNumExprArg`.

If there is no value to return for `expr`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `expr` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloObjective) const
```

This member function returns the value of an objective in the current solution of the invoking algorithm. For example, to access the objective, use the member function `getValue(obj)` where `obj` is an instance of the class `IloObjective`.

If there is no value to return for `obj`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `obj` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloIntVar) const
```

This member function returns the numerical value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloNumVar) const
```

This member function returns the numeric variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloNumVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public void getValues(const IloIntVarArray,
                    IloNumArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of

the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public void getValues(const IloNumVarArray,
                     IloNumArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public IloBool isExtracted(const IloExtractable) const
```

This member function returns `IloTrue` if `extr` has been extracted for the invoking algorithm; otherwise, it returns `IloFalse`.

```
public ostream & out() const
```

This member function returns a reference to the stream currently used for logging. General output from the invoking algorithm is accessible through this member function. `IloAlgorithm::out` is initialized with the value of `IloEnv::out`.

```
public void printTime() const
```

This member function directs the output of the member function `getTime` to an output channel of the invoking algorithm. (The member function `getTime` accesses the elapsed time in seconds since the most recent reset of the invoking algorithm.)

```
public void resetTime() const
```

This member function resets the timer on the invoking algorithm. The type of timer is platform dependent. On Windows systems, the time is elapsed wall clock time. On UNIX systems, the time is CPU time.

```
public void setError(ostream &)
```

This member function sets the stream for errors generated by the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cerr`.

```
public void setOut(ostream &)
```

This member function redirects the `out()` stream with the stream given as a parameter.

This member function can be used with `getNullStream` to suppress screen output by redirecting it to the null stream.

```
public void setWarning(ostream &)
```

This member function sets the stream for warnings from the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cout`.

```
public IloBool solve() const
```

This member function solves the current model in the invoking algorithm. In other words, `solve` works with all extractable objects extracted from the model for the algorithm. The member function returns `IloTrue` if it finds a solution (not necessarily an optimal one). Here is an example of its use:

```
if (algo.solve()) {
    algo.out() << "Status is " << algo.getStatus() << endl;
};
```

If an objective of the model has been extracted into the invoking algorithm, this member function solves the model to optimality. If there is currently no objective, this member function searches for the first feasible solution. A feasible solution is not necessarily optimal, though it satisfies all constraints.

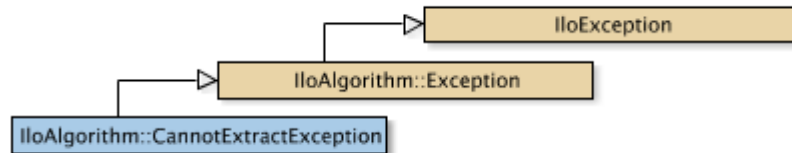
```
public ostream & warning() const
```

This member function returns a reference to the stream currently used for warnings from the invoking algorithm. `IloAlgorithm::warning` is initialized with the value of `IloEnv::warning`.

IloAlgorithm::CannotExtractException

Category Inner Class

InheritancePath



Definition File ilconcert/iloalg.h

Summary The class of exceptions thrown if an object cannot be extracted from a model.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited methods from IloException
IloException::end , IloException::getMessage

Description If an attempt to extract an object from a model fails, this exception is thrown.

Methods

public void **end**()

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

public const IloAlgorithmI * **getAlgorithm**() const

The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

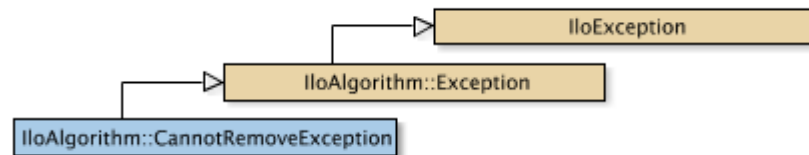
public [IloExtractableArray](#) & **getExtractables**()

The member function **getExtractables** returns the extractable objects that triggered the exception.

IloAlgorithm::CannotRemoveException

Category Inner Class

InheritancePath



Definition File ilconcert/iloalg.h

Summary The class of exceptions thrown if an object cannot be removed from a model.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited methods from IloException
IloException::end , IloException::getMessage

Description If an attempt to remove an extractable object from a model fails, this exception is thrown.

Methods

public void **end()**

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

public const IloAlgorithmI * **getAlgorithm()** const

The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

public [IloExtractableArray](#) & **getExtractables()**

The member function **getExtractables** returns the extractable objects that triggered the exception.

IloAlgorithm::Exception

Category Inner Class

InheritancePath

Definition File ilconcert/iloalg.h

Summary The base class of exceptions thrown by classes derived from IloAlgorithm.

Constructor Summary

public	Exception (const char * str)

Inherited methods from [IloException](#)

IloException::end , IloException::getMessage

Description

IloAlgorithm is the base class of algorithms in Concert Technology.

The class `IloAlgorithm::Exception`, derived from the class `IloException`, is the base class of exceptions thrown by classes derived from IloAlgorithm.

On platforms that support C++ exceptions, when exceptions are enabled, the member function [IloAlgorithm::extract](#) will throw an exception if you attempt to extract an unsuitable object from your model for an algorithm. An extractable object is unsuitable for an algorithm if there is no member function to extract the object from your model to that algorithm.

For example, an attempt to extract more than one objective into an algorithm that accepts only one objective will throw an exception.

Similarly, the member function [getValue](#) will throw an exception if you attempt to access the value of a variable that has not yet been bound to a value.

See Also [IloAlgorithm](#), [IloException](#)

Constructors

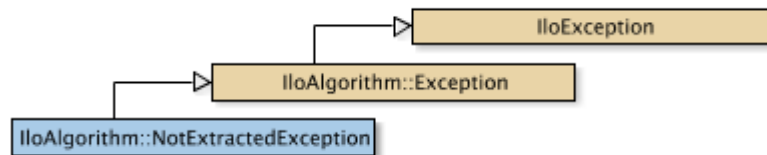
public **Exception**(const char * str)

This constructor creates an exception thrown from a member of IloAlgorithm. The exception contains the message string str, which can be queried with the member function [IloException::getMessage](#).

IloAlgorithm::NotExtractedException

Category Inner Class

InheritancePath



Definition File ilconcert/iloalg.h

Summary The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.

Constructor Summary

public	NotExtractedException (const IloAlgorithmI *,const IloExtractable)

Method Summary

public const IloAlgorithmI *	getAlgorithm() const
public const IloExtractable &	getExtractable()

Inherited methods from [IloException](#)

[IloException::end](#), [IloException::getMessage](#)

Description If an expression, numeric variable, objective, or array of extractable objects has no value in the current solution of an algorithm, this exception is thrown.

Constructors public **NotExtractedException**(const IloAlgorithmI *,
const [IloExtractable](#))

The constructor `NotExtractedException` creates an exception thrown from the algorithm object `alg` for the extractable object `extr`.

Methods

```
public const IloAlgorithmI * getAlgorithm() const
```

The member function `getAlgorithm` returns the algorithm from which the exception was thrown.

```
public const IloExtractable & getExtractable()
```

The member function `getExtractable` returns the extractable object that triggered the exception.

IloAlgorithm::Status

Category Inner Enumeration

Definition File ilconcert/iloalg.h

Synopsis

```

Status{
    Unknown,
    Feasible,
    Optimal,
    Infeasible,
    Unbounded,
    InfeasibleOrUnbounded,
    Error
};

```

Summary An enumeration for the class [IloAlgorithm](#).

Description [IloAlgorithm](#) is the base class of algorithms in Concert Technology, and [IloAlgorithm::Status](#) is an enumeration limited in scope to the class [IloAlgorithm](#). The member function [getStatus](#) returns a status indicating information about the current model and the solution.

Unknown indicates that the algorithm has no information about the solution of the model.

Feasible indicates that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions [getValue](#) access this feasible solution.

Optimal indicates that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions [getValue](#) access this optimal solution.

Infeasible indicates that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

Unbounded indicates that the algorithm proved the model unbounded.

InfeasibleOrUnbounded indicates that the model is infeasible or unbounded.

Error indicates that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also [IloAlgorithm](#), [operator](#)

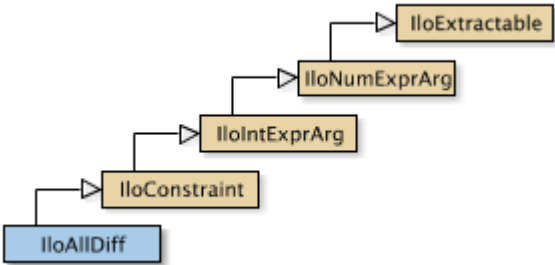
Fields

Unknown
Feasible
Optimal
Infeasible
Unbounded
InfeasibleOrUnbounded
Error

IloAllDiff

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Summary A constraint that forces constrained integer variables to assume different values in a model.

Constructor Summary	
public	IloAllDiff()
public	IloAllDiff(IloAllDiffI * impl)
public	IloAllDiff(const IloEnv env, const IloIntVarArray vars, const char * name=0)

Method Summary	
public IloAllDiffI *	getImpl() const

Inherited methods from IloConstraint	
IloConstraint::getImpl	

Inherited methods from [IloIntExprArg](#)

[IloIntExprArg::getImpl](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of this class is a constraint that forces constrained *integer* variables to assume different values from one another in a model. In other words, no two of those integer variables will have the same integer value when this constraint is satisfied.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

What Is Extracted

All the variables (that is, instances of [IloNumVar](#) or one of its subclasses) that have appeared as a parameter of a constructor of `IloAlldiff` and all variables that have been explicitly added to the instance of `IloAlldiff` will be extracted by an algorithm (such as an instance of `IloSolver`, documented in the ILOG Solver Reference Manual) that extracts that constraint.

See Also

[IloAdd](#), [IloConstraint](#), [IloDiff](#)

Constructors

`public IloAlldiff()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloAllDiff(IloAllDiffI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloAllDiff(const IloEnv env,  
                  const IloIntVarArray vars,  
                  const char * name=0)
```

This constructor creates a constraint that forces all the integer variables in **vars** to assume different values from each other. If **vars** is empty, this constructor creates an empty instance of **IloAllDiff**, and then you must fill the constraint; that is, you must put variables into the array. You must add this constraint to a model and extract the model for an algorithm in order for it to be taken into account.

Methods

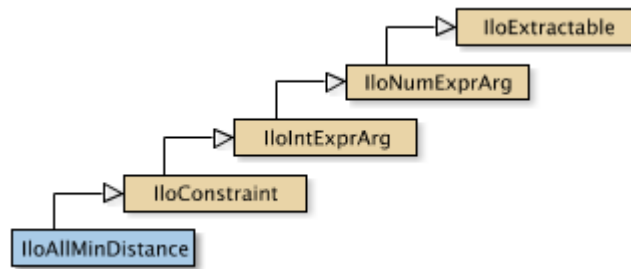
```
public IloAllDiffI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloAllMinDistance

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Summary A constraint on the minimum absolute distance between a pair of variables in an array.

Constructor Summary

public	IloAllMinDistance()
public	IloAllMinDistance(IloAllMinDistanceI * impl)
public	IloAllMinDistance(const IloEnv env, const IloIntVarArray vars, IloInt k, const char * name=0)

Method Summary

public IloAllMinDistanceI *	getImpl() const

Inherited methods from [IloConstraint](#)

IloConstraint::getImpl

Inherited methods from [IloIntExprArg](#)[IloIntExprArg::getImpl](#)**Inherited methods from [IloNumExprArg](#)**[IloNumExprArg::getImpl](#)**Inherited methods from [IloExtractable](#)**[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)**Description**

An instance of the class `IloAllMinDistance` is a constraint that makes sure that the absolute distance between any pair of variables in an array of constrained numeric variables will be greater than or equal to a given integer.

See Also[IloAllDiff](#)**Constructors**

```
public IloAllMinDistance()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloAllMinDistance(IloAllMinDistanceI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloAllMinDistance(const IloEnv env,  
                        const IloIntVarArray vars,  
                        IloInt k,  
                        const char * name=0)
```

This constructor returns a constraint that insures that the absolute distance between any pair of variables in the array `vars` will be greater than or equal to `k`. You must add this constraint to a model and extract the model for an algorithm in order for it to be taken into account.

Methods

```
public IloAllMinDistanceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

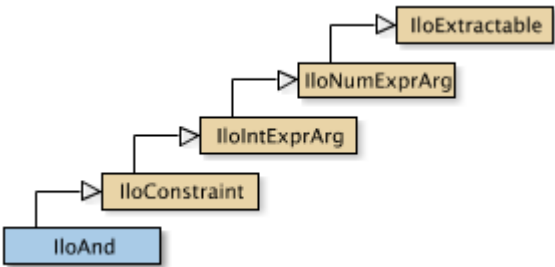
IloAllNullIntersect

Category	Global Function
Definition File	ilconcert/iloanyset.h
Synopsis	<pre>public IloConstraint IloAllNullIntersect(const IloEnv env, const IloAnySetVarArray vars) public IloConstraint IloAllNullIntersect(const IloEnv env, const IloIntSetVarArray vars)</pre>
Description	This function creates and returns a constraint (an instance of IloConstraint) for use in a model. The constraint insures that for the sets in the array <code>vars</code>, the intersection of <code>vars[i]</code> with <code>vars[j]</code> will be empty for all <code>i</code> and <code>j</code> when this constraint is satisfied. In order for the constraint to take effect, you must add it to a model with the template IloAdd or the member function IloModel::add and extract the model for an algorithm with the member function IloAlgorithm::extract.

IloAnd

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Summary A conjunctive constraint that defines a logical conjunctive-AND among other constraints.

Constructor Summary	
public	IloAnd()
public	IloAnd(IloAndI * impl)
public	IloAnd(const IloEnv env,const char * name=0)

Method Summary	
public void	add(const IloConstraintArray array) const
public void	add(const IloConstraint constraint) const
public IloAndI *	getImpl() const
public void	remove(const IloConstraintArray array) const
public void	remove(const IloConstraint constraint) const

Inherited methods from [IloConstraint](#)

<code>IloConstraint::getImpl</code>

Inherited methods from <code>IloIntExprArg</code>

<code>IloIntExprArg::getImpl</code>

Inherited methods from <code>IloNumExprArg</code>

<code>IloNumExprArg::getImpl</code>

Inherited methods from <code>IloExtractable</code>

<code>IloExtractable::end</code> , <code>IloExtractable::getEnv</code> , <code>IloExtractable::getId</code> , <code>IloExtractable::getImpl</code> , <code>IloExtractable::getName</code> , <code>IloExtractable::getObject</code> , <code>IloExtractable::setName</code> , <code>IloExtractable::setObject</code>

Description

An instance of `IloAnd` represents a conjunctive constraint. In other words, it defines a logical conjunctive-AND among any number of constraints. It lets you represent a constraint on constraints in your model. Since an instance of `IloAnd` is a constraint itself, you can build up extensive conjunctions by adding constraints to an instance of `IloAnd` by means of the member function `add`. You can also remove constraints from an instance of `IloAnd` by means of the member function `remove`.

The elements of a conjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

Conjunction of Goals

If you want to represent the conjunction of goals (rather than constraints) in your model, then you should consider the function `IloAndGoal` (documented in the ILOG Solver Reference Manual).

What Is Extracted

All the constraints (that is, instances of [IloConstraint](#) or one of its subclasses) that have been added to a conjunctive constraint (an instance of `IloAnd`) and that have not been removed from it will be extracted when an algorithm such as `IloSolver` (documented in the ILOG Solver Reference Manual) extracts the constraint.

Example

For example, you may write:

```
IloAnd and(env);
and.add(constraint1);
and.add(constraint2);
and.add(constraint3);
```

Those lines are equivalent to :

```
IloAnd and = constraint1 && constraint2 && constraint3;
```

See Also

[IloConstraint](#), [IloOr](#), [operator &&](#)

Constructors

```
public IloAnd()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloAnd(IloAndI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloAnd(const IloEnv env,
             const char * name=0)
```

This constructor creates a conjunctive constraint for use in the environment `env`. In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

The optional argument `name` is set to `0` by default.

Methods

```
public void add(const IloConstraintArray array) const
```

This member function makes all the elements in `array` elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to all the elements of `array`.

Note: *The member function `add` notifies Concert Technology algorithms about this change to the invoking object.*

```
public void add(const IloConstraint constraint) const
```

This member function makes `constraint` one of the elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to `constraint`.

Note: *The member function `add` notifies Concert Technology algorithms about this change to the invoking object.*

```
public IloAndI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloConstraintArray array) const
```

This member function removes all the elements of `array` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to any of those elements.

Note: *The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.*

```
public void remove(const IloConstraint constraint) const
```

This member function removes `constraint` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to `constraint`.

Note: *The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.*

IloAny

Category	Type Definition
Definition File	ilconcert/ilosys.h
Synopsis	<code>void * IloAny</code>
Description	This type definition represents objects in a model handled by Concert Technology enumerated variables and by Concert Technology set variables. A pointer to any object, whether a predefined Concert Technology type or an instance of a C++ class, is implicitly converted to <code>IloAny</code>. By using this type, you can be sure that these components of your application will port without any source change across different hardware platforms.
See Also	IloModel

IloAnyArray

Category Class

InheritancePath



Definition File ilconcert/iloany.h

Summary The array class of the enumerated type definition IloAny.

Constructor Summary	
public	IloAnyArray (IloDefaultArrayI * i=0)
public	IloAnyArray (const IloAnyArray & copy)
public	IloAnyArray (const IloEnv env, IloInt n=0)
public	IloAnyArray (const IloEnv env, IloInt n, const IloAny p0, const IloAny p1, ...)

Method Summary	
public void	add (const IloAny p)
public IloBool	contains (IloAny e) const

Description

For each basic type, Concert Technology defines a corresponding array class. [IloAnyArray](#) is the array class of the basic enumerated type definition ([IloAny](#)) for a model.

Instances of [IloAnyArray](#) are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

If you would like to represent a set of enumerated values (that is, no repeated elements, no order among elements), consider an instance of [IloAnySet](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also [IloAny](#), [IloAnySet](#), [operator>>](#), [operator](#)

Constructors

```
public IloAnyArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloAnyArray(const IloAnyArray & copy)
```

This copy constructor creates a handle to the array of pointers indicated by `copy`.

```
public IloAnyArray(const IloEnv env,  
                  IloInt n=0)
```

This constructor creates an array of `n` elements, all of which are empty handles.

```
public IloAnyArray(const IloEnv env,  
                  IloInt n,  
                  const IloAny p0,  
                  const IloAny p1,  
                  ...)
```

This constructor creates an array of `n` elements for use in a model.

Methods

```
public void add(const IloAny p)
```

This member function appends `p` to the invoking array.

```
public IloBool contains(IloAny e) const
```

This member function checks whether the value is contained or not.

IloAnyBinaryPredicate

Category Class

InheritancePath

IloAnyBinaryPredicate

Definition File ilconcert/ilotupleset.h

Summary A class to define binary predicates on objects in a model.

Constructor Summary

public	IloAnyBinaryPredicate()
public	IloAnyBinaryPredicate(IloAnyBinaryPredicateI * impl)

Method Summary

public IloAnyBinaryPredicateI *	getImpl() const
public IloBool	isTrue (const IloAny val1, const IloAny val2)
public void	operator= (const IloAnyBinaryPredicate & h)

Description This class makes it possible for you to define binary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as [isTrue](#)) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, [IloAnyBinaryPredicate](#), and an implementation class, such as [IloAnyBinaryPredicateI](#), where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of [IloAnyBinaryPredicateI](#) allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloAnyBinaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `isTrue` must check) must be two.

See Also [IloTableConstraint](#)

Constructors

```
public IloAnyBinaryPredicate()
```

This constructor creates an empty binary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Any attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

```
public IloAnyBinaryPredicate(IloAnyBinaryPredicateI * impl)
```

This constructor creates a handle object (an instance of the class `IloAnyBinaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IlcAnyPredicateI`, documented in the *ILOG Solver Reference Manual*).

Methods

```
public IloAnyBinaryPredicateI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool isTrue(const IloAny val1,
                    const IloAny val2)
```

This member function returns `IloTrue` if the values `val1` and `val2` make the invoking binary predicate valid. It returns `IloFalse` otherwise.

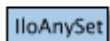
```
public void operator=(const IloAnyBinaryPredicate & h)
```

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

IloAnySet

Category Class

InheritancePath



Definition File ilconcert/iloanyset.h

Summary A class whose instances represent a set of enumeration values.

Constructor Summary

public	IloAnySet (const IloEnv env, const IloAnyArray array, IloBool withIndex= IloFalse)
public	IloAnySet (const IloEnv env, IloBool withIndex= IloFalse)

Method Summary

public void	add (IloAnySet set)
public void	add (IloAny elt)
public IloBool	contains (IloAnySet set) const
public IloBool	contains (IloAny elt) const
public IloAnySet	copy () const
public void	empty ()
public IloEnv	getEnv () const
public IloAnySetI *	getImpl () const
public IloAny	getNext (IloAny value, IloInt n=1) const
public IloAny	getNextC (IloAny value, IloInt n=1) const
public IloAny	getPrevious (IloAny value, IloInt n=1) const
public IloAny	getPreviousC (IloAny value, IloInt n=1) const
public IloInt	getSize () const
public IloBool	intersects (IloAnySet set) const
public void	remove (IloAnySet set)
public void	remove (IloAny elt)
public void	setIntersection (IloAnySet set)

public void	setIntersection(IloAny elt)

Inner Class	
IloAnySet::IloAnySet::Iterator or	An iterator to traverse the elements of IloAnySet.

Description

An instance of this class represents a set of enumerated values. The same enumerated value will not appear more than once in a set. The elements of a set are not ordered. The class [IloAnySet::Iterator](#) offers you a way to traverse the elements of such a set.

If you are considering modeling issues where you want to represent repeated elements or where you want to exploit an indexed order among the elements, then you might want to look at the class [IloAnyArray](#) instead of this class for sets.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloAny](#), [IloAnyArray](#), [IloAnySet::Iterator](#)

Constructors

```
public IloAnySet(const IloEnv env,
                const IloAnyArray array,
                IloBool withIndex=IloFalse)
```

This constructor creates a set of enumerated values for `env` from the elements in `array`.

```
public IloAnySet(const IloEnv env,
                IloBool withIndex=IloFalse)
```

This constructor creates an empty set (no elements) for `env`. You must use the member function `IloAnySet::add` to fill this set with elements.

Methods

```
public void add(IloAnySet set)
```

This member function adds `set` to the invoking set. By `add`s, we mean that the invoking set becomes the union of its former elements and the elements of `set`.

```
public void add(IloAny elt)
```

This member function adds `elt` to the invoking set. By `add`s, we mean that the invoking set becomes the union of its former elements and the new `elt`.

```
public IloBool contains(IloAnySet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` intersects the invoking set. The value one indicates that the invoking set contains all the elements of `set`, and that the intersection of the invoking set with `set` is precisely `set`. The value zero indicates that the intersection of the invoking set `set` is not precisely `set`.

```
public IloBool contains(IloAny elt) const
```

This member function returns a Boolean value (zero or one) that indicates whether `elt` is an element of the invoking set. The value one indicates that the invoking set contains `elt`; the value zero indicates that the invoking set does not contain `elt`.

```
public IloAnySet copy() const
```

This member functions creates a clone of the array.

```
public void empty()
```

This member function removes the elements from the invoking set. In other words, the invoking set becomes the empty set.

```
public IloEnv getEnv() const
```

This member function returns the environment of the invoking set.

```
public IloAnySetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public IloAny getNext(IloAny value,  
                     IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it throws an exception

See also `getNextC`, `getPreviousC` for circular search.

```
public IloAny getNextC(IloAny value,  
                     IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it will give you the first value (circular search)

See also `getNext`, `getPrevious`.

```
public IloAny getPrevious(IloAny value,  
                         IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it throws an exception

See also getNextC, getPreviousC for circular search.

```
public IloAny getPreviousC(IloAny value,
                           IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it will give you the last value (circular search)

See also getNext, getPrevious.

```
public IloInt getSize() const
```

This member function returns an integer indicating the size of the invoking set (that is, how many elements it contains).

```
public IloBool intersects(IloAnySet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` intersects the invoking set. The value one indicates that the intersection of `set` and the invoking set is not empty (at least one element in common); the value zero indicates that the intersection of `set` and the invoking set is empty (no elements in common).

```
public void remove(IloAnySet set)
```

This member function removes all the elements of `set` from the invoking set.

```
public void remove(IloAny elt)
```

This member function removes `elt` from the invoking set.

```
public void setIntersection(IloAnySet set)
```

This member function changes the invoking set so that it includes only the elements of `set`. In other words, the invoking set becomes the intersection of its former elements with the elements of `set`.

```
public void setIntersection(IloAny elt)
```

This member function changes the invoking set so that it includes only the element indicated by `elt`. In other words, the invoking set becomes the intersection of its former elements with `elt`.

IloAnySet::Iterator

Category Inner Class

InheritancePath

Definition File ilconcert/iloanyset.h

Summary An iterator to traverse the elements of I loAnySet.

Constructor Summary	
public	Iterator (const IloAnySetI * coll)
public	Iterator (const IloAnySet coll)
public	Iterator (IloGenAlloc * heap, const IloAnySetI * coll)

Description An instance of the nested class `IloAnySet::Iterator` is an iterator that traverses the elements of a finite set of pointers (an instance of [IloAnySet](#)).

See Also [IloAnySet](#)

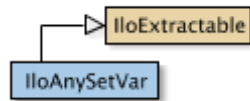
Constructors

```
public Iterator(const IloAnySetI * coll)
.
public Iterator(const IloAnySet coll)
.
public Iterator(IloGenAlloc * heap,
               const IloAnySetI * coll)
.
```

IloAnySetVar

Category Class

InheritancePath



Definition File ilconcert/iloanyset.h

Summary Class to represent a set of enumerated values as a constrained variable.

Constructor Summary	
public	IloAnySetVar()
public	IloAnySetVar(IloIntSetVarI * impl)
public	IloAnySetVar(const IloEnv env, const IloAnyArray possible, const char * name=0)
public	IloAnySetVar(const IloEnv env, const IloAnyArray possible, const IloAnyArray required, const char * name=0)
public	IloAnySetVar(const IloAnyCollection possible, const char * name=0)
public	IloAnySetVar(const IloAnyCollection possible, const IloAnyCollection required, const char * name=0)

Method Summary	
public void	addPossible(IloAny elt) const
public void	addRequired(IloAny elt) const
public IloIntSetVarI *	getImpl() const
public void	getPossibleSet(IloAnySet set) const
public IloAnySet	getPossibleSet() const
public IloAnySet::Iterator	getPossibleSetIterator() const
public void	getRequiredSet(IloAnySet set) const
public IloAnySet	getRequiredSet() const

public IloAnySet::Iterator	getRequiredSetIterator() const
public void	removePossible(IloAny elt) const
public void	removeRequired(IloAny elt) const

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description

An instance of this class offers a convenient way to represent a set of enumerated values as a constrained variable in Concert Technology.

A constrained variable representing a set of enumerated values (that is, an instance of [IloAnySetVar](#)) is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set. This class offers member functions for accessing the required and possible elements of a set of enumerated values.

The function [IloCard](#) offers you a way to constrain the number of elements in a set variable. That is, [IloCard](#) constrains the cardinality of a set variable.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloAnySet](#), [IloAnySetVarArray](#), [IloCard](#), [IloEqIntersection](#),
[IloEqUnion](#), [IloExtractable](#), [IloMember](#), [IloModel](#), [IloNotMember](#),
[IloNullIntersect](#), [IloSubset](#), [IloSubsetEq](#)

Constructors

```
public IloAnySetVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloAnySetVar(IloIntSetVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloAnySetVar(const IloEnv env,  
                  const IloAnyArray possible,  
                  const char * name=0)
```

This constructor creates a constrained set variable from the values in `possible` and makes the set variable part of the environment indicated by `env`, where the set consists

of enumerated values. By default, its `name` is indicated by the empty string, but you can indicate a `name` of your own choice.

```
public IloAnySetVar(const IloEnv env,
                  const IloAnyArray possible,
                  const IloAnyArray required,
                  const char * name=0)
```

With this constructor, you can indicate both the `required` and the `possible` sets that characterize the instance of `IloAnySetVar` that it creates. By default, its `name` is indicated by the empty string, but you can indicate a `name` of your own choice.

```
public IloAnySetVar(const IloAnyCollection possible,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

```
public IloAnySetVar(const IloAnyCollection possible,
                  const IloAnyCollection required,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

Methods

```
public void addPossible(IloAny elt) const
```

This member function adds `elt` to the set of possible elements of the invoking set variable.

Note: *The member function `addPossible` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void addRequired(IloAny elt) const
```

This member function adds `elt` to the set of required elements of the invoking set variable.

Note: *The member function `addRequired` notifies Concert Technology algorithms about this change of this invoking object.*

```
public IloIntSetVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.


```
public void getPossibleSet(IloAnySet set) const
```

This member function accesses the possible elements of the invoking set variable and puts those elements into its parameter `set`.

```
public IloAnySet getPossibleSet() const
```

This member function returns the possible elements of the invoking set variable.

```
public IloAnySet::Iterator getPossibleSetIterator() const
```

This member function returns an `IloAnySet::Iterator` to traverse the possible elements of the invoking set variable.

```
public void getRequiredSet(IloAnySet set) const
```

This member function accesses the required elements of the invoking set variable and puts those elements into its parameter `set`.

```
public IloAnySet getRequiredSet() const
```

This member function returns the required elements of the invoking set variable.

```
public IloAnySet::Iterator getRequiredSetIterator() const
```

This member function returns an `IloAnySet::Iterator` to traverse the required elements of the invoking set variable.

```
public void removePossible(IloAny elt) const
```

This member function removes `elt` as a possible element of the invoking set variable.

Note: The member function `removePossible` notifies Concert Technology algorithms about this change of this invoking object.

```
public void removeRequired(IloAny elt) const
```

This member function removes `elt` as a required element of the invoking set variable.

Note: The member function `removeRequired` notifies Concert Technology algorithms about this change of this invoking object.

IloAnySetVarArray

Category Class

InheritancePath



Definition File ilconcert/iloanyset.h

Summary The array class of the set variable class IloAnySetVar.

Constructor Summary

public	IloAnySetVarArray (IloDefaultArrayI * i=0)
public	IloAnySetVarArray (const IloEnv env, IloInt n=0)

Method Summary

public void	add (IloInt more, const IloAnySetVar x)
public void	add (const IloAnySetVar x)
public void	add (const IloAnySetVarArray array)
public IloAnySetVar	operator[] (IloInt i) const
public IloAnySetVar &	operator[] (IloInt i)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description For each basic type, Concert Technology defines a corresponding array class. [IloAnySetVarArray](#) is the array class of the set variable class for enumerated values ([IloAnySetVar](#)) in a model.

Instances of `IloAnySetVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

For more information on arrays, see the concept `Arrays`

See Also [IloAnySetVar](#), [IloModel](#), [operator](#)

Constructors

```
public IloAnySetVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of set variables, where each set is a set of enumerated values. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloAnySetVarArray(const IloEnv env,  
                        IloInt n=0)
```

This constructor creates an extensible array of `n` set variables, where each set is a set of enumerated values. Initially, the `n` elements are empty handles.

Methods

```
public void add(IloInt more,  
               const IloAnySetVar x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloAnySetVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloAnySetVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloAnySetVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `const` operator

```
IloAnySetVar operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloAnySetVar & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

IloAnyTernaryPredicate

Category Class

InheritancePath



Definition File ilconcert/ilotupleset.h

Summary A class to define ternary predicates on objects in a model.

Constructor Summary	
public	IloAnyTernaryPredicate()
public	IloAnyTernaryPredicate(IloAnyTernaryPredicate I * impl)

Method Summary	
public IloAnyTernaryPredicateI *	getImpl() const
public IloBool	isTrue (const IloAny val1,const IloAny val2,const IloAny val3)
public void	operator= (const IloAnyTernaryPredicate & h)

Description This class makes it possible for you to define ternary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as [isTrue](#)) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. A ternary predicate checks an ordered set of three objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, [IloAnyTernaryPredicate](#), and an implementation class, such as [IloAnyTernaryPredicateI](#), where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of [IloAnyTernaryPredicateI](#) allocated in a Concert Technology

environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloAnyTernaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `isTrue` must check) must be three.

See Also [IloTableConstraint](#)

Constructors

```
public IloAnyTernaryPredicate()
```

This constructor creates an empty ternary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Any attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

```
public IloAnyTernaryPredicate(IloAnyTernaryPredicateI * impl)
```

This constructor creates a handle object (an instance of the class `IloAnyTernaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IloAnyTernaryPredicateI`).

Methods

```
public IloAnyTernaryPredicateI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool isTrue(const IloAny val1,
                    const IloAny val2,
                    const IloAny val3)
```

This member function returns `IloTrue` if the values `val1`, `val2`, and `val3` make the invoking ternary predicate valid. It returns `IloFalse` otherwise.

```
public void operator=(const IloAnyTernaryPredicate & h)
```

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

IloAnyTupleSet

Category Class

InheritancePath

IloAnyTupleSet

Definition File ilconcert/ilotupleset.h

Constructor Summary

public	IloAnyTupleSet (const IloEnv env, const IloInt arity)

Method Summary

public IloBool	add (const IloAnyArray tuple) const
public IloAnyTupleSetI *	getImpl () const
public IloBool	isIn (const IloAnyArray tuple) const
public IloBool	remove (const IloAnyArray tuple) const

Description

A tuple is an ordered set of values represented by an array. A *set* of enumerated tuples in a model is represented by an instance of [IloAnyTupleSet](#). That is, the elements of a tuple *set* are tuples of enumerated values (such as pointers). The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, [IloAnyTupleSet](#) manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function [isIn](#) to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- ◆ When you create the set, you must specify the arity of the tuple-elements it contains.

- ◆ You use the member function `IloAnyTupleSet::add` to add tuples to the set. You can add tuples to the set in a model; you cannot add tuples to an instance of this class during a search, nor inside a constraint, nor inside a goal.

Concert Technology will throw an exception (an instance of `IloSolver::SolverErrorException`) if you attempt:

- ◆ to add a tuple with a different number of variables from the arity of the set;
- ◆ to search for a tuple with an arity different from the set arity.

You do not have to worry about memory allocation. If you respect these conventions, Concert Technology manages allocation and de-allocation transparently for you.

See `IlcIntTupleSet` in the *ILOG Solver Reference Manual*.

See Also

[IloAnyTupleSetIterator](#), [IloTableConstraint](#), [IloExtractable](#)

Constructors

```
public IloAnyTupleSet(const IloEnv env,
                    const IloInt arity)
```

This constructor creates a set of tuples (an instance of the class `IloAnyTupleSet`) with the arity indicated by `arity`.

Methods

```
public IloBool add(const IloAnyArray tuple) const
```

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set. You may use this member function to add tuples to the invoking set in a model; you may not add tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *ILOG Solver Reference Manual*.

```
public IloAnyTupleSetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public IloBool isIn(const IloAnyArray tuple) const
```

This member function returns `IloTrue` if `tuple` belongs to the invoking set. Otherwise, it returns `IloFalse`. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set.

```
public IloBool remove(const IloAnyArray tuple) const
```

This member function removes `tuple` from the invoking set in a model. You may use this member function to remove tuples from the invoking set in a model; you may not

remove tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *ILOG Solver Reference Manual*.

IloAnyTupleSetIterator

Category Class

InheritancePath



Definition File ilconcert/ilotupleset.h

Constructor Summary	
public	IloAnyTupleSetIterator (const IloEnv env, IloAnyTupleSet tset)

Method Summary	
public IloAnyArray	operator * () const

Description An instance of the class `IloAnyTupleSetIterator` is an iterator that traverses the elements of a finite set of tuples of enumerated values (instance of [IloAnyTupleSet](#)).

See `IlcAnyTupleSet` in the *ILOG Solver Reference Manual*.

Constructors public `IloAnyTupleSetIterator`(const [IloEnv](#) env, [IloAnyTupleSet](#) tset)

This constructor creates an iterator associated with `tSet` to traverse its elements.

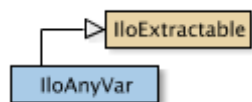
Methods public [IloAnyArray](#) `operator *`() const

This operator returns the current element, the one to which the invoking iterator points.

IloAnyVar

Category Class

InheritancePath



Definition File ilconcert/iloany.h

Summary A class to represent an enumerated variable.

Constructor Summary

public	IloAnyVar()
public	IloAnyVar(IloNumVarI * impl)
public	IloAnyVar(const IloEnv env, const IloAnyArray array, const char * name=0)
public	IloAnyVar(const IloAnyCollection array, const char * name=0)

Method Summary

public IloNumVarI *	getImpl() const
public void	getPossibleValues(IloAnyArray values) const
public void	setPossibleValues(const IloAnyArray values)

Inherited methods from [IloExtractable](#)

IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject	

Description

An instance of this class offers a convenient way to represent an enumerated variable in Concert Technology.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also

[IloAny](#), [IloAnyVarArray](#), [IloExtractable](#), [IloModel](#)

Constructors

```
public IloAnyVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloAnyVar(IloNumVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloAnyVar(const IloEnv env,
                const IloAnyArray array,
                const char * name=0)
```

This constructor creates a constrained enumerated variable from the values in `array` and makes the variable part of the environment indicated by `env`. By default, its name is indicated by the empty string, but you can indicate a `name` of your own choice.

```
public IloAnyVar(const IloAnyCollection array,
                const char * name=0)
```

This constructor creates a constrained enumerated variable from the given collection.

Methods

```
public IloNumVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getPossibleValues(IloAnyArray values) const
```

This member function accesses the possible values of the invoking enumerated variable and puts those values into its parameter `values`.

```
public void setPossibleValues(const IloAnyArray values)
```

This member function sets `values` as the domain of the invoking enumerated variable.

IloAnyVarArray

Category Class

InheritancePath



Definition File ilconcert/iloany.h

Summary A class to represent an array of enumerated variables.

Constructor Summary

public	IloAnyVarArray (IloDefaultArrayI * i=0)
public	IloAnyVarArray (const IloEnv env, IloInt n=0)

Method Summary

public void	add (IloInt more, const IloAnyVar x)
public void	add (const IloAnyVar x)
public void	add (const IloAnyVarArray array)
public IloAnyVar	operator[] (IloInt i) const
public IloAnyVar &	operator[] (IloInt i)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description For each basic type, Concert Technology defines a corresponding array class. [IloAnyVarArray](#) is the array class of the enumerated variable class ([IloAnyVar](#)) for a model. The parent class for [IloAnyVarArray](#) is the class [IloExtractableArray](#).

Instances of `IloAnyVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also [IloAnyVar](#), [IloModel](#), [operator](#)

Constructors

```
public IloAnyVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of enumerated variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloAnyVarArray(const IloEnv env,  
                    IloInt n=0)
```

This constructor creates an extensible array of `n` enumerated variables. Initially, the `n` elements are empty handles.

Methods

```
public void add(IloInt more,  
              const IloAnyVar x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloAnyVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloAnyVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloAnyVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `const` operator

```
IloAnyVar operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloAnyVar & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

IloArcCos

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloArcCos(const IloNumExprArg arg)
public IloNum IloCos(IloNum val)
public IloNum IloSin(IloNum val)
public IloNum IloTan(IloNum val)
public IloNum IloArcCos(IloNum val)
public IloNum IloArcSin(IloNum val)
public IloNum IloArcTan(IloNum val)
public IloNumExprArg IloSin(const IloNumExprArg arg)
public IloNumExprArg IloCos(const IloNumExprArg arg)
public IloNumExprArg IloTan(const IloNumExprArg arg)
public IloNumExprArg IloArcSin(const IloNumExprArg arg)
public IloNumExprArg IloArcTan(const IloNumExprArg arg)
```

Description Concert Technology offers predefined functions that return an expression from a trigonometric function on an expression. These predefined functions also return a numeric value from a trigonometric function on a numeric value as well.

Programming Hint

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

IloArray

Category Class

InheritancePath

IloArray< X>

Definition File ilconcert/iloenv.h

Summary A template to create classes of arrays for elements of a given class.

Constructor Summary

public	IloArray (IloEnv env, IloInt max=0)

Method Summary

public void	add (IloArray < X > ax) const
public void	add (IloInt more, X x) const
public void	add (X x) const
public void	clear ()
public void	end ()
public IloEnv	getEnv () const
public IloInt	getSize () const
public X &	operator[] (IloInt i)
public const X &	operator[] (IloInt i) const
public void	remove (IloInt first, IloInt nb=1)

Description

This C++ template creates a class of arrays for elements of a given class. In other words, you can use this template to create arrays of Concert Technology objects; you can also use this template to create arrays of arrays (that is, multidimensional arrays).

In its synopsis, X represents a class, x is an instance of the class X. This template creates the array class ([IloArrayX](#)) for any class in Concert Technology, including classes with names in the form [IloXArray](#), such as [IloExtractableArray](#). Concert Technology predefines the array classes listed here. The member functions defined by this template are documented in each of those predefined classes.

- ◆ IloAnyArray
- ◆ IloAnySetVarArray
- ◆ IloAnyVarArray
- ◆ IloBoolArray
- ◆ IloBoolVarArray
- ◆ IloConstraintArray
- ◆ IloExprArray
- ◆ IloExtractableArray
- ◆ IloFloatArray
- ◆ IloFloatVarArray
- ◆ IloIntArray
- ◆ IloIntVarArray
- ◆ IloNumColumnArray
- ◆ IloNumVarArray
- ◆ IloRangeArray
- ◆ IloSemiContVarArray
- ◆ IloSolutionArray
- ◆ IloSOS1Array
- ◆ IloSOS2Array

The classes you create in this way consist of extensible arrays. That is, you can add elements to the array as needed.

Deleting Arrays

The member function `end` created by this template deletes only the array; the member function `delete` does not delete the elements of the array.

Copying Arrays

Like certain other Concert Technology classes, a class of arrays created by `IloArray` is a handle class corresponding to an implementation class. In other words, an instance of an `IloArray` class is a handle pointing to a corresponding implementation object. More than one handle may point to the same implementation object. As long as you do not modify any of the handles, each of them will still point to the same implementation object.

However, if you modify an array, then Concert Technology considers whether the array is the sole handle pointing to its implementation object or whether the array is one of many handles pointing to the same implementation object.

If the array is the sole handle pointing to its implementation object, Concert Technology modifies the array.

If the array is one handle among many pointing to the same implementation object, then Concert Technology first creates a copy of the implementation object corresponding to the array (the handle you are modifying) and then modifies the copy. Consequently, the other handles continue to point to the original, unmodified implementation object.

In other words, when Concert Technology copies an array, it uses a lazy copy mechanism.

Input and Output of Multidimensional Arrays

The template operator `>>` makes it possible to read numerical values from a file in the format `[x, y, z, ...]` where `x, y, z` are the results of the operator `>>` for class `X`. Class `X` must provide a default constructor for operator `>>` to work. That is, the statement `X x;` must work for `X`. This input operator is limited to numerical values.

Likewise, the template operator `<<` makes it possible to write to a file in the format `[x, y, z, ...]` where `x, y, z` are the results of the operator `<<` for class `X`. (This output operator is *not* limited to numerical values, as the input operator is.)

These two operators make it possible to read and write multidimensional arrays of numerical values like this:

```
IloArray<IloArray<IloIntArray> >
```

(Notice the space between `>>` at the end of that statement. It is necessary in C++.)

However, there is a practical limit of four on the number of dimensions supported by the input operator for reading multidimensional arrays. This limit is due to the inability of certain C++ compilers to support templates correctly. Specifically, you can read input by means of the input operator for multidimensional arrays of one, two, three, or four dimensions. There is no such limit on the number of dimensions with respect to the output operator for multidimensional arrays.

Constructors

```
public IloArray(IloEnv env,
               IloInt max=0)
```

This constructor creates an array of `max` elements, all of which are empty handles.

Methods

```
public void add(IloArray< X > ax) const
```

This member function appends the elements in `ax` to the invoking array.

```
public void add(IloInt more,
               X x) const
```

This member function appends *x* to the invoking array *multiple* times. The parameter *more* indicates how many times.

```
public void add(X x) const
```

This member function appends *x* to the invoking array.

```
public void clear()
```

This member function removes all the elements from the invoking array. In other words, it produces an empty array.

```
public void end()
```

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object. After a call to this member function, you cannot use the invoking extractable object again.

```
public IloEnv getEnv() const
```

This member function returns the environment where the invoking array was created. The elements of the invoking array belong to the same environment.

```
public IloInt getSize() const
```

This member function returns an integer indicating the size of the invoking array. An empty array has size 0 (zero).

```
public X & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index *i*.

```
public const X & operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index *i*. Concert Technology uses the `const` operator

```
IloArray operator[] (IloInt i) const;
```

on `const` arrays.

```
public void remove(IloInt first,
                  IloInt nb=1)
```

This member function removes elements from the invoking array. It begins removing elements at the index indicated by *first*, and it removes *nb* elements (*nb* = 1 by default).

IloBarrier

Category Class

InheritancePath



Definition File ilconcert/ilothread.h

Summary A system class to synchronize threads at a specified number.

Constructor Summary	
public	IloBarrier (int count)

Method Summary	
public int	wait ()

Description The class `IloBarrier` provides synchronization primitives adapted to Concert Technology. A barrier, an instance of this class, serves as a rendezvous for a specific number of threads. After you create a barrier for *n* threads, the first *n*-1 threads to reach that barrier will be blocked. The *n*th thread to arrive at the barrier completes the synchronization and wakes up the *n*-1 threads already waiting at that barrier. When the *n*th thread arrives, the barrier resets itself. Any other thread that arrives at this point is blocked and will participate in a new barrier of size *n*.

See `ILOUSEMT` for details about the compilation macro to use with instances of this class.

Note: *The class `IloBarrier` has nothing to do with the ILOG CPLEX barrier optimizer.*

System Class

`IloBarrier` is a system class.

Most Concert Technology classes actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloBarrier`, differ from that Concert Technology pattern. `IloBarrier` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloBarrier` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloBarrier` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloBarrier` in a Concert Technology environment because the destructor for that instance of `IloBarrier` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloBarrier` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloBarrier

Instances of `IloBarrier` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloBarrier` by calling the delete operator when your application no longer needs those objects.

See Also [IloCondition](#), [IloFastMutex](#), [ILOUSEMT](#)

Constructors `public IloBarrier(int count)`

This constructor creates an instance of `IloBarrier` of size `count` and allocates it on the C++ heap (not in a Concert Technology environment).

Methods `public int wait()`

The first `count - 1` calls to this member function block the calling thread. The last call (that is, the call numbered `count`) wakes up all the `count - 1` waiting threads. Once a thread has been woken up, it leaves the barrier. When a thread leaves the barrier (that is, when it returns from the `wait` call), it will return either 1 (one) or 0 (zero). If the thread returns 0, the barrier is not yet empty. If the thread returns 1, it was the last thread at the barrier.

A non-empty barrier contains blocked threads or exiting threads.

IloBaseEnvMutex

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Summary A class to initialize multithreading in an application.

Method Summary	
public virtual void	lock()
public virtual void	unlock()

Description An instance of this base class in the function [IloInitMT](#) initializes multithreading in a Concert Technology application. For a general purpose mutex, see the class [IloFastMutex](#).

See Also [IloFastMutex](#), [IloInitMT](#)

Methods public virtual void **lock()**
This member function locks a mutex.
public virtual void **unlock()**
This member function unlocks a mutex.

IloBool

Category	Type Definition
Definition File	ilconcert/ilosys.h
Synopsis	IloInt IloBool
Description	This type definition represents Boolean values in Concert Technology. Those values are <code>IloTrue</code> and <code>IloFalse</code>. Booleans are, in fact, integers of type IloInt. <code>IloFalse</code> is 0 (zero), and <code>IloTrue</code> is 1 (one). This type anticipates the built-in <code>bool</code> type proposed for standard C++. By using this type, you can be sure that the Concert Technology components of your application will port in this respect without source changes across different hardware platforms.
See Also	IloBoolArray , IloInt , IloModel , IloNum

IloBoolAbstraction

Category Global Function

Definition File ilconcert/ilomodel.h

Synopsis

```
public IloConstraint IloBoolAbstraction(const IloEnv env,
    const IloBoolVarArray y,
    const IloIntArray values)
public IloConstraint IloBoolAbstraction(const IloEnv env,
    const IloBoolVarArray avars,
    const IloAnyVarArray vars,
    const IloAnyArray values)
```

Summary A function to create a *constraint* that abstracts an array of Boolean variables.

Description This function creates and returns a constraint that abstracts an array of constrained Boolean variables in a model. It differs from `IloAbstraction` in that its `y`-array is an array of Boolean variables (also known as 0-1 variables or binary variables). Like `IloAbstraction`, for each element `x[i]`, there is a variable `y[i]` corresponding to the abstraction of `x[i]` with respect to an array of values. That is,

```
x[i] = v with v in values if and only if y[i] = IloTrue;
x[i] = v with v not in values if and only if y[i] = IloFalse.
```

This constraint maintains a many-to-one mapping that makes it possible to define constraints that impinge only on a particular set of values from the domains of constrained variables.

Example

For simplicity, assume that an array `x` consists of three elements with the domains {3}, {4}, and {5}. Assume that the values we are interested in are {4, 8, 12, 16}. Then `IloBoolAbstraction` produces the elements of the array `y`, like this:

X	&	Values	Y
---		-----	-----
3		4	IloFalse
4		8	IloTrue
5		12	IloFalse
		16	

Adding a Constraint to a Model, Extracting a Model for an Algorithm

In order for the constraint returned by `IloBoolAbstraction` to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

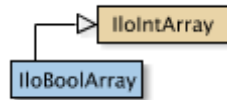
Exceptions

If the arrays `x` and `y` are not the same size, this function throws the exception `IloBoolAbstraction::InvalidArraysException`.

IloBoolArray

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Constructor Summary

public	IloBoolArray (IloArrayI * i=0)
public	IloBoolArray (const IloEnv env, IloInt n=0)
public	IloBoolArray (const IloEnv env, IloInt n, const IloBool v0)
public	IloBoolArray (const IloEnv env, IloInt n, const IloBool v0, const IloBool v1...)

Method Summary

public void	add (IloInt more, const IloBool x)
public void	add (const IloBool x)
public void	add (const IloBoolArray x)

Inherited methods from [IloIntArray](#)

[IloIntArray::contains](#), [IloIntArray::contains](#), [IloIntArray::discard](#), [IloIntArray::discard](#), [IloIntArray::operator\(\)](#), [IloIntArray::operator\[\]](#), [IloIntArray::operator\[\]](#), [IloIntArray::operator\[\]](#), [IloIntArray::toNumArray](#)

Description [IloBoolArray](#) is the array class of the basic Boolean class for a model. It is a handle class. The implementation class for [IloBoolArray](#) is the undocumented class [IloBoolArrayI](#).

Instances of `IloBoolArray` are extensible. (They differ from instances of `IlcBoolArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where “automatic” has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloBoolArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

See Also

[IloBool](#)

Constructors

```
public IloBoolArray(IloArrayI * i=0)
```

This constructor creates an array of Boolean values from an implementation object.

```
public IloBoolArray(const IloEnv env,
                   IloInt n=0)
```

This constructor creates an array of `n` Boolean values for use in a model in the environment indicated by `env`. By default, its elements are empty handles.

```
public IloBoolArray(const IloEnv env,
                   IloInt n,
                   const IloBool v0)
```

This constructor creates an array of `n` Boolean values.

```
public IloBoolArray(const IloEnv env,
                   IloInt n,
                   const IloBool v0,
                   const IloBool v1...) )
```

This constructor creates an array of `n` Boolean values; the elements of the new array take the corresponding values: `v0`, `v1`, ..., `v(n-1)`.

Methods

```
public void add(IloInt more,  
               const IloBool x)
```

This member function appends `x` to the invoking array of Boolean values; it appends `x` `more` times.

```
public void add(const IloBool x)
```

This member function appends the value `x` to the invoking array.

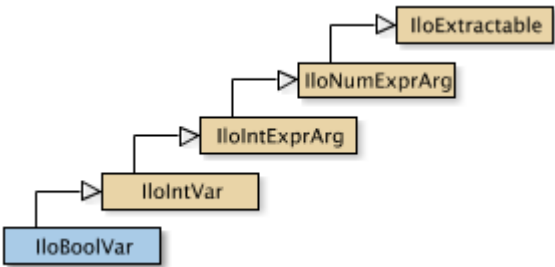
```
public void add(const IloBoolArray x)
```

This member function appends the values in the array `x` to the invoking array.

IloBoolVar

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary	
public	<code>IloBoolVar(IloEnv env,IloInt min=0,IloInt max=1,const char * name=0)</code>
public	<code>IloBoolVar(IloEnv env,const char * name)</code>
public	<code>IloBoolVar(const IloAddNumVar & column,const char * name=0)</code>

Inherited methods from <code>IloIntVar</code>
<code>IloIntVar::getImpl</code> , <code>IloIntVar::getLB</code> , <code>IloIntVar::getMax</code> , <code>IloIntVar::getMin</code> , <code>IloIntVar::getUB</code> , <code>IloIntVar::setBounds</code> , <code>IloIntVar::setLB</code> , <code>IloIntVar::setMax</code> , <code>IloIntVar::setMin</code> , <code>IloIntVar::setPossibleValues</code> , <code>IloIntVar::setUB</code>

Inherited methods from <code>IloIntExprArg</code>
<code>IloIntExprArg::getImpl</code>

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of this class represents a constrained Boolean variable in a Concert Technology model. Boolean variables are also known as binary decision variables. They can assume the values 0 (zero) or 1 (one).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

What Is Extracted

An instance of `IloBoolVar` is extracted by `IloSolver` (documented in the *ILOG Solver Reference Manual*) as an instance of the class `IlcBoolVar` (also documented in the *ILOG Solver Reference Manual*).

An instance of `IloBoolVar` is extracted by `IloCplex` as a column representing a numeric variable of type `Bool` with bounds as indicated by `IloBoolVar`.

See Also

[IloIntVar](#), [IloNumVar](#)

Constructors

```
public IloBoolVar(IloEnv env,
                 IloInt min=0,
                 IloInt max=1,
                 const char * name=0)
```

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, the Boolean variable assumes a value of 0.0 (zero) or 1.0. By default, its name is indicated by the empty string, but you can indicate a name of your own choice.

```
public IloBoolVar(IloEnv env,
                 const char * name)
```

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, its name is indicated by the empty string, but you can indicate a name of your own choice.

```
public IloBoolVar(const IloAddNumVar & column,  
                  const char * name=0)
```

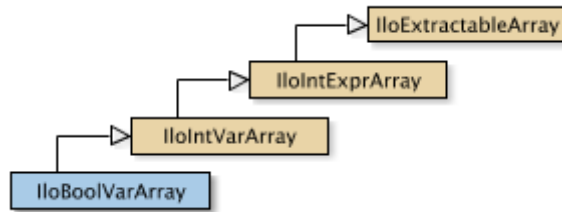
This constructor creates an instance of `IloBoolVar` like this:

```
IloNumVar(column, 0.0, 1.0, ILOBOOL, name);
```


IloBoolVarArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloBoolVarArray (IloDefaultArrayI * i=0)
public	IloBoolVarArray (const IloEnv env, IloInt n=0)
public	IloBoolVarArray (const IloEnv env, const IloNumColumnArray columnarray)

Method Summary

public void	add (IloInt more, const IloBoolVar x)
public void	add (const IloBoolVar x)
public void	add (const IloBoolVarArray x)
public IloBoolVar	operator[] (IloInt i) const
public IloBoolVar &	operator[] (IloInt i)

Inherited methods from [IloIntVarArray](#)

IloIntVarArray::add , IloIntVarArray::add , IloIntVarArray::add , IloIntVarArray::endElements , IloIntVarArray::operator() , IloIntVarArray::operator[] , IloIntVarArray::operator[] , IloIntVarArray::operator[] , IloIntVarArray::toNumVarArray	

Inherited methods from [IloIntExprArray](#)

[IloIntExprArray::add](#), [IloIntExprArray::add](#), [IloIntExprArray::add](#),
[IloIntExprArray::endElements](#), [IloIntExprArray::operator\(\)](#),
[IloIntExprArray::operator\[\]](#), [IloIntExprArray::operator\[\]](#),
[IloIntExprArray::operator\[\]](#)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description

For each basic type, Concert Technology defines a corresponding array class.

IloBoolVarArray is the array class of the Boolean variable class for a model. It is a handle class.

Instances of **IloBoolVarArray** are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloBoolVar](#)

Constructors

```
public IloBoolVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of Boolean variables.

```
public IloBoolVarArray(const IloEnv env,  
                      IloInt n=0)
```

This constructor creates an extensible array of `n` Boolean variables.

```
public IloBoolVarArray(const IloEnv env,  
                      const IloNumColumnArray columnarray)
```

This constructor creates an extensible array of Boolean variables from a column array.

Methods

```
public void add(IloInt more,  
               const IloBoolVar x)
```

This member function appends `x` to the invoking array of Boolean variables; it appends `x` `more` times.

```
public void add(const IloBoolVar x)
```

This member function appends the value `x` to the invoking array.

```
public void add(const IloBoolVarArray x)
```

This member function appends the variables in the array `x` to the invoking array.

```
public IloBoolVar operator[](IloInt i) const
```

This operator returns a reference to the extractable object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `const` operator

```
IloBoolVar operator[] (IloInt i) const;
```

on `const` arrays.

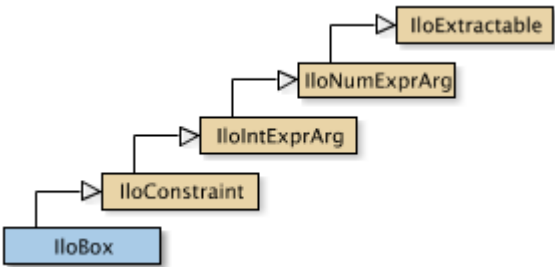
```
public IloBoolVar & operator[](IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position indicated by the index `i`.

IloBox

Category Class

InheritancePath



Definition File ilconcert/ilobox.h

Summary A class of multidimensional boxes for multidimensional placement problems.

Constructor Summary	
public	IloBox()
public	IloBox(IloBoxI * impl)
public	IloBox(const IloEnv env,IloInt dimensions,const IloIntVarArray origin,const IloIntArray size)

Method Summary	
public IloConstraint	contains(IloBox box)
public IloInt	getDimensions()
public IloBoxI *	getImpl() const
public IloConstraint	getNotOverlapConstraint()
public IloIntVar	getOrigin(IloInt dimension)
public IloInt	getSize(IloInt dimension)
public IloConstraint	notOverlapInDimension(IloBox box1,IloBox box2,IloInt dimension)
public IloConstraint	overlapInDimension(IloBox box1,IloBox box2,IloInt dimension)

public IloConstraint	precedenceInDimension(IloBox box1,IloBox box2,IloInt dimension)

Inherited methods from IloConstraint
IloConstraint::getImpl

Inherited methods from IloIntExprArg
IloIntExprArg::getImpl

Inherited methods from IloNumExprArg
IloNumExprArg::getImpl

Inherited methods from IloExtractable
IloExtractable::end, IloExtractable::getEnv, IloExtractable::getId, IloExtractable::getImpl, IloExtractable::getName, IloExtractable::getObject, IloExtractable::setName, IloExtractable::setObject

Description

Instances of the class `IloBox` are multidimensional boxes that appear in multidimensional placement problems. To solve packing or placement problems, you may need to be able to place boxes within a given container. In such a situation, both the boxes to place and the container to hold them are instances of the class `IloBox`.

To specify the containment relation that a given container holds a given box, use the member function `contains`.

See `IloCBox` documented in the *ILOG Solver Reference Manual*.

See `IloCFilterLevelConstraint` documented in the *ILOG Solver Reference Manual*.

Constructors

public `IloBox()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloBox(IloBoxI * impl)
```

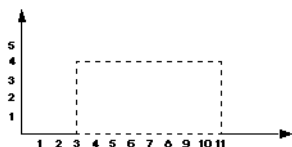
This constructor creates a handle object from a pointer to an implementation object.

```
public IloBox(const IloEnv env,
              IloInt dimensions,
              const IloIntVarArray origin,
              const IloIntArray size)
```

This constructor creates a box according to the specifications passed in the parameters. The parameter `dimensions` indicates the number of dimensions the box has. The arrays `origin` and `size` must contain the same number of elements as the number of dimensions of the box. In dimension i , the box extends from `origin[i]` to `origin[i] + size[i]`. For example, the statement

```
IloBox(env, 2, IloIntVarArray (env, 2, 3, 0), IloIntArray(env, 2, 8, 4));
```

creates a box as shown in this illustration.



Methods

```
public IloConstraint contains(IloBox box)
```

This member function creates a constraint that requires the invoking box to contain `box`. The parameter `box` should have the same number of dimensions as the invoking box.

```
public IloInt getDimensions()
```

This member function returns the number of dimensions of the invoking box.

```
public IloBoxI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloConstraint getNotOverlapConstraint()
```

This member function returns a constraint that specifies that none of the boxes contained in the invoking box can overlap along any of their dimensions.

The filter level of this constraint can be set using `IloSolver::setFilterLevel`. Currently two filter levels are allowed, `IlcBasic`, the default, and `IlcMedium`,

which is slower but propagates more. You can specify the default filter level of this constraint using the method `IloSolver::setDefaultFilterLevel`. Refer to the *ILOG Solver Reference Manual* for details about these methods.

```
public IloIntVar getOrigin(IloInt dimension)
```

This member function returns the origin of the invoking box along the dimension indicated by `dimension`.

```
public IloInt getSize(IloInt dimension)
```

This member function returns the length of the invoking box along the dimension indicated by `dimension`.

```
public IloConstraint notOverlapInDimension(IloBox box1,
                                           IloBox box2,
                                           IloInt dimension)
```

This member function returns a constraint that specifies that if `box1` and `box2` are contained in the invoking box, then they do not overlap along the dimension indicated by `dimension`.

```
public IloConstraint overlapInDimension(IloBox box1,
                                          IloBox box2,
                                          IloInt dimension)
```

This member function returns a constraint that specifies that if `box1` and `box2` are contained in the invoking box, then they must overlap along the dimension indicated by `dimension`.

```
public IloConstraint precedenceInDimension(IloBox box1,
                                             IloBox box2,
                                             IloInt dimension)
```

If `box1` and `box2` are contained in the invoking box, then this member function returns a constraint that specifies the precedence of `box1` over `box2` along the dimension indicated by `dimension`.

IloCard

Category	Global Function
Definition File	ilconcert/iloanyset.h
Synopsis	<pre>public IloIntVar IloCard(IloAnySetVar vars)</pre>
Description	This function creates and returns a constrained numeric variable that represents the number of elements in vars. In other words, it constrains the cardinality of a set variable. For example, to constrain mySet to contain four or more elements, you can use IloCard in the following way: <pre>model.add (IloCard (mySet) >= 4);</pre>

IloCeil

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<code>public IloNum IloCeil(IloNum val)</code>
Description	This function computes the least integer value not less than <code>val</code> .

Examples:

```
IloCeil(IloInfinity) is IloInfinity.  
IloCeil(-IloInfinity) is -IloInfinity.  
IloCeil(0) is 0.  
IloCeil(0.4) is 1.  
IloCeil(-0.4) is 0.  
IloCeil(0.5) is 1.  
IloCeil(-0.5) is 0.  
IloCeil(0.6) is 1.  
IloCeil(-0.6) is 0.
```

IloCondition

Category Class

InheritancePath



Definition File ilconcert/ilothread.h

Constructor Summary

public	IloCondition()

Method Summary

public void	broadcast()
public void	notify()
public void	wait(IloFastMutex * m)

Description

The class `IloCondition` provides synchronization primitives adapted to Concert Technology for use in a parallel application.

See `ILOUSEMT` for details about the `compilation` macro to use with instances of this class.

An instance of the class `IloCondition` allows several threads to synchronize on a specific event. In this context, inter-thread communication takes place through signals. A thread expecting a condition of the computation state (say, `conditionC`) to be true before it executes a `treatmentT` can wait until the condition is true. When computation reaches a state where `conditionC` holds, then another thread can signal this fact by notifying a single waiting thread or by broadcasting to all the waiting threads that `conditionC` has now been met.

The conventional template for waiting on `conditionC` looks like this:

```
mutex.lock();
while (conditionC does not hold)
    condition.wait(&mutex);
doTreatmentT();
```

```
mutex.unlock();
```

That template has the following properties:

- ◆ The whole fragment is a critical section so that the evaluation of `conditionC` is protected. (Indeed, it would be unsafe to evaluate `conditionC` while at the same time another thread modifies the computation state and affects the truth value of `conditionC`.) The pair of member functions `IloFastMutex::lock` and `IloFastMutex::unlock` delimit the critical section.
- ◆ When a thread enters the `wait` call, the mutex is automatically unlocked by the system.
- ◆ The loop that repeatedly checks `conditionC` is essential to the correctness of the code fragment. It protects against the following possibility: between the time that a thread modifies the computation state (so that `conditionC` holds) and notifies a waiting thread and the moment the waiting thread wakes up, the computation state might have been changed by another thread, and `conditionC` might very well be false.
- ◆ Upon returning from the `wait` call, the mutex is locked. The operation of waking up and locking the mutex is atomic. In other words, nothing can happen between the waking and the locking.

System Class

`IloCondition` is a system class.

Most Concert Technology classes actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation on the Concert Technology heap are managed by an instance of `IloEnv`.

However, system classes, such as `IloCondition`, differ from that Concert Technology pattern. `IloCondition` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloCondition` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloCondition` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloCondition` on the Concert Technology heap because the destructor for that instance of `IloCondition` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects on the Concert Technology heap.

For example, it is not a good idea to make an instance of `IloCondition` part of a conventional Concert Technology model allocated on the Concert Technology heap because that instance will not automatically be de-allocated from the Concert Technology heap along with the other Concert Technology objects.

De-allocating Instances of `IloCondition`

Instances of `IloCondition` differ from the usual Concert Technology objects because they are not allocated on the Concert Technology heap, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloCondition` by calling the delete operator when your application no longer needs those objects.

See Also [IloFastMutex](#), `ILOUSEMT`

Constructors `public IloCondition()`

This constructor creates an instance of `IloCondition` and allocates it on the C++ heap (not in a Concert Technology environment). The instance contains data structures specific to an operating system.

Methods `public void broadcast()`

This member function wakes all threads currently waiting on the invoking condition. If there are no threads waiting, this member function does nothing.

`public void notify()`

This member function wakes one of the threads currently waiting on the invoking condition.

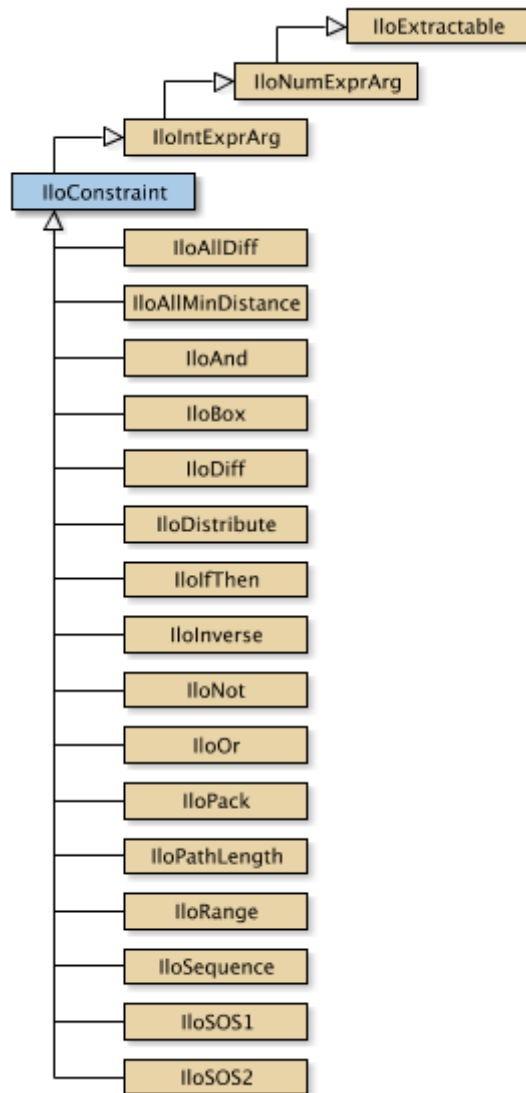
`public void wait(IloFastMutex * m)`

This member function first puts the calling thread to sleep while it unlocks the mutex `m`. Then, when either of the member functions `broadcast` or `notify` wakes up that thread, this member function acquires the lock on `m` and returns.

IloConstraint

Category Class

InheritancePath



Definition File `ilconcert/iloexpression.h`

Constructor Summary

public	IloConstraint()
public	IloConstraint(IloConstraintI * impl)

Method Summary

public IloConstraintI *	IloConstraint::getImpl() const

Inherited methods from [IloIntExprArg](#)

IloIntExprArg::getImpl

Inherited methods from [IloNumExprArg](#)

IloNumExprArg::getImpl

Inherited methods from [IloExtractable](#)

IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description

An instance of this class is a constraint in a model. To create a constraint, you can:

- ◆ use a constructor from a subclass of [IloConstraint](#), such as [IloRange](#), [IloAllDiff](#), etc. For example:

```
IloAllDiff allDiff(env, vars);
```

- ◆ use a logical operator between constraints to return a constraint. For example, you can use the logical operators on other constraints, like this:

```
IloOr or = myConstraint1 ||
myConstraint2;
```

- ◆ use an arithmetic operator between a numeric variable and an expression to return a constraint. For example, you can use the arithmetic operators on numeric variables or expressions, like this:

```
IloRange rng = ( x + 3*y <= 7 );
```

After you create a constraint, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function [IloModel::add](#) or the template [IloAdd](#). Then extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also [IloConstraintArray](#), [IloModel](#), [IloRange](#)

Constructors

```
public IloConstraint()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloConstraint(IloConstraintI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

Methods

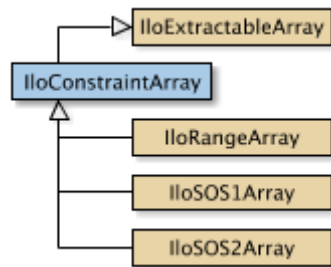
```
public IloConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloConstraintArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloConstraintArray (IloDefaultArrayI * i=0)
public	IloConstraintArray (const IloConstraintArray & copy)
public	IloConstraintArray (const IloEnv env, IloInt n=0)

Method Summary

public void	add (IloInt more, const IloConstraint x)
public void	add (const IloConstraint x)
public void	add (const IloConstraintArray x)
public IloConstraint	operator[] (IloInt i) const
public IloConstraint &	operator[] (IloInt i)

Inherited methods from [IloExtractableArray](#)


```
IloExtractableArray::add, IloExtractableArray::add,
IloExtractableArray::add, IloExtractableArray::endElements
```

Description

For each basic type, Concert Technology defines a corresponding array class. `IloConstraintArray` is the array class of constraints for a model.

Instances of `IloConstraintArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

Arrays

See Also

[IloConstraint](#), [operator>>](#), [operator](#)

Constructors

```
public IloConstraintArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloConstraintArray(const IloConstraintArray & copy)
```

This copy constructor makes a copy of the array indicated by `copy`.

```
public IloConstraintArray(const IloEnv env,
                        IloInt n=0)
```

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

```
public void add(IloInt more,
               const IloConstraint x)
```

This member function appends `constraint` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloConstraint x)
```

This member function appends `constraint` to the invoking array.

```
public void add(const IloConstraintArray x)
```

This member function appends the elements in `array` to the invoking array.

```
public IloConstraint operator[](IloInt i) const
```

This operator returns a reference to the constraint located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `CONST` operator

```
IloConstraint operator[] (IloInt i) const;
```

on `CONST` arrays.

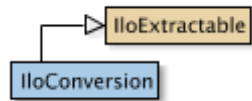
```
public IloConstraint & operator[](IloInt i)
```

This operator returns a reference to the constraint located in the invoking array at the position indicated by the index `i`.

IloConversion

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary	
public	IloConversion()
public	IloConversion(IloConversionI * impl)
public	IloConversion(const IloEnv env, const IloNumVar var, IloNumVar::Type t, const char * name=0)
public	IloConversion(const IloEnv env, const IloNumVarArray vars, IloNumVar::Type t, const char * name=0)
public	IloConversion(const IloEnv env, const IloIntVarArray vars, IloNumVar::Type t, const char * name=0)

Method Summary	
public IloConversionI *	getImpl() const

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description

An instance of this class offers you a means to change the type of a numeric variable. For example, in a model (an instance of `IloModel`) extracted for an algorithm (such as an instance of the class `IloCplex`), you may want to convert the type of a given numeric variable (an instance of `IloNumVar`) from `ILOFLOAT` to `ILOINT` or to `ILOBOOL` (or from `IloNumVar::Float` to `IloNumVar::Int` or to `IloNumVar::Bool`). Such a change is known as a conversion.

After you create a conversion, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` or the template `IloAdd`. Then extract the model for an algorithm (such as an instance of `IloCplex`) with the member function `IloAlgorithm::extract`.

Multiple Type Conversions of the Same Variable

You can convert the type of a numeric variable in a model. To do so, create an instance of `IloConversion` and add it to the model. You can also convert the type of a numeric variable after the model has been extracted for an algorithm (such as an instance of `IloCplex`).

An instance of `IloCplex` will not accept more than one type conversion of the same variable. That is, you can change the type once, but not twice, in a single instance of `IloCplex`. Attempts to convert the type of the same variable more than once will throw the exception `IloCplex::MultipleConversionException`.

In situations where you want to change the type of a numeric variable more than once (for example, from Boolean to integer to floating-point), there are these possibilities:

- ◆ You can remove a prior conversion of a given variable in a given model. To do so, use its member function `IloExtractable::end` to delete it and optionally add a new conversion.
- ◆ You can apply different conversions to a given variable in more than one model, like this:

```
IloNumVar x(env, 0, 10, ILOBOOL);
IloRange rng = (x <= 10);
IloModel mdl1(env);
mdl1.add(rng);
mdl1.add(IloConversion(env, x, ILOINT));
IloCplex cplex1(mdl1);
IloModel mdl2(env);
mdl2.add(rng);
mdl2.add(IloConversion(env, x, ILOFLOAT));
IloCplex cplex2(mdl2);
```

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also [IloModel](#)

Constructors

```
public IloConversion()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloConversion(IloConversionI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloConversion(const IloEnv env,
                    const IloNumVar var,
                    IloNumVar::Type t,
                    const char * name=0)
```

This constructor accepts a numeric variable and a type; it creates a handle to a type conversion to change the type of the variable `var` to the type indicated by `t`. You may use the parameter `name` to name the type conversion so that you can refer to it by a string identifier.

```
public IloConversion(const IloEnv env,
                    const IloNumVarArray vars,
                    IloNumVar::Type t,
                    const char * name=0)
```

This constructor accepts an array of numeric variables and a type; it creates a handle to a type conversion to change the type of each variable in the array `vars` to the type indicated by `t`. You may use the parameter `name` to name the type conversion so that you can refer to it by a string identifier.

```
public IloConversion(const IloEnv env,
                    const IloIntVarArray vars,
                    IloNumVar::Type t,
                    const char * name=0)
```

This constructor accepts an array of integer variables and a type; it creates a handle to a type conversion to change the type of each variable in the array `vars` to the type indicated by `t`. You may use the parameter `name` to name the type conversion so that you can refer to it by a string identifier.

Methods

```
public IloConversionI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloCplex::Status

Category	Inner Type Definition
Definition File	ilcplex/ilocplexi.h
Synopsis	CplexStatus Status
Summary	An enumeration for the class IloAlgorithm .
Description	IloAlgorithm is the base class of algorithms in Concert Technology, and IloAlgorithm::Status is an enumeration limited in scope to the class IloAlgorithm. The member function getStatus returns a status indicating information about the current model and the solution. Unknown indicates that the algorithm has no information about the solution of the model. Feasible indicates that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions getValue access this feasible solution. Optimal indicates that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions getValue access this optimal solution. Infeasible indicates that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model. Unbounded indicates that the algorithm proved the model unbounded. InfeasibleOrUnbounded indicates that the model is infeasible or unbounded. Error indicates that an error occurred and, on platforms that support exceptions, that an exception has been thrown.
See Also	IloAlgorithm , operator

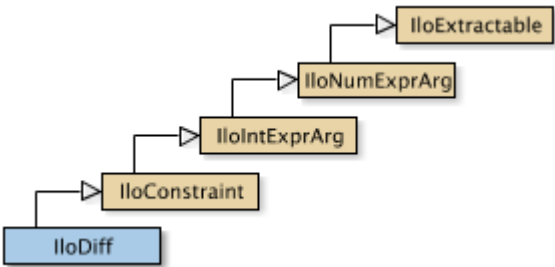
IloDeleterMode

Category	Enumeration
Definition File	ilconcert/iloenv.h
Synopsis	<pre> IloDeleterMode{ IloLinearDeleterMode, IloSafeDeleterMode, IloRecursiveDeleterMode, IloSmartDeleterMode }; </pre>
Summary	An enumeration to set the mode of an <code>IloDeleter</code> .
Description	This enumeration allows you to set the <code>IloDeleter</code> mode. The modes <code>IloRecursiveDeleterMode</code> and <code>IloSafeDeleterMode</code> are not documented and should not be used. You can set the mode using the member function <code>IloEnv::setDeleter</code>. For a description of deletion in ILOG Concert Technology, refer to Deletion of Extractables.
Fields	<pre> IloLinearDeleterMode IloSafeDeleterMode IloRecursiveDeleterMode IloSmartDeleterMode </pre>

IloDiff

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloDiff()
public	IloDiff(IloDiffI * impl)
public	IloDiff(const IloEnv env,const IloNumExprArg expr1,const IloNumExprArg expr2,const char * name=0)
public	IloDiff(const IloEnv env,const IloNumExprArg expr1,IloNum val,const char * name=0)

Method Summary	
public IloDiffI *	getImpl() const

Inherited methods from IloConstraint	
IloConstraint::getImpl	

Inherited methods from [IloIntExprArg](#)

[IloIntExprArg::getImpl](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of this class is a constraint that enforces inequality (that is, “not equal” as indicated by !=) in Concert Technology.

To create a constraint, you can:

- ◆ use the inequality operator != on constrained variables (instances of [IloNumVar](#) and its subclasses) or expressions (instances of [IloExpr](#) and its subclasses).
- ◆ use a constructor from this class.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also

[IloAllDiff](#), [IloConstraint](#), [IloExpr](#), [IloNumVar](#)

Constructors

```
public IloDiff()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloDiff(IloDiffI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloDiff(const IloEnv env,
              const IloNumExprArg expr1,
              const IloNumExprArg expr2,
              const char * name=0)
```

This constructor creates a constraint that enforces inequality (!=) in a model between the two expressions that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

```
public IloDiff(const IloEnv env,
              const IloNumExprArg expr1,
              IloNum val,
              const char * name=0)
```

This constructor creates a constraint that enforces inequality (!=) in a model between the expression `expr1` and the floating-point value that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

Methods

```
public IloDiffI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

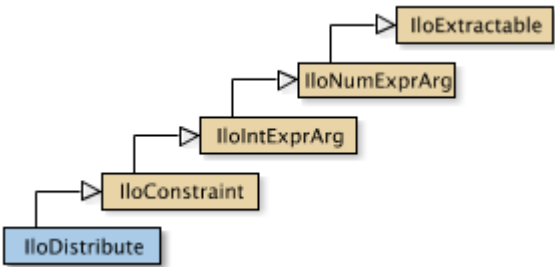
IloDisableNANDetection

Category	Global Function
Definition File	ilconcert/ilosys.h
Synopsis	<pre>public void IloDisableNANDetection()</pre>
Description	This function disables NaN (Not a number) detection.

IloDistribute

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloDistribute()
public	IloDistribute(IloDistributeI * impl)
public	IloDistribute(const IloEnv env, const IloIntVarArray cards, const IloIntArray values, const IloIntVarArray vars, const char * name=0)
public	IloDistribute(const IloEnv env, const IloIntVarArray cards, const IloIntVarArray vars, const char * name=0)

Method Summary	
public IloDistributeI *	getImpl() const

Inherited methods from IloConstraint	
IloConstraint::getImpl	

Inherited methods from [IloIntExprArg](#)[IloIntExprArg::getImpl](#)**Inherited methods from [IloNumExprArg](#)**[IloNumExprArg::getImpl](#)**Inherited methods from [IloExtractable](#)**
[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)
Description

An instance of this class is a counting constraint in a model. You can use an instance of this class to count the number of occurrences of several values among the constrained variables in an array of constrained variables. You can also use an instance of this class to force the constrained variables of an array to assume values in such a way that only a limited number of the constrained variables assume each value.

For example, if we have five cars to paint in three available colors, then we might refer to the cars as `c1`, `c2`, `c3`, `c4`, `c5`, and the colors as `p1`, `p2`, `p3`. If we can allow no more than three cars to be painted `p1`, exactly three cars to be painted `p2`, and no more than one car to be painted `p3`, then we can represent our problem informally in terms of this constraint like this:

```
cards = [[0,3], [3,3], [0,1]]
values = [p1, p2, p3]
vars = [c1, c2, c3, c4, c5]
```

In more formal terms, the constrained variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`. More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `values[i]` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value at

`values[i]`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain the value at `values[i]` in their domain.

The arrays `cards` and `values` must be the same length; otherwise, Concert Technology throws an exception on platforms that support C++ exceptions when exceptions are enabled.

When an instance of this class is created by a constructor with only `cards` and `vars` as arguments (that is, there is no `values` parameter), then the array of values that are being counted must be an array of consecutive integers starting with 0 (zero). In that case, for each `i`, `cards[i]` is equal to the number of occurrences of `i` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value `i`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain `i` in their domain.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also [IloConstraint](#), [IloSequence](#)

Constructors

```
public IloDistribute()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloDistribute(IloDistributeI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloDistribute(const IloEnv env,
                    const IloIntVarArray cards,
                    const IloIntArray values,
                    const IloIntVarArray vars,
                    const char * name=0)
```

This constructor creates a counting constraint in a model. You must use the template [IloAdd](#) or the member function [IloModel::add](#) to add this constraint to a model and then use [IloAlgorithm::extract](#) to extract the model for an algorithm in order for the constraint to be taken into account.

The arrays `cards` and `values` must be the same length; otherwise Concert Technology throws the exception `InvalidArraysException`.

```
public IloDistribute(const IloEnv env,
                    const IloIntVarArray cards,
                    const IloIntVarArray vars,
```

```
const char * name=0)
```

This constructor creates a counting constraint in a model. You must use the template [IloAdd](#) or the member function [IloModel::add](#) to add this constraint to a model and then use [IloAlgorithm::extract](#) to extract the model for an algorithm in order for the constraint to be taken into account.

Methods

```
public IloDistributeI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

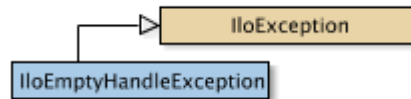
IloDiv

Category	Global Function
Definition File	ilconcert/iloexpression.h
Synopsis	<pre>public IloIntExprArg IloDiv(const IloIntExprArg x, const IloIntExprArg y) public IloIntExprArg IloDiv(const IloIntExprArg x, IloInt y) public IloIntExprArg IloDiv(IloInt x, const IloIntExprArg y)</pre>
Description	This function is available for integer division. For numerical division, use operator/ .

IloEmptyHandleException

Category Class

InheritancePath



Definition File `ilconcert/iloenv.h`

Summary The class of exceptions thrown if an empty handle is passed.

Constructor Summary	
public	IloEmptyHandleException()
public	IloEmptyHandleException(const char * message)

Inherited methods from IloException
IloException::end , IloException::getMessage

Description The exception `IloEmptyHandleException` is thrown if an empty handle is passed as parameter to a method, function, or class constructor.

Constructors `public IloEmptyHandleException()`

`public IloEmptyHandleException(const char * message)`

This constructor creates an exception containing the message string `message`.

IloEnableNANDetection

Category	Global Function
Definition File	ilconcert/ilosys.h
Synopsis	<pre>public void IloEnableNANDetection()</pre>
Description	This function enables NaN (Not a number) detection.

IloEndMT

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<pre>public void IloEndMT()</pre>
Description	This function ends multithreading in a Concert Technology application.

IloEnv

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Summary The class of environments for models or algorithms in Concert Technology.

Constructor Summary

public	IloEnv()
public	IloEnv(IloEnvI * impl)

Method Summary

public IloBool	assumeStrictNumericalDivision() const
public void	assumeStrictNumericalDivision(IloBool val) const
public void	IloEnv::end()
public ostream &	IloEnv::error() const
public IloExtractableI *	getExtractable(IloInt id)
public IloEnvI *	getImpl() const
public IloInt	getMemoryUsage() const
public IloRandom	getRandom() const
public IloNum	getTime() const
public IloInt	getTotalMemoryUsage() const
public const char *	getVersion() const
public ostream &	IloEnv::out() const
public void	printTime() const
public void	IloEnv::setDeleter(IloDeleterMode mode) const
public void	setNormalizer(IloBool val) const
public void	unsetDeleter() const
public ostream &	IloEnv::warning() const

Description

An instance of this class is an environment, managing memory and identifiers for modeling objects. Every Concert Technology object, such as an extractable object, a model, or an algorithm, must belong to an environment. In C++ terms, when you construct a model (an instance of [IloModel](#)) or an algorithm (an instance of [IloCplex](#) documented in this manual or [IloSolver](#), documented in the ILOG Solver Reference Manual, for example), then you must pass one instance of [IloEnv](#) as a parameter of that constructor.

Environment and Memory Management

An environment (an instance of [IloEnv](#)) efficiently manages memory allocations for the objects constructed with that environment as a parameter. For example, when Concert Technology objects in your model are extracted by an algorithm, those extracted objects are handled as efficiently as possible with respect to memory management; there is no unnecessary copying that might cause memory explosions in your application on the part of Concert Technology.

When your application deletes an instance of [IloEnv](#), Concert Technology will automatically delete all models and algorithms depending on that environment as well. You delete an environment by calling the member function `env.end`.

The memory allocated for Concert Technology arrays, expressions, sets, and columns is not freed until all references to these objects have terminated and the objects themselves have been deleted.

Certain classes documented in this manual, such as [IloFastMutex](#), are known as system classes. They do not belong to a Concert Technology environment; in other words, an instance of [IloEnv](#) is *not* a parameter in their constructors. As a consequence, a Concert Technology environment does *not* attempt to manage their memory allocation and de-allocation; a call of `IloEnv::end` will *not* delete an instance of a system class. These system classes are clearly designated in this documentation, and the appropriate constructors and destructors for them are documented in this manual as well.

Environment and Initialization

An instance of [IloEnv](#) in your application initializes certain data structures and modeling facilities for Concert Technology. For example, [IloEnv](#) initializes the symbolic constant [IloInfinity](#).

The environment also indicates the current assumptions about normalization or the reduction of terms in linear expressions. For an explanation of this concept, see the concept [Normalization: Reducing Linear Terms](#)

Environment and Communication Streams

An instance of [IloEnv](#) in your application initializes the default output streams for general information, for error messages, and for warnings.

Environment and Extractable Objects

Every extractable object in your problem must belong to an instance of `IloEnv`. In C++ terms, in the constructor of certain extractable objects that you create, such as a constrained variable, you must pass an instance of `IloEnv` as a parameter to indicate which environment the extractable object belongs to. An extractable object (that is, an instance of `IloExtractable` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Two different environments cannot share the same extractable object.

You can extract objects from only one environment into a given algorithm. In other words, algorithms do not extract objects from two or more different environments.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also [IloException](#), [IloModel](#), operator `new`

Constructors `public IloEnv()`

This constructor creates an environment to manage the extractable objects in Concert Technology.

`public IloEnv(IloEnvI * impl)`

This constructor creates an environment (a handle) from its implementation object.

Methods `public IloBool assumeStrictNumericalDivision() const`

This member function checks whether the arithmetic division operator `/` is restricted to numerical division only. If it returns `IloTrue`, operator `/` is restricted, and integer division is accessible only through the `IloDiv` function.

If it returns `IloFalse`, operator `/` is not restricted.

The default value is `IloTrue`.

`public void assumeStrictNumericalDivision(IloBool val) const`

This member function specifies whether the arithmetic division operator `/` is restricted to numerical division only. If `val` is `IloTrue`, operator `/` is restricted, and integer division is accessible only through the `IloDiv` function.

If `val` is `IloFalse`, operator `/` is not restricted.

The default value is `IloTrue`.

`public void end()`

When you call this member function, it cleans up the invoking environment. In other words, it deletes all the extractable objects (instances of `IloExtractable` and its subclasses) created in that environment and frees the memory allocated for them. It also deletes all algorithms (instances of `IloAlgorithm` and its subclasses) created in that environment and frees memory allocated for them as well, including the representations of extractable objects extracted for those algorithms.

The memory allocated for Concert Technology arrays, expressions, sets, and columns is not freed until all references to these objects have terminated and the objects themselves have been deleted.

```
public ostream & error() const
```

This member function returns a reference to the output stream currently used for error messages from the invoking environment. It is initialized as `cerr`.

```
public IloExtractableI * getExtractable(IloInt id)
```

This member function returns the extractable associated with the specified identifier `id`.

```
public IloEnvI * getImpl() const
```

This member function returns the implementation object of the invoking environment.

```
public IloInt getMemoryUsage() const
```

This member function returns a value in bytes indicating how full the heap is.

```
public IloRandom getRandom() const
```

Each instance of `IloEnv` contains a random number generator, an instance of the class `IloRandom`. This member function returns that `IloRandom` instance.

```
public IloNum getTime() const
```

This member function returns the amount of time elapsed in seconds since the construction of the invoking environment. (The member function `printTime` directs this information to the output stream of the invoking environment.)

```
public IloInt getTotalMemoryUsage() const
```

This member function returns a value in bytes indicating how large the heap is.

```
public const char * getVersion() const
```

This member function returns a string indicating the version of ILOG Concert Technology.

```
public ostream & out() const
```

This member function returns a reference to the output stream currently used for logging. General output from the invoking environment is accessible through this member function. By default, the logging output stream is defined by an instance of `IloEnv` as `cout`.

```
public void printTime() const
```

This member function directs the output of the member function `getTime` to the output stream of the invoking environment. (The member function `getTime` accesses the elapsed time in seconds since the creation of the invoking environment.)

```
public void setDeleter(IloDeleterMode mode) const
```

This member function sets the mode for the deletion of extractables, as described in the concept `Deletion of Extractables`. The mode can be `IloLinearDeleterMode` or `IloSafeDeleterMode`.

```
public void setNormalizer(IloBool val) const
```

This member function turns on or off the facilities in Concert Technology for normalizing linear expressions. Normalizing linear expressions is also known as reducing the terms of a linear expression. In this context, a linear expression that does not contain multiple terms with the same variable is said to be normalized. The concept in this manual offers examples of this idea.

When `val` is `IloTrue`, (the default), then Concert Technology analyzes linear expressions to determine whether any variable appears more than once in a given linear expression. It then combines terms in the linear expression to eliminate any duplication of variables. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing in the case of duplicated variables in the terms of a linear expression.

When `val` is `IloFalse`, then Concert Technology assumes that all linear expressions in the invoking environment have already been processed to reduce them to their most efficient form. In other words, Concert Technology assumes that linear expressions have been normalized. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

```
public void unsetDeleter() const
```

This member function unsets the mode for the deletion of extractables, as described in the concept `Deletion of Extractables`.

```
public ostream & warning() const
```

This member function returns a reference to the output stream currently used for warnings from the invoking environment. By default, the warning output stream is defined by an instance of `IloEnv` as `cout`.

IloEnvironmentMismatch

Category Class

InheritancePath



Definition File `ilconcert/iloenv.h`

Constructor Summary

public	IloEnvironmentMismatch()
public	IloEnvironmentMismatch(const char * message)

Inherited methods from [IloException](#)

[IloException::end](#), [IloException::getMessage](#)

Description The `IloEnvironmentMismatch` exception is thrown if you try to build an object using objects from another environment.

Constructors

```

public IloEnvironmentMismatch()

public IloEnvironmentMismatch(const char * message)
  
```

IloEqIntersection

Category Global Function

Definition File ilconcert/iloanysset.h

Synopsis

```
public IloConstraint IloEqIntersection(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySetVar var2,
    const IloAnySetVar var3)
public IloConstraint IloEqIntersection(const IloEnv env,
    const IloIntSetVar var1,
    const IloIntSetVar var2,
    const IloIntSetVar var3)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in Concert Technology. The constraint forces the intersection of the sets `var2` and `var3` to be precisely the elements of the set `intersection`.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloEqPartition

Category Global Function

Definition File ilconcert/iloanyset.h

Synopsis

```
public IloConstraint IloEqPartition(const IloEnv env,
    const IloAnySetVar var,
    const IloAnySetVarArray vars)
public IloConstraint IloEqPartition(const IloEnv,
    const IloIntSetVar var,
    const IloIntSetVarArray vars)
```

Description These functions create and return a constraint. When that constraint is posted, it insures that the value of the variable `var` will be required by exactly one set variable of the array `vars`. In this context, a constraint will be posted after it has been added to a model and extracted by a solver (for example, an instance of `IloSolver` documented in the *ILOG Solver Reference Manual*).

IloEqUnion

Category Global Function

Definition File ilconcert/iloanyset.h

Synopsis

```
public IloConstraint IloEqUnion(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySetVar var2,
    const IloAnySetVar var3)
public IloConstraint IloEqUnion(const IloEnv env,
    const IloAnySetVar var,
    const IloAnySetVarArray vars)
public IloConstraint IloEqUnion(const IloEnv env,
    const IloIntSetVar var1,
    const IloIntSetVar var2,
    const IloIntSetVar var3)
public IloConstraint IloEqUnion(const IloEnv,
    const IloIntSetVar var,
    const IloIntSetVarArray vars)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. When its parameters are two sets of variables, such as `var2` and `var3`, the constraint forces the union of the sets `var2` and `var3` to be precisely the elements of the set `unionSet`. Likewise, when its parameters include an array of set variables, such as `vars`, the constraint forces the union of the *elements* of that array to be `unionSet`.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Category	Class

```

classDiagram
    class IloException
    class IloAlgorithm_Exception["IloAlgorithm::Exception"]
    class IloEmptyHandleException
    class IloEnvironmentMismatch
    class IloNumExpr_NonLinearExpression["IloNumExpr::NonLinearExpression"]
    IloException <|-- IloAlgorithm_Exception
    IloException <|-- IloEmptyHandleException
    IloException <|-- IloEnvironmentMismatch
    IloException <|-- IloNumExpr_NonLinearExpression
  
```

Constructor Summary	
protected	IloException (const char * message=0, IloBool deleteMessage= IloFalse)

Description	This class is the base class for exceptions in Concert Technology. An instance of this class represents an exception on platforms that support exceptions when exceptions are enabled.

Constructors

```
protected IloException(const char * message=0,  
                        IloBool deleteMessage=IloFalse)
```

```
Methods           public virtual void end()
```

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

```
public virtual const char * getMessage() const
```

This member function returns the message (a character string) of the invoking exception.

IloExponent

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloExponent(const IloNumExprArg arg)
public IloNum IloExponent(IloNum val)
```

Description

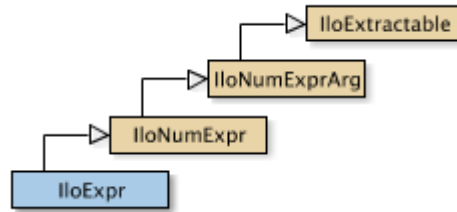
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

IloExponent returns the exponentiation of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `exp`.

IloExpr

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloExpr()
public	IloExpr(IloNumExprI * expr)
public	IloExpr(const IloNumLinExprTerm term)
public	IloExpr(const IloIntLinExprTerm term)
public	IloExpr(IloNumExprArg)
public	IloExpr(const IloEnv env, IloNum=0)

Method Summary

public IloNum	getConstant() const
public IloNumLinTermI *	getImpl() const
public IloExpr::LinearIterator	getLinearIterator() const
public IloBool	isNormalized() const
public IloInt	normalize() const
public IloExpr &	operator *=(IloNum val)
public IloExpr &	operator +=(const IloIntLinExprTerm term)
public IloExpr &	operator +=(const IloNumLinExprTerm term)
public IloExpr &	operator +=(const IloIntVar var)
public IloExpr &	operator +=(const IloNumVar var)
public IloExpr &	operator +=(const IloNumExprArg expr)

public IloExpr &	operator+=(IloNum val)
public IloExpr &	operator-=(const IloIntLinExprTerm term)
public IloExpr &	operator-=(const IloNumLinExprTerm term)
public IloExpr &	operator-=(const IloIntVar var)
public IloExpr &	operator-=(const IloNumVar var)
public IloExpr &	operator-=(const IloNumExprArg expr)
public IloExpr &	operator-=(IloNum val)
public IloExpr &	operator/=(IloNum val)
public void	remove(const IloNumVarArray vars)
public void	setConstant(IloNum cst)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, IloNumArray values)
public void	setNumConstant(IloNum constant)

Inherited methods from [IloNumExpr](#)

[IloNumExpr::getImpl](#), [IloNumExpr::operator *](#), [IloNumExpr::operator +=](#),
[IloNumExpr::operator +=](#), [IloNumExpr::operator -=](#), [IloNumExpr::operator -=](#),
[IloNumExpr::operator /=](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Inner Class

<code>IloExpr::IloExpr::LinearIterator</code>	An iterator over the linear part of an expression.

Description

An instance of this class represents an expression in a model. An instance of `IloExpr` is a handle.

Expressions in Environments

The variables in an expression must all belong to the same environment as the expression itself. In other words, you must not mix variables from different environments within the same expression.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

Programming Hint: Creating Expressions

In addition to using a constructor of this class to create an expression, you may also initialize an instance of `IloExpr` as a C++ expression built from variables of a model. For example:

```
IloNumVar x;
IloNumVar y;
IloExpr expr = x + y;
```

Programming Hint: Empty Handles and Null Expressions

This statement creates an empty handle:

```
IloExpr e1;
```

You must initialize it before you use it. For example, if you attempt to use it in this way:

```
e1 += 10; // BAD IDEA
```

Without the compiler option `-DNDEBUG`, that line will cause an `assert` statement to fail because you are attempting to use an empty handle.

In contrast, the following statement

```
IloExpr e2(env);
```

creates a handle to a null expression. You can use this handle to build up an expression, for example, in this way:

```
e2 += 10; // OK
```

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, $x + 3y$ is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, $x + 3y + 2x$. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `setNormalizer(IloTrue)`.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `setNormalizer(IloFalse)`.

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of this class to fail if you do not compile with the flag `-DNDEBUG`.

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of a failed assertion in case of duplicates.

See Also

[IloExprArray](#), [IloModel](#)

Constructors

```
public IloExpr()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloExpr(IloNumExprI * expr)
```

This constructor creates an expression from a pointer to the implementation class of numeric expressions `IloNumExprI*`.

```
public IloExpr(const IloNumLinExprTerm term)
```

This constructor creates an integer expression with linear terms using the undocumented class `IloNumLinExprTerm`.

```
public IloExpr(const IloIntLinExprTerm term)
```

This constructor creates an integer expression with linear terms using the undocumented class `IloIntLinExprTerm`.

```
public IloExpr(IloNumExprArg)
```

This constructor creates an expression using the undocumented class `IloNumExprArg`.

```
public IloExpr(const IloEnv env,
               IloNum=0)
```

This constructor creates an expression in the environment indicated by `env`. It may be used to build other expressions from variables belonging to `env`. You must not mix variables of different environments within an expression.

Methods

```
public IloNum getConstant() const
```

This member function returns the constant term in the invoking expression.

```
public IloNumLinTermI * getImpl() const
```

This member function returns the implementation object of the invoking enumerated variable.

```
public IloExpr::LinearIterator getLinearIterator() const
```

This methods returns a linear iterator on the invoking expression.

```
public IloBool isNormalized() const
```

This member function returns `IloTrue` if the invoking expression has been normalized using `normalize`.

```
public IloInt normalize() const
```

This member function normalizes the invoking linear expression. Normalizing is sometimes known as reducing the terms of a linear expression. That is, if there is more than one linear term using the same variable in the invoking linear expression, then this member function merges those linear terms into a single term expressed in that variable. The return value indicates the number of merged terms.

For example, $1*x + 17*y - 3*x$ becomes $17*y - 2*x$, and the member function returns 1 (one).

If you attempt to use this member function on a nonlinear expression, it throws an exception.

```
public IloExpr & operator *=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`.

```
public IloExpr & operator+=(const IloIntLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator+=(const IloNumLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator+=(const IloIntVar var)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator+=(const IloNumVar var)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator+=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator+=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x + \dots$

```
public IloExpr & operator-=(const IloIntLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x - \dots$

```
public IloExpr & operator-=(const IloNumLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x - \dots$

```
public IloExpr & operator-=(const IloIntVar var)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x - \dots$

```
public IloExpr & operator-=(const IloNumVar var)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x - \dots$

```
public IloExpr & operator-=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop.
It is more efficient than $x = x - \dots$

```
public IloExpr & operator-=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x - \dots$.

```
public IloExpr & operator/=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x / \dots$.

```
public void remove(const IloNumVarArray vars)
```

This member function removes all occurrences of all variables listed in the array `vars` from the invoking expression. For linear expressions, the effect of this member function is equivalent to setting the coefficient for all the variables listed in `vars` to 0 (zero).

```
public void setConstant(IloNum cst)
```

This member function assigns `cst` as the constant term in the invoking expression.

```
public void setLinearCoef(const IloNumVar var,
                          IloNum value)
```

This member function assigns `value` as the coefficient of `var` in the invoking expression if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a non linear expression. An attempt to do so will cause Concert Technology to throw an exception.

```
public void setLinearCoefs(const IloNumVarArray vars,
                           IloNumArray values)
```

For each of the variables in `vars`, this member function assigns the corresponding value of `values` as its linear coefficient if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a nonlinear expression. An attempt to do so will cause Concert Technology to throw an exception.

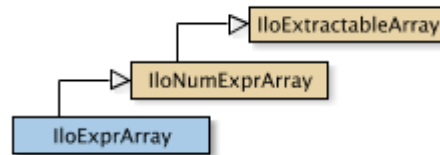
```
public void setNumConstant(IloNum constant)
```

This member function assigns `cst` as the constant term in the invoking expression.

IloExprArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloExprArray (IloDefaultArrayI * i=0)
public	IloExprArray (const IloEnv env, IloInt n=0)

Method Summary

public IloNumExprArg	operator() (IloNumExprArg anIntegerVariable) const

Inherited methods from [IloNumExprArray](#)

[IloNumExprArray::add](#), [IloNumExprArray::add](#), [IloNumExprArray::add](#),
[IloNumExprArray::endElements](#), [IloNumExprArray::operator\(\)](#),
[IloNumExprArray::operator\[\]](#)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description

For each basic type, Concert Technology defines a corresponding array class. `IloExprArray` is the array class of the expressions class ([IloExpr](#)) for a model.

Instances of `IloExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloExpr](#)

Constructors

```
public IloExprArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array of expressions for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloExprArray(const IloEnv env,  
                   IloInt n=0)
```

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

Methods

```
public IloNumExprArg operator()(IloNumExprArg anIntegerVariable) const
```

This subscripting operator is deprecated. It must be replaced by the operator `IloNumExprArg operator[] (IloIntExprArg anIntegerVariable) const;`

IloExpr::LinearIterator

Category Inner Class

InheritancePath

IloExpr::LinearIterator

Definition File ilconcert/iloexpression.h

Summary An iterator over the linear part of an expression.

Method Summary	
public IloNum	getCoef() const
public IloNumVar	getVar() const
public IloBool	ok() const
public void	operator++()

Description An instance of the nested class IloExpr::LinearIterator is an iterator that traverses the linear part of an expression.

Example

Start with an expression that contains both linear and non linear terms:

```
IloExpr e = 2*x + 3*y + cos(x);
```

Now define a linear iterator for the expression:

```
IloExpr::LinearIterator it(e);
```

That constructor creates a linear iterator initialized on the first linear term in `e`, that is, the term $(2 \cdot x)$. Consequently, a call to the member function `ok` returns `IloTrue`.

```
it.ok(); // returns IloTrue
```

A call to the member function `getCoef` returns the coefficient of the current linear term.

```
it.getCoef(); // returns 2 from the term (2*x)
```

Likewise, the member function `getVar` returns the handle of the variable of the current linear term.

```
it.getVar(); // returns handle of x from the term (2*x)
```

A call to the `operator++` at this point advances the iterator to the next linear term, $(3*y)$. The iterator ignores nonlinear terms in the expression.

```
++it; // goes to next linear term (3*y)
it.ok(); // returns IloTrue
it.getCoef(); // returns 3 from the term (3*y)
it.getVar(); // returns handle of y from the term (3*y)
++it; // goes to next linear term, if there is one in the expression
it.ok(); // returns IloFalse because there is no linear term
```

Methods

```
public IloNum getCoef() const
```

This member function returns the coefficient of the current term.

```
public IloNumVar getVar() const
```

This member function returns the variable of the current term.

```
public IloBool ok() const
```

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

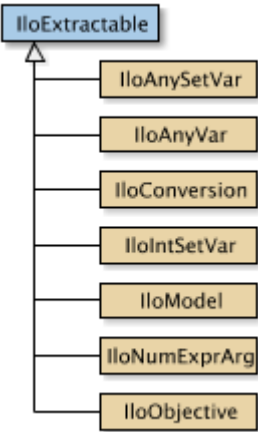
```
public void operator++()
```

This operator advances the iterator to point to the next term of the linear part of the expression.

IloExtractable

Category Class

InheritancePath



Definition File ilconcert/iloextractable.h

Constructor Summary	
public	IloExtractable (IloExtractableI * obj=0)

Method Summary	
public void	IloExtractable::end()
public IloEnv	IloExtractable::getEnv() const
public IloInt	IloExtractable::getId() const
public IloExtractableI *	IloExtractable::getImpl() const
public const char *	IloExtractable::getName() const
public IloAny	IloExtractable::getObject() const
public void	IloExtractable::setName (const char * name) const
public void	IloExtractable::setObject (IloAny obj) const

Description

This class is the base class of all extractable objects (that is, instances of such classes as [IloConstraint](#), [IloNumVar](#), and so forth).

Instances of subclasses of this class represent objects (such as constraints, constrained variables, objectives, and so forth) that can be extracted by Concert Technology from your model for use by your application in Concert Technology algorithms.

Not every algorithm can extract every extractable object of a model. For example, a model may include more than one objective, but you can extract only one objective for an instance of [IloCplex](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

Adding Extractable Objects

Generally, for an extractable object to be taken into account by one of the algorithms in Concert Technology, you must add the extractable object to a model with the member function [IloModel::add](#) and extract the model for the algorithm with the member function [IloAlgorithm::extract](#).

Environment and Extractable Objects

Every extractable object in your model must belong to one instance of [IloEnv](#). An extractable object (that is, an instance of [IloExtractable](#) or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Notification

When you change an extractable object, for example by removing it from a model, Concert Technology notifies algorithms that have extracted the model containing this extractable object about the change. Member functions that carry out such notification are noted in this documentation.

See Also

[IloEnv](#), [IloGetClone](#), [IloModel](#)

Constructors

```
public IloExtractable(IloExtractableI * obj=0)
```

This constructor creates a handle to the implementation object.

Methods

```
public void end()
```

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object.

After a call to this member function, you can not use the invoking extractable object again.

Note: *The member function end notifies Concert Technology algorithms about the destruction of this invoking object.*

```
public IloEnv getEnv() const
```

This member function returns the environment to which the invoking extractable object belongs. An extractable object belongs to exactly one environment; different environments can not share the same extractable object.

```
public IloInt getId() const
```

This member function returns the ID of the invoking extractable object.

```
public IloExtractableI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public const char * getName() const
```

This member function returns a character string indicating the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

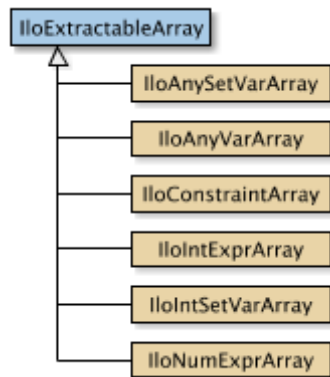
```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterwards. Normally, `obj` contains user data pertinent to the invoking object.

IloExtractableArray

Category Class

InheritancePath



Definition File ilconcert/iloextractable.h

Constructor Summary

public	IloExtractableArray (IloDefaultArrayI * i=0)
public	IloExtractableArray (const IloExtractableArray & r)
public	IloExtractableArray (const IloEnv env, IloInt n=0)

Method Summary

public void	IloExtractableArray::add (IloInt more, const IloExtractable x)
public void	IloExtractableArray::add (const IloExtractable x)
public void	IloExtractableArray::add (const IloExtractableArray x)
public void	IloExtractableArray::endElements ()

Description

An instance of this class is an array of extractable objects (instances of the class [IloExtractable](#) or its subclasses).

Instances of `IloExtractableArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

For information on arrays, see the concept [Arrays](#)

See Also

[IloArray](#), [IloExtractable](#), [operator](#)

Constructors

```
public IloExtractableArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloExtractableArray(const IloExtractableArray & r)
```

This copy constructor creates a handle to the array of extractable objects indicated by `r`.

```
public IloExtractableArray(const IloEnv env,  
                          IloInt n=0)
```

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

```
public void add(IloInt more,  
               const IloExtractable x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloExtractable x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloExtractableArray x)
```

This member function appends the elements in the array `x` to the invoking array.

```
public void endElements()
```

This member function calls [IloExtractable::end](#) for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

IloFastMutex

Category Class

InheritancePath



Definition File ilconcert/ilothread.h

Constructor Summary	
public	IloFastMutex()

Method Summary	
public int	isLocked()
public void	IloFastMutex::lock()
public void	IloFastMutex::unlock()

Description

The class `IloFastMutex` provides synchronization primitives adapted to the needs of Concert Technology. In particular, an instance of the class `IloFastMutex` is a non-recursive mutex that implements mutual exclusion from critical sections of code in multithreaded applications. The purpose of a mutex is to guarantee that concurrent calls to a critical section of code in a multithreaded application are serialized. If a critical section of code is protected by a mutex, then two (or more) threads cannot execute the critical section simultaneously. That is, an instance of this class makes it possible for you to serialize potentially concurrent calls.

Concert Technology implements a mutex by using a single resource that you lock when your application enters the critical section and that you unlock when you leave. Only one thread can own that resource at a given time.

See `ILOUSEMT` for details about the compilation macro to use with instances of this class.

Protection by a Mutex

A critical section of code in a multithreaded application is protected by a mutex when that section of code is encapsulated by a pair of calls to the member functions `IloFastMutex::lock` and `IloFastMutex::unlock`.

In fact, we say that a pair of calls to the member functions `lock` and `unlock` defines a critical section. The conventional way of defining a critical section looks like this:

```
mutex.lock();
while (conditionC does not hold)
    condition.wait(&mutex);
doTreatmentT();
mutex.unlock();
```

The class `IloCondition` provides synchronization primitives to express conditions in critical sections of code.

State of a Mutex

A mutex (an instance of `IloFastMutex`) has a state; the state may be locked or unlocked. You can inquire about the state of a mutex to determine whether it is locked or unlocked by using the member function `isLocked`. When a thread enters a critical section of code in a multithreaded application and then locks the mutex defining that critical section, we say that the thread owns that lock and that lock belongs to the thread until the thread unlocks the mutex.

Exceptions

The member functions `IloFastMutex::lock` and `IloFastMutex::unlock` can throw C++ exceptions when exceptions are enabled on platforms that support them. These are the possible exceptions:

- ◆ **IloMutexDeadlock**: Instances of `IloFastMutex` are not recursive. Consequently, if a thread locks a mutex and then attempts to lock that mutex again, the member function `lock` throws the exception `MutexDeadlock`. On platforms that do not support exceptions, it causes the application to exit.
- ◆ **IloMutexNotOwner**: The thread that releases a given lock (that is, the thread that unlocks a mutex) must be the same thread that locked the mutex in the first place. For example, if a thread A takes lock L and thread B attempts to unlock L, then the member function `unlock` throws the exception `MutexNotOwner`. On platforms that do not support exceptions, it causes the application to exit.
- ◆ **IloMutexNotOwner**: The member function `unlock` throws this exception whenever a thread attempts to unlock an instance of `IloFastMutex` that is not already locked. On platforms that do not support exceptions, it causes the application to exit.

System Class: Memory Management

`IloFastMutex` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in internal data structures of Concert Technology are managed by an instance of `IloEnv`.

However, system classes, such as `IloFastMutex`, differ from that pattern. `IloFastMutex` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloFastMutex` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloFastMutex` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloFastMutex` in the Concert Technology environment because the destructor for that instance of `IloFastMutex` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in the Concert Technology environment. In other words, allocation of any instance of `IloFastMutex` in the Concert Technology environment will produce memory leaks.

For example, it is not a good idea to make an instance of `IloFastMutex` part of a conventional Concert Technology model allocated in the Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of `IloFastMutex`

Instances of `IloFastMutex` differ from the usual Concert Technology objects because they are not allocated in the Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloFastMutex` by calling the delete operator when your application no longer needs those objects.

See Also [IloBarrier](#), [IloCondition](#), `ILOUSEMT`

Constructors `public IloFastMutex()`

This constructor creates an instance of `IloFastMutex` and allocates it on the C++ heap (not in the Concert Technology environment). This mutex contains operating system-specific resources to represent a lock. You may use this mutex for purposes that are private to a process. Its behavior is undefined for inter-process locking.

Methods `public int isLocked()`

This member function returns a Boolean value that indicates the state of the invoking mutex. That is, it tells you whether the mutex is locked by the calling thread (0) or unlocked (1) or locked by a thread other than the calling thread (also 1).

```
public void lock()
```

This member function acquires a lock for the invoking mutex on behalf of the calling thread. That lock belongs to the calling thread until the member function `unlock` is called.

If you call this member function and the invoking mutex has already been locked, then the calling thread is suspended until the first lock is released.

```
public void unlock()
```

This member function releases the lock on the invoking mutex, if there is such a lock.

If you call this member function on a mutex that has not been locked, then this member function throws an exception if C++ exceptions have been enabled on a platform that supports exceptions. Otherwise, it causes the application to exit.

IloFloatArray

Category Macro

Synopsis `IloFloatArray()`

Description `IloFloatArray` is the array class of the basic floating-point class for a model. It is a handle class. The implementation class for `IloFloatArray` is the undocumented class `IloFloatArrayI`.

Instances of `IloFloatArray` are extensible. (They differ from instances of `IlcFloatArray` in this respect. `IlcFloatArray` is documented in the *ILOG Solver Reference Manual*.)

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated on the Concert Technology heap associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where “automatic” has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloFloatArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

See Also [IloNum](#)

IloFloatVar

Category Macro

Synopsis `IloFloatVar()`

Description An instance of this class represents a constrained floating-point variable in Concert Technology.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also [IloFloatVarArray](#), [IloNumVar](#)

IloFloatVarArray

Category	Macro
Synopsis	IloFloatVarArray()
Summary	The array class of IloFloatVar.
Description	For each basic type, Concert Technology defines a corresponding array class. IloFloatVarArray is the array class of the floating-point variable class for a model. It is a handle class. Instances of IloFloatVarArray are extensible. Most member functions in this class contain <code>assert</code> statements. For an explanation of the macro <code>NDEBUG</code> (a way to turn on or turn off these <code>assert</code> statements), see the concept <code>Assert and NDEBUG</code>.
See Also	IloFloatVar

IloFloor

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<code>public IloNum IloFloor(IloNum val)</code>
Description	This function computes the largest integer value not greater than <code>val</code> .

Examples:

```
IloFloor(IloInfinity) is IloInfinity.  
IloFloor(-IloInfinity) is -IloInfinity.  
IloFloor(0) is 0.  
IloFloor(0.4) is 0.  
IloFloor(-0.4) is -1.  
IloFloor(0.5) is 0.  
IloFloor(-0.5) is -1.  
IloFloor(0.6) is 0.  
IloFloor(-0.6) is -1.
```

IloFunction

Category Class

InheritancePath

`IloFunction< X,Y>`

Definition File `ilconcert/iloset.h`

Description Concert Technology offers you the means to define classes of functions that map instances of one class `X` to instances of another class `Y`.

This C++ template creates a class of handles to the implementation class built by the template `IloFunctionI`.

Normally, you subclass the class `IloFunctionI<X, Y>` and in doing so, you define its pure virtual member function:

```
virtual Y getValue(X);
```

Then you use this template to define a handle to that class of type `IloFunction<X, Y>`.

Definition file: `<ilconcert/iloset.h>`

IloGetClone

Category	Global Function
Definition File	ilconcert/iloextractable.h
Synopsis	<pre>public X IloGetClone(const IloEnv env, const X x)</pre>
Description	This C++ template creates a clone (that is, an exact copy) of an instance of the class X.

IloHalfPi

Category Macro

Synopsis **IloHalfPi()**

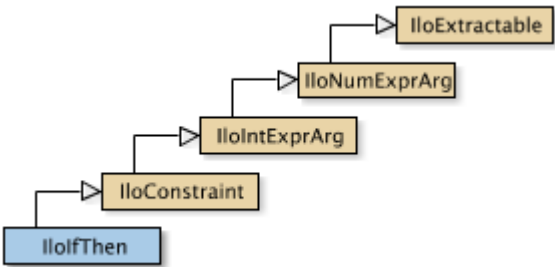
Description Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloHalfPi;          // = 1.57079632679489661923
```

IloIfThen

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	<code>IloIfThen()</code>
public	<code>IloIfThen(IloIfThenI * impl)</code>
public	<code>IloIfThen(const IloEnv env, const IloConstraint left, const IloConstraint right, const char * name=0)</code>

Method Summary	
public IloIfThenI *	<code>getImpl() const</code>

Inherited methods from IloConstraint	
<code>IloConstraint::getImpl</code>	

Inherited methods from [IloIntExprArg](#)

[IloIntExprArg::getImpl](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of `IloIfThen` represents a condition constraint. Generally, a condition constraint is composed of an if part (the conditional statement or left side) and a then part (the consequence or right side).

In order for a constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also

[IloConstraint](#)

Constructors

```
public IloIfThen()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloIfThen(IloIfThenI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIfThen(const IloEnv env,  
                const IloConstraint left,  
                const IloConstraint right,  
                const char * name=0)
```

This constructor creates a condition constraint in the environment indicated by `env`. The parameter `left` indicates the if part of the condition. The parameter `right` indicates the then part of the condition. The string `name` indicates the name of the constraint; it is set to `0` by default. For the constraint to take effect, you must add it to a model and extract the model for an algorithm.

Methods

```
public IloIfThenI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloInitMT

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<pre>public void IloInitMT() public void IloInitMT(IloBaseEnvMutex *)</pre>
Description	This function initializes multithreading in a Concert Technology application.

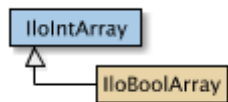
IloInt

Category	Type Definition
Definition File	ilconcert/ilosys.h
Synopsis	long IloInt
Description	This type definition represents signed integers in Concert Technology.
See Also	IloBool , IloModel , IloNum

IloIntArray

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Constructor Summary

public	IloIntArray (IloArrayI * i=0)
public	IloIntArray (const IloEnv env, IloInt n=0)
public	IloIntArray (const IloEnv env, IloInt n, IloInt v0, IloInt v1...)

Method Summary

public IloBool	IloIntArray::contains (IloIntArray ax) const
public IloBool	IloIntArray::contains (IloInt value) const
public void	IloIntArray::discard (IloIntArray ax)
public void	IloIntArray::discard (IloInt value)
public IloIntExprArg	IloIntArray::operator() (IloIntExprArg intExp) const
public IloIntExprArg	IloIntArray::operator[] (IloIntExprArg intExp) const
public IloInt &	IloIntArray::operator[] (IloInt i)
public const IloInt &	IloIntArray::operator[] (IloInt i) const
public IloNumArray	IloIntArray::toNumArray () const

Description [IloIntArray](#) is the array class of the basic integer class for a model. It is a handle class. The implementation class for [IloIntArray](#) is the undocumented class [IloIntArrayI](#).

Instances of `IloIntArray` are extensible. (They differ from instances of `IlcIntArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where “automatic” has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloIntArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

`IloIntArray` inherits additional methods from the template `IloArray`:

- ◆ [add](#)
- ◆ [add](#)
- ◆ [clear](#)
- ◆ [getEnv](#)
- ◆ [getSize](#)
- ◆ [remove](#)
- ◆ [operator\[\]](#)
- ◆ [operator\[\]](#)

See Also [IloInt](#)

Constructors `public IloIntArray(IloArrayI * i=0)`

This constructor creates an array of integers from an implementation object.

```
public IloIntArray(const IloEnv env,
                  IloInt n=0)
```

This constructor creates an array of n integers for use in a model in the environment indicated by `env`. By default, its elements are empty handles.

```
public IloIntArray(const IloEnv env,
                  IloInt n,
                  IloInt v0,
                  IloInt v1...)

```

This constructor creates an array of n integers; the elements of the new array take the corresponding values: $v_0, v_1, \dots, v_{(n-1)}$.

Methods

```
public IloBool contains(IloIntArray ax) const

```

This member function checks whether the value is contained or not.

```
public IloBool contains(IloInt value) const

```

This member function checks whether the value is contained or not.

```
public void discard(IloIntArray ax)

```

This member function removes elements from the invoking array. It removes the array `ax`.

```
public void discard(IloInt value)

```

This member function removes elements from the invoking array. It removes the element.

```
public IloIntExprArg operator()(IloIntExprArg intExp) const

```

This subscripting operator is deprecated. It must be replaced by the operator `IloIntExprArg operator[] (IloIntExprArg intExp) const`;

```
public IloIntExprArg operator[](IloIntExprArg intExp) const

```

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call A the invoking array. When `intExp` is bound to the value i , then the domain of the expression is the domain of $A[i]$. More generally, the domain of the expression is the union of the domains of the expressions $A[i]$ where the i are in the domain of `intExp`.

This operator is also known as an element constraint.

```
public IloInt & operator[](IloInt i)

```

This operator returns a reference to the object located in the invoking array at the position indicated by the index i .

```
public const IloInt & operator[](IloInt i) const

```

This operator returns a reference to the object located in the invoking array at the position indicated by the index i . Concert Technology uses the `CONST` operator

```
IloArray operator[] (IloInt i) const;
```

on `const` arrays.

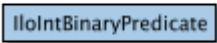
```
public IloNumArray toNumArray() const
```

This constructor creates an array of integers from an array of numeric values.

IloIntBinaryPredicate

Category Class

InheritancePath



Definition File ilconcert/ilotupleset.h

Constructor Summary	
public	IloIntBinaryPredicate()
public	IloIntBinaryPredicate(IloIntBinaryPredicateI *impl)

Method Summary	
public IloIntBinaryPredicateI *	getImpl() const
public IloBool	isTrue (const IloInt val1,const IloInt val2)
public void	operator= (const IloIntBinaryPredicate & h)

Description This class makes it possible for you to define binary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as [isTrue](#)) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, [IloIntBinaryPredicate](#), and an implementation class, such as [IloIntBinaryPredicateI](#), where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of [IloIntBinaryPredicateI](#) allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloIntBinaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `isTrue` must check) must be two.

See Also [IloTableConstraint](#)

Constructors

```
public IloIntBinaryPredicate()
```

This constructor creates an empty binary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Int attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

```
public IloIntBinaryPredicate(IloIntBinaryPredicateI * impl)
```

This constructor creates a handle object (an instance of the class `IloIntBinaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IloIntPredicateI`, documented in the *ILOG Solver Reference Manual*).

Methods

```
public IloIntBinaryPredicateI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool isTrue(const IloInt val1,
                    const IloInt val2)
```

This member function returns `IloTrue` if the values `val1` and `val2` make the invoking binary predicate valid. It returns `IloFalse` otherwise.

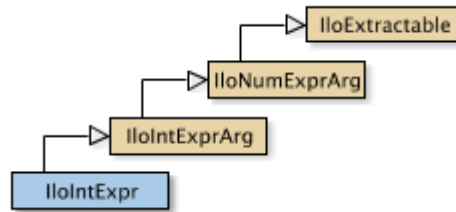
```
public void operator=(const IloIntBinaryPredicate & h)
```

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

IloIntExpr

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The class of integer expressions in Concert Technology 2.0.

Constructor Summary

public	IloIntExpr()
public	IloIntExpr(IloIntExprI * impl)
public	IloIntExpr(const IloIntExprArg arg)
public	IloIntExpr(const IloIntLinExprTerm term)
public	IloIntExpr(const IloEnv env, IloInt n=0)

Method Summary

public IloIntExprI *	getImpl() const
public IloIntExpr &	operator *=(IloInt val)
public IloIntExpr &	operator +=(const IloIntExprArg expr)
public IloIntExpr &	operator +=(IloInt val)
public IloIntExpr &	operator -=(const IloIntExprArg expr)
public IloIntExpr &	operator -=(IloInt val)

Inherited methods from **IloIntExprArg**

[IloIntExprArg::getImpl](#)

Inherited methods from **IloNumExprArg**

[IloNumExprArg::getImpl](#)

Inherited methods from **IloExtractable**

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

Integer expressions in Concert Technology 2.0 are represented using objects of type **IloIntEpr**.

Constructors

```
public IloIntExpr()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloIntExpr(IloIntExprI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIntExpr(const IloIntExprArg arg)
```

This constructor creates an integer expression using the undocumented class **IloIntExprArg**.

```
public IloIntExpr(const IloIntLinExprTerm term)
```

This constructor creates an integer expression with linear terms using the undocumented class **IloIntLinExprTerm**.

```
public IloIntExpr(const IloEnv env,  
                 IloInt n=0)
```

This constructor creates an integer expression in the environment **env**.

Methods

```
public IloIntExprI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloIntExpr & operator *=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x * \dots$.

```
public IloIntExpr & operator +=(const IloIntExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x + \dots$.

```
public IloIntExpr & operator +=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x + \dots$.

```
public IloIntExpr & operator -=(const IloIntExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x - \dots$.

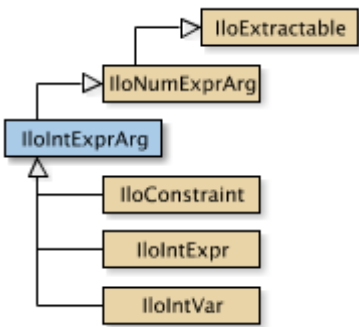
```
public IloIntExpr & operator -=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than $x = x - \dots$.

IloIntExprArg

Category Class

InheritancePath



Definition File `ilconcert/iloexpression.h`

Summary A class used internally in Concert Technology.

Constructor Summary	
public	IloIntExprArg()
public	IloIntExprArg(IloIntExprI * impl)

Method Summary	
public IloIntExprI *	IloIntExprArg::getImpl() const

Inherited methods from IloNumExprArg	
IloNumExprArg::getImpl	

Inherited methods from **IloExtractable**

```
IloExtractable::end, IloExtractable::getEnv, IloExtractable::getId,
IloExtractable::getImpl, IloExtractable::getName,
IloExtractable::getObject, IloExtractable::setName,
IloExtractable::setObject
```

Description Concert Technology uses instances of these classes internally as temporary objects when it is parsing a C++ expression in order to build an instance of **IloIntExpr**. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as **IloSum** or **IloScaleProd**, can be an instance of this class.

Constructors `public IloIntExprArg()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloIntExprArg(IloIntExprI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

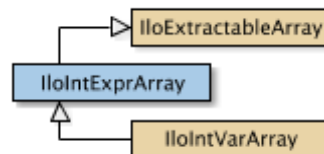
Methods `public IloIntExprI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

IloIntExprArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The array class of IloIntExpr.

Constructor Summary

public	IloIntExprArray (IloDefaultArrayI * i=0)
public	IloIntExprArray (const IloEnv env, IloInt n=0)

Method Summary

public void	IloIntExprArray::add (IloInt more, const IloIntExpr x)
public void	IloIntExprArray::add (const IloIntExpr x)
public void	IloIntExprArray::add (const IloIntExprArray array)
public void	IloIntExprArray::endElements ()
public IloIntExprArg	IloIntExprArray::operator ()(IloIntExprArg anIntegerExpr) const
public IloIntExprArg	IloIntExprArray::operator [](IloIntExprArg anIntegerExpr) const
public IloIntExpr	IloIntExprArray::operator [](IloInt i) const
public IloIntExpr &	IloIntExprArray::operator [](IloInt i)

Inherited methods from **IloExtractableArray**

`IloExtractableArray::add`, `IloExtractableArray::add`,
`IloExtractableArray::add`, `IloExtractableArray::endElements`

Description

For each basic type, Concert Technology defines a corresponding array class. `IloNumExprArray` is the array class of the integer expressions class (`IloNumExpr`) for a model.

Instances of `IloIntExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructors

```
public IloIntExprArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloIntExprArray(const IloEnv env,  
                      IloInt n=0)
```

This copy constructor creates a handle to the array of extractable objects indicated by copy.

Methods

```
public void add(IloInt more,  
               const IloIntExpr x)
```

This member function appends x to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloIntExpr x)
```

This member function appends x to the invoking array.

```
public void add(const IloIntExprArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloIntExprArg operator()(IloIntExprArg anIntegerExpr) const
```

This subscripting operator is deprecated. It must be replaced by the operator `IloIntExprArg operator[] (IloIntExprArg anIntegerExpr) const`;

```
public IloIntExprArg operator[] (IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call *A* the invoking array. When `anIntegerExpr` is bound to the value *i*, the domain of the expression is the domain of *A[i]*. More generally, the domain of the expression is the union of the domains of the expressions *A[i]* where the *i* are in the domain of `anIntegerExpr`.

This operator is also known as an element constraint.

```
public IloIntExpr operator[] (IloInt i) const
```

This operator returns a reference to the extractable object located in the invoking array at the position indicated by the index *i*. Concert Technology uses the `const` operator

```
IloIntExpr operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloIntExpr & operator[] (IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position indicated by the index *i*.

IloIntSet

Category Class

InheritancePath

IloIntSet

Definition File ilconcert/iloset.h

Constructor Summary

public	IloIntSet (const IloEnv env, const IloIntArray array, IloBool withIndex= IloFalse)
public	IloIntSet (const IloEnv env, const IloNumArray array, IloBool withIndex= IloFalse)
public	IloIntSet (const IloEnv env, IloBool withIndex= IloFalse)
public	IloIntSet (IloIntSetI * impl=0)

Method Summary

public void	add (IloIntSet set)
public void	add (IloInt elt)
public IloBool	contains (IloIntSet set) const
public IloBool	contains (IloInt elt) const
public void	empty ()
public IloIntSetI *	getImpl () const
public IloInt	getNext (IloInt value, IloInt n=1) const
public IloInt	getNextC (IloInt value, IloInt n=1) const
public IloInt	getPrevious (IloInt value, IloInt n=1) const
public IloInt	getPreviousC (IloInt value, IloInt n=1) const
public IloInt	getSize () const
public IloBool	intersects (IloIntSet set) const
public void	remove (IloIntSet set)
public void	remove (IloInt elt)
public void	setIntersection (IloIntSet set)
public void	setIntersection (IloInt elt)

Inner Class	
IloIntSet::IloIntSet::Iterat or	

Description An instance of this class offers a convenient way to represent a set of integer values as a constrained variable in Concert Technology.

A constrained variable representing a set of integer values (that is, an instance of [IloIntSetVar](#)) is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set. This class offers member functions for accessing the required and possible elements of a set of integer values.

The function [IloCard](#) offers you a way to constrain the number of elements in a set variable. That is, [IloCard](#) constrains the cardinality of a set variable.

See Also [IloCard](#), [IloEqIntersection](#), [IloEqUnion](#), [IloExtractable](#), [IloMember](#), [IloModel](#), [IloNotMember](#), [IloNullIntersect](#), [IloIntSetVarArray](#), [IloSubset](#), [IloSubsetEq](#)

Constructors

```
public IloIntSet(const IloEnv env,
                const IloIntArray array,
                IloBool withIndex=IloFalse)
```

This constructor creates a set of integer values in the environment `env` from the elements in `array`.

```
public IloIntSet(const IloEnv env,
                const IloNumArray array,
                IloBool withIndex=IloFalse)
```

This constructor creates a set of numeric values in the environment `env` from the elements in `array`.

```
public IloIntSet(const IloEnv env,
                IloBool withIndex=IloFalse)
```

This constructor creates an empty set (no elements) in the environment `env`. You must use the member function `IloIntSet::add` to fill this set with elements.

```
public IloIntSet(IloIntSetI * impl=0)
```

This constructor creates a handle to a set of integer values from its implementation object.

Methods

```
public void add(IloIntSet set)
```

This member function adds `set` to the invoking set. Here, "adds" means that the invoking set becomes the union of its former elements and the elements of `set`.

To calculate the arithmetic sum of values in an array, use the function `IloSum`.

```
public void add(IloInt elt)
```

This member function adds `elt` to the invoking set. Here, "adds" means that the invoking set becomes the union of its former elements and the new `elt`.

```
public IloBool contains(IloIntSet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` contains the invoking set. The value one indicates that the invoking set contains all the elements of `set`, and that the intersection of the invoking set with `set` is precisely `set`. The value zero indicates that the intersection of the invoking set and `set` is not precisely `set`.

```
public IloBool contains(IloInt elt) const
```

This member function returns a Boolean value (zero or one) that indicates whether `elt` is an element of the invoking set. The value one indicates that the invoking set contains `elt`; the value zero indicates that the invoking set does not contain `elt`.

```
public void empty()
```

This member function removes the elements from the invoking set. In other words, the invoking set becomes the empty set.

```
public IloIntSetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking set.

```
public IloInt getNext(IloInt value,  
                    IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it throws an exception

See also `getNextC`, `getPreviousC` for circular search.

```
public IloInt getNextC(IloInt value,  
                    IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it will give you the first value (circular search)

See also `getNext`, `getPrevious`.


```
public IloInt getPrevious(IloInt value,
                        IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it throws an exception

See also getNextC, getPreviousC for circular search.

```
public IloInt getPreviousC(IloInt value,
                        IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it will give you the last value (circular search)

See also getNext, getPrevious.

```
public IloInt getSize() const
```

This member function returns an integer indicating the size of the invoking set (that is, how many elements it contains).

```
public IloBool intersects(IloIntSet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` intersects the invoking set. The value one indicates that the intersection of `set` and the invoking set is not empty (at least one element in common); the value zero indicates that the intersection of `set` and the invoking set is empty (no elements in common).

```
public void remove(IloIntSet set)
```

This member function removes all the elements of `set` from the invoking set.

```
public void remove(IloInt elt)
```

This member function removes `elt` from the invoking set.

```
public void setIntersection(IloIntSet set)
```

This member function changes the invoking set so that it includes only the elements of `set`. In other words, the invoking set becomes the intersection of its former elements with the elements of `set`.

```
public void setIntersection(IloInt elt)
```

This member function changes the invoking set so that it includes only the element indicated by `elt`. In other words, the invoking set becomes the intersection of its former elements with `elt`.

IloIntSet::Iterator

Category Inner Class

InheritancePath

Definition File ilconcert/iloset.h

Constructor Summary	
public	Iterator (const IloIntSetI * coll)
public	Iterator (const IloIntSet coll)
public	Iterator (IloGenAlloc * heap, const IloIntSetI * coll)

Description An instance of the nested class `IloIntSet::Iterator` is an iterator that traverses the elements of a finite set of numeric values (an instance of [IloIntSet](#)).

See Also [IloIntSet](#)

Constructors

```

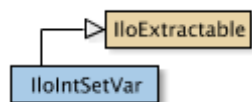
public Iterator(const IloIntSetI * coll)
.
public Iterator(const IloIntSet coll)
.
public Iterator(IloGenAlloc * heap,
               const IloIntSetI * coll)
.

```

IloIntSetVar

Category Class

InheritancePath



Definition File ilconcert/iloset.h

Summary The class IloIntSetVar.

Constructor Summary	
public	<code>IloIntSetVar()</code>
public	<code>IloIntSetVar(IloIntSetVarI * impl)</code>
public	<code>IloIntSetVar(const IloEnv env, const IloIntArray array, const char * name=0)</code>
public	<code>IloIntSetVar(const IloEnv env, const IloIntArray possible, const IloIntArray required, const char * name=0)</code>
public	<code>IloIntSetVar(const IloEnv env, const IloNumArray array, const char * name=0)</code>
public	<code>IloIntSetVar(const IloEnv env, const IloNumArray possible, const IloNumArray required, const char * name=0)</code>
public	<code>IloIntSetVar(const IloIntCollection possible, const char * name=0)</code>
public	<code>IloIntSetVar(const IloIntCollection possible, const IloIntCollection required, const char * name=0)</code>
public	<code>IloIntSetVar(const IloNumCollection possible, const char * name=0)</code>
public	<code>IloIntSetVar(const IloNumCollection possible, const IloNumCollection required, const char * name=0)</code>

Method Summary	
public void	addPossible(IloInt elt) const
public void	addRequired(IloInt elt) const
public IloIntSetVarI *	getImpl() const
public void	getPossibleSet(IloIntSet set) const
public IloIntSet	getPossibleSet() const
public void	getRequiredSet(IloIntSet set) const
public IloIntSet	getRequiredSet() const
public void	removePossible(IloInt elt) const
public void	removeRequired(IloInt elt) const

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description

An instance of this class represents a set of integer values. The same integer value will not appear more than once in a set. The elements of a set are not ordered.

A constrained variable representing a set of integer values (that is, an instance of [IloIntSetVar](#)) is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set. This class offers member functions for accessing the required and possible elements of a set of integer values.

The function [IloCard](#) offers you a way to constrain the number of elements in a set variable. That is, [IloCard](#) constrains the cardinality of a set variable.

Constructors

```
public IloIntSetVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloIntSetVar(IloIntSetVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIntSetVar(const IloEnv env,  
                  const IloIntArray array,  
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. By default, its name is indicated by the empty string, but you can indicate a `name` of your choice.

```
public IloIntSetVar(const IloEnv env,
                  const IloIntArray possible,
                  const IloIntArray required,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. The array `possible` indicates the set of possible elements of the set variable; the array `required` specifies the set of required elements of the set variable. By default, its name is indicated by the empty string, but you can indicate a `name` of your choice.

```
public IloIntSetVar(const IloEnv env,
                  const IloNumArray array,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. By default, its name is indicated by the empty string, but you can indicate a `name` of your choice.

```
public IloIntSetVar(const IloEnv env,
                  const IloNumArray possible,
                  const IloNumArray required,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. The numeric array `possible` indicates the set of possible elements of the set variable; the numeric array `required` specifies the set of required elements of the set variable. By default, its name is indicated by the empty string, but you can indicate a `name` of your choice.

```
public IloIntSetVar(const IloIntCollection possible,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

```
public IloIntSetVar(const IloIntCollection possible,
                  const IloIntCollection required,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

```
public IloIntSetVar(const IloNumCollection possible,
                  const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

```
public IloIntSetVar(const IloNumCollection possible,
                   const IloNumCollection required,
                   const char * name=0)
```

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

Methods

```
public void addPossible(IloInt elt) const
```

This member function adds `elt` to the set of possible elements of the invoking set variable.

Note: *The member function `addPossible` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void addRequired(IloInt elt) const
```

This member function adds `elt` to the set of required elements of the invoking set variable.

Note: *The member function `addRequired` notifies Concert Technology algorithms about this change of this invoking object.*

```
public IloIntSetVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getPossibleSet(IloIntSet set) const
```

This member function accesses the possible elements of the invoking set variable and puts those elements into its parameter `set`.

```
public IloIntSet getPossibleSet() const
```

This member function returns the possible elements of the invoking set variable.

```
public void getRequiredSet(IloIntSet set) const
```

This member function accesses the possible elements of the invoking set variable and puts those elements into its parameter `set`.

```
public IloIntSet getRequiredSet() const
```

This member function returns the required elements of the invoking set variable.

```
public void removePossible(IloInt elt) const
```

This member function removes `elt` as a possible element of the invoking set variable.

Note: *The member function `removePossible` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void removeRequired(IloInt elt) const
```

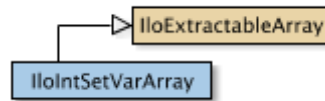
This member function removes `elt` as a required element of the invoking set variable.

Note: *The member function `removeRequired` notifies Concert Technology algorithms about this change of this invoking object.*

IloIntSetVarArray

Category Class

InheritancePath



Definition File ilconcert/iloset.h

Summary The array class of the set variable class for integer values.

Constructor Summary

public	IloIntSetVarArray (IloDefaultArrayI * i=0)
public	IloIntSetVarArray (const IloEnv env, IloInt n=0)

Method Summary

public void	add (IloInt more, const IloIntSetVar x)
public void	add (const IloIntSetVar x)
public void	add (const IloIntSetVarArray array)
public IloIntSetVar	operator[] (IloInt i) const
public IloIntSetVar &	operator[] (IloInt i)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description For each basic type, Concert Technology defines a corresponding array class. [IloIntSetVarArray](#) is the array class of the set variable class for integer values ([IloIntSetVar](#)) in a model.

Instances of `IloIntSetVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Constructors

```
public IloIntSetVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of set variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloIntSetVarArray(const IloEnv env,  
                        IloInt n=0)
```

This constructor creates an extensible array of `n` set variables, where each set is a set of integer values.

Methods

```
public void add(IloInt more,  
               const IloIntSetVar x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloIntSetVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloIntSetVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloIntSetVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `CONST` operator

```
IloIntSetVar operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloIntSetVar & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

IloIntTernaryPredicate

Category Class

InheritancePath



Definition File ilconcert/ilotupleset.h

Constructor Summary	
public	IloIntTernaryPredicate()
public	IloIntTernaryPredicate(IloIntTernaryPredicate I * impl)

Method Summary	
public IloIntTernaryPredicateI *	getImpl() const
public IloBool	isTrue (const IloInt val1,const IloInt val2,const IloInt val3)
public void	operator= (const IloIntTernaryPredicate & h)

Description This class makes it possible for you to define ternary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as [isTrue](#)) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. A ternary predicate checks an ordered set of three objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, [IloIntTernaryPredicate](#), and an implementation class, such as [IloIntTernaryPredicateI](#), where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of [IloIntTernaryPredicateI](#) allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloIntTernaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `isTrue` must check) must be three.

See Also

[IloTableConstraint](#)

Constructors

```
public IloIntTernaryPredicate()
```

This constructor creates an empty ternary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. An attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

```
public IloIntTernaryPredicate(IloIntTernaryPredicateI * impl)
```

This constructor creates a handle object (an instance of the class `IloIntTernaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IloIntTernaryPredicateI`).

Methods

```
public IloIntTernaryPredicateI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool isTrue(const IloInt val1,
                    const IloInt val2,
                    const IloInt val3)
```

This member function returns `IloTrue` if the values `val1`, `val2`, and `val3` make the invoking ternary predicate valid. It returns `IloFalse` otherwise.

```
public void operator=(const IloIntTernaryPredicate & h)
```

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

IloIntTupleSet

Category Class

InheritancePath

IloIntTupleSet

Definition File ilconcert/ilotupleset.h

Constructor Summary

public	IloIntTupleSet (const IloEnv env, const IloInt arity)

Method Summary

public IloBool	add (const IloIntArray tuple) const
public IloIntTupleSetI *	getImpl () const
public IloBool	isIn (const IloIntArray tuple) const
public IloBool	remove (const IloIntArray tuple) const

Description

A tuple is an ordered set of values represented by an array. A *set* of enumerated tuples in a model is represented by an instance of [IloIntTupleSet](#). That is, the elements of a tuple *set* are tuples of enumerated values (such as pointers). The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, [IloIntTupleSet](#) manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function [isIn](#) to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- ◆ When you create the set, you must specify the arity of the tuple-elements it contains.

- ◆ You use the member function `IloIntTupleSet::add` to add tuples to the set. You can add tuples to the set in a model; you cannot add tuples to an instance of this class during a search, nor inside a constraint, nor inside a goal.

Concert Technology will throw an exception (an instance of `IloSolver::SolverErrorException`) if you attempt:

- ◆ to add a tuple with a different number of variables from the arity of the set;
- ◆ to search for a tuple with an arity different from the set arity.

You do not have to worry about memory allocation. If you respect these conventions, Concert Technology manages allocation and de-allocation transparently for you.

See `IlcIntTupleSet` in the *ILOG Solver Reference Manual*.

See Also

[IloIntTupleSetIterator](#), [IloTableConstraint](#), [IloExtractable](#)

Constructors

```
public IloIntTupleSet(const IloEnv env,
                    const IloInt arity)
```

This constructor creates a set of tuples (an instance of the class `IloIntTupleSet`) with the arity indicated by `arity`.

Methods

```
public IloBool add(const IloIntArray tuple) const
```

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set. You may use this member function to add tuples to the invoking set in a model; you may not add tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *ILOG Solver Reference Manual*.

```
public IloIntTupleSetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public IloBool isIn(const IloIntArray tuple) const
```

This member function returns `IloTrue` if `tuple` belongs to the invoking set. Otherwise, it returns `IloFalse`. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set.

```
public IloBool remove(const IloIntArray tuple) const
```

This member function removes `tuple` from the invoking set in a model. You may use this member function to remove tuples from the invoking set in a model; you may not

remove tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *ILOG Solver Reference Manual*.

IloIntTupleSetIterator

Category Class

InheritancePath



Definition File ilconcert/ilotupleset.h

Constructor Summary	
public	IloIntTupleSetIterator (const IloEnv env, IloIntTupleSet tset)

Method Summary	
public IloIntArray	operator * () const

Description An instance of the class `IloIntTupleSetIterator` is an iterator that traverses the elements of a finite set of tuples of enumerated values (instance of [IloIntTupleSet](#)).

See `IlcIntTupleSet` in the *ILOG Solver Reference Manual*.

Constructors public `IloIntTupleSetIterator`(const [IloEnv](#) env, [IloIntTupleSet](#) tset)

This constructor creates an iterator associated with `tSet` to traverse its elements.

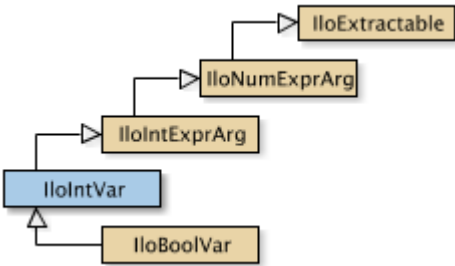
Methods public [IloIntArray](#) `operator *`() const

This operator returns the current element, the one to which the invoking iterator points.

IloIntVar

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary	
public	IloIntVar()
public	IloIntVar(IloNumVarI * impl)
public	IloIntVar(IloEnv env, IloInt vmin=0, IloInt vmax=IloIntMax, const char * name=0)
public	IloIntVar(const IloAddNumVar & var, IloInt lowerBound=0, IloInt upperBound=IloIntMax, const char * name=0)
public	IloIntVar(const IloEnv env, const IloIntArray values, const char * name=0)
public	IloIntVar(const IloAddNumVar & var, const IloIntArray values, const char * name=0)
public	IloIntVar(const IloNumVar var)
public	IloIntVar(const IloIntCollection coll, const char * name=0)
public	IloIntVar(const IloIntRange coll, const char * name=0)

Method Summary	
public IloNumVarI *	IloIntVar::getImpl() const

public IloNum	IloIntVar::getLB() const
public IloInt	IloIntVar::getMax() const
public IloInt	IloIntVar::getMin() const
public IloNum	IloIntVar::getUB() const
public void	IloIntVar::setBounds(IloInt lb, IloInt ub) const
public void	IloIntVar::setLB(IloNum min) const
public void	IloIntVar::setMax(IloInt max) const
public void	IloIntVar::setMin(IloInt min) const
public void	IloIntVar::setPossibleValues(const IloIntArray values) const
public void	IloIntVar::setUB(IloNum max) const

Inherited methods from IloIntExprArg

IloIntExprArg::getImpl

Inherited methods from IloNumExprArg

IloNumExprArg::getImpl

Inherited methods from IloExtractable

IloExtractable::end, IloExtractable::getEnv, IloExtractable::getId,
IloExtractable::getImpl, IloExtractable::getName,
IloExtractable::getObject, IloExtractable::setName,
IloExtractable::setObject

Description

An instance of this class represents a constrained integer variable in a Concert Technology model. If you are looking for a class of numeric variables that may assume integer values and may be relaxed to assume floating-point values, then consider the class [IloNumVar](#). If you are looking for a class of binary decision variables (that is, variables that assume only the values 0 (zero) or 1 (one), then consider the class [IloBoolVar](#).

Bounds of an Integer Variable

The lower and upper bound of an instance of this class is an integer.

What Is Extracted

An instance of `IloIntVar` is extracted by `IloSolver` (documented in the *ILOG Solver Reference Manual*) as an instance of `IlcIntVar` (also documented in the *ILOG Solver Reference Manual*).

An instance of `IloIntVar` is extracted by `IloCplex` as a column representing a numeric variable of type `Int` with bounds as indicated by `IloIntVar`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

Note: When numerical bounds are given to an integer variable (an `IloIntVar` or an `IloNumVar` with `Type = Int`) in the constructors of via a modifier (`setUB`, `setLB`, `setBounds`), they are inward rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

See Also

[IloBoolVar](#), [IloNumVar](#)

Constructors

```
public IloIntVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloIntVar(IloNumVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIntVar(IloEnv env,
                IloInt vmin=0,
                IloInt vmax=IloIntMax,
                const char * name=0)
```

This constructor creates an instance of `IloIntVar` like this:

```
IloNumVar(env, vmin, vmax, ILOINT, name);
```

```
public IloIntVar(const IloAddNumVar & var,
                IloInt lowerBound=0,
                IloInt upperBound=IloIntMax,
                const char * name=0)
```

This constructor creates an instance of `IloIntVar` like this:

```
IloNumVar(column, lowerBound, upperBound, ILOINT, name);
```

```
public IloIntVar(const IloEnv env,
```

```
const IloIntArray values,
const char * name=0)
```

This constructor calls upon its corresponding `IloNumVar` constructor.

```
public IloIntVar(const IloAddNumVar & var,
const IloIntArray values,
const char * name=0)
```

This constructor calls upon its corresponding `IloNumVar` constructor.

```
public IloIntVar(const IloNumVar var)
```

This constructor creates a new handle on `var` if it is of type `ILOINT`. Otherwise, an exception is thrown.

```
public IloIntVar(const IloIntCollection coll,
const char * name=0)
```

This constructor creates an instance of `IloIntVar` from the given collection

*

```
public IloIntVar(const IloIntRange coll,
const char * name=0)
```

This constructor creates an instance of `IloIntVar` from the given collection

*

Methods

```
public IloNumVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking variable.

```
public IloInt getMax() const
```

This member function returns the maximal value of the invoking variable.

```
public IloInt getMin() const
```

This member function returns the minimal value of the invoking variable.

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking variable.

```
public void setBounds(IloInt lb,
IloInt ub) const
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

Note: *The member function `setBounds` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

```
public void setLB(IloNum min) const
```

This member function sets `min` as the lower bound of the invoking variable.

Note: *The member function `setLB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

```
public void setMax(IloInt max) const
```

This member function returns the minimal value of the invoking variable to `max`.

Note: *The member function `setMax` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

```
public void setMin(IloInt min) const
```

This member function returns the minimal value of the invoking variable to `min`.

Note: *The member function `setMin` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

```
public void setPossibleValues(const IloIntArray values) const
```

This member function sets `values` as the domain of the invoking integer variable.

Note: *The member function `setPossibleValues` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

```
public void setUB(IloNum max) const
```

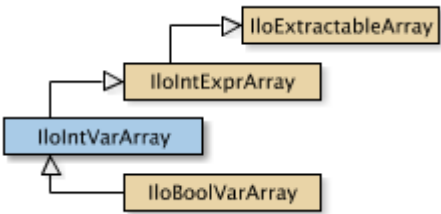
This member function sets `max` as the upper bound of the invoking variable.

Note: *The member function `setUB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.*

IloIntVarArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The array class of IloIntVar.

Constructor Summary	
public	IloIntVarArray (IloDefaultArrayI * i=0)
public	IloIntVarArray (const IloEnv env, IloInt n=0)
public	IloIntVarArray (const IloEnv env,const IloIntArray lb,const IloIntArray ub)
public	IloIntVarArray (const IloEnv env, IloInt lb,const IloIntArray ub)
public	IloIntVarArray (const IloEnv env,const IloIntArray lb, IloInt ub)
public	IloIntVarArray (const IloEnv env, IloInt n, IloInt lb, IloInt ub)
public	IloIntVarArray (const IloEnv env,const IloNumColumnArray columnarray)
public	IloIntVarArray (const IloEnv env,const IloNumColumnArray columnarray,const IloNumArray lb,const IloNumArray ub)

Method Summary	
public void	IloIntVarArray::add (IloInt more,const IloIntVar x)

public void	IloIntVarArray::add (const IloIntVar x)
public void	IloIntVarArray::add (const IloIntVarArray x)
public void	IloIntVarArray::endElements ()
public IloIntExprArg	IloIntVarArray::operator ()(IloIntExprArg anIntegerExpr) const
public IloIntVar	IloIntVarArray::operator [(IloInt i) const
public IloIntVar &	IloIntVarArray::operator [(IloInt i)
public IloIntExprArg	IloIntVarArray::operator [(IloIntExprArg anIntegerExpr) const
public IloNumVarArray	IloIntVarArray::toNumVarArray () const

Inherited methods from [IloIntExprArray](#)

[IloIntExprArray::add](#), [IloIntExprArray::add](#), [IloIntExprArray::add](#),
[IloIntExprArray::endElements](#), [IloIntExprArray::operator](#)(),
[IloIntExprArray::operator](#)[], [IloIntExprArray::operator](#)[],
[IloIntExprArray::operator](#)[]

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description

For each basic type, Concert Technology defines a corresponding array class.

[IloIntVarArray](#) is the array class of the integer variable class for a model. It is a handle class.

Instances of [IloIntVarArray](#) are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloIntVar](#)

Constructors

```
public IloIntVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of integer variables.

```
public IloIntVarArray(const IloEnv env,  
                    IloInt n=0)
```

This constructor creates an extensible array of n integer variables.

```
public IloIntVarArray(const IloEnv env,
                    const IloIntArray lb,
                    const IloIntArray ub)
```

This constructor creates an extensible array of integer variables with lower and upper bounds as indicated.

```
public IloIntVarArray(const IloEnv env,
                    IloInt lb,
                    const IloIntArray ub)
```

This constructor creates an extensible array of integer variables with a lower bound and an array of upper bounds as indicated.

```
public IloIntVarArray(const IloEnv env,
                    const IloIntArray lb,
                    IloInt ub)
```

This constructor creates an extensible array of integer variables with an array of lower bounds and an upper bound as indicated.

```
public IloIntVarArray(const IloEnv env,
                    IloInt n,
                    IloInt lb,
                    IloInt ub)
```

This constructor creates an extensible array of n integer variables, with a lower and an upper bound as indicated.

```
public IloIntVarArray(const IloEnv env,
                    const IloNumColumnArray columnarray)
```

This constructor creates an extensible array of integer variables from a column array.

```
public IloIntVarArray(const IloEnv env,
                    const IloNumColumnArray columnarray,
                    const IloNumArray lb,
                    const IloNumArray ub)
```

This constructor creates an extensible array of integer variables with lower and upper bounds as indicated from a column array.

Methods

```
public void add(IloInt more,
               const IloIntVar x)
```

This member function appends x to the invoking array of integer variables; it appends x `more` times.

```
public void add(const IloIntVar x)
```

This member function appends the value x to the invoking array.

```
public void add(const IloIntVarArray x)
```


This member function appends the variables in the array `x` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloIntExprArg operator()(IloIntExprArg anIntegerExpr) const
```

This subscripting operator is deprecated. It must be replaced by the operator `IloIntExprArg operator[] (IloIntExprArg anIntegerExpr) const`;

```
public IloIntVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `const` operator

```
IloIntVar operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloIntVar & operator[](IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position indicated by the index `i`.

```
public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element constraint.

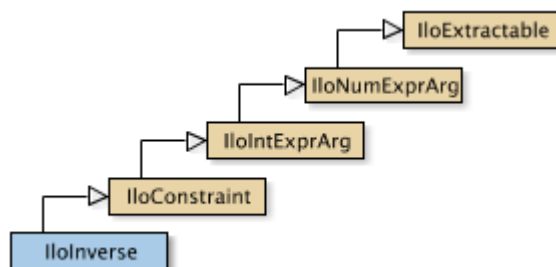
```
public IloNumVarArray toNumVarArray() const
```

This member function copies the invoking array into a new `IloNumVarArray`.

IloInverse

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary

public	IloInverse()
public	IloInverse(IloInverseI * impl)
public	IloInverse(const IloEnv env, const IloIntVarArray f, const IloIntVarArray invf, const char * name=0)

Method Summary

public IloInverseI *	getImpl() const

Inherited methods from [IloConstraint](#)

IloConstraint::getImpl

Inherited methods from [IloIntExprArg](#)

[IloIntExprArg::getImpl](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of `IloInverse` represents an inverse constraint. Informally, we say that an inverse constraint works on two arrays, say, `f` and `invf`, so that an element of `f` composed with the corresponding element of `invf` produces the index of that element.

In formal terms, if the length of the array `f` is `n`, and the length of the array `invf` is `m`, then the inverse constraint insures that:

- ◆ for all `i` in the interval `[0, n-1]`, if `f[i]` is in `[0, m-1]` then `invf[f[i]] == i`;
- ◆ for all `j` in the interval `[0, m-1]`, if `invf[j]` is in `[0, n-1]` then `f[invf[j]] == j`.

In order for a constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also

[IloConstraint](#)

Constructors

```
public IloInverse()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloInverse(IloInverseI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloInverse(const IloEnv env,  
                  const IloIntArray f,  
                  const IloIntArray invf,  
                  const char * name=0)
```

This constructor creates an inverse constraint that if the length of the array `f` is `n`, and the length of the array `invf` is `m`, then this function returns a constraint that insures that:

- ◆ for all `i` in the interval `[0, n-1]`, if `f[i]` is in `[0, m-1]` then `invf[f[i]] == i`;
- ◆ for all `j` in the interval `[0, m-1]`, if `invf[j]` is in `[0, n-1]` then `f[invf[j]] == j`.

Methods

```
public IloInverseI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

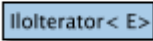
IloIsNAN

Category	Global Function
Definition File	ilconcert/ilosys.h
Synopsis	<pre>public int IloIsNAN(double)</pre>
Description	This function tests if a double value is a NaN (Not a number).

Iloiterator

Category Class

InheritancePath



Definition File ilconcert/iloiterator.h

Summary A template to create iterators for a class of extractable objects.

Constructor Summary	
public	<code>IloIterator</code> (const <code>IloEnv</code> env, <code>IloBool</code> withSubClasses= <code>IloTrue</code>)

Method Summary	
public <code>IloBool</code>	<code>ok()</code>
public void	<code>operator++()</code>

Description This template creates iterators for a given class of extractable objects (denoted by E in the template) within an instance of `IloEnv`.

By default, an iterator created in this way will traverse instances of E and of its subclasses. You can prevent the iterator from traversing instances of subclasses of E (that is, you can limit its effect) by setting the parameter `withSubClasses` to `IloFalse` in the constructor of the iterator.

While an iterator created in this way is working, you must not create nor destroy any extractable objects in the instance of `IloEnv` where it is working. In other words, an iterator created in this way works only in a stable environment.

An iterator created with this template differs from an instance of `IloModel::Iterator`. An instance of `IloModel::Iterator` works only on extractable objects (instances of `IloExtractable` or its subclasses) that have explicitly been added to a model (an instance of `IloModel`). In contrast, an iterator created with this template will work on all extractable objects within a given environment, whether or not they have been explicitly added to a model.

See Also [IloEnv](#), [IloExtractable](#), [IloModel](#), [IloModel::Iterator](#)

Constructors

```
public IloIterator(const IloEnv env,
                  IloBool withSubClasses=IloTrue)
```

This template constructor creates an iterator for instances of the class `E`. When the parameter `withSubClasses` is `IloTrue` (its default value), the iterator will also work on instances of the subclasses of `E`. When `withSubClasses` is `IloFalse`, the iterator works only on instances of `E`.

Example

Here is an example of an iterator created by this template for the class [IloNumVar](#).

```
typedef IloIterator<IloNumVar> IloNumVarIterator;
void displayAllVars(IloEnv env) {
    for (IloNumVarIterator it(env); it.ok(); ++it) {
        IloNumVar ext = *it;
        cout << ext;
    }
}
```

Methods

```
public IloBool ok()
```

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

```
public void operator++()
```

This operator advances the iterator to point to the next value in the iteration.

IloLinearTimeConverter

Category Class

InheritancePath



IloLinearTimeConverter

Definition File ilconcert/ilodate.h

Description This class is used to convert dates and durations to doubles. It performs linear conversion according to the following rule:

num = *duration* / *unit* (for durations)

num = (*date* - *origin*) / *unit* (for dates)

IloLog

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloLog(const IloNumExprArg arg)
public IloNum IloLog(IloNum val)
```

Description Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

IloLog returns the natural logarithm of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `log`.

IloMax

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNum IloMax(const IloNumArray vals)
public IloNum IloMax(IloNum val1,
                    IloNum val2)
public IloInt IloMax(const IloIntArray vals)
public IloNumExprArg IloMax(const IloNumExprArray exprs)
public IloIntExprArg IloMax(const IloIntExprArray exprs)
public IloNumExprArg IloMax(const IloNumExprArg x,
                           const IloNumExprArg y)
public IloNumExprArg IloMax(const IloNumExprArg x,
                           IloNum y)
public IloNumExprArg IloMax(IloNum x,
                           const IloNumExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x,
                           const IloIntExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x,
                           IloInt y)
public IloNumExprArg IloMax(const IloIntExprArg x,
                           IloNum y)
public IloIntExprArg IloMax(const IloIntExprArg x,
                           int y)
public IloIntExprArg IloMax(IloInt x,
                           const IloIntExprArg y)
public IloNumExprArg IloMax(IloNum x,
                           const IloIntExprArg y)
public IloIntExprArg IloMax(int x,
                           const IloIntExprArg y)
```

Description These functions compare their arguments and return the greatest value.

IloMaximize

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloObjective IloMaximize(const IloEnv env,
                                IloNum constant=0.0,
                                const char * name=0)
public IloObjective IloMaximize(const IloEnv env,
                                const IloNumExprArg expr,
                                const char * name=0)
```

Description This function defines a maximization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Maximize`. However, an instance of `IloObjective` created by `IloMaximize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

IloMember

Category Global Function

Definition File ilconcert/iloanysset.h

Synopsis

```
public IloConstraint IloMember(const IloEnv env,
    const IloAnyVar var1,
    const IloAnySetVar var2)
public IloConstraint IloMember(const IloEnv env,
    IloAny val,
    const IloAnySetVar var2)
public IloConstraint IloMember(const IloEnv,
    const IloNumExprArg expr,
    const IloNumArray elements)
public IloConstraint IloMember(const IloEnv,
    const IloIntExprArg expr,
    const IloIntArray elements)
public IloConstraint IloNotMember(const IloEnv,
    const IloNumExprArg expr,
    const IloNumArray elements)
public IloConstraint IloNotMember(const IloEnv,
    const IloIntExprArg expr,
    const IloIntArray elements)
public IloConstraint IloMember(const IloEnv,
    const IloIntVar var1,
    const IloIntSetVar var2)
public IloConstraint IloMember(const IloEnv,
    IloInt var1,
    const IloIntSetVar var2)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. The constraint forces `element` to be a member of `setVar`.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloMin

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```

public IloNum IloMin(const IloNumArray vals)
public IloNum IloMin(IloNum val1,
                    IloNum val2)
public IloInt IloMin(const IloIntArray vals)
public IloNumExprArg IloMin(const IloNumExprArray exprs)
public IloIntExprArg IloMin(const IloIntExprArray exprs)
public IloNumExprArg IloMin(const IloNumExprArg x,
                           const IloNumExprArg y)
public IloNumExprArg IloMin(const IloNumExprArg x,
                           IloNum y)
public IloNumExprArg IloMin(IloNum x,
                           const IloNumExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x,
                           const IloIntExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x,
                           IloInt y)
public IloNumExprArg IloMin(const IloIntExprArg x,
                           IloNum y)
public IloIntExprArg IloMin(const IloIntExprArg x,
                           int y)
public IloIntExprArg IloMin(IloInt x,
                           const IloIntExprArg y)
public IloNumExprArg IloMin(IloNum x,
                           const IloIntExprArg y)
public IloIntExprArg IloMin(int x,
                           const IloIntExprArg y)

```

Description These functions compare their arguments and return the least value. When its argument is an array, the function compares the *elements* of that array and returns the least value.

IloMinimize

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloObjective IloMinimize(const IloEnv env,
                                IloNum constant=0.0,
                                const char * name=0)
public IloObjective IloMinimize(const IloEnv env,
                                const IloNumExprArg expr,
                                const char * name=0)
```

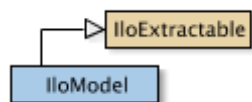
Description This function defines a minimization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Minimize`. However, an instance of `IloObjective` created by `IloMinimize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

IloModel

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary

public	IloModel()
public	IloModel(IloModelI * impl)
public	IloModel(const IloEnv env, const char * name=0)

Method Summary

public const IloExtractableArray &	IloModel::add(const IloExtractableArray & x) const
public IloExtractable	add(const IloExtractable x) const
public IloModelI *	getImpl() const
public void	remove(const IloExtractableArray x) const
public void	remove(const IloExtractable x) const

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Inner Class	
IloModel::IloModel::Iterator	

Description

An instance of this class represents a model. A model consists of the extractable objects such as constraints, constrained variables, objectives, and possibly other modeling objects, that represent a problem. Concert Technology extracts information from a model and passes the information in an appropriate form to algorithms that solve the problem. (For information about extracting objects into algorithms, see the member function [IloAlgorithm::extract](#) and the template [IloAdd](#).)

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

Models and Submodels

With Concert Technology, you may create more than one model in a given environment (an instance of [IloEnv](#)). In fact, you can create submodels. That is, you can add one model to another model within the same environment.

What Is Extracted from a Model

All the extractable objects (that is, instances of [IloExtractable](#) or one of its subclasses) that have been added to a model (an instance of `IloModel`) and that have not been removed from it will be extracted when an algorithm extracts the model. An instance of the nested class [IloModel::Iterator](#) accesses those extractable objects.

See Also

[IloEnv](#), [IloExtractable](#), [IloModel::Iterator](#)

Constructors

```
public IloModel()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloModel(IloModelI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloModel(const IloEnv env,
               const char * name=0)
```

This constructor creates a model. By default, the name of the model is the empty string, but you can attribute a name to the model at its creation.

Methods

```
public const IloExtractableArray & add(const IloExtractableArray & x) const
```

This member function adds the array of extractable objects to the invoking model.

Note: *The member function `add` notifies Concert Technology algorithms about this addition to the model.*

```
public IloExtractable add(const IloExtractable x) const
```

This member function adds the extractable object to the invoking model.

Note: *The member function `add` notifies Concert Technology algorithms about this addition to the model.*

```
public IloModelI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloExtractableArray x) const
```

This member function removes the array of extractable objects from the invoking model.

Note: *The member function `remove` notifies Concert Technology algorithms about this removal from the model.*

```
public void remove(const IloExtractable x) const
```

This member function removes the extractable object from the invoking model.

Note: *The member function `remove` notifies Concert Technology algorithms about this removal from the model.*

IloModel::Iterator

Category Inner Class

InheritancePath

Definition File ilconcert/ilomodel.h

Constructor Summary

public	Iterator (const IloModel model)

Method Summary

public IloBool	ok() const
public IloExtractable	operator * ()
public void	operator++ ()

Description An instance of this nested class is an iterator capable of traversing the extractable objects in a model.

An iterator of this class differs from one created by the template [IloIterator](#). Instances of [IloIterator](#) traverse all the extractable objects of a given class (indicated by E in the template) within a given environment (an instance of [IloEnv](#)), whether or not those extractable objects have been explicitly added to a model. Instances of [IloModel::Iterator](#) traverse only those extractable objects that have explicitly been added to a given model (an instance of [IloModel](#)).

See Also [IloIterator](#), [IloModel](#)

Constructors public **Iterator**(const [IloModel](#) model)

This constructor creates an iterator to traverse the extractable objects in the model indicated by `model`.

Methods public [IloBool](#) **ok()** const

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

public [IloExtractable](#) **operator ***()

This operator returns the current extractable object, the one to which the invoking iterator points.

```
public void operator++()
```

This operator advances the iterator to point to the next extractable object in the model.

IloMonotonicDecreasingNumExpr

Category	Global Function
Definition File	ilconcert/iloexpression.h
Synopsis	<pre>public IloNumExprArg IloMonotonicDecreasingNumExpr(IloNumExprArg node, IloNumFunction f, IloNumFunction invf)</pre>
Description	This function creates a new constrained expression equal to $f(x)$. The arguments <code>f</code> and <code>invf</code> must be pointers to functions of type <code>IloNumFunction</code>. Those two functions must be inverses of one another, that is, $\text{invf}(f(x)) == x \text{ and } f(\text{invf}(x)) == x \text{ for all } x.$ Those two functions must also be monotonically decreasing. <code>IloMonotonicDecreasingNumExpr</code> does <i>not</i> verify whether <code>f</code> and <code>invf</code> are inverses of one another. It does <i>not</i> verify whether they are monotonically decreasing either. The effects of this function are reversible.

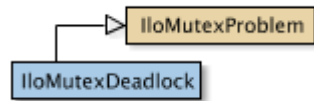
IloMonotonicIncreasingNumExpr

Category	Global Function
Definition File	ilconcert/iloexpression.h
Synopsis	<pre>public IloNumExprArg IloMonotonicIncreasingNumExpr(IloNumExprArg node, IloNumFunction f, IloNumFunction invf)</pre>
Description	This function creates a new constrained expression equal to $f(x)$. The arguments <code>f</code> and <code>invf</code> must be pointers to functions of type <code>IlcFloatFunction</code>. Those two functions must be inverses of one another, that is, $\text{invf}(f(x)) == x \text{ and } f(\text{invf}(x)) == x \text{ for all } x.$ <code>IloMonotonicIncreasingNumExpr</code> does <i>not</i> verify whether <code>f</code> and <code>invf</code> are inverses of one another. It does <i>not</i> verify whether they are monotonically increasing either.

IloMutexDeadlock

Category Class

InheritancePath



Definition File `ilconcert/ilothread.h`

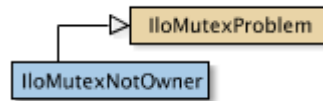
Summary The class of exceptions thrown due to mutex deadlock.

Description This is the class of exceptions thrown if two or more threads become deadlocked waiting for a mutex owned by the other(s).

IloMutexNotOwner

Category Class

InheritancePath



Definition File `ilconcert/ilothread.h`

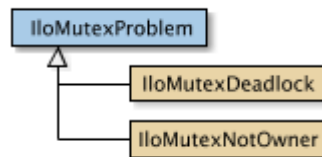
Summary The class of exceptions thrown.

Description The class of exceptions thrown if a thread attempts to unlock a mutex that it does not own.

IloMutexProblem

Category Class

InheritancePath



Definition File ilconcert/ilothread.h

Constructor Summary

public	IloMutexProblem (const char * msg)

Description

The class `IloMutexProblem` is part of the hierarchy of classes representing exceptions in Concert Technology. Concert Technology uses instances of this class when an error occurs with respect to a mutex, an instance of [IloFastMutex](#).

An exception is thrown; it is not allocated in a Concert Technology environment; it is not allocated on the C++ heap. It is not necessary for you as a programmer to delete an exception explicitly. Instead, the system calls the constructor of the exception to create it, and the system calls the destructor of the exception to delete it.

When exceptions are enabled on a platform that supports C++ exceptions, an instance of `IloMutexProblem` makes it possible for Concert Technology to throw an exception in case of error. On platforms that do not support C++ exceptions, an instance of this class makes it possible for Concert Technology to exit in case of error.

Throwing and Catching Exceptions

Exceptions are thrown by value. They are not allocated on the C++ heap, nor in a Concert Technology environment. The correct way to catch an exception is to catch a reference to the error (indicated by the ampersand &), like this:

```
catch(IloMutexProblem& error);
```

See Also [IloException](#), [IloFastMutex](#)

Constructors

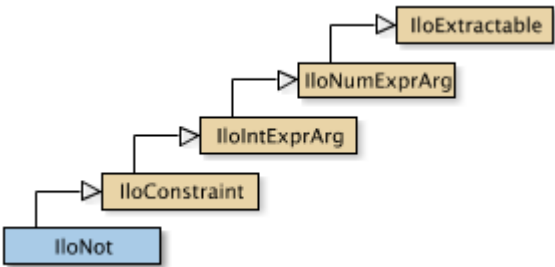
```
public IloMutexProblem(const char * msg)
```

This constructor creates an instance of `IloMutexProblem` to represent an exception in case of an error involving a mutex. This instance is not allocated on C++ heap; it is not allocated in a Concert Technology environment either.

IloNot

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloNot()
public	IloNot(IloNotI * impl)

Method Summary	
public IloNotI *	getImpl() const

Inherited methods from IloConstraint	
IloConstraint::getImpl	

Inherited methods from IloIntExprArg	
IloIntExprArg::getImpl	

Inherited methods from IloNumExprArg
IloNumExprArg::getImpl

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description	The class <code>IloNot</code> represents a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.
See Also	operator!
Constructors	<pre>public IloNot()</pre> This constructor creates an empty handle. You must initialize it before you use it. <pre>public IloNot(IloNotI * impl)</pre> This constructor creates a handle object from a pointer to an implementation object.
Methods	<pre>public IloNotI * getImpl() const</pre> This member function returns a pointer to the implementation object of the invoking handle.

IloNotMember

Category Global Function

Definition File ilconcert/iloanysset.h

Synopsis

```
public IloConstraint IloNotMember(const IloEnv env,
    const IloAnyVar var1,
    const IloAnySetVar var2)
public IloConstraint IloNotMember(const IloEnv env,
    IloAny val,
    const IloAnySetVar var2)
public IloConstraint IloNotMember(const IloEnv,
    const IloIntVar var1,
    const IloIntSetVar var2)
public IloConstraint IloNotMember(const IloEnv,
    IloInt var1,
    const IloIntSetVar var2)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. The constraint forces `element` not to be a member of `setVar`.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloNullIntersect

Category Global Function

Definition File ilconcert/iloanyset.h

Synopsis

```
public IloConstraint IloNullIntersect(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySetVar var2)
public IloConstraint IloNullIntersect(const IloEnv env,
    const IloAnySet var1,
    const IloAnySetVar var2)
public IloConstraint IloNullIntersect(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySet var2)
public IloConstraint IloNullIntersect(const IloEnv,
    const IloIntSetVar var1,
    const IloIntSetVar var2)
public IloConstraint IloNullIntersect(const IloEnv,
    const IloIntSet var1,
    const IloIntSetVar var2)
public IloConstraint IloNullIntersect(const IloEnv,
    const IloIntSetVar var1,
    const IloIntSet var2)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. The constraint forces the set `var1` to have no elements in common with the set `var2`. In other words, the intersection of `var1` with `var2` will be empty when this constraint is satisfied.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloNum

Category	Type Definition
Definition File	ilconcert/ilosys.h
Synopsis	double IloNum
Description	This type definition represents numeric values as floating-point numbers in Concert Technology.
See Also	IloModel , IloInt

IloNumArray

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Constructor Summary	
public	IloNumArray (IloArrayI * i=0)
public	IloNumArray (const IloNumArray & cpy)
public	IloNumArray (const IloEnv env, IloInt n=0)
public	IloNumArray (const IloEnv env, IloInt n, IloNum f0, IloNum f1, ...)

Method Summary	
public IloBool	contains (IloNum value) const
public IloNumExprArg	operator ()(IloNumExprArg intExp) const
public IloNum &	operator [](IloInt i)
public const IloNum &	operator [](IloInt i) const
public IloNumExprArg	operator [](IloIntExprArg intExp) const
public IloIntArray	toIntArray () const

Description For each basic type, Concert Technology defines a corresponding array class. [IloNumArray](#) is the array class of the basic floating-point class ([IloNum](#)) for a model.

Instances of [IloNumArray](#) are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

[IloNumArray](#) inherits additional methods from the template [IloArray](#):

- ◆ [add](#)
- ◆ [add](#)
- ◆ [add](#)
- ◆ [clear](#)
- ◆ [getEnv](#)
- ◆ [getSize](#)
- ◆ [remove](#)
- ◆ [operator\[\]](#)
- ◆ [operator\[\]](#)

See Also [IloNum](#), [operator>>](#), [operator](#)

Constructors

```
public IloNumArray(IloArrayI * i=0)
```

This constructor creates an empty array of floating-point numbers for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloNumArray(const IloNumArray & cpy)
```

This copy constructor creates a handle to the array of floating-point objects indicated by `copy`.

```
public IloNumArray(const IloEnv env,
                  IloInt n=0)
```

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

```
public IloNumArray(const IloEnv env,
                  IloInt n,
                  IloNum f0,
                  IloNum f1,
                  ...)
```

This constructor creates an array of `n` floating-point objects for use in a model.

Methods

```
public IloBool contains(IloNum value) const
```

This member function checks whether the value is contained or not.

```
public IloNumExprArg operator()(IloNumExprArg intExp) const
```


This subscripting operator is deprecated. It must be replaced by the operator `IloNumExprArg operator[] (IloIntExprArg intExp) const`;

```
public IloNum & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

```
public const IloNum & operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `const` operator

```
IloArray operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloNumExprArg operator[](IloIntExprArg intExp) const
```

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `intExp` is bound to the value `i`, then the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `intExp`.

This operator is also known as an element constraint.

```
public IloIntArray toIntArray() const
```

This member function copies the invoking array to a new `IloIntVarArray`, checking the type of the variables during the copy.

IloNumColumn

Category Class

InheritancePath

IloNumColumn

Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloNumColumn (const IloEnv env)
public	IloNumColumn (const IloAddNumVar & var)

Method Summary

public void	clear() const
public	operator const IloAddNumVar &() const
public IloNumColumn &	operator+=(const IloAddValueToRange & rhs)
public IloNumColumn &	operator+=(const IloAddNumVar & rhs)
public IloNumColumn &	operator+=(const IloNumColumn & rhs)

Description

An instance of this class helps you design a model through column representation. In other words, you can create a model by defining each of its columns as an instance of this class. In particular, an instance of `IloNumColumn` enables you to build a column for a numeric variable (an instance of `IloNumVar`) with information about the extractable objects (such as objectives, constraints, etc.) where that numeric variable may eventually appear, even if the numeric variable has not yet been created.

Usually you populate a column (an instance of this class) with objects returned by the `operator()` of the class (such as `IloObjective::operator()`) where you want to install the newly created variable, as in the examples below.

An instance of `IloNumColumn` keeps a list of those objects returned by `operator()`. In other words, an instance of `IloNumColumn` knows the extractable objects where a numeric variable will be added when it is created.

When you create a new instance of [IloNumVar](#) with an instance of `IloNumColumn` as a parameter, then Concert Technology adds the newly created numeric variable to all the extractable objects (such as constraints, ranges, objectives, etc.) for which an instance of `IloAddNumVar` will be added to this instance of `IloNumColumn`. Note that `IloNumColumn` does not support normalization, as normalization is not well defined for constraints such as `IloSOS1` and `IloAllDiff`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

For information on columnwise modeling, see the concept [Column-Wise Modeling](#).

See Also [IloNumVar](#), [IloObjective](#), [IloRange](#)

Constructors

```
public IloNumColumn(const IloEnv env)
```

This constructor creates an empty column in the environment `env`.

```
public IloNumColumn(const IloAddNumVar & var)
```

This constructor creates a column and adds `var` to it.

Methods

```
public void clear() const
```

This member function removes (from the invoking column) its list of extractable objects.

```
public operator const IloAddNumVar &() const
```

This casting operator allows you to use instances of `IloNumColumn` in column expressions. It accepts an extractable object, such as an objective (an instance of [IloObjective](#)) or a constraint (an instance of [IloConstraint](#)). It returns the object derived from `IloAddNumVar` and needed to represent the extractable object in column format.

```
public IloNumColumn & operator+=(const IloAddValueToRange & rhs)
```

This operator adds the appropriate instances of `IloAddValueToRange` for the right-hand side `rhs` to the invoking column.

Examples:

To use an instance of this class to create a column with a coefficient of 2 in the objective, with 10 in `range1`, and with 3 in `range2`, set:

```
IloNumColumn col = obj(2) + range1(10) + range2(3);
```

To use an instance of this class to create a numeric variable corresponding to the column with lower bound 0 (zero) and upper bound 10:

```
IloNumVar var(env, col, 0, 10);
```

Another example:

```
IloNumColumn col1(env);
IloNumColumn col2 = rng7(3.1415);
col1 += obj(1.0);
col1 += rng(-12.0);
col2 += rng2(13.7) + rng3(14.7);
col2 += col1;
```

```
public IloNumColumn & operator+=(const IloAddNumVar & rhs)
```

This operator adds the appropriate instances of `IloAddNumVar` for the right-hand side `rhs` to the invoking column.

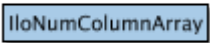
```
public IloNumColumn & operator+=(const IloNumColumn & rhs)
```

This operator assigns the right-hand side `rhs` to the invoking column.

IloNumColumnArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary	
public	IloNumColumnArray (IloDefaultArrayI * i=0)
public	IloNumColumnArray (const IloEnv env, IloInt n=0)
public	IloNumColumnArray (const IloNumColumnArray & h)

Description

For each basic type, Concert Technology defines a corresponding array class. `IloNumColumnArray` is the array class of the column representation class for a model. The implementation class for `IloNumColumnArray` is the undocumented class `IloNumColumnArrayI`.

Instances of `IloNumColumnArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also [IloModel](#), [IloNumColumn](#)

Constructors

```
public IloNumColumnArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of columns. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloNumColumnArray(const IloEnv env,
                        IloInt n=0)
```

This constructor creates an array of n elements. Initially, the n elements are empty handles.

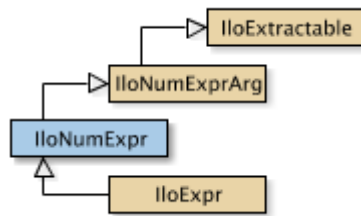
```
public IloNumColumnArray(const IloNumColumnArray & h)
```

This copy constructor creates a handle to the array of column objects indicated by **copy**.

IloNumExpr

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The class of numeric expressions in a Concert model.

Constructor Summary

public	IloNumExpr()
public	IloNumExpr(IloNumExprI * impl)
public	IloNumExpr(const IloNumExprArg expr)
public	IloNumExpr(const IloEnv env, IloNum=0)
public	IloNumExpr(const IloNumLinExprTerm term)
public	IloNumExpr(const IloIntLinExprTerm term)
public	IloNumExpr(const IloExpr & expr)

Method Summary

public IloNumExprI *	IloNumExpr::getImpl() const
public IloNumExpr &	IloNumExpr::operator *=(IloNum val)
public IloNumExpr &	IloNumExpr::operator+=(const IloNumExprArg expr)
public IloNumExpr &	IloNumExpr::operator+=(IloNum val)
public IloNumExpr &	IloNumExpr::operator-=(const IloNumExprArg expr)
public IloNumExpr &	IloNumExpr::operator-=(IloNum val)
public IloNumExpr &	IloNumExpr::operator/=(IloNum val)

Inherited methods from IloNumExprArg
IloNumExprArg::getImpl

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Inner Class	
IloNumExpr::IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numerical constant of a non-linear expression is set or queried.

Description Numeric expressions in Concert Technology 2.0 are represented using the class `IloNumExpr`.

Constructors

public **IloNumExpr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloNumExpr**(IloNumExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloNumExpr**(const [IloNumExprArg](#) expr)

This constructor creates a numeric expression using the undocumented class `IloNumExprArg`.

public **IloNumExpr**(const [IloEnv](#) env,
 [IloNum](#)=0)

This constructor creates a numeric expression in the environment indicated by `env`. It may be used to build other expressions from variables belonging to `env`.

public **IloNumExpr**(const IloNumLinExprTerm term)

This constructor creates a numeric expression using the undocumented class `IloNumLinExprTerm`.


```
public IloNumExpr(const IloIntLinExprTerm term)
```

This constructor creates a numeric expression using the undocumented class `IloIntLinExprTerm`.

```
public IloNumExpr(const IloExpr & expr)
```

This is the copy constructor for this class.

Methods

```
public IloNumExprI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNumExpr & operator *=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`.

```
public IloNumExpr & operator +=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`.

```
public IloNumExpr & operator +=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`.

```
public IloNumExpr & operator -=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`.

```
public IloNumExpr & operator -=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`.

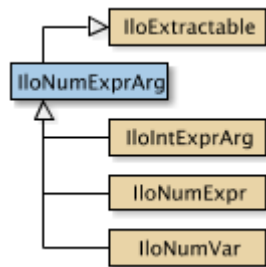
```
public IloNumExpr & operator /=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x / ...`.

IloNumExprArg

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary A class used internally in Concert Technology.

Constructor Summary	
public	IloNumExprArg()
public	IloNumExprArg(IloNumExprI * impl)

Method Summary	
public IloNumExprI *	IloNumExprArg::getImpl() const

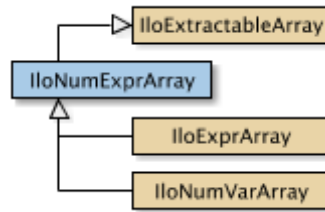
Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description	Concert Technology uses instances of this class internally as temporary objects when it is parsing a C++ expression in order to build an instance of <code>IloNumExpr</code>. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as <code>IloSum</code> or <code>IloScalProd</code>, can be an instance of this class.
Constructors	<pre>public IloNumExprArg()</pre> This constructor creates an empty handle. You must initialize it before you use it. <pre>public IloNumExprArg(IloNumExprI * impl)</pre> This constructor creates a handle object from a pointer to an implementation object.
Methods	<pre>public IloNumExprI * getImpl() const</pre> This member function returns a pointer to the implementation object of the invoking handle.

IloNumExprArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The class IloNumExprArray.

Constructor Summary

public	IloNumExprArray (IloDefaultArrayI * i=0)
public	IloNumExprArray (const IloEnv env, IloInt n=0)

Method Summary

public void	IloNumExprArray::add (IloInt more, const IloNumExpr x)
public void	IloNumExprArray::add (const IloNumExpr x)
public void	IloNumExprArray::add (const IloNumExprArray array)
public void	IloNumExprArray::endElements ()
public IloNumExprArg	IloNumExprArray::operator ()(IloNumExprArg anIntegerExpr) const
public IloNumExprArg	IloNumExprArray::operator [(IloIntExprArg anIntegerExpr) const

Inherited methods from **IloExtractableArray**

`IloExtractableArray::add`, `IloExtractableArray::add`,
`IloExtractableArray::add`, `IloExtractableArray::endElements`

Description

For each basic type, Concert Technology defines a corresponding array class. **IloNumExprArray** is the array class of the numeric expressions class (**IloNumExpr**) for a model.

Instances of **IloNumExprArray** are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructors

```
public IloNumExprArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty array of numeric expressions for use in a model. You cannot create instances of the undocumented class **IloDefaultArrayI**. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloNumExprArray(const IloEnv env,  
                      IloInt n=0)
```

This constructor creates an array of *n* elements. Initially, the *n* elements are empty handles.

Methods

```
public void add(IloInt more,  
               const IloNumExpr x)
```

This member function appends *x* to the invoking array. The parameter *more* indicates how many times.

```
public void add(const IloNumExpr x)
```

This member function appends *x* to the invoking array.

```
public void add(const IloNumExprArray array)
```

This member function appends the elements in *array* to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloNumExprArg operator()(IloNumExprArg anIntegerExpr) const
```

This subscripting operator is deprecated. It must be replaced by the operator `IloNumExprArg operator[] (IloIntExprArg anIntegerExpr) const`;

```
public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const
```

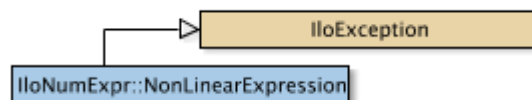
This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element constraint.

IloNumExpr::NonLinearExpression

Category Inner Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The class of exceptions thrown if a numerical constant of a non-linear expression is set or queried.

Method Summary	
public const IloNumExprArg	getExpression() const

Inherited methods from IloException
IloException::end , IloException::getMessage

DescriptionMethods public const [IloNumExprArg](#) [getExpression\(\)](#) const

The member function `getExprUsed` returns the expression involved in the exception.

IloNumFunction

Category	Type Definition
Definition File	ilconcert/ilosys.h
Synopsis	<code>IloNum(* IloNumFunction)(IloNum)</code>
Description	This C++ type represents a pointer to a function that takes an argument of type IloNum and returns an object of type IloNum . This type is the type of the function passed as a parameter to the function IloMonotonicDecreasingNumExpr .
See Also	IloMonotonicDecreasingNumExpr , IloMonotonicIncreasingNumExpr

IloNumSet

Category Class

InheritancePath

IloNumSet

Definition File ilconcert/iloset.h

Summary The class IloNumSet.

Constructor Summary

public	IloNumSet (const IloEnv env, const IloNumArray array, IloBool withIndex= IloFalse)
public	IloNumSet (const IloEnv env, IloBool withIndex= IloFalse)
public	IloNumSet (IloNumSetI * impl=0)

Method Summary

public void	add (IloNumSet set)
public void	add (IloNum elt)
public IloBool	contains (IloNumSet set) const
public IloBool	contains (IloNum elt) const
public void	empty ()
public IloNumSetI *	getImpl () const
public IloNum	getNext (IloNum value, IloInt n=1) const
public IloNum	getNextC (IloNum value, IloInt n=1) const
public IloNum	getPrevious (IloNum value, IloInt n=1) const
public IloNum	getPreviousC (IloNum value, IloInt n=1) const
public IloInt	getSize () const
public IloBool	intersects (IloNumSet set) const
public void	remove (IloNumSet set)
public void	remove (IloNum elt)
public void	setIntersection (IloNumSet set)
public void	setIntersection (IloNum elt)

Description

An instance of this class represents a set of numeric values. The same numeric value will not appear more than once in a set. The elements of a set are not ordered. The class `Iterator` offers you a way to traverse the elements of such a set.

If you are considering modeling issues where you want to represent repeated elements or where you want to exploit an indexed order among the elements, then you might want to look at the class `IloNumArray` instead of this class for sets.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also

`IloNumArray`, `Iterator`, `IloNum`, `operator`

Constructors

```
public IloNumSet(const IloEnv env,
                 const IloNumArray array,
                 IloBool withIndex=IloFalse)
```

This constructor creates a set of numeric values in the environment `env` from the elements in `array`.

```
public IloNumSet(const IloEnv env,
                 IloBool withIndex=IloFalse)
```

This constructor creates an empty set (no elements) in the environment `env`. You must use the member function `IloNumSet::add` to fill this set with elements.

```
public IloNumSet(IloNumSetI * impl=0)
```

This constructor creates a handle to a set of numeric values from its implementation object.

Methods

```
public void add(IloNumSet set)
```

This member function adds `set` to the invoking set. By "adds," we mean that the invoking set becomes the union of its former elements and the elements of `set`.

To calculate the arithmetic sum of values in an array, use the function `IloSum`.

```
public void add(IloNum elt)
```

This member function adds `elt` to the invoking set. By "adds," we mean that the invoking set becomes the union of its former elements and the new `elt`.

```
public IloBool contains(IloNumSet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` intersects the invoking set. The value one indicates that the invoking set contains all the elements of `set`, and that the intersection of the invoking set with `set` is precisely `set`. The value zero indicates that the intersection of the invoking set `set` is not precisely `set`.

```
public IloBool contains(IloNum elt) const
```

This member function returns a Boolean value (zero or one) that indicates whether `elt` is an element of the invoking set. The value one indicates that the invoking set contains `elt`; the value zero indicates that the invoking set does not contain `elt`.

```
public void empty()
```

This member function removes the elements from the invoking set. In other words, the invoking set becomes the empty set.

```
public IloNumSetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking set.

```
public IloNum getNext(IloNum value,  
                    IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it throws an exception

See also `getNextC`, `getPreviousC` for circular search.

```
public IloNum getNextC(IloNum value,  
                     IloInt n=1) const
```

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it will give you the first value (circular search)

See also `getNext`, `getPrevious`.

```
public IloNum getPrevious(IloNum value,  
                        IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it throws an exception

See also `getNextC`, `getPreviousC` for circular search.

```
public IloNum getPreviousC(IloNum value,  
                          IloInt n=1) const
```

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it will give you the last value (circular search)

See also getNext, getPrevious.

```
public IloInt getSize() const
```

This member function returns an integer indicating the size of the invoking set (that is, how many elements it contains).

```
public IloBool intersects(IloNumSet set) const
```

This member function returns a Boolean value (zero or one) that indicates whether `set` intersects the invoking set. The value one indicates that the intersection of `set` and the invoking set is not empty (at least one element in common); the value zero indicates that the intersection of `set` and the invoking set is empty (no elements in common).

```
public void remove(IloNumSet set)
```

This member function removes all the elements of `set` from the invoking set.

```
public void remove(IloNum elt)
```

This member function removes `elt` from the invoking set.

```
public void setIntersection(IloNumSet set)
```

This member function changes the invoking set so that it includes only the elements of `set`. In other words, the invoking set becomes the intersection of its former elements with the elements of `set`.

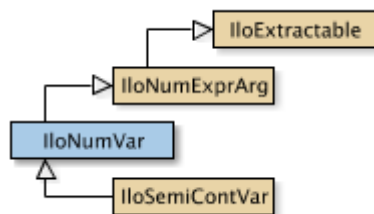
```
public void setIntersection(IloNum elt)
```

This member function changes the invoking set so that it includes only the element indicated by `elt`. In other words, the invoking set becomes the intersection of its former elements with `elt`.

IloNumVar

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary	
public	<code>IloNumVar()</code>
public	<code>IloNumVar(IloNumVarI * impl)</code>
public	<code>IloNumVar(const IloEnv env, IloNum lb=0, IloNum ub=IloInfinity, IloNumVar::Type type=Float, const char * name=0)</code>
public	<code>IloNumVar(const IloEnv env, IloNum lowerBound, IloNum upperBound, const char * name)</code>
public	<code>IloNumVar(const IloAddNumVar & var, IloNum lowerBound=0.0, IloNum upperBound=IloInfinity, IloNumVar::Type type=Float, const char * name=0)</code>
public	<code>IloNumVar(const IloEnv env, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)</code>
public	<code>IloNumVar(const IloAddNumVar & var, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)</code>
public	<code>IloNumVar(const IloConstraint constraint)</code>
public	<code>IloNumVar(const IloNumCollection coll, const char * name=0)</code>
public	<code>IloNumVar(const IloNumRange coll, const char * name=0)</code>

Method Summary	
public IloNumVarI *	getImpl() const
public IloNum	getLB() const
public void	getPossibleValues(IloNumArray values) const
public IloNumVar::Type	getType() const
public IloNum	getUB() const
public void	setBounds(IloNum lb, IloNum ub) const
public void	setLB(IloNum num) const
public void	setPossibleValues(const IloNumArray values) const
public void	setUB(IloNum num) const

Inherited methods from IloNumExprArg
IloNumExprArg::getImpl

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Inner Enumeration	
IloNumVar::Type	

Description

An instance of this class represents a numeric variable in a model. A numeric variable may be either an integer variable or a floating-point variable; that is, a numeric variable has a type, a value of the nested enumeration [IloNumVar::Type](#). By default, its type is `Float`. It also has a lower and upper bound. A numeric variable cannot assume values less than its lower bound, nor greater than its upper bound.

If you are looking for a class of variables that can assume only constrained integer values, consider the class [IloIntVar](#). If you are looking for a class of binary decision

variables that can assume only the values 0 (zero) or 1 (one), then consider the class [IloBoolVar](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

Programming Hint

For each enumerated value in the nested enumeration [IloNumVar::Type](#), Concert Technology offers an equivalent predefined C++ `#define` to make programming easier. For example, in your applications, you may write either statement:

```
IloNumVar x(env, 0, 17, IloNumVar::Int); // using the enumeration
IloNumVar x(env, 0, 17, ILOINT);        // using the #define
```

Note: When numerical bounds are given to an integer variable (an *IloIntVar* or an *IloNumVar* with `Type = Int`) in the constructors of via a modifier (*setUB*, *setLB*, *setBounds*), they are inward rounded to an integer value. *LB* is rounded down and *UB* is rounded up.

See Also

[IloBoolVar](#), [IloIntVar](#), [IloModel](#), [IloNumVarArray](#),
[IloNumVar::Type](#)

Constructors

```
public IloNumVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloNumVar(IloNumVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloNumVar(const IloEnv env,
                 IloNum lb=0,
                 IloNum ub=IloInfinity,
                 IloNumVar::Type type=Float,
                 const char * name=0)
```

This constructor creates a constrained numeric variable and makes it part of the environment `env`. By default, the numeric variable ranges from 0.0 (zero) to the symbolic constant `IloInfinity`, but you can indicate other upper and lower bounds yourself. By default, the numeric variable assumes floating-point values. However, you can constrain it to be an integer by setting its `type = Int`. By default, its `name` is indicated by the empty string, but you can indicate a `name` of your own choice.

```
public IloNumVar(const IloEnv env,
```

```
IloNum lowerBound,
IloNum upperBound,
const char * name)
```

This constructor creates a constrained numeric variable and makes it part of the environment `env`. The bounds of the variable are set by `lowerBound` and `upperBound`. By default, its name is indicated by the empty string, but you can indicate a `name` of your own choice.

```
public IloNumVar(const IloAddNumVar & var,
                IloNum lowerBound=0.0,
                IloNum upperBound=IloInfinity,
                IloNumVar::Type type=Float,
                const char * name=0)
```

This constructor creates a constrained numeric variable in column format. For more information on adding columns to a model, refer to the concept *Column-Wise Modeling*.

```
public IloNumVar(const IloEnv env,
                const IloNumArray values,
                IloNumVar::Type type=Float,
                const char * name=0)
```

This constructor creates a constrained discrete numeric variable and makes it part of the environment `env`. The new discrete variable will be limited to values in the set indicated by the array `values`. By default, its name is indicated by the empty string, but you can indicate a name of your own choice. You can use the member function `setPossibleValues` with instances created by this constructor.

```
public IloNumVar(const IloAddNumVar & var,
                const IloNumArray values,
                IloNumVar::Type type=Float,
                const char * name=0)
```

This constructor creates a constrained *discrete* numeric variable from `var` by limiting its domain to the values indicated in the array `values`. You may use the member function `setPossibleValues` with instances created by this constructor.

```
public IloNumVar(const IloConstraint constraint)
```

This constructor creates a constrained numeric variable which is equal to the truth value of `constraint`. The truth value of `constraint` is either 0 for `IloFalse` or 1 for `IloTrue`. You can use this constructor to cast a constraint to a constrained numeric variable. That constrained numeric variable can then be used like any other constrained numeric variable. It is thus possible to express sums of constraints, for example. The following line expresses the idea that all three variables cannot be equal:

```
model.add((x != y) + (y != z) + (z != x) >= 2);
```

```
public IloNumVar(const IloNumCollection coll,
```



```
const char * name=0)
```

This constructor creates a constrained *discrete* numeric variable from the given collection

```
public IloNumVar(const IloNumRange coll,
                 const char * name=0)
```

This constructor creates a constrained *discrete* numeric variable from the given collection

Methods

```
public IloNumVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking numeric variable.

```
public void getPossibleValues(IloNumArray values) const
```

This member function accesses the possible values of the invoking numeric variable and puts them in the array `values`.

```
public IloNumVar::Type getType() const
```

This member function returns the type of the invoking numeric variable, indicating whether it is integer (`Int`) or floating-point (`Float`).

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking numeric variable.

```
public void setBounds(IloNum lb,
                     IloNum ub) const
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

Note: *The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this numeric variable.*

```
public void setLB(IloNum num) const
```

This member function sets `num` as the lower bound of the invoking numeric variable.

Note: *The member function `setLB` notifies Concert Technology algorithms about this change of bound in this numeric variable.*

```
public void setPossibleValues(const IloNumArray values) const
```

This member function sets `values` as the domain of the invoking discrete numeric variable. This member function can be called only on instances of `IloNumVar` that have been created with one of the two *discrete* constructors; that is, instances of `IloNumVar` which have been defined by an explicit array of discrete values.

Note: *The member function `setPossibleValues` notifies Concert Technology algorithms about this change of domain in this discrete numeric variable.*

```
public void setUB(IloNum num) const
```

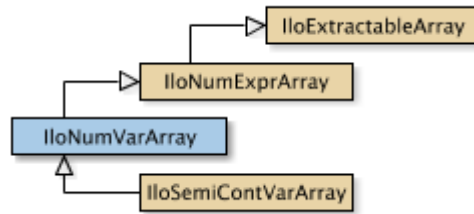
This member function sets `num` as the upper bound of the invoking numeric variable.

Note: *The member function `setUB` notifies Concert Technology algorithms about this change of bound in this numeric variable.*

IloNumVarArray

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Summary The array class of IloNumVar.

Constructor Summary	
public	IloNumVarArray (IloDefaultArrayI * i=0)
public	IloNumVarArray (const IloEnv env, IloInt n=0)
public	IloNumVarArray (const IloEnv env, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray (const IloEnv env, IloNum lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray (const IloEnv env, const IloNumArray lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray (const IloEnv env, IloInt n, IloNum lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray (const IloEnv env, const IloNumColumnArray columnarray, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray (const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)

Method Summary	
public void	add(IloInt more, const IloNumVar x)
public void	add(const IloNumVar x)
public void	add(const IloNumVarArray array)
public void	endElements()
public IloNumExprArg	operator()(IloNumExprArg anIntegerExpr) const
public IloNumExprArg	operator[](IloIntExprArg anIntegerExpr) const
public void	setBounds(const IloNumArray lb, const IloNumArray ub)
public IloIntExprArray	toIntExprArray() const
public IloIntVarArray	toIntVarArray() const
public IloNumExprArray	toNumExprArray() const

Inherited methods from IloNumExprArray
IloNumExprArray::add , IloNumExprArray::add , IloNumExprArray::add , IloNumExprArray::endElements , IloNumExprArray::operator() , IloNumExprArray::operator[]

Inherited methods from IloExtractableArray
IloExtractableArray::add , IloExtractableArray::add , IloExtractableArray::add , IloExtractableArray::endElements

Description

For each basic type, Concert Technology defines a corresponding array class. `IloNumVarArray` is the array class of the numeric variable class for a model.

Instances of `IloNumVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also

[IloAllDiff](#), [IloModel](#), [IloNumVar](#), [operator>>](#), [operator](#)

Constructors

public `IloNumVarArray`(`IloDefaultArrayI * i=0`)

This constructor creates an empty extensible array of numeric variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) optionally or to use 0 (zero) as a default value of a parameter in a constructor.

```
public IloNumVarArray(const IloEnv env,
                     IloInt n=0)
```

This constructor creates an extensible array of `n` numeric variables in `env`. Initially, the `n` elements are empty handles.

```
public IloNumVarArray(const IloEnv env,
                     const IloNumArray lb,
                     const IloNumArray ub,
                     IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the array `lb` must be the same as the length of the array `ub`. In other words, `lb.getSize() == ub.getSize()`. This constructor will construct an array with the number of elements in the array `ub`.

```
public IloNumVarArray(const IloEnv env,
                     IloNum lb,
                     const IloNumArray ub,
                     IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the new array will be the same as the length of the array `ub`.

```
public IloNumVarArray(const IloEnv env,
                     const IloNumArray lb,
                     IloNum ub,
                     IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env,
                     IloInt n,
                     IloNum lb,
                     IloNum ub,
                     IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of `n` numeric variables in `env` with lower and upper bounds and type as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env,
                     const IloNumColumnArray columnarray,
```

```
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables with `type` as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env,
                    const IloNumColumnArray columnarray,
                    const IloNumArray lb,
                    const IloNumArray ub,
                    IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables with lower and upper bounds and `type` as indicated. The instances of `IloNumVar` to fill this array are constructed at the same time.

Methods

```
public void add(IloInt more,
              const IloNumVar x)
```

This member function appends `x` to the invoking array `multiple` times. The parameter `more` indicates how many times.

```
public void add(const IloNumVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloNumVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloNumExprArg operator()(IloNumExprArg anIntegerExpr) const
```

This subscripting operator is deprecated. It must be replaced by the operator `IloNumExprArg operator[] (IloIntExprArg anIntegerExpr) const`;

```
public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element constraint.

```
public void setBounds(const IloNumArray lb,
                    const IloNumArray ub)
```

For each element in the invoking array, this member function sets `lb` as the lower bound and `ub` as the upper bound of the corresponding numeric variable in the invoking array.

Note: *The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this array of numeric variables.*

```
public IloIntExprArray toIntExprArray() const
```

This member function copies the invoking array to a new `IloIntExprArray`, checking the type of the variables during the copy.

```
public IloIntVarArray toIntVarArray() const
```

This member function copies the invoking array to a new `IloIntVarArray`, checking the type of the variables during the copy.

```
public IloNumExprArray toNumExprArray() const
```

This member function copies the invoking array to a new `IloNumExprArray`, checking the type of the variables during the copy.

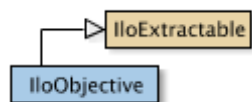
IloNumVar::Type

Category	Inner Enumeration
Definition File	ilconcert/iloexpression.h
Synopsis	<pre> Type{ Int, Float, Bool }; </pre>
Description	This nested enumeration enables you to indicate whether an instance of IloNumVar is of type integer (<code>Int</code>), Boolean (<code>Bool</code>), or floating-point (<code>Float</code>). Programming Hint For each enumerated value in <code>IloNumVar::Type</code>, there is a predefined equivalent C++ <code>#define</code> in the Concert Technology include files to make programming easier. For example, instead of writing <code>IloNumVar::Int</code> in your application, you can write <code>ILOINT</code>. Likewise, <code>ILOFLOAT</code> is defined for <code>IloNumVar::Float</code> and <code>ILOBOOL</code> for <code>IloNumVar::Bool</code>.
See Also	IloNumVar
Fields	<pre> Int Float Bool </pre>

IloObjective

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary

public	IloObjective()
public	IloObjective(IloObjectiveI * impl)
public	IloObjective(const IloEnv env, IloNum constant=0.0, IloObjective::Sense sense=Minimize, const char * name=0)
public	IloObjective(const IloEnv env, const IloNumExprArg expr, IloObjective::Sense sense=Minimize, const char * name=0)

Method Summary

public IloNum	getConstant() const
public IloNumExprArg	getExpr() const
public IloObjectiveI *	getImpl() const
public IloObjective::Sense	getSense() const
public IloAddValueToObj	IloObjective::operator()(IloNum value)
public IloAddValueToObj	operator>()
public void	setConstant(IloNum constant)
public void	setExpr(const IloNumExprArg)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
public void	setSense(IloObjective::Sense sense)

Inherited methods from **IloExtractable**

```
IloExtractable::end, IloExtractable::getEnv, IloExtractable::getId,
IloExtractable::getImpl, IloExtractable::getName,
IloExtractable::getObject, IloExtractable::setName,
IloExtractable::setObject
```

Inner Enumeration

```
IloObjective::Sense
```

Description

An instance of this class is an objective in a model. An objective consists of its `sense` (indicating whether it is a minimization or maximization) and an expression. The expression may be a constant.

An objective belongs to the environment that the variables in its expression belong to. Generally, you will create an objective, add it to a model, and extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the objective (an instance of `IloObjective`) will be extracted when an algorithm such as `IloCplex`, documented in the `ILOG CPLEX Reference Manual`, extracts the objective.

Multiple Objectives

You may create more than one objective in a model, for example, by creating more than one group. However, certain algorithms, such as an instance of `IloCplex`, will throw an exception (on a platform that supports C++ exceptions, when exceptions are enabled) if you attempt to extract more than one objective at a time.

Also see the functions `IloMaximize` and `IloMinimize` for “short cuts” to create objectives.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, $x + 3y$ is a linear expression of two terms consisting of two variables. In some linear expressions, a given variable may appear in more than one term, for example, $x + 3y + 2x$. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function [setNormalizer](#).

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing for certain member functions of this class in case of duplicates.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume no duplicates. You set this mode with the member function [setNormalizer](#).

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also [IloMaximize](#), [IloMinimize](#), [IloModel](#), [IloObjective::Sense](#)

Constructors

```
public IloObjective()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloObjective(IloObjectiveI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloObjective(const IloEnv env,
                  IloNum constant=0.0,
                  IloObjective::Sense sense=Minimize,
                  const char * name=0)
```

This constructor creates an objective consisting of a `constant` and belonging to `env`. The sense of the objective (whether it is a minimization or maximization) is indicated by `sense`; by default, it is a minimization. You may supply a `name` for the objective; by default, its `name` is the empty string. This constructor is useful when you want to create an empty objective and fill it later by column-wise modeling.

```
public IloObjective(const IloEnv env,
                   const IloNumExprArg expr,
                   IloObjective::Sense sense=Minimize,
                   const char * name=0)
```

This constructor creates an objective to add to a model from `expr`.

After you create an objective from an expression with this constructor, you must use the member function `add` explicitly to add your objective to your model or to a group in order for the objective to be taken into account.

Note: *When it accepts an expression as a parameter, this constructor checks the setting of `setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.*

Methods

```
public IloNum getConstant() const
```

This member function returns the constant term from the expression of the invoking objective.

```
public IloNumExprArg getExpr() const
```

This member function returns the expression of the invoking `IloObjective` object.

```
public IloObjectiveI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloObjective::Sense getSense() const
```

This member function returns the sense of the invoking objective, indicating whether the objective is a minimization (`Minimize`) or a maximization (`Maximize`).

```
public IloAddValueToObj operator()(IloNum value)
```

This casting operator uses a floating-point `value` to create an instance of `IloAddNumVar` or one of its subclasses and to add that `value` to that instance. See the concept `Column-Wise Modeling` for an explanation of how to use this operator in column-wise modeling.

```
public IloAddValueToObj operator>()
```

This casting operator uses a floating-point `value` to create an instance of `IloAddNumVar` or one of its subclasses and to add that `value` to that instance. If no argument is given, it assumes 1.0. See the concept `Column-Wise Modeling` for an explanation of how to use this operator in column-wise modeling.

```
public void setConstant(IloNum constant)
```

This member function sets `constant` as the constant term in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setConstant` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void setExpr(const IloNumExprArg)
```

This member function sets the expression of the invoking `IloObjective` object.

```
public void setLinearCoef(const IloNumVar var,  
                          IloNum value)
```

This member function sets `value` as the linear coefficient of the variable `var` in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.*

If you attempt to use `setLinearCoef` on a nonlinear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars,  
                          const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.*

If you attempt to use `setLinearCoef` on a non linear expression, Concert Technology will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setSense(IloObjective::Sense sense)
```

This member function sets `sense` to indicate whether the invoking objective is a maximization (`Maximize`) or minimization (`Minimize`), and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setSense` notifies Concert Technology algorithms about this change of this invoking object.*

IloObjective::Sense

Category Inner Enumeration

Definition File `ilconcert/ilolinear.h`

Synopsis

```
Sense{
    Minimize,
    Maximize
};
```

Description An instance of the class [IloObjective](#) represents an objective in a model. This nested enumeration is limited in scope to that class, and its values indicate the sense of an objective—whether it is a minimization (`Minimize`) or a maximization (`Maximize`).

See Also [IloObjective](#)

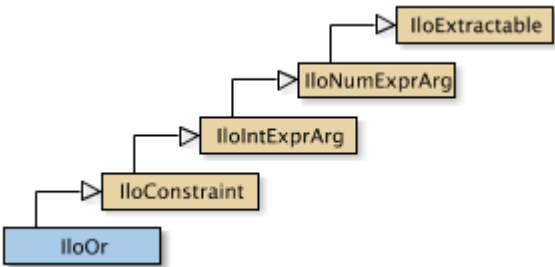
Fields

```
Minimize
Maximize
```

IloOr

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloOr()
public	IloOr(IloOrI * impl)
public	IloOr(const IloEnv env,const char * name=0)

Method Summary	
public void	add(const IloConstraintArray cons) const
public void	add(const IloConstraint con) const
public IloOrI *	getImpl() const
public void	remove(const IloConstraintArray cons) const
public void	remove(const IloConstraint con) const

Inherited methods from IloConstraint
IloConstraint::getImpl

Inherited methods from [IloIntExprArg](#)

[IloIntExprArg::getImpl](#)

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of `IloOr` represents a disjunctive constraint. In other words, it defines a disjunctive-OR among any number of constraints. Since an instance of `IloOr` is a constraint itself, you can build up extensive disjunctions by adding constraints to an instance of `IloOr` by means of the member function [add](#). You can also remove constraints from an instance of `IloOr` by means of the member function [remove](#).

The elements of a disjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

Disjunctive Goals

If you would like to represent a disjunctive-OR as a goal (rather than a constraint), then you should consider the function `IloOrGoal`, documented in the [ILOG Solver Reference Manual](#).

What Is Extracted

All the constraints (that is, instances of [IloConstraint](#) or one of its subclasses) that have been added to a disjunctive constraint (an instance of `IloOr`) and that have not

been removed from it will be extracted when an algorithm such as `IloSolver` (documented in the ILOG Solver Reference Manual) extracts the constraint.

Example

For example, you may write:

```
IloOr or(env);
or.add(constraint1);
or.add(constraint2);
or.add(constraint3);
```

Those lines are equivalent to :

```
IloOr or = constraint1 || constraint2 || constraint3;
```

See Also [IloAnd](#), [IloConstraint](#), [operator ||](#)

Constructors

```
public IloOr()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloOr(IloOrI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloOr(const IloEnv env,
            const char * name=0)
```

This constructor creates a disjunctive constraint for use in `env`. The optional argument `name` is set to `0` by default.

Methods

```
public void add(const IloConstraintArray cons) const
```

This member function makes all the elements in `array` elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to all the elements of `array`.

Note: *The member function `add` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void add(const IloConstraint con) const
```

This member function makes `constraint` one of the elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to `constraint`.

Note: *The member function `add` notifies Concert Technology algorithms about this change of this invoking object.*

```
public IloOrI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloConstraintArray cons) const
```

This member function removes all the elements of `array` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to any of those elements.

Note: *The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void remove(const IloConstraint con) const
```

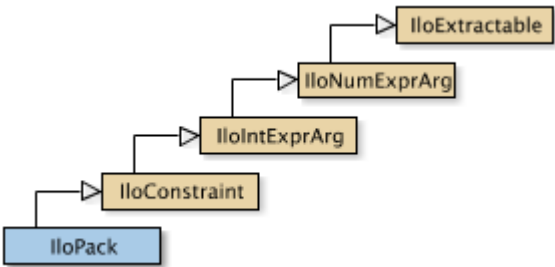
This member function removes `constraint` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to `constraint`.

Note: *The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.*

IloPack

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloPack()
public	IloPack(IloPackI * impl)
public	IloPack(const IloEnv env,const IloIntExprArray load,const IloIntExprArray where,const IloIntArray weight,const char * name=0)
public	IloPack(const IloEnv env,const IloIntExprArray load,const IloIntExprArray where,const IloIntArray weight,const IloIntExpr used,const char * name=0)
public	IloPack(const IloEnv env,const IloIntExprArray load,const IloIntExprArray where,const IloIntArray weight,const IloIntSetVar used,const char * name=0)

Method Summary	
public IloPackI *	getImpl() const

Inherited methods from [IloConstraint](#)[IloConstraint::getImpl](#)**Inherited methods from [IloIntExprArg](#)**[IloIntExprArg::getImpl](#)**Inherited methods from [IloNumExprArg](#)**[IloNumExprArg::getImpl](#)**Inherited methods from [IloExtractable](#)**

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

The `IloPack` constraint maintains the load of a set of containers or bins, given a set of weighted items and an assignment of items to containers.

Consider that we have n items and m containers. Each item i has an integer weight $w[i]$ and a constrained integer variable $p[i]$ associated with it, indicating in which container (numbered contiguously from 0) item i is to be placed. No item can be split up, and so an item can go in only one container. Associated with each container j is an integer variable $l[j]$ representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed. Finally, the number, or indeed the set of containers used can be specified. A container is used if at least one item is placed in the container in question.

Constructors

```
public IloPack()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloPack(IloPackI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloPack(const IloEnv env,  
               const IloIntExprArray load,  
               const IloIntExprArray where,  
               const IloIntArray weight,  
               const char * name=0)
```

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers.

should be placed in, counting from 0. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

Parameters :

•env

The environment used for allocating the constraint.

•load

The array of loads, one for each container.

•where

For each item, which container it should be placed in, counting from 0.

•weight

An array of non-negative weights, one for each item. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

•name

The name of the constraint.

```
public IloPack(const IloEnv env,  
               const IloIntExprArray load,  
               const IloIntExprArray where,  
               const IloIntArray weight,  
               const IloIntExpr used,  
               const char * name=0)
```

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers. The number of containers used is also maintained.

should be placed in, counting from 0. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

Parameters :

•env

The environment used for allocating the constraint.

- load

The array of loads, one for each container.

- where

For each item, which container it should be placed in, counting from 0.

- weight

An array of non-negative weights, one for each item. **where** and **weight** must be the same size, otherwise an instance of **IloException** is thrown.

- used

The number of containers used.

- name

The name of the constraint.

```
public IloPack(const IloEnv env,
               const IloIntExprArray load,
               const IloIntExprArray where,
               const IloIntArray weight,
               const IloIntSetVar used,
               const char * name=0)
```

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers. The set of containers used is also maintained.

should be placed in, counting from 0. **where** and **weight** must be the same size, otherwise an instance of **IloException** is thrown. can be obtained from the set by taking the cardinality of the set.

See Also

IloCard

Parameters :

- env

The environment used for allocating the constraint.

See Also

IloCard

- load

The array of loads, one for each container.

See Also

IloCard

- where

For each item, which container it should be placed in, counting from 0.

See Also

IloCard

•weight

An array of non-negative weights, one for each item. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

See Also

IloCard

•used

The set of containers used. The number of containers used can be obtained from the set by taking the cardinality of the set.

See Also

IloCard

•name

The name of the constraint.

See Also

IloCard

Methods

```
public IloPackI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloPartition

Category Global Function

Definition File ilconcert/iloanysset.h

Synopsis

```
public IloConstraint IloPartition(const IloEnv env,
                                const IloAnySetVarArray vars)
public IloConstraint IloPartition(const IloEnv env,
                                const IloAnySetVarArray vars,
                                const IloAnyArray vals)
public IloConstraint IloPartition(const IloEnv,
                                const IloIntSetVarArray vars)
public IloConstraint IloPartition(const IloEnv,
                                const IloIntSetVarArray vars,
                                const IloIntArray vals)
```

Description This function creates and returns a constraint. When that constraint is posted, it insures that each value of the array `vals` will be required by exactly one set variable of the array `vars`.

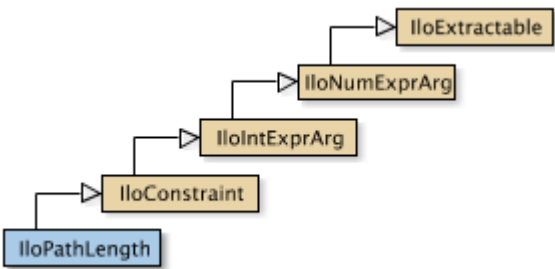
If the argument `vals` is not mentioned, the array `vals` is formed by the union of the values involved in the set variables of `vars`.

In this context, a constraint will be posted after it has been added to a model and extracted by a solver (for example, an instance of `IloSolver` documented in the *ILOG Solver Reference Manual*).

IloPathLength

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary	
public	IloPathLength()
public	IloPathLength(IloPathLengthI * impl)
public	IloPathLength(const IloEnv env,const IloIntVarArray next,const IloNumVarArray cumul,IloPathTransitFunction transit,IloInt nbPaths=1,const char * name=0)
public	IloPathLength(const IloEnv env,const IloIntVarArray next,const IloIntVarArray cumul,IloPathTransitFunction transit,IloInt nbPaths=1,const char * name=0)
public	IloPathLength(const IloEnv env,const IloIntVarArray next,const IloNumVarArray cumul,IloPathTransitI * pathTransit,IloInt nbPaths=1,const char * name=0)
public	IloPathLength(const IloEnv env,const IloIntVarArray next,const IloIntVarArray cumul,IloPathTransitI * pathTransit,IloInt nbPaths=1,const char * name=0)

Method Summary

public IloPathLengthI *	getImpl() const

Inherited methods from IloConstraint
IloConstraint::getImpl

Inherited methods from IloIntExprArg
IloIntExprArg::getImpl

Inherited methods from IloNumExprArg
IloNumExprArg::getImpl

Inherited methods from IloExtractable
IloExtractable::end , IloExtractable::getEnv , IloExtractable::getId , IloExtractable::getImpl , IloExtractable::getName , IloExtractable::getObject , IloExtractable::setName , IloExtractable::setObject

Description An instance of this class is a path constraint in Concert Technology.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

What IloPathLength Does Not Do

The path-length constraint does not determine whether there is a path between nodes in a graph; rather, it constrains accumulations (such as flow) along a path. The filtering algorithm associated with this constraint works on the accumulation variables in the array lengths.

If you are looking for a Hamiltonian path, for example, (that is, one in which each node is visited exactly once), consider using instead the constraint [IloAllDiff](#) on the variables in the array `next`.

What IloPathLength Does

If we are given

- ◆ a set of n nodes, known as N ,
- ◆ a maximum number of paths among those nodes, `maxNbPaths`,
- ◆ a set of `maxNbPaths` nodes, known as S , for starting nodes,
- ◆ a set of `maxNbPaths` nodes, known as E , for ending nodes,

then a path constraint insures that there exist at most `maxNbPaths` paths starting from a node in S , visiting nodes in N , and ending at a node in E . Furthermore, each node will be *visited* only once, has only one predecessor and only one successor, and each node *belongs* to a path that starts from a node in S and ends at a node in E .

In particular, in an instance of `IloPath`, in the arrays `next` and `cumul`,

- ◆ the indices in $[0, n-1]$ correspond to the nodes of N ,
- ◆ the indices in $[n, n+\text{maxNbPaths}-1]$ correspond to the nodes of E ,
- ◆ and the indices in $[n+\text{maxNbPaths}, n+2*\text{maxNbPaths}-1]$ correspond to the nodes of S .

In other words, the size of `next` and `cumul` is $n+2*\text{maxNbPaths}$.

`next[i]` is the node following node i on the current path. `cumul[i]` is the accumulated cost from the beginning of the path to node i . The argument `transit` indicates the *transition function*.

When this constraint is satisfied, it insures that for all indices i in the range $[0, n-1]$ or in $[n+\text{maxNbPaths}, n+2*\text{maxNbPaths}-1]$, if `next[i]==j` and j is in $[0, n+\text{maxNbPaths}-1]$, then `cumul[i] + transit.transit(i,j) <= cumul[j]`.

When i is in the range $[n, n+\text{maxNbPaths}-1]$, `next[i]` has no meaning because the nodes in E do not have successors, of course. In this case, the constraint deals with them by setting `next[i]` to $i+\text{maxNbPaths}$ (that is, nodes of S).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also

[IloAllDiff](#), [IloConstraint](#), [IloPathTransitFunction](#), [IloPathTransitI](#)

Constructors

```
public IloPathLength()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloPathLength(IloPathLengthI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloPathLength(const IloEnv env,
                    const IloIntVarArray next,
                    const IloNumVarArray cumul,
                    IloPathTransitFunction transit,
                    IloInt nbPaths=1,
                    const char * name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception [InvalidArraysException](#).

```
public IloPathLength(const IloEnv env,
                    const IloIntVarArray next,
                    const IloIntVarArray cumul,
                    IloPathTransitFunction transit,
                    IloInt nbPaths=1,
                    const char * name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception [InvalidArraysException](#).

```
public IloPathLength(const IloEnv env,
                    const IloIntVarArray next,
                    const IloNumVarArray cumul,
                    IloPathTransitI * pathTransit,
                    IloInt nbPaths=1,
                    const char * name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

```
public IloPathLength(const IloEnv env,
                    const IloIntVarArray next,
                    const IloIntVarArray cumul,
                    IloPathTransitI * pathTransit,
                    IloInt nbPaths=1,
                    const char * name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

Methods

```
public IloPathLengthI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloPathTransitFunction

Category	Type Definition
Definition File	ilconcert/ilomodel.h
Synopsis	<code>IloNum(* IloPathTransitFunction)(IloInt i, IloInt j)</code>
Description	This C++ type definition represents a pointer to a function that takes two arguments and returns a floating-point number. The two arguments are the indices of nodes. The function should return a <i>transit cost</i> for connecting the two nodes. This transit cost can be the distance between the nodes or the cost of a path visiting either of the nodes. This kind of function is known as a <i>transit function</i>.
See Also	IloPathLength , IloPathTransitI

IloPathTransitI

Category Class

InheritancePath

IloPathTransitI

Definition File ilconcert/ilomodel.h

Constructor Summary

public	IloPathTransitI()

Method Summary

public virtual IloPathTransitI *	makeClone (IloEnvI *) const
public virtual IloNum	transit (IloInt i, IloInt j)

Description You can define the transit function in a path constraint (the cost for linking two nodes together).

This class is the implementation class for **IloPathTransit**, the class of object that defines a transit function for the path constraint. The virtual member function **transit** in **IlcPathTransitI** returns the transition cost for connecting two nodes together.

To express new transit functions, you can define a subclass of **IlcPathTransitI**. If this transition can be expressed by an evaluation function, then you can use the predefined **IloPathFunction** for that purpose.

Most member functions in this class contain **assert** statements. For an explanation of the macro **NDEBUG** (a way to turn on or turn off these **assert** statements), see the concept **Assert** and **NDEBUG**.

See Also [IloPathLength](#), [IloPathTransitFunction](#)

Constructors public **IloPathTransitI**()

This constructor creates an implementation of a transit function.

Methods

```
public virtual IloPathTransitI * makeClone(IloEnvI *) const
```

This virtual member function returns a copy of the invoking object.

```
public virtual IloNum transit(IloInt i,  
                               IloInt j)
```

This virtual member function returns the transition cost from node *i* to node *j*. Its default implementation returns 0 (zero) as the value of every transition.

IloPi

Category Macro

Synopsis **IloPi()**

Description Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloPi;          // = 3.14159265358979323846
```

IloPiecewiseLinear

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node,
    const IloNumArray point,
    const IloNumArray slope,
    IloNum a,
    IloNum fa)
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node,
    IloNum firstSlope,
    const IloNumArray point,
    const IloNumArray value,
    IloNum lastSlope)
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node,
    const IloNumArray point,
    const IloNumArray value)
```

Description

The function `IloPiecewiseLinear` creates an expression node to represent a continuous or discontinuous piecewise linear function f of the variable x . The array `point` contains the n breakpoints of the function such that $point[i-1] \leq point[i]$ for $i = 1, \dots, n-1$. The array `slope` contains the $n+1$ slopes of the $n+1$ segments of the function. The values `a` and `fa` must be coordinates of a point such that $fa = f(a)$.

When $point[i-1] = point[i]$, there is a step at the x -coordinate $point[i-1]$ and its height is $slope[i-1]$.

Example

The expression

```
IloPiecewiseLinear(x, IloNumArray(env, 2, 10., 20.),
    IloNumArray(env, 3, 0.3, 1., 2.),
    0, 0);
```

defines a piecewise linear function f having two breakpoints at $x = 10$ and $x = 20$, and three segments; their slopes are 0.3 , 1 , and 2 . The first segment has infinite length and ends at the point $(x = 10, f(x) = 3)$ since $f(0) = 0$. The second segment starts at the point $(x = 10, f(x) = 3)$ and ends at the point $(x = 20, f(x) = 13)$, where the third segment starts.

IloPower

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloPower(const IloNumExprArg x,  
                             const IloNumExprArg y)  
public IloNumExprArg IloPower(const IloNumExprArg x,  
                             IloNum y)  
public IloNumExprArg IloPower(IloNum x,  
                             const IloNumExprArg y)
```

Description Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

IloPower returns the result of raising its **base** argument to the power of its **exponent** argument, that is, **base^{exponent}**. If **base** is a floating-point value or variable, then **exponent** must be greater than or equal to 0 (zero).

IloQuarterPi

Category Macro

Synopsis **IloQuarterPi()**

Description Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloQuarterPi;    // = 0.78539816339744830962
```

IloRandom

Category Class

InheritancePath

IloRandom

Definition File ilconcert/ilorandom.h

Constructor Summary

public	IloRandom()
public	IloRandom (const IloEnv env, IloInt seed=0)
public	IloRandom (IloRandomI * impl)
public	IloRandom (const IloRandom & rand)

Method Summary

public void	end()
public IloEnv	getEnv() const
public IloNum	getFloat() const
public IloRandomI *	getImpl() const
public IloInt	getInt (IloInt n) const
public const char *	getName() const
public IloAny	getObject() const
public void	reSeed (IloInt seed)
public void	setName (const char * name) const
public void	setObject (IloAny obj) const

Description

This handle class produces streams of pseudo-random numbers. You can use objects of this class to create a search with a random element. You can create any number of instances of this class; these instances insure reproducible results in multithreaded applications, where the use of a single source for random numbers creates problems.

See [IloRandomize](#) in the *ILOG Solver Reference Manual*.

Constructors

public **IloRandom**()

This constructor creates a random number generator; it is initially an empty handle. You must assign this handle before you use its member functions.

```
public IloRandom(const IloEnv env,
                 IloInt seed=0)
```

This constructor creates an object that generates random numbers. You can seed the generator by supplying a value for the integer argument `seed`.

```
public IloRandom(IloRandomI * impl)
```

This constructor creates a handle object (an instance of the class `IloRandom`) from a pointer to an implementation object (an instance of the class `IloRandomI`).

```
public IloRandom(const IloRandom & rand)
```

This constructor creates a handle object from a reference to a random number generator. After execution, both the newly constructed handle and `rand` point to the same implementation object.

Methods

```
public void end()
```

This member function releases all memory used by the random number generator. After a call to this member function, you should not use the generator again.

```
public IloEnv getEnv() const
```

This member function returns the environment associated with the implementation class of the invoking generator.

```
public IloNum getFloat() const
```

This member function returns a floating-point number drawn uniformly from the interval $[0..1)$.

```
public IloRandomI * getImpl() const
```

This member function returns the implementation object of the invoking handle.

```
public IloInt getInt(IloInt n) const
```

This member function returns an integer drawn uniformly from the interval $[0..n)$.

```
public const char * getName() const
```

This member function returns a character string indicating the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public void reSeed(IloInt seed)
```

This member function re-seeds the random number generator with `seed`.

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

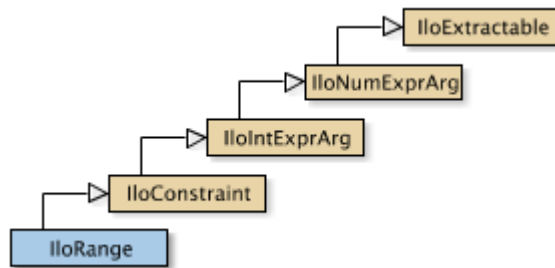
```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterwards. Normally, `obj` contains user data pertinent to the invoking object.

IloRange

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary

public	IloRange()
public	IloRange(IloRangeI * impl)
public	IloRange(const IloEnv env, IloNum lb, IloNum ub, const char * name=0)
public	IloRange(const IloEnv env, IloNum lhs, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)
public	IloRange(const IloEnv env, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)

Method Summary

public IloNumExprArg	getExpr() const
public IloRangeI *	getImpl() const
public IloNum	getLB() const
public IloNum	getUB() const
public IloAddValueToRange	operator()(IloNum value) const
public void	setBounds(IloNum lb, IloNum ub)
public void	setExpr(const IloNumExprArg expr)
public void	setLB(IloNum lb)

public void	<code>setLinearCoef</code> (const <code>IloNumVar</code> var, <code>IloNum</code> value)
public void	<code>setLinearCoefs</code> (const <code>IloNumVarArray</code> vars, const <code>IloNumArray</code> values)
public void	<code>setUB</code> (<code>IloNum</code> ub)

Inherited methods from `IloConstraint`

`IloConstraint::getImpl`

Inherited methods from `IloIntExprArg`

`IloIntExprArg::getImpl`

Inherited methods from `IloNumExprArg`

`IloNumExprArg::getImpl`

Inherited methods from `IloExtractable`

`IloExtractable::end`, `IloExtractable::getEnv`, `IloExtractable::getId`,
`IloExtractable::getImpl`, `IloExtractable::getName`,
`IloExtractable::getObject`, `IloExtractable::setName`,
`IloExtractable::setObject`

Description

An instance of this class is a range in a model, that is, a constraint of the form:

`lowerBound <= expression <= upperBound`

You can create a range from the constructors in this class or from the arithmetic operators on numeric variables (instances of `IloNumVar` and its subclasses) or on expressions (instances of `IloExpr` and its subclasses):

◆ operator `<=`

- ◆ operator >=
- ◆ operator ==

After you create a constraint, such as an instance of `IloRange`, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` to add the range to a model and the member function `IloAlgorithm::extract` to extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the range (an instance of `IloRange`) will be extracted when an algorithm such as `IloCplex`, documented in the ILOG CPLEX Reference Manual, extracts the range.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, $x + 3y$. In some linear expressions, a given variable may appear in more than one term, for example, $x + 3y + 2x$. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes linear expressions that your application passes it, and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function `setNormalizer(IloTrue)`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion in some of the member functions of this class failing in the case of duplicates.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `setNormalizer(IloFalse)`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also [IloConstraint](#), [IloRangeArray](#)

Constructors

```
public IloRange()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloRange(IloRangeI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloRange(const IloEnv env,
               IloNum lb,
               IloNum ub,
               const char * name=0)
```

This constructor creates an empty range constraint. Before you use this constraint, you must fill the range. The optional argument `name` is set to `0` by default.

After you create a range constraint, you must explicitly add it to a model in order for it to be taken into account. To do so, use the member function [IloModel::add](#).

```
public IloRange(const IloEnv env,
               IloNum lhs,
               const IloNumExprArg expr,
               IloNum rhs=IloInfinity,
               const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class [IloNumExprArg](#)) and its upper bound (`rhs`). The default bound for `rhs` is the symbolic constant `IloInfinity`. The optional argument `name` is set to `0` by default.

Note: When it accepts an expression as a parameter, this constructor checks the setting of [setNormalizer](#) to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

```
public IloRange(const IloEnv env,
               const IloNumExprArg expr,
               IloNum rhs=IloInfinity,
               const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class [IloNumExprArg](#)) and its upper bound (`rhs`). Its lower bound (`lhs`) will be `-IloInfinity`. The default bound for `rhs` is `IloInfinity`. The optional argument `name` is set to `0` by default.

Note: When it accepts an expression as a parameter, this constructor checks the setting of `setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

Methods

```
public IloNumExprArg getExpr() const
```

This member function returns the expression of the invoking `IloRange` object.

```
public IloRangeI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking range.

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking range.

```
public IloAddValueToRange operator()(IloNum value) const
```

This operator creates the objects needed internally to represent a range in column-wise modeling. See the concept `Column-Wise Modeling` for an explanation of how to use this operator in column-wise modeling.

```
public void setBounds(IloNum lb,
                     IloNum ub)
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: The member function `setBounds` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setExpr(const IloNumExprArg expr)
```

This member function sets `expr` as the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

```
public void setLB(IloNum lb)
```

This member function sets `lb` as the lower bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setLB` notifies Concert Technology algorithms about this change of this invoking object.*

```
public void setLinearCoef(const IloNumVar var,
                          IloNum value)
```

This member function sets `value` as the linear coefficient of the variable `var` in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.*

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars,
                           const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.*

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setUB(IloNum ub)
```

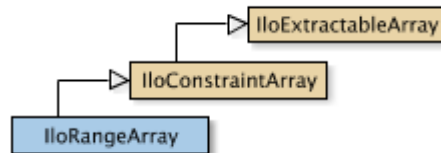
This member function sets `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note: *The member function `setUB` notifies Concert Technology algorithms about this change of this invoking object.*

IloRangeArray

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary	
public	<code>IloRangeArray(IloDefaultArrayI * i=0)</code>
public	<code>IloRangeArray(const IloEnv env, IloInt n=0)</code>
public	<code>IloRangeArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub)</code>
public	<code>IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, const IloNumArray ub)</code>
public	<code>IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloNumArray ub)</code>
public	<code>IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, IloNum ub)</code>
public	<code>IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, IloNum ub)</code>
public	<code>IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, const IloIntArray ub)</code>
public	<code>IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloIntArray ub)</code>
public	<code>IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, IloNum ub)</code>
public	<code>IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumArray ub)</code>

public	IloRangeArray (const IloEnv env, const IloIntArray lbs, const IloIntArray ub)
public	IloRangeArray (const IloEnv env, IloNum lb, const IloNumArray ub)
public	IloRangeArray (const IloEnv env, const IloNumArray lbs, IloNum ub)
public	IloRangeArray (const IloEnv env, IloNum lb, const IloIntArray ub)
public	IloRangeArray (const IloEnv env, const IloIntArray lbs, IloNum ub)

Method Summary	
public void	add (IloInt more, const IloRange range)
public void	add (const IloRange range)
public void	add (const IloRangeArray array)
public IloNumColumn	operator() (const IloNumArray vals)
public IloNumColumn	operator() (const IloIntArray vals)
public IloRange	operator[] (IloInt i) const
public IloRange &	operator[] (IloInt i)
public void	setBounds (const IloIntArray lbs, const IloIntArray ub)
public void	setBounds (const IloNumArray lbs, const IloNumArray ub)

Inherited methods from IloConstraintArray
add , add , add , operator[] , operator[]

Inherited methods from IloExtractableArray
IloExtractableArray::add , IloExtractableArray::add , IloExtractableArray::add , IloExtractableArray::endElements

Description

For each basic type, Concert Technology defines a corresponding array class. [IloRangeArray](#) is the array class of ranges for a model.

Instances of `IloRangeArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

For information on arrays, see the concept `Arrays`

Note: *`IloRangeArray` has access to member functions defined in the `IloArray` template.*

See Also

`IloRange`, `operator>>`, `operator`

Constructors

```
public IloRangeArray(IloDefaultArrayI * i=0)
```

This default constructor creates an empty range array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloRangeArray(const IloEnv env,
                    IloInt n=0)
```

This constructor creates an array of `n` elements, each of which is an empty handle.

```
public IloRangeArray(const IloEnv env,
                    IloInt n,
                    IloNum lb,
                    IloNum ub)
```

This constructor creates an array of `n` elements, each with the lower bound `lb` and the upper bound `ub`.

```
public IloRangeArray(const IloEnv env,
                    const IloNumArray lbs,
                    const IloNumExprArray rows,
                    const IloNumArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. It uses the corresponding elements of the arrays `lbs` and `ubs` to set the lower and upper bounds of elements in the new array. The length of `rows` must equal the length of `lbs` and `ubs`.

```
public IloRangeArray(const IloEnv env,
                    IloNum lb,
                    const IloNumExprArray rows,
                    const IloNumArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

```
public IloRangeArray(const IloEnv env,
                    const IloNumArray lbs,
                    const IloNumExprArray rows,
                    IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

```
public IloRangeArray(const IloEnv env,
                    IloNum lb,
                    const IloNumExprArray rows,
                    IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of every element in the new array will be `ub`.

```
public IloRangeArray(const IloEnv env,
                    const IloIntArray lbs,
                    const IloNumExprArray rows,
                    const IloIntArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. It uses the corresponding elements of the arrays `lbs` and `ubs` to set the lower and upper bounds of elements in the new array. The length of `rows` must equal the length of `lbs` and `ubs`.

```
public IloRangeArray(const IloEnv env,
                    IloNum lb,
                    const IloNumExprArray rows,
                    const IloIntArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

```
public IloRangeArray(const IloEnv env,
                    const IloIntArray lbs,
                    const IloNumExprArray rows,
                    IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each

element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

```
public IloRangeArray(const IloEnv env,
                    const IloNumArray lbs,
                    const IloNumArray ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env,
                    const IloIntArray lbs,
                    const IloIntArray ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env,
                    IloNum lb,
                    const IloNumArray ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env,
                    const IloNumArray lbs,
                    IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env,
                    IloNum lb,
                    const IloIntArray ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env,
                    const IloIntArray lbs,
                    IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

Methods

```
public void add(IloInt more,
               const IloRange range)
```

This member function appends `range` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloRange range)
```

This member function appends `range` to the invoking array.

```
public void add(const IloRangeArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloNumColumn operator()(const IloNumArray vals)
```

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

```
public IloNumColumn operator()(const IloIntArray vals)
```

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

```
public IloRange operator[(IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `CONST` operator

```
IloRange operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloRange & operator[(IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

```
public void setBounds(const IloIntArray lbs,
                    const IloIntArray ubs)
```

This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note: *The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.*

```
public void setBounds(const IloNumArray lbs,  
                     const IloNumArray ubs)
```

This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note: *The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.*

IloRound

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<code>public IloNum IloRound(IloNum val)</code>
Description	This function computes the nearest integer value to <code>val</code> . Halfway cases are rounded to the larger in magnitude.

Examples:

```

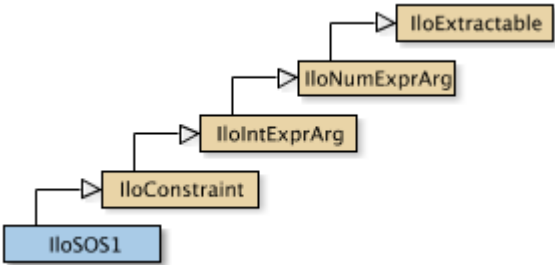
IloRound(IloInfinity) is IloInfinity.
IloRound(-IloInfinity) is -IloInfinity.
IloRound(0) is 0.
IloRound(0.4) is 0.
IloRound(-0.4) is 0.
IloRound(0.5) is 1.
IloRound(-0.5) is -1.
IloRound(0.6) is 1.
IloRound(-0.6) is -1.
IloRound(1e300) is 1e300.
IloRound(1.0000001e6) is 1e6.
IloRound(1.0000005e6) is 1.000001e6.

```

IloSOS1

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary	
public	IloSOS1()
public	IloSOS1(IloSOS1I * impl)
public	IloSOS1(const IloEnv env,const char * name=0)
public	IloSOS1(const IloEnv env,const IloNumVarArray vars,const char * name=0)
public	IloSOS1(const IloEnv env,const IloNumVarArray vars,const IloNumArray vals,const char * name=0)

Method Summary	
public IloSOS1I *	getImpl() const
public void	getNumVars(IloNumVarArray variables) const
public void	getValues(IloNumArray values) const

Inherited methods from [IloConstraint](#)

<code>IloConstraint::getImpl</code>

Inherited methods from <code>IloIntExprArg</code>

<code>IloIntExprArg::getImpl</code>

Inherited methods from <code>IloNumExprArg</code>

<code>IloNumExprArg::getImpl</code>

Inherited methods from <code>IloExtractable</code>

<code>IloExtractable::end</code> , <code>IloExtractable::getEnv</code> , <code>IloExtractable::getId</code> , <code>IloExtractable::getImpl</code> , <code>IloExtractable::getName</code> , <code>IloExtractable::getObject</code> , <code>IloExtractable::setName</code> , <code>IloExtractable::setObject</code>

Description

This handle class represents special ordered sets of type 1 (SOS1). A special ordered set of type 1 specifies a set of variables, and only one among them may take a non zero value. You may assign a weight to each variable in an SOS1. This weight indicates an order among the variables. If you do not specify any weights to indicate order among the variables, then Concert Technology considers the order in which you gave the variables to the constructor of this set and the order in which you added variables later.

When you extract a model (an instance of `IloModel`) for an instance of `IloCplex` (documented in the ILOG CPLEX Reference Manual), it will use the order for branching on variables.

For more details about SOS1, see the ILOG CPLEX Reference and User's Manuals.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also

[IloSOS1Array](#), [IloSOS2](#)

Constructors

`public IloSOS1()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSOS1(IloSOS1I * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSOS1(const IloEnv env,  
               const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). You must add the variables to this set for them to be taken into account.

```
public IloSOS1(const IloEnv env,  
               const IloNumVarArray vars,  
               const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). The set includes each of the variables specified in the array **vars**.

```
public IloSOS1(const IloEnv env,  
               const IloNumVarArray vars,  
               const IloNumArray vals,  
               const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). The set includes the variables specified in the array **vars**. The corresponding value in **vals** indicates the weight of each variable in **vars**.

Methods

```
public IloSOS1I * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getNumVars(IloNumVarArray variables) const
```

This member function accesses the variables in a special ordered set of type 1 (SOS1) and puts those variables into its parameter **variables**.

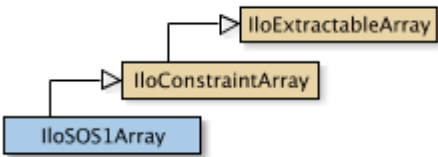
```
public void getValues(IloNumArray values) const
```

This member function accesses the weights of the variables in a special ordered set of type 1 (SOS1) and puts those weights into its parameter **values**.

IloSOS1Array

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary	
public	IloSOS1Array (IloDefaultArrayI * i=0)
public	IloSOS1Array (const IloEnv env, IloInt n=0)

Method Summary	
public void	add (IloInt more,const IloSOS1 & x)
public void	add (const IloSOS1 & x)
public void	add (const IloSOS1Array & x)
public IloSOS1	operator[] (IloInt i) const
public IloSOS1 &	operator[] (IloInt i)

Inherited methods from IloConstraintArray
add , add , add , operator[] , operator[]

Inherited methods from IloExtractableArray

```
IloExtractableArray::add, IloExtractableArray::add,
IloExtractableArray::add, IloExtractableArray::endElements
```

Description

For each basic type, Concert Technology defines a corresponding array class. `IloSOS1Array` is the array class of special ordered sets of type 1 (SOS1) for a model.

Instances of `IloSOS1Array` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

For information on arrays, see the concept `Arrays`

See Also

[IloSOS1](#), [operator>>](#), [operator](#)

Constructors

```
public IloSOS1Array(IloDefaultArrayI * i=0)
```

This default constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloSOS1Array(const IloEnv env,
                   IloInt n=0)
```

This constructor creates an array of `n` empty elements in the environment `env`.

Methods

```
public void add(IloInt more,
               const IloSOS1 & x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloSOS1 & x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloSOS1Array & x)
```

This member function appends the elements in `array` to the invoking array.

```
public IloSOS1 operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `CONST` operator

```
IloSOS1 operator[] (IloInt i) const;
```

on `const` arrays.

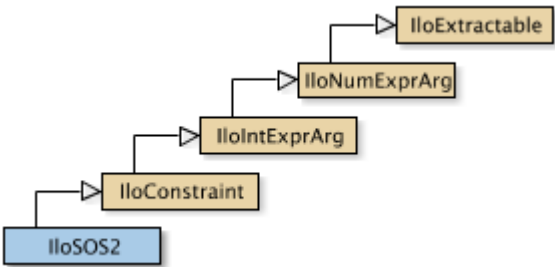
```
public IloSOS1 & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

IloSOS2

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary	
public	IloSOS2()
public	IloSOS2(IloSOS2I * impl)
public	IloSOS2(const IloEnv env,const char * name=0)
public	IloSOS2(const IloEnv env,const IloNumVarArray vars,const char * name=0)
public	IloSOS2(const IloEnv env,const IloNumVarArray vars,const IloNumArray vals,const char * name=0)

Method Summary	
public IloSOS2I *	getImpl() const
public void	getNumVars(IloNumVarArray variables) const
public void	getValues(IloNumArray values) const

Inherited methods from [IloConstraint](#)

<code>IloConstraint::getImpl</code>

Inherited methods from <code>IloIntExprArg</code>

<code>IloIntExprArg::getImpl</code>

Inherited methods from <code>IloNumExprArg</code>

<code>IloNumExprArg::getImpl</code>

Inherited methods from <code>IloExtractable</code>

<code>IloExtractable::end, IloExtractable::getEnv, IloExtractable::getId,</code> <code>IloExtractable::getImpl, IloExtractable::getName,</code> <code>IloExtractable::getObject, IloExtractable::setName,</code> <code>IloExtractable::setObject</code>

Description

This handle class represents special ordered sets of type 2 (SOS2). A special ordered set of type 2 specifies a set of variables, and only two among them may take a non zero value. These two variables must be adjacent. You may assign a weight to each variable in an SOS2. This weight indicates an order among the variables. Concert Technology asserts adjacency with respect to this assigned order. If you do not specify any weights to indicate order and adjacency among the variables, then Concert Technology considers the order in which you gave the variables to the constructor of this set and the order in which you added variables later (for example, by column generation).

When you extract a model (an instance of `IloModel`) for an instance of `IloCplex` (documented in the ILOG CPLEX Reference Manual), it will use the order of the SOS2 for branching on variables.

For more details about SOS2, see the ILOG CPLEX Reference and User's Manuals. Special ordered sets of type 2 (SOS2) commonly appear in models of piecewise linear functions. Concert Technology provides direct support for piecewise linear models in `IloPiecewiseLinear`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also [IloPiecewiseLinear](#), [IloSOS1](#), [IloSOS2Array](#)

Constructors

```
public IloSOS2()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSOS2(IloSOS2I * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSOS2(const IloEnv env,
               const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). You must add the variables to this set for them to be taken into account.

```
public IloSOS2(const IloEnv env,
               const IloNumVarArray vars,
               const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). The set includes each of the variables specified in the array `vars`.

```
public IloSOS2(const IloEnv env,
               const IloNumVarArray vars,
               const IloNumArray vals,
               const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). The set includes the variables specified in the array `vars`. The corresponding value in `vals` indicates the weight of each variable in `vars`.

Methods

```
public IloSOS2I * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getNumVars(IloNumVarArray variables) const
```

This member function accesses the variables in a special ordered set of type 2 (SOS2) and puts those variables into its parameter `variables`.

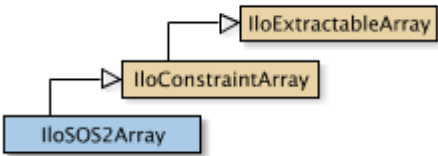
```
public void getValues(IloNumArray values) const
```

This member function accesses the weights of the variables in a special ordered set of type 2 (SOS2) and puts those weights into its parameter `values`.

IloSOS2Array

Category Class

InheritancePath



Definition File ilconcert/ilolinear.h

Constructor Summary	
public	IloSOS2Array (IloDefaultArrayI * i=0)
public	IloSOS2Array (const IloEnv env, IloInt num=0)

Method Summary	
public void	add (IloInt more,const IloSOS2 x)
public void	add (const IloSOS2 x)
public void	add (const IloSOS2Array array)
public IloSOS2	operator[] (IloInt i) const
public IloSOS2 &	operator[] (IloInt i)

Inherited methods from IloConstraintArray
add , add , add , operator[] , operator[]

Inherited methods from IloExtractableArray

```
IloExtractableArray::add, IloExtractableArray::add,
IloExtractableArray::add, IloExtractableArray::endElements
```

Description

For each basic type, Concert Technology defines a corresponding array class. `IloSOS2Array` is the array class of special ordered sets of type 2 (SOS2) for a model.

Instances of `IloSOS2Array` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

For information on arrays, see the concept `Arrays`

See Also

[IloSOS2](#), [operator>>](#), [operator](#)

Constructors

```
public IloSOS2Array(IloDefaultArrayI * i=0)
```

This default constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloSOS2Array(const IloEnv env,
                   IloInt num=0)
```

This constructor creates an array of `num` empty elements in the environment `env`.

Methods

```
public void add(IloInt more,
               const IloSOS2 x)
```

This member function appends `x` to the invoking array multiple times. The parameter `more` indicates how many times.

```
public void add(const IloSOS2 x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloSOS2Array array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloSOS2 operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`. Concert Technology uses the `CONST` operator

```
IloSOS2 operator[] (IloInt i) const;
```

on `const` arrays.

```
public IloSOS2 & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position indicated by the index `i`.

IloScalProd

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloScalProd(const IloNumArray values,
                                const IloNumVarArray vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars,
                                const IloNumArray values)
public IloNumExprArg IloScalProd(const IloNumArray values,
                                const IloIntVarArray vars)
public IloNumExprArg IloScalProd(const IloIntVarArray vars,
                                const IloNumArray values)
public IloNumExprArg IloScalProd(const IloIntArray values,
                                const IloNumVarArray vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars,
                                const IloIntArray values)
public IloNumExprArg IloScalProd(const IloNumExprArray leftExprs,
                                const IloNumExprArray rightExprs)
```

Description This function returns an instance of `IloNumExprArg`, the internal building block of an expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

IloScalProd

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloIntExprArg IloScalProd(const IloIntArray values,
                                const IloIntVarArray vars)
public IloIntExprArg IloScalProd(const IloIntVarArray vars,
                                const IloIntArray values)
public IloIntExprArg IloScalProd(const IloIntExprArray leftExprs,
                                const IloIntExprArray rightExprs)
```

Description This function returns an instance of [IloIntExprArg](#), the internal building block of an integer expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

IloScalProd

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNum IloScalProd(const IloNumArray vals1,
                          const IloNumArray vals2)
public IloNum IloScalProd(const IloIntArray vals1,
                          const IloNumArray vals2)
public IloNum IloScalProd(const IloNumArray vals1,
                          const IloIntArray vals2)
```

Description This function returns a numeric value representing the scalar product of numeric values in the arrays `vals1` and `vals2`.

IloScalProd

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloInt IloScalProd(const IloIntArray vals1,
                        const IloIntArray vals2)
```

Description This function returns an integer value representing the scalar product of integer values in the arrays `vals1` and `vals2`.

IloSemaphore

Category Class

InheritancePath



Definition File ilconcert/ilothread.h

Constructor Summary	
public	IloSemaphore (int value=0)

Method Summary	
public void	post ()
public int	tryWait ()
public void	wait ()

Description

The class `IloSemaphore` provides synchronization primitives adapted to Concert Technology. This class supports inter-thread communication in multithread applications. An instance of this class, a semaphore, is a counter; its value is always positive. This counter can be incremented or decremented. You can always increment a semaphore, and incrementing is a non-blocking operation. However, the value of the counter cannot be negative, so any thread that attempts to decrement a semaphore whose counter is already equal to 0 (zero) is blocked until another thread increments the semaphore.

See `ILOUSEMT` for details about the compilation macro to use with instances of this class.

System Class

`IloSemaphore` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloSemaphore`, differ from that pattern. `IloSemaphore` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloSemaphore` are not automatically de-allocated by a call to the member function `IloEnv::end`. You must explicitly destroy instances of `IloSemaphore` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloSemaphore` in a Concert Technology environment because the destructor for that instance of `IloSemaphore` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloSemaphore` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of `IloSemaphore`

Instances of `IloSemaphore` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloSemaphore` by calling the delete operator when your application no longer needs those objects.

See Also [IloBarrier](#), [IloCondition](#), [ILOUSEMT](#)

Constructors

```
public IloSemaphore(int value=0)
```

This constructor creates an instance of `IloSemaphore`, initializes it to `value`, and allocates it on the C++ heap (not in a Concert Technology environment). If you do not pass a `value` parameter, the constructor initializes the semaphore at 0 (zero).

Methods

```
public void post()
```

This member function increments the invoking semaphore by 1 (one). If there are threads blocked at this semaphore, then this member function wakes one of them.

```
public int tryWait()
```

This member function attempts to decrement the invoking semaphore by 1 (one). If this decrement leaves the counter positive, then the call succeeds and returns 1 (one). If the decrement would make the counter strictly negative, then the decrement does not occur, the call fails, and the member function returns 0 (zero).

```
public void wait()
```

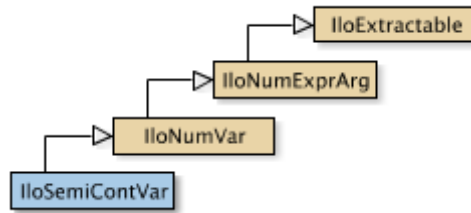
This member function decrements the invoking semaphore by 1 (one).

If this decrement would make the semaphore strictly negative, then this member function blocks the calling thread. The thread wakes up when the member function can safely decrement the semaphore without causing the counter to become negative (for example, if another entity increments the semaphore). If this member function cannot decrement the invoking semaphore, then it leads to deadlock.

IloSemiContVar

Category Class

InheritancePath



Definition File ilconcert/iloexpression.h

Constructor Summary

public	IloSemiContVar ()
public	IloSemiContVar (IloSemiContVarI * impl)
public	IloSemiContVar (const IloEnv env, IloNum sclb, IloNum ub, IloNumVar::Type type= ILOFLOAT , const char * name=0)
public	IloSemiContVar (const IloAddNumVar & var, IloNum sclb, IloNum ub, IloNumVar::Type type= ILOFLOAT , const char * name=0)

Method Summary

public IloSemiContVarI *	getImpl () const
public IloNum	getSemiContLB () const
public void	setSemiContLB (IloNum sclb) const

Inherited methods from [IloNumVar](#)

getImpl , getLB , getPossibleValues , getType , getUB , setBounds , setLB , setPossibleValues , setUB

Inherited methods from [IloNumExprArg](#)

[IloNumExprArg::getImpl](#)

Inherited methods from [IloExtractable](#)

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of this class represents a constrained semicontinuous variable in a Concert Technology model. Semicontinuous variables derive from [IloNumVar](#), the class of numeric variables.

A semicontinuous variable may be 0 (zero) or it may take a value within an interval defined by its semicontinuous lower and upper bound. Conventionally, semicontinuous variables are defined as floating-point variables, but you can designate an instance of [IloSemiContVar](#) as integer by using the type indicator it inherits from [IloNumVar](#). In that case, Concert Technology will impose an integrality constraint on the semicontinuous variable for you, thus further restricting the feasible set of values to 0 (zero) and the integer values in the interval defined by the semicontinuous lower and upper bound.

Note: When numerical bounds are given to an integer variable (an [IloIntVar](#) or an [IloNumVar](#) with `Type = Int`) in the constructors of via a modifier (`setUB`, `setLB`, `setBounds`), they are inward rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

In an instance of [IloNumVar](#), `lb` denotes the lower bound of the variable, and `ub` denotes its upper bound. In an instance of the derived class [IloSemiContVar](#), `scLB` denotes the semicontinuous lower bound.

In formal terms, if $lb \leq 0$, then a semicontinuous variable is a numeric variable with the feasible set of $\{0, [scLB, ub]\}$, where $0 < scLB < ub$; otherwise, for other values of `lb`, the feasible set of a semicontinuous variable is the intersection of the interval $[lb, ub]$ with the set $\{0, [scLB, ub]\}$. The semicontinuous lower bound `scLB` may differ from the lower bound of an ordinary numeric variable in that the semicontinuous variable is restricted to the semicontinuous region. For example, the

table below shows you the bounds of a semicontinuous variable and the corresponding feasible region.

Examples of bounds on semicontinuous variables and their feasible regions

These conditions	define these feasible regions
$lb == ub < sclb$	$\{0\}$ if $lb==ub==0$ or empty set if $lb==ub!=0$
$lb < 0 < sclb < ub$	$\{0, [sclb, ub]\}$
$0 < lb < sclb < ub$	$[sclb, ub]$
$0 < sclb < lb < ub$	$[lb, ub]$

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept [Assert and NDEBUG](#).

See Also

[IloNumVar](#)

Constructors

```
public IloSemiContVar()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSemiContVar(IloSemiContVarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSemiContVar(const IloEnv env,
                    IloNum sclb,
                    IloNum ub,
                    IloNumVar::Type type=ILOFLOAT,
                    const char * name=0)
```

This constructor creates an instance of `IloSemiContVar` from its `sclb` (that is, its semicontinuous lower bound) and its upper bound `ub`. By default, its type is floating-point, but you can use `ILOINT` to indicate integer; in that case, Concert Technology will impose an integrality constraint on the variable. The value for `lb` is set to zero.

```
public IloSemiContVar(const IloAddNumVar & var,
                    IloNum sclb,
                    IloNum ub,
                    IloNumVar::Type type=ILOFLOAT,
                    const char * name=0)
```

This constructor creates an instance of `IloSemiContVar` from the prototype `var`.

Methods

```
public IloSemiContVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getSemiContLB() const
```

This member function returns the semicontinuous lower bound (that is, its `scLb`) of the invoking semicontinuous variable.

```
public void setSemiContLB(IloNum sclb) const
```

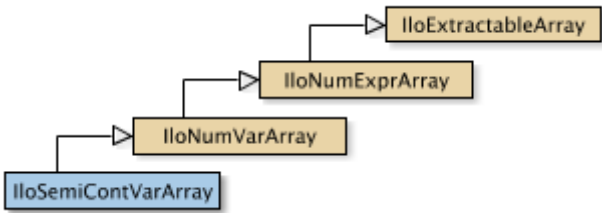
This member function makes `scLb` the semicontinuous lower bound of the invoking semicontinuous variable.

Note: *The member function `setSemiContinuousLb` notifies Concert Technology algorithms about this change of this invoking object.*

IloSemiContVarArray

Category Class

InheritancePath



Definition File iloconcert/iloexpression.h

Constructor Summary	
public	IloSemiContVarArray (IloDefaultArrayI * i=0)
public	IloSemiContVarArray (const IloEnv env)
public	IloSemiContVarArray (const IloEnv env, IloInt n)
public	IloSemiContVarArray (const IloEnv env, IloInt n, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT)
public	IloSemiContVarArray (const IloEnv env,const IloNumColumnArray columnarray,const IloNumArray sclb,const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)

Method Summary	
public void	add (IloInt more,const IloSemiContVar x)
public void	add (const IloSemiContVar x)
public void	add (const IloSemiContVarArray array)

Inherited methods from [IloNumVarArray](#)

[add](#), [add](#), [add](#), [endElements](#), [operator\(\)](#), [operator\[\]](#), [setBounds](#),
[toIntExprArray](#), [toIntVarArray](#), [toNumExprArray](#)

Inherited methods from [IloNumExprArray](#)

[IloNumExprArray::add](#), [IloNumExprArray::add](#), [IloNumExprArray::add](#),
[IloNumExprArray::endElements](#), [IloNumExprArray::operator\(\)](#),
[IloNumExprArray::operator\[\]](#)

Inherited methods from [IloExtractableArray](#)

[IloExtractableArray::add](#), [IloExtractableArray::add](#),
[IloExtractableArray::add](#), [IloExtractableArray::endElements](#)

Description

For each basic type, Concert Technology defines a corresponding array class. `IloSemiContVarArray` is the array class of the semicontinuous numeric variable class for a model.

Instances of `IloSemiContVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

See Also

[IloSemiContVar](#), [IloModel](#), [IloNumVar](#), [operator>>](#), [operator](#)

Constructors

```
public IloSemiContVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of semicontinuous numeric variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As a parameter in this default constructor, it allows you to pass 0 (zero) as a value to an optional parameter in functions and member functions that accept an array as a parameter.

```
public IloSemiContVarArray(const IloEnv env)
```


This constructor creates an extensible array of `n` semicontinuous numeric variables in `env`. Initially, the array contains zero elements.

```
public IloSemiContVarArray(const IloEnv env,
                          IloInt n)
```

This constructor creates an extensible array of `n` semicontinuous numeric variables in `env`. Initially, the `n` elements are empty handles.

```
public IloSemiContVarArray(const IloEnv env,
                          IloInt n,
                          IloNum sclb,
                          IloNum ub,
                          IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of `n` semicontinuous numeric variables in the environment `env`. Each element of the array has a semicontinuous lower bound of `sclb` and an upper bound of `ub`. The type (whether integer, Boolean, or floating-point) of each element is indicated by `type`. The default type is floating-point.

```
public IloSemiContVarArray(const IloEnv env,
                          const IloNumColumnArray columnarray,
                          const IloNumArray sclb,
                          const IloNumArray ub,
                          IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of semicontinuous numeric variables from a column array in the environment `env`. The array `sclb` indicates the corresponding semicontinuous lower bound, and the array `ub` indicates the corresponding upper bound for each new element. The parameter `type` indicates the type (whether integer, Boolean, or floating point) of each new element. The default type is floating-point.

Methods

```
public void add(IloInt more,
               const IloSemiContVar x)
```

This member function appends `x` to the invoking array `multiple` times. The parameter `more` indicates how many times.

```
public void add(const IloSemiContVar x)
```

This member function appends `x` to the invoking array.

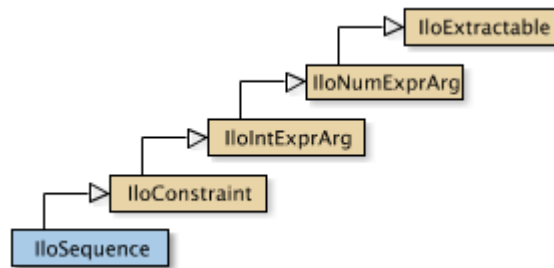
```
public void add(const IloSemiContVarArray array)
```

This member function appends the elements in `array` to the invoking array.

IloSequence

Category Class

InheritancePath



Definition File ilconcert/ilomodel.h

Constructor Summary

public	IloSequence()
public	IloSequence(IloSequenceI * impl)
public	IloSequence(const IloEnv env, IloInt nbMin, IloInt nbMax, IloInt seqWidth, const IloIntVarArray vars, const IloIntArray values, const IloIntVarArray cards, const char * name=0)

Method Summary

public IloSequenceI *	getImpl() const

Inherited methods from [IloConstraint](#)

IloConstraint::getImpl

Inherited methods from [IloIntExprArg](#)[IloIntExprArg::getImpl](#)**Inherited methods from [IloNumExprArg](#)**[IloNumExprArg::getImpl](#)**Inherited methods from [IloExtractable](#)**

[IloExtractable::end](#), [IloExtractable::getEnv](#), [IloExtractable::getId](#),
[IloExtractable::getImpl](#), [IloExtractable::getName](#),
[IloExtractable::getObject](#), [IloExtractable::setName](#),
[IloExtractable::setObject](#)

Description

An instance of this class represents a sequence constraint in a model. As you can see from the parameters of its constructor, an instance of this class enables you to constrain:

- ◆ the minimum number of allowable values in the sequence,
- ◆ the maximum number of allowable values in the sequence,
- ◆ the frequency of allowable values (that is, how often a value occurs in the sequence),
- ◆ the number of elements in the sequence.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert` and `NDEBUG`.

See Also[IloConstraint](#), [IloDistribute](#)**Constructors**

```
public IloSequence()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSequence(IloSequenceI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSequence(const IloEnv env,
                  IloInt nbMin,
                  IloInt nbMax,
                  IloInt seqWidth,
                  const IloIntVarArray vars,
                  const IloIntArray values,
                  const IloIntVarArray cards,
                  const char * name=0)
```

This constructor creates a sequence constraint in an environment. The parameter `nbMin` indicates a minimum number of allowable values, and `nbMax` indicates a maximum number of allowable values. The parameter `seqWidth` indicates the number of elements in a sequence. The parameter `cards` indicates an array of cardinalities (that is, how many occurrences).

In the new constraint created by this class, the constrained variables in the array `cards` will be equal to the number of occurrences in the array `vars` of the values in the array `values` such that for each sequence of `seqWidth` (a number) consecutive constrained variables of `vars`, at least `nbMin` and at most `nbMax` `values` are assigned to a constrained variable of the sequence.

The arrays `cards` and `values` must be the same length; otherwise, on platforms where C++ exceptions are supported and exceptions are enabled, Concert Technology throws the exception `InvalidArraysException`.

Methods

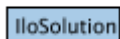
```
public IloSequenceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

IloSolution

Category Class

InheritancePath



Definition File ilconcert/ilosolution.h

Constructor Summary

public	IloSolution()
public	IloSolution(IloSolutionI * impl)
public	IloSolution(const IloSolution & solution)
public	IloSolution(IloEnv env, const char * name=0)

Method Summary

public void	add(IloAnySetVarArray a) const
public void	add(IloAnySetVar var) const
public void	add(IloAnyVarArray a) const
public void	add(IloAnyVar var) const
public void	add(IloNumVarArray a) const
public void	add(IloNumVar var) const
public void	add(IloObjective objective) const
public IloBool	contains(IloExtractable extr) const
public void	copy(IloExtractable extr, IloSolution solution) const
public void	copy(IloSolution solution) const
public void	end()
public IloEnv	getEnv() const
public IloSolutionI *	getImpl() const
public IloNum	getMax(IloNumVar var) const
public IloNum	getMin(IloNumVar var) const
public const char *	getName() const
public IloAny	getObject() const
public IloObjective	getObjective() const

public IloNum	getObjectiveValue() const
public IloNumVar	getObjectiveVar() const
public IloAnySet	getPossibleSet(IloAnySetVar var) const
public IloAnySet	getRequiredSet(IloAnySetVar var) const
public IloAny	getValue(IloAnyVar var) const
public IloNum	getValue(IloNumVar var) const
public IloNum	getValue(IloObjective obj) const
public IloBool	isBetterThan(IloSolution solution) const
public IloBool	isBound(IloAnySetVar var) const
public IloBool	isBound(IloNumVar var) const
public IloBool	isEquivalent(IloExtractable extr, IloSolution solution) const
public IloBool	isEquivalent(IloSolution solution) const
public IloBool	isObjectiveSet() const
public IloBool	isRestorable(IloExtractable extr) const
public IloBool	isWorseThan(IloSolution solution) const
public IloSolution	makeClone(IloEnv env) const
public void	operator=(const IloSolution & solution)
public void	remove(IloExtractableArray extr) const
public void	remove(IloExtractable extr) const
public void	restore(IloExtractable extr, IloAlgorithm algorithm) const
public void	restore(IloAlgorithm algorithm) const
public void	setFalse(IloBoolVar var) const
public void	setMax(IloNumVar var, IloNum max) const
public void	setMin(IloNumVar var, IloNum min) const
public void	setName(const char * name) const
public void	setNonRestorable(IloExtractableArray array) const
public void	setNonRestorable(IloExtractable extr) const
public void	setObject(IloAny obj) const
public void	setObjective(IloObjective objective) const
public void	setPossibleSet(IloAnySetVar var, IloAnySet possible) const
public void	setRequiredSet(IloAnySetVar var, IloAnySet required) const
public void	setRestorable(IloExtractableArray array) const
public void	setRestorable(IloExtractable ex) const
public void	setTrue(IloBoolVar var) const
public void	setValue(IloAnyVar var, IloAny value) const
public void	setValue(IloNumVar var, IloNum value) const

public void	setValue(IloObjective objective, IloNum value) const
public void	store(IloExtractable extr, IloAlgorithm algorithm) const
public void	store(IloAlgorithm algorithm) const
public void	unsetObjective() const

Inner Class	
IloSolution::IloSolution::Iterator	

Description

Instances of this class store solutions to problems. The fundamental property of `IloSolution` is its ability to transfer stored values from or to the active objects associated with it. In particular, the member function `store` stores the values from algorithm variables while the member function `restore` instantiates the actual variables with stored values. Variables in the solution may be selectively restored. This class also offers member functions to copy and to compare solutions.

Information about these classes of variables can be stored in an instance of `IloSolution`:

- ◆ `IloAnySet`: the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- ◆ `IloAnyVar`: the value of the variable is stored.
- ◆ `IloBoolVar`: the value (true or false) of the variable is stored. Some of the member functions for `IloBoolVar` are covered by the member function for `IloNumVar`, as `IloBoolVar` is a subclass of `IloNumVar`. For example, there is no explicit member function to add objects of type `IloBoolVar`.
- ◆ `IloIntSetVar`: the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- ◆ `IloNumVar`: the lower and upper bounds are stored; when the variable is bound, the current lower and upper bound are equivalent.
- ◆ `IloObjective`: the value of the objective is stored. Objectives are never restored; operations such as `setRestorable` cannot change this. More than one instance of `IloObjective` can be added to a solution,. In such cases, there is the notion of an active objective, which is returned by `IloSolution::getObjective`. The active objective typically specifies the optimization criterion for the problem to

which the solution object is a solution. For example, the ILOG Solver class `IloImprove` uses the notion of an active objective.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept `Assert and NDEBUG`.

Objects of type `IloSolution` have a scope, comprising the set of variables that have their values stored in the solution. The scope is given *before* the basic operations of storing and restoring are performed, via `add` and `remove` methods. For example,

```
IloNumVar var(env);
IloSolution soln(env);
solution.add(var);
```

creates a numerical variable and a solution and adds the variable to the solution. Arrays of variables can also be added to the solution. For example,

```
IloNumVarArray arr(env, 10, 0, 1);
soln.add(arr);
```

adds 10 variables with range `[0...1]`. When an array of variables is added to the solution, the array object itself is not present in the scope of the solution; only the elements are present. If the solution is then stored by means of `soln.store(algorithm)`, the values of variable `var` and `arr[0]` to `arr[9]` are saved. Any attempt to add a variable that is already present in a solution throws an exception, an instance of `IloException`.

Accessors allow access to the stored values of the variables, regardless of the state (or existence) of the algorithm they were stored from. For example,

```
cout << "arr[3] = " << soln.getValue(arr[3]) << endl;
```

Any attempt to access a variable that is not present in the solution throws an instance of `IloException`.

A variable or an array of variables can be removed from a solution. For example,

```
soln.remove(var);
```

removes `var` from the scope of the solution; and


```
soln.remove(arr);
```

removes `arr[0]` to `arr[9]` from the solution.

Any attempt to remove a variable that is not present in the solution throws an instance of `IloException`.

See `IloRestoreSolution` in the *ILOG Solver Reference Manual*.

See `IloStoreSolution` in the *ILOG Solver Reference Manual*.

See Also [IloAnySetVar](#), [IloAnyVar](#), [IloNumVar](#), [IloIntSetVar](#), [IloObjective](#)

Constructors

```
public IloSolution()
```

This constructor creates a solution whose implementation pointer is 0 (zero). The handle must be assigned before its methods can be used.

```
public IloSolution(IloSolutionI * impl)
```

This constructor creates a handle object (an instance of the class `IloSolution`) from a pointer to an implementation object (an instance of the class `IloSolutionI`).

```
public IloSolution(const IloSolution & solution)
```

This constructor creates a handle object from a reference to a solution. After execution, both the newly constructed handle and `solution` point to the same implementation object.

```
public IloSolution(IloEnv env,
                  const char * name=0)
```

This constructor creates an instance of the `IloSolution` class. The optional argument `name`, if supplied, becomes the name of the created object.

Methods

```
public void add(IloAnySetVarArray a) const
```

This member function adds each element of `array` to the invoking solution.

```
public void add(IloAnySetVar var) const
```

This member function adds the set variable `var` to the invoking solution.

```
public void add(IloAnyVarArray a) const
```

This member function adds each element of `array` to the invoking solution.

```
public void add(IloAnyVar var) const
```

This member function adds the variable `var` to the invoking solution.

```
public void add(IloNumVarArray a) const
```

This member function adds each element of `array` to the invoking solution.

```
public void add(IloNumVar var) const
```

This member function adds the variable `var` to the invoking solution.

```
public void add(IloObjective objective) const
```

This member function adds `objective` to the invoking solution. If the solution has no active objective, then `objective` becomes the active objective. Otherwise, the active objective remains unchanged.

```
public IloBool contains(IloExtractable extr) const
```

This member function returns `IloTrue` if `extr` is present in the invoking object. Otherwise, it returns `IloFalse`.

```
public void copy(IloExtractable extr,  
                 IloSolution solution) const
```

This member function copies the saved value of `extr` from `solution` to the invoking solution. If `extr` does not exist in either `solution` or the invoking object, this member function throws an instance of `IloException`. The restorable status of `extr` is not copied.

```
public void copy(IloSolution solution) const
```

For each variable that has been added to `solution`, this member function copies its saved data to the invoking solution. If a particular extractable does not already exist in the invoking solution, it is automatically added first. If variables were added to the invoking solution, their restorable status is the same as in `solution`. Otherwise, their status remains unchanged in the invoking solution.

```
public void end()
```

This member function deallocates the memory used to store the solution. If you no longer need a solution, calling this member function can reduce memory consumption.

```
public IloEnv getEnv() const
```

This member function returns the environment specified when the invoking object was constructed.

```
public IloSolutionI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking solution.

```
public IloNum getMax(IloNumVar var) const
```

This member function returns the maximal value of the variable `var` in the invoking solution.

```
public IloNum getMin(IloNumVar var) const
```

This member function returns the minimal value of the variable `var` in the invoking solution.

```
public const char * getName() const
```

This member function returns a character string indicating the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public IloObjective getObjective() const
```

This member function returns the *active* objective as set via a previous call to `add` or `setObjective(IloObjective)`. If there is no active objective, an empty handle is returned.

```
public IloNum getObjectiveValue() const
```

This member function returns the saved value of the current active objective. It can be seen as performing the action `getValue(getObjective())`.

```
public IloNumVar getObjectiveVar() const
```

If the active objective corresponds to a simple `IloNumVar`, this member function returns that variable. If there is no active objective or if the objective is not a simple variable, an empty handle is returned.

```
public IloAnySet getPossibleSet(IloAnySetVar var) const
```

This member function returns the set of possible values for the variable `var`, as stored in the invoking solution.

```
public IloAnySet getRequiredSet(IloAnySetVar var) const
```

This member function returns the set of required values for the variable `var`, as stored in the invoking solution.

```
public IloAny getValue(IloAnyVar var) const
```

This member function returns the value of the variable `var` in the invoking solution.

```
public IloNum getValue(IloNumVar var) const
```

This member function returns the value of the variable `var` in the invoking solution. If the saved minimum and maximum of the variable are not equal, this member function throws an instance of `IloException`.

```
public IloNum getValue(IloObjective obj) const
```

This member function returns the saved value of objective `objective` in the invoking solution.

```
public IloBool isBetterThan(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly higher quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

```
public IloBool isBound(IloAnySetVar var) const
```

This member function returns `IloTrue` if the stored required and possible sets for the set variable `var` are equal in the invoking solution. Otherwise, it returns `IloFalse`.

```
public IloBool isBound(IloNumVar var) const
```

This member function returns `IloTrue` if `var` takes a single value in the invoking solution. Otherwise, it returns `IloFalse`.

```
public IloBool isEquivalent(IloExtractable extr,
                           IloSolution solution) const
```

This member function returns `IloTrue` if the saved value of `extr` is the same in the invoking solution and `solution`. Otherwise, it returns `IloFalse`. If `extr` does not exist in either `solution` or the invoking object, the member function throws an instance of `IloException`.

```
public IloBool isEquivalent(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking object and `solution` contain the same variables with the same saved values. Otherwise, it returns `IloFalse`.

```
public IloBool isObjectiveSet() const
```

This member function returns `IloTrue` if the invoking solution has an active objective. Otherwise, it returns `IloFalse`.

```
public IloBool isRestorable(IloExtractable extr) const
```

This member function returns `IloFalse` if `setNonRestorable(extr)` was called more recently than `setRestorable(extr)`. Otherwise, it returns `IloTrue`. This member function always returns `IloFalse` when it is passed an `IloObjective` object.

```
public IloBool isWorseThan(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly lower quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

```
public IloSolution makeClone(IloEnv env) const
```

This member function allocates a new solution on `env` and adds to it all variables that were added to the invoking object. The “restorable” status of all variables in the clone is the same as that in the invoking solution. Likewise, the active objective in the clone is the same as that in the invoking solution. The newly created solution is returned.

```
public void operator=(const IloSolution & solution)
```

This operator assigns an address to the handle pointer of the invoking solution. That address is the location of the implementation object of `solution`. After the execution of this operator, the invoking solution and `solution` both point to the same implementation object.

```
public void remove(IloExtractableArray extr) const
```

This member function removes each element of `array` from the invoking solution. If the invoking solution does not contain all elements of `array`, the member function throws an instance of `IloException`.

```
public void remove(IloExtractable extr) const
```

This member function removes extractable `extr` from the invoking solution. If the invoking solution does not contain `extr`, the member function throws an instance of `IloException`.

```
public void restore(IloExtractable extr,  
                  IloAlgorithm algorithm) const
```

This member function restores the value of the extractable corresponding to `extr` by reference to `algorithm`. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the ILOG Solver class `IloSolver`, this member function can only be used during search. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

```
public void restore(IloAlgorithm algorithm) const
```

This member function uses `algorithm` to instantiate the variables in the invoking solution with their saved values. The value of any objective added to the solution is not restored. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the ILOG Solver class `IloSolver`, this member function can only be used during search.

```
public void setFalse(IloBoolVar var) const
```

This member function sets the stored value of `var` to `IloFalse` in the invoking solution.

```
public void setMax(IloNumVar var,  
                  IloNum max) const
```

This member function sets the maximal value of the variable `var` in the invoking solution to `max`.

```
public void setMin(IloNumVar var,  
                  IloNum min) const
```

This member function sets the minimal value of the variable `var` in the invoking solution to `min`.

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

```
public void setNonRestorable(IloExtractableArray array) const
```

This member function indicates to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, no elements of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state.

```
public void setNonRestorable(IloExtractable extr) const
```

This member function indicates to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, `extr` will not be restored. When a variable is added to a solution, it is added in a “restorable” state.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterwards. Normally, `obj` contains user data pertinent to the invoking object.

```
public void setObjective(IloObjective objective) const
```

This member function adds `objective` to the invoking solution, if it is not already present, and sets the active objective to `objective`.

```
public void setPossibleSet(IloAnySetVar var,  
                           IloAnySet possible) const
```

This member function sets the stored possible values for `var` as `possible` in the invoking solution.

```
public void setRequiredSet(IloAnySetVar var,  
                           IloAnySet required) const
```

This member function sets the stored required values for `var` as `required` in the invoking solution.

```
public void setRestorable(IloExtractableArray array) const
```

This member function indicates to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, the appropriate element(s) of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of this type are never restored.

```
public void setRestorable(IloExtractable ex) const
```

This member function indicates to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or

`restore(IloExtractable, IloAlgorithm)`, `extr` will be restored. When a variable is added to a solution, it is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of this type are never restored.

```
public void setTrue(IloBoolVar var) const
```

This member function sets the stored value of `var` to `IloTrue` in the invoking solution.

```
public void setValue(IloAnyVar var,
                    IloAny value) const
```

This member function sets the value of the variable `var` to `value` in the invoking solution.

```
public void setValue(IloNumVar var,
                    IloNum value) const
```

This member function sets the value (both minimum and maximum) of the variable `var` to `value` in the invoking solution.

```
public void setValue(IloObjective objective,
                    IloNum value) const
```

This member function sets the value of `objective` as stored in the invoking solution to `value`. This member function should be used with care and only when the objective value of the solution is known exactly.

```
public void store(IloExtractable extr,
                 IloAlgorithm algorithm) const
```

This member function stores the value of the extractable corresponding to `extr` by reference to `algorithm`. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

```
public void store(IloAlgorithm algorithm) const
```

This member function stores the values of the objects added to the solution by reference to `algorithm`.

```
public void unsetObjective() const
```

This member function asserts that there should be no active objective in the invoking solution, although the previous active object is still present. A new active objective can be set via `add` or `setObjective`.

IloSolutionArray

Category	Type Definition
Definition File	ilconcert/ilosolution.h
Synopsis	<code>IloSimpleArray< IloSolution > IloSolutionArray</code>
Description	This type definition represents arrays of instances of IloSolution. Instances of <code>IloSolutionArray</code> are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.
See Also	IloSolution

IloSolution::Iterator

Category Inner Class

InheritancePath

IloSolution::Iterator

Definition File ilconcert/ilosolution.h

Constructor Summary

public	Iterator (IloSolution solution)

Method Summary

public IloBool	ok () const
public IloExtractable	operator * () const
public Iterator &	operator++ ()

Description

[Iterator](#) is a class nested in the class [IloSolution](#). It allows you to traverse the variables in a solution. The iterator scans the objects in the same order as they were added to the solution.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterwards is undefined.

◆ `iter` can be safely used after the following code has executed:

```
IloExtractable elem = *iter;
++iter;
solution.remove(elem);
```

◆ `iter` cannot be safely used after the following code has executed:

```
solution.remove(*iter); // bad idea
++iter;
```

See Also [IloIterator](#), [IloSolution](#)

Constructors `public Iterator(IloSolution solution)`

This constructor creates an iterator to traverse the variables of `solution`. The iterator traverses variables in the same order they were added to `solution`.

Methods `public IloBool ok() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all variables have been scanned by the iterator.

`public IloExtractable operator *() const`

This operator returns the extractable object corresponding to the variable located at the current iterator position. If all variables have been scanned, this operator returns an empty handle.

`public Iterator & operator++()`

This operator moves the iterator to the next variable in the solution.

IloSolutionIterator

Category Class

InheritancePath



Definition File ilconcert/ilosolution.h

Constructor Summary	
public	IloSolutionIterator (IloSolution s)

Method Summary	
public E	operator *() const
public void	operator++ ()

Description This template class creates a typed iterator over solutions. You can use this iterator to discover all extractable objects added to a solution and of a particular type. The type is denoted by `E` in the template.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterwards is undefined.

An iterator created with this template differs from an instance of [IloSolution::Iterator](#). An instance of [IloSolution::Iterator](#) works on all extractable objects within a given solution (an instance of [IloSolution](#)). In contrast, an iterator created with this template only iterates over extractable objects of the specified type.

See Also [IloSolution](#), [IloSolution::Iterator](#)

Constructors public **IloSolutionIterator**([IloSolution](#) s)
This constructor creates an iterator for instances of the class `E`.

Methods public E **operator** *() const

This operator returns the current element, the one to which the invoking iterator points. This current element is a handle to an extractable object (not a pointer to the implementation object).

```
public void operator++()
```

This operator advances the iterator by one position.

IloSolutionManip

Category Class

InheritancePath



Definition File ilconcert/ilosolution.h

Constructor Summary	
public	IloSolutionManip (IloSolution solution, IloExtractable extr)

Description An instance of this class accesses a specific part of a solution so that you can display it. You construct the class `IloSolutionManip` from a solution and an extractable object. You use the `operator<<` with this constructed class to display information stored on the specified extractable object in the solution.

See Also [IloSolution](#), [operator](#)

Constructors public `IloSolutionManip`([IloSolution](#) solution, [IloExtractable](#) extr)

This constructor creates an instance of `IloSolutionManip` from the solution indicated by `solution` and from the extractable object `extr`. The constructor throws an exception (an instance of [IloException](#)) if `extr` has not been added to `solution`. You can use the `operator<<` with the newly created object to display the information in `extr` stored in `solution`.

IloSquare

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloSquare(const IloNumExprArg arg)
public IloNum IloSquare(IloNum val)
public IloInt IloSquare(IloInt val)
public IloInt IloSquare(int val)
public IloIntExprArg IloSquare(const IloIntExprArg arg)
```

Description Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

IloSquare returns the square of its argument (that is, $val*val$ or $expr*expr$).

What Is Extracted

IloSquare is extracted by an instance of IloSolver (documented in the *ILOG Solver Reference Manual*) as an instance of IlcSquare.

IloSubset

Category Global Function

Definition File ilconcert/iloanyset.h

Synopsis

```
public IloConstraint IloSubset(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySetVar var2)
public IloConstraint IloSubset(const IloEnv env,
    const IloAnySet var1,
    const IloAnySetVar var2)
public IloConstraint IloSubset(const IloEnv env,
    const IloAnySetVar var1,
    const IloAnySet var2)
public IloConstraint IloSubset(const IloEnv,
    const IloIntSetVar var1,
    const IloIntSetVar var2)
public IloConstraint IloSubset(const IloEnv,
    const IloIntSet var1,
    const IloIntSetVar var2)
public IloConstraint IloSubset(const IloEnv,
    const IloIntSetVar var1,
    const IloIntSet var2)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. That constraint forces **var1** to be strictly a subset of **var2**. That is, there is at least one element of **var2** not in **var1**, and all elements of **var1** are in **var2**.

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloSubsetEq

Category Global Function

Definition File ilconcert/iloanyset.h

Synopsis

```
public IloConstraint IloSubsetEq(const IloEnv env,
                                const IloAnySetVar var1,
                                const IloAnySetVar var2)
public IloConstraint IloSubsetEq(const IloEnv env,
                                const IloAnySet var1,
                                const IloAnySetVar var2)
public IloConstraint IloSubsetEq(const IloEnv env,
                                const IloAnySetVar var1,
                                const IloAnySet var2)
public IloConstraint IloSubsetEq(const IloEnv,
                                const IloIntSetVar var1,
                                const IloIntSetVar var2)
public IloConstraint IloSubsetEq(const IloEnv,
                                const IloIntSet var1,
                                const IloIntSetVar var2)
public IloConstraint IloSubsetEq(const IloEnv,
                                const IloIntSetVar var1,
                                const IloIntSet var2)
```

Description This function creates and returns a constraint (an instance of [IloConstraint](#)) for use in a model. That constraint forces `var1` to be a subset of `var2`. (The set `var1` may be equal to `var2`; every element of `var1` is also an element of `var2`.)

In order for the constraint to take effect, you must add it to a model with the template [IloAdd](#) or the member function [IloModel::add](#) and extract the model for an algorithm with the member function [IloAlgorithm::extract](#).

IloSum

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg IloSum(const IloNumExprArray exprs)
public IloIntExprArg IloSum(const IloIntExprArray exprs)
public IloNum IloSum(const IloNumArray values)
public IloInt IloSum(const IloIntArray values)
```

Description These functions return a numeric value representing the sum of numeric values in the array `vals`, or an instance of `IloNumExprArg`, the internal building block of an expression, representing the sum of the variables in the arrays `exprs` or `values`.

IloTableConstraint

Category Global Function

Definition File ilconcert/ilotupleaset.h

Synopsis

```

public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVarArray vars,
    const IloIntTupleSet set,
    IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVar var1,
    const IloIntVar var2,
    const IloIntTupleSet set,
    IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVar var1,
    const IloIntVar var2,
    const IloIntVar var3,
    const IloIntTupleSet set,
    IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVarArray vars,
    const IloIntTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVar var1,
    const IloIntVar var2,
    const IloIntVar var3,
    const IloIntTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVarArray vars,
    const IloIntBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVar var1,
    const IloIntVar var2,
    const IloIntBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloIntVar y,
    const IloIntArray a,
    const IloNumVar x)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloAnyVarArray vars,
    const IloAnyTupleSet set,
    IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloAnyVarArray vars,
    const IloAnyTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,
    const IloAnyVarArray vars,
    const IloAnyBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env,

```

```

const IloAnyVar y,
const IloAnyArray a,
const IloNumVar x)

```

Description

This function can be used to define simple constraints that are not predefined. It creates and returns a constraint for use in an ILOG Concert Technology model. That constraint is defined for *all* the constrained variables in the array `vars` or for the single constrained variable `y`.

This kind of constraint is sometimes known as an *element* constraint.

The semantics of that generic constraint can be indicated in either one of several ways:

- ◆ by a predicate; in that case, the argument `pred` indicates that predicate;
- ◆ by the values that satisfy the constraint; in that case, the argument `set` indicates the combinations of values that satisfy the constraint, and the argument `compatible` must be `IloTrue`;
- ◆ by the values that do *not* satisfy the constraint; in that case, the argument `set` indicates the unsatisfactory combinations of values, and the argument `compatible` must be `IloFalse`;
- ◆ by making the constrained variable `y` equal to the element of the array `a` at the index indicated by `x`. In other words, `y=a[x]`;

The order of the constrained variables in the array `vars` is important because the same order is respected in the predicate `pred` or the `set`. That is, `IloTableConstraint` passes an array of values to the member function `isTrue` for a predicate or to the member function `isIn` for a set, where the first such value is a value of `vars[0]`, the second is a value of `vars[1]`, and in general, the *ith* value is a value of the constrained variable `vars[i]`.

To avoid exceptions, you must observe the following conditions:

- ◆ If the function is called with a predicate `pred` as an argument, the size of the array of constrained variables must be three.
- ◆ The size of `vars` is must be the same as the size of the `set`.

IloThreeHalfPi

Category Macro

Synopsis **IloThreeHalfPi()**

Description Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloThreeHalfPi; // = 4.71238898038468985769
```

IloTimer

Category Class

InheritancePath



Definition File ilconcert/iloenv.h

Constructor Summary	
public	IloTimer (const IloEnv & env)
public	IloTimer (IloEnvI * env)

Method Summary	
public IloEnv	getEnv () const
public IloNum	getTime () const
public void	reset ()
public IloNum	restart ()
public IloNum	start ()
public IloNum	stop ()

Description An instance of **IloTimer** represents a timer in a Concert Technology model. It works like a stop watch.

See Also [IloEnv](#)

Constructors public **IloTimer**(const [IloEnv](#) & env)
This constructor creates a timer.
public **IloTimer**(IloEnvI * env)
This constructor creates an instance of the class **IloTimer**

Methods public [IloEnv](#) **getEnv**() const

This member function returns the environment in which the invoking timer was constructed.

```
public IloNum getTime() const
```

This member function returns the accumulated time, in seconds, since one of these conditions:

- ◆ the first call of the member function `start` after construction of the invoking timer;
- ◆ the most recent call to the member function `restart`;
- ◆ a call to `reset`.

```
public void reset()
```

This member function sets the elapsed time of the invoking timer to 0.0. It also stops the clock.

```
public IloNum restart()
```

This member function returns the accumulated time, resets the invoking timer to 0.0, and starts the timer again. In other words, the member function `restart` is equivalent to the member function `reset` followed by `start`.

```
public IloNum start()
```

This member function makes the invoking timer resume accumulating time. It returns the time accumulated so far.

```
public IloNum stop()
```

This member function stops the invoking timer so that it no longer accumulates time.

IloTwoPi

Category Macro

Synopsis **IloTwoPi()**

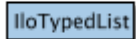
Description Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloTwoPi;          // = 6.28318530717958647692
```

IloTypedList

Category Class

InheritancePath



IloTypedList

Definition File ilconcert/ilotypedlist.h

Description A list of extractables of the same type, with possibly a sibling and a first son list. It holds the extractables that are instances of a specific class: the instances of the sub-classes are in the son lists.

IloTypedListManagerI

Category Class

InheritancePath



IloTypedListManagerI

Definition File ilconcert/ilotypedlist.h

Description The class `IloTypedListManager` manages the `IloTypedList`. It classifies the extractables of a model to get a tree of lists of extractables grouped by type.

A model must have a pointer to an instance `IloTypedListManager` that is initialized to 0, and allocate such an instance when an iterator is created.

Creation and removal of extractable must check if an `IloTypedListManager` exists, and respectively call `add()` or `remove()` on it. The model may ask the `IloTypedListManager` to clear itself, which implies that all the extractables will be scanned again if an iterator is subsequently created.

operator &&

Category Global Function

Definition File ilconcert/ilomodel.h

Synopsis

```
public IloAnd operator &&(const IloConstraint constraint1,  
    const IloConstraint constraint2)
```

Description This overloaded C++ operator creates a conjunctive constraint that represents the conjunction of its two parameters. The constraint can represent a conjunction of two constraints; of a constraint and another conjunction; or of two conjunctions. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator *

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumLinExprTerm operator *(const IloNumVar x,
                                     IloInt num)
public IloNumLinExprTerm operator *(IloInt num,
                                     const IloNumVar x)
public IloNumLinExprTerm operator *(const IloIntVar x,
                                     IloNum num)
public IloNumLinExprTerm operator *(IloNum num,
                                     const IloIntVar x)
public IloIntLinExprTerm operator *(const IloIntVar x,
                                     IloInt num)
public IloNumExprArg operator *(const IloNumExprArg x,
                                 const IloNumExprArg y)
public IloNumExprArg operator *(const IloNumExprArg x,
                                 IloNum y)
public IloNumExprArg operator *(IloNum x,
                                 const IloNumExprArg y)
public IloIntExprArg operator *(const IloIntExprArg x,
                                 const IloIntExprArg y)
public IloIntExprArg operator *(const IloIntExprArg x,
                                 IloInt y)
```

Description This overloaded C++ operator returns an expression equal to the product of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

operator new

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<pre>public void * operator new(size_t sz, const IloEnv & env)</pre>
Description	ILOG Concert Technology offers this overloaded C++ <code>new</code> operator. This operator is overloaded to allocate data on internal data structures associated with an invoking environment (an instance of <code>IloEnv</code>). The memory used by objects allocated with this overloaded operator is automatically reclaimed when you call the member function <code>IloEnv::end</code>. As a developer, you must <i>not</i> delete objects allocated with this operator because of this automatic freeing of memory. In other words, you must <i>not</i> use the <code>delete</code> operator for objects allocated with this overloaded <code>new</code> operator. The use of this overloaded <code>new</code> operator is not obligatory in Concert Technology applications. You will see examples of its use in the user's manuals that accompany the ILOG optimization products.

operator!

Category Global Function

Definition File ilconcert/ilomodel.h

Synopsis

```
public IloConstraint operator!(const IloConstraint constraint)
```

Description This overloaded C++ operator returns a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator!=

Category Global Function

Definition File ilconcert/iloany.h

Synopsis

```

public IloConstraint operator!=(const IloAnyVar var1,
                                const IloAnyVar var2)
public IloConstraint operator!=(const IloAnyVar var1,
                                IloAny val)
public IloConstraint operator!=(IloAny val,
                                const IloAnyVar var1)
public IloConstraint operator!=(const IloAnySetVar var1,
                                const IloAnySetVar var2)
public IloConstraint operator!=(const IloAnySetVar var1,
                                const IloAnySet set)
public IloConstraint operator!=(const IloAnySet set,
                                const IloAnySetVar var1)
public IloDiff operator!=(IloNumExprArg arg1,
                           IloNumExprArg arg2)
public IloDiff operator!=(IloNumExprArg arg,
                           IloNum val)
public IloDiff operator!=(IloNum val,
                           IloNumExprArg arg)
public IloConstraint operator!=(const IloIntSetVar var1,
                                const IloIntSetVar var2)
public IloConstraint operator!=(const IloIntSetVar var,
                                const IloIntSet set)
public IloConstraint operator!=(const IloIntSet set,
                                const IloIntSetVar var)

```

Description This overloaded C++ operator constrains its two arguments to be unequal (that is, different from each other). In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator%

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloIntExprArg operator%(const IloIntExprArg x,  
                                const IloIntExprArg y)
```

Description This operator returns an instance of [IloIntExprArg](#), the internal building block of an expression, representing the modulo of the expressions x and y.

operator%

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloIntExprArg operator%(const IloIntExprArg x,  
                                IloInt y)
```

Description This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the expression `x` and the integer value `y`.

operator%

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloIntExprArg operator%(IloInt x,  
    const IloIntExprArg y)
```

Description This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the integer value `x` and the expression `y`.

operator+

Category	Global Function
Definition File	ilconcert/iloexpression.h
Synopsis	<pre> public IloNumExprArg operator+(const IloNumExprArg x, const IloNumExprArg y) public IloNumExprArg operator+(const IloNumExprArg x, IloNum y) public IloNumExprArg operator+(IloNum x, const IloNumExprArg y) public IloIntExprArg operator+(const IloIntExprArg x, const IloIntExprArg y) public IloIntExprArg operator+(const IloIntExprArg x, IloInt y) public IloIntExprArg operator+(IloInt x, const IloIntExprArg y) </pre>
Description	This overloaded C++ operator returns an expression equal to the sum of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

operator-

Category	Global Function
Definition File	ilconcert/iloexpression.h
Synopsis	<pre> public IloNumExprArg operator-(const IloNumExprArg x, const IloNumExprArg y) public IloNumExprArg operator-(const IloNumExprArg x, IloNum y) public IloNumExprArg operator-(IloNum x, const IloNumExprArg y) public IloIntExprArg operator-(const IloIntExprArg x, const IloIntExprArg y) public IloIntExprArg operator-(const IloIntExprArg x, IloInt y) public IloIntExprArg operator-(IloInt x, const IloIntExprArg y) </pre>
Description	This overloaded C++ operator returns an expression equal to the difference of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

operator/

Category Global Function

Definition File ilconcert/iloexpression.h

Synopsis

```
public IloNumExprArg operator/(const IloNumExprArg x,  
                               const IloNumExprArg y)  
public IloNumExprArg operator/(const IloNumExprArg x,  
                               IloNum y)  
public IloNumExprArg operator/(IloNum x,  
                               const IloNumExprArg y)
```

Description This overloaded C++ operator returns an expression equal to the quotient of its arguments. Its arguments may be numeric values or numeric variables. For integer division, use [IloDiv](#).

operator<

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloConstraint operator<(IloNumExprArg base,
                               IloNumExprArg base2)
public IloConstraint operator<(IloNumExprArg base,
                               IloNum val)
public IloConstraint operator<(IloNum val,
                               const IloNumExprArg expr)
public IloConstraint operator<(IloIntExprArg base,
                               IloIntExprArg base2)
public IloConstraint operator<(IloIntExprArg base,
                               IloInt val)
```

Description This overloaded C++ operator constrains its first argument to be strictly less than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator<<

Category	Global Function
Definition File	ilconcert/iloalg.h
Synopsis	<pre> public ostream & operator<<(ostream & out, IloAlgorithm::Status st) public ostream & operator<<(ostream & out, const IloArray< X > & a) public ostream & operator<<(ostream & out, const IloNumExpr & ext) public ostream & operator<<(ostream & os, const IloRandom & r) public ostream & operator<<(ostream & stream, const IloSolution & solution) public ostream & operator<<(ostream & stream, const IloSolutionManip & fragment) public ostream & operator<<(ostream & o, const IloException & e) </pre>
Description	This overloaded C++ operator directs output to an output stream.

operator<=

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloConstraint operator<=(IloNumExprArg base,
                                IloNumExprArg base2)
public IloRange operator<=(IloNumExprArg base,
                            IloNum val)
public IloRangeBase operator<=(IloNum val,
                                const IloNumExprArg expr)
public IloRange operator<=(const IloRangeBase base,
                            IloNum val)
```

Description This overloaded C++ operator constrains its first argument to be less than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator==

Category Global Function

Definition File ilconcert/iloany.h

Synopsis

```
public IloConstraint operator==(const IloAnyVar var1,
                                const IloAnyVar var2)
public IloConstraint operator==(const IloAnyVar var1,
                                IloAny val)
public IloConstraint operator==(IloAny val,
                                const IloAnyVar var1)
public IloConstraint operator==(const IloAnySetVar var1,
                                const IloAnySetVar var2)
public IloConstraint operator==(const IloAnySetVar var1,
                                const IloAnySet set)
public IloConstraint operator==(const IloAnySet set,
                                const IloAnySetVar var1)
public IloConstraint operator==(const IloIntSetVar var1,
                                const IloIntSetVar var2)
public IloConstraint operator==(const IloIntSetVar var1,
                                const IloIntSet set)
public IloConstraint operator==(const IloIntSet set,
                                const IloIntSetVar var)
```

Description This overloaded C++ operator constrains its two arguments to be equal. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator>

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloConstraint operator>(IloNumExprArg base,
                               IloNumExprArg base2)
public IloConstraint operator>(IloNumExprArg base,
                               IloNum val)
public IloConstraint operator>(IloNum val,
                               IloNumExprArg eb)
public IloConstraint operator>(IloIntExprArg base,
                               IloIntExprArg base2)
public IloConstraint operator>(IloIntExprArg base,
                               IloInt val)
public IloConstraint operator>(IloInt val,
                               IloIntExprArg eb)
```

Description This overloaded C++ operator constrains its first argument to be strictly greater than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator>=

Category Global Function

Definition File ilconcert/ilolinear.h

Synopsis

```
public IloConstraint operator>=(IloNumExprArg base,
                                IloNumExprArg base2)
public IloRange operator>=(IloNumExprArg expr,
                            IloNum val)
public IloRange operator>=(IloNum val,
                            IloNumExprArg eb)
```

Description This overloaded C++ operator constrains its first argument to be greater than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

operator>>

Category	Global Function
Definition File	ilconcert/iloenv.h
Synopsis	<pre>public istream & operator>>(istream & in, IloNumArray & a) public istream & operator>>(istream & in, IloIntArray & a)</pre>
Description	This overloaded C++ operator directs input to an input stream.

operator||

Category	Global Function
Definition File	ilconcert/ilomodel.h
Synopsis	<pre>public IloOr operator (const IloConstraint constraint1, const IloConstraint constraint2)</pre>
Description	This overloaded C++ operator creates a disjunctive constraint that represents the disjunction of its two parameters. The constraint can represent a disjunction of two constraints; of a constraint and another disjunction; or of two disjunctions. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Group optim.concert.xml

The ILOG Concert Serialization API.

Classes Summary	
IloXmlContext	
IloXmlInfo	
IloXmlReader	
IloXmlWriter	

DescriptionIloXmlContext

Category	Class
InheritancePath	



Definition File	ilconcert/iloxmlcontext.h

Constructor Summary

public	IloXmlContext(IloEnv env)
public	IloXmlContext(IloXmlContextI * impl=0)

Method Summary	
public void	end()
public IloInt	getChildIdReadError()
public const char *	getChildTagReadError()
public IloIntArray	getIdListReadError()
public const IloXmlContextI *	getImpl()
public IloInt	getParentIdReadError()
public const char *	getParentTagReadError()
public IloAnyArray	getTagListReadError()
public const char *	getWriteError()
public int	getWritePrecision()
public IloBool	readExtractable(IloXmlReader reader, IloXmlElement * element)
public IloBool	readModel(IloModel model, istream & file)
public IloBool	readModel(IloModel model, const char * fileName)
public IloBool	readModelAndSolution(IloModel model, const char * modelFileName, IloSolution solution, const char * solutionFileName)
public IloBool	readRtti(IloXmlReader reader, IloXmlElement * element)
public IloBool	readSolution(IloSolution solution, istream & file)
public IloBool	readSolution(IloSolution solution, const char * fileName)
public IloBool	readSolutionValue(IloSolution solution, IloXmlElement * root, IloXmlReader reader)
public void	registerXML(IloTypeIndex index, IloXmlInfo * xmlinfo)
public void	registerXMLArray(IloXmlInfo * xmlinfo)
public IloBool	setWriteMode(IloInt mode)
public void	setWritePrecision(int writePrecision)
public IloBool	writeExtractable(const IloExtractableI * it, IloXmlWriter writer, IloXmlElement * masterElement)

public IloBool	writeModel(const IloModel model,const char * fileName)
public IloBool	writeModelAndSolution(const IloModel model,const char * modelFileName,const IloSolution solution,const char * solutionFileName)
public IloBool	writeRtti(const IloRtti * it,IloXmlWriter writer,IloXmlElement * masterElement)
public IloBool	writeSolution(const IloSolution solution,const char * fileName)
public void	writeSolutionValue(const IloExtractable it,const IloSolution solution,IloXmlWriter writer)

Description

An instance of `IloXmlContext` allows you to serialize an `IloModel` or an `IloSolution` in XML.

You can write an `IloModel` using `writeModel`, write an `IloSolution` using `writeSolution`, or write both using `writeModelAndSolution`.

You can read an `IloModel` in XML using `readModel`, read an `IloSolution` in XML using `readSolution`, or read both using `readModelAndSolution`.

Other products should add their own serialization class and add them to the plug-in using the member functions `registerXML` and `registerXMLArray`.

Examples

For example, you can write:

```
IloModel model(env);
IloSolution solution(env);
...;
IloXmlContext context(env);
context.writeModel(model, "model.xml");
context.writeSolution(solution, "solution.xml");
```

or you can write

```
IloModel model(env);
IloSolution solution(env);
IloXmlContext context(env);
context.readModel(model, "model.xml");
context.readSolution(solution, "solution.xml");
```

See Also [IloXmlReader](#), [IloXmlWriter](#), [IloXmlInfo](#)

Constructors `public IloXmlContext(IloEnv env)`

This constructor creates an XML context and makes it part of the environment `env`.

`public IloXmlContext(IloXmlContextI * impl=0)`

This constructor creates a XML context from its implementation object.

Methods `public void end()`

This member function deletes the invoking XML context.

`public IloInt getChildIdReadError()`

This member function returns the XML ID of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public const char * getChildTagReadError()`

This member function returns the XML tag of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public IloIntArray getIdListReadError()`

This member function returns the XML ID list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

`public const IloXmlContextI * getImpl()`

This member function returns the `IloXmlContextI` implementation.

`public IloInt getParentIdReadError()`

This member function returns the XML ID of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public const char * getParentTagReadError()`

This member function returns the XML tag of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public IloAnyArray getTagListReadError()`

This member function returns the XML tag list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

`public const char * getWriteError()`

This member function returns the name of the extractable called in cases where a problem occurs when reading an `IloModel`.

`public int getWritePrecision()`

This member function returns the write precision for floats

```
public IloBool readExtractable(IloXmlReader reader,
                              IloXmlElement * element)
```

@deprecated

See Also

[readRtti](#)

```
public IloBool readModel(IloModel model,
                        istream & file)
```

This member function reads `model` from an XML stream.

```
public IloBool readModel(IloModel model,
                        const char * fileName)
```

This member function reads `model` from the XML file `fileName`.

```
public IloBool readModelAndSolution(IloModel model,
                                    const char * modelFileName,
                                    IloSolution solution,
                                    const char * solutionFileName)
```

This member function reads `model` and `solution` from their respective XML files, `modelFileName` and `solutionFileName`.

```
public IloBool readRtti(IloXmlReader reader,
                       IloXmlElement * element)
```

This member function tries to read all extractables from the XML element.

```
public IloBool readSolution(IloSolution solution,
                           istream & file)
```

This member function reads `solution` from an XML stream.

Note: This member function only works if a model has already been serialized.

```
public IloBool readSolution(IloSolution solution,
                           const char * fileName)
```

This member function reads `solution` from the XML file `fileName`.

Note: This member function only works if a model has already been serialized.

```
public IloBool readSolutionValue(IloSolution solution,
```



```
IloXmlElement * root,
IloXmlReader reader)
```

This member function reads an `IloSolution` object from an XML element.

```
public void registerXML(IloTypeIndex index,
IloXmlInfo * xmlinfo)
```

This member function registers the serialization class of an extractable with a linked ID, usually its RTTI index. In write mode, the RTTI index is used to catch the correct serialization class.

In read mode, `IloXmlInfo::getTagName` is used to link the correct serialization class to the correct tag.

```
IlpXmlContext context(env);
context.registerXML(IloAllDiffI::GetTypeIndex(), new (env)
IloXmlInfo_AllDiff(context));
```

```
public void registerXMLArray(IloXmlInfo * xmlinfo)
```

This member function registers the serialization class of an array of extractables with a linked ID.

```
context.registerXMLArray(new (env) IloXmlInfo_SOS2Array(context));
```

```
public IloBool setWriteMode(IloInt mode)
```

This member function sets the write mode. The write mode can be set to `NoUnknown` or `EvenUnknown`. `NoUnknown` throws an exception if an attempt is made to serialize an unknown extractable. `EvenUnknown` writes a `Unknown` tag with the name of the extractable in a type attribute.

```
public void setWritePrecision(int writePrecision)
```

This member function sets the write precision for floats. By default, there is no rounding mode on an `IloNum` or an `IloNumArray`. You can also choose the no rounding mode with the `IloNoRoundingMode` constant.

```
public IloBool writeExtractable(const IloExtractableI * it,
IloXmlWriter writer,
IloXmlElement * masterElement)
```

@deprecated

See Also

[writeRtti](#)

```
public IloBool writeModel(const IloModel model,
const char * fileName)
```

This member function writes `model` to the file `fileName` in XML format.

```
public IloBool writeModelAndSolution(const IloModel model,
                                     const char * modelFileName,
                                     const IloSolution solution,
                                     const char * solutionFileName)
```

This member function writes `model` to the file `modelFileName` and `solution` to the file `solutionFileName` in XML format.

```
public IloBool writeRtti(const IloRtti * it,
                        IloXmlWriter writer,
                        IloXmlElement * masterElement)
```

This member function writes a specified extractable. It is used from the serialization class of an extractable to write a embedded extractable.

The `IloOr` object calls this method on its constrained `vars`.

See Also

[IloXmlInfo::writeRtti](#)

```
public IloBool writeSolution(const IloSolution solution,
                             const char * fileName)
```

This member function writes `solution` to the file `fileName` in XML format.

```
public void writeSolutionValue(const IloExtractable it,
                              const IloSolution solution,
                              IloXmlWriter writer)
```

This member function writes a specified extractable of a solution in XML. It is used from the serialization class of an extractable to write an embedded extractable.

See Also

[IloXmlInfo::writeSolutionValue](#)

IloXmlInfo

Category Class

InheritancePath

IloXmlInfo

Definition File ilconcert/iloxmlabstract.h

Constructor Summary

public	IloXmlInfo (IloXmlContextI * context,const char * version=0)
public	IloXmlInfo ()

Method Summary

public IloBool	checkAttExistence (IloXmlReader reader,IloXmlElement * element,const char * attribute)
public IloBool	checkExprExistence (IloXmlReader reader,IloXmlElement * element,const char * attribute, IloInt & id)
public IloXmlContextI *	getContext ()
protected IloBool	getIntValArray (IloXmlReader reader,IloXmlElement * element, IloIntArray & intArray)
protected IloBool	getNumValArray (IloXmlReader reader,IloXmlElement * element, IloNumArray & numArray)
public IloBool	getRefInChild (IloXmlReader reader,IloXmlElement * element, IloInt & id)
public virtual const char *	getTag ()
public virtual IloXmlElement *	getTagElement (IloXmlWriter writer,const IloRtti * exprI)
public static const char *	IloXmlInfo::getTagName ()
protected IloNumVar::Type	getVarType (IloXmlReader reader,IloXmlElement * element)
protected const char *	getVersion ()

protected virtual IloRtti *	<code>read(IloXmlReader reader, IloXmlElement * element)</code>
public virtual IloExtractableArray *	<code>readArrayFromXml(IloXmlReader reader, IloXmlElement * element)</code>
public IloBool	<code>readExtractable(IloXmlReader reader, IloXmlElement * element)</code>
public virtual IloRtti *	<code>readFrom(IloXmlReader reader, IloXmlElement * element)</code>
public virtual IloExtractableI *	<code>readFromXml(IloXmlReader reader, IloXmlElement * element)</code>
public IloBool	<code>readRtti(IloXmlReader reader, IloXmlElement * element)</code>
public virtual IloBool	<code>readSolution(IloXmlReader reader, IloSolution solution, IloXmlElement * element)</code>
protected virtual IloExtractableI *	<code>readXml(IloXmlReader reader, IloXmlElement * element)</code>
protected virtual IloExtractableArray *	<code>readXmlArray(IloXmlReader reader, IloXmlElement * element)</code>
protected IloXmlElement *	<code>setBoolArray(IloXmlWriter writer, const IloBoolArray Array)</code>
public IloXmlElement *	<code>setCommonArrayXml(IloXmlWriter writer, const IloExtractableArray * extractable)</code>
public IloXmlElement *	<code>setCommonValueXml(IloXmlWriter writer, const IloRtti * exprI)</code>
public IloXmlElement *	<code>setCommonXml(IloXmlWriter writer, const IloRtti * exprI)</code>
protected IloXmlElement *	<code>setIntArray(IloXmlWriter writer, const IloIntArray Array)</code>
protected IloXmlElement *	<code>setIntSet(IloXmlWriter writer, const IloIntSet Array)</code>
protected IloXmlElement *	<code>setNumArray(IloXmlWriter writer, const IloNumArray Array)</code>
protected IloXmlElement *	<code>setNumSet(IloXmlWriter writer, const IloNumSet Array)</code>
protected void	<code>setVersion(const char * version)</code>
public void	<code>setXml(IloXmlWriter writer, IloXmlElement * element, const IloRtti * exprI)</code>
public virtual int	<code>write(IloXmlWriter writer, const IloExtractableArray * extractable, IloXmlElement * masterElement)</code>
public virtual IloBool	<code>write(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)</code>
public IloBool	<code>writeExtractable(IloXmlWriter writer, IloXmlElement * element, const IloExtractable extractable, const char * attribute=0)</code>

public virtual IloBool	writeRef(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)
public IloBool	IloXmlInfo::writeRtti(IloXmlWriter writer, IloXmlElement * element, const IloRtti * rtti, const char * attribute=0)
public virtual void	writeSolution(IloXmlWriter writer, const IloSolution solution, const IloExtractable extractable)
public void	IloXmlInfo::writeSolutionValue(IloXmlWriter writer, const IloSolution solution, IloXmlElement * element, const IloRtti * rtti, const char * attribute)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS2Array array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS1Array array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSemiContVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloConstraintArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloRangeArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntSetVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloBoolVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntVarArray array)

public virtual IloBool	writeXml (IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)
public virtual IloBool	writeXmlRef (IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)

Description The class **IloXmlInfo** allows you to serialize an **IloModel** or an **IloSolution** in XML.

Constructors public **IloXmlInfo**(**IloXmlContextI** * context,
const char * version=0)

This constructor creates an instance of the handle class **IloXmlInfo** from a pointer to an instance of the undocumented implementation class **IloXmlContextI**.

public **IloXmlInfo**()

This constructor creates an empty instance of the handle class **IloXmlInfo**.

Methods public **IloBool** **checkAttExistence**(**IloXmlReader** reader,
IloXmlElement * element,
const char * attribute)

Given a specified attribute, this member function checks **element** to establish whether the attribute exists. If the attribute does not exist, this member function throws an exception.

You can use this member function to dynamically validate an XML element.

public **IloBool** **checkExprExistence**(**IloXmlReader** reader,
IloXmlElement * element,
const char * attribute,
IloInt & id)

Given a specified attribute, this member function checks **element** to establish whether the attribute exists, fills the **id**, and checks in the XML context memory whether an object with this **id** exists.

You can use this member function to dynamically validate an XML element.

Example: in the **read** method of the **IloDiff**, check that the **IdRef** object is already serialized

public **IloXmlContextI** * **getContext**()

This member function returns the related **IloXmlContextI** of the constructor.

protected **IloBool** **getIntValArray**(**IloXmlReader** reader,
IloXmlElement * element,
IloIntArray & intArray)

This member function returns the contained `IloIntArray` in the XML element `element`.

See Also [`IloXmlReader::string2IntArray`](#)

```
protected IloBool getNumValArray(IloXmlReader reader,
                                   IloXmlElement * element,
                                   IloNumArray & numArray)
```

This member function returns the `IloNumArray` in the XML element `element`.

See Also [`IloXmlReader::string2NumArray`](#)

```
public IloBool getRefInChild(IloXmlReader reader,
                              IloXmlElement * element,
                              IloInt & id)
```

Given an XML element, this member function checks for the first value `id` or `RefId` in the element and its children.

```
public virtual const char * getTag()
```

This member function returns the related XML tag.

```
public virtual IloXmlElement * getTagElement(IloXmlWriter writer,
                                              const IloRtti * exprI)
```

For backward compatibility with 2.0 and the XML for `IloExtractable` objects, if this method is not specialized, by default the `getTagElement` method with `IloExtractableI` will be called

```
public static const char * getTagName()
```

This static member function returns the linked XML tag of this serialization class.

```
protected IloNumVar::Type getVarType(IloXmlReader reader,
                                       IloXmlElement * element)
```

This member function returns the type of an `IloNumVar` - `IloFloat`, `IloInt`, or `IloBool` - in the XML element `element`.

```
protected const char * getVersion()
```

This member function returns the version of the object.

```
protected virtual IloRtti * read(IloXmlReader reader,
                                 IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

For backward compatibility with Concert 2.0 and the XML for `IloExtractable` objects, by default the method `readXml` with `IloExtractableI` will be called

```
public virtual IloExtractableArray * readArrayFromXml(IloXmlReader reader,
```

```
IloXmlElement * element)
```

This member function reads an array of `IloRtti*` from the given XML element.

This is the method to specialize when writing a serialization class for an array of extractables.

```
public IloBool readExtractable(IloXmlReader reader,
                              IloXmlElement * element)
```

@deprecated

```
public virtual IloRtti * readFrom(IloXmlReader reader,
                                  IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `readRtti`; it then calls `readXml`.

For backward compatibility with Concert 2.0 and the XML for `IloExtractable` objects, by default the method `readFromXml` with `IloExtractableI` will be called

```
public virtual IloExtractableI * readFromXml(IloXmlReader reader,
                                              IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `readRtti`; it then calls `readXml`.

@deprecated

```
public IloBool readRtti(IloXmlReader reader,
                       IloXmlElement * element)
```

This member function asks the XML context to read the `IloRtti` in the child element and then calls `readFromXml` to read the parent extractable.

```
public virtual IloBool readSolution(IloXmlReader reader,
                                    IloSolution solution,
                                    IloXmlElement * element)
```

This member function reads a variable for `IloSolution` from the XML element element.

```
protected virtual IloExtractableI * readXml(IloXmlReader reader,
                                             IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

@deprecated

```
protected virtual IloExtractableArray * readXmlArray(IloXmlReader reader,
                                                      IloXmlElement * element)
```


This member function reads an array of `IloRtti*` from the given XML element.

It is called by the XML context. It first asks the XML context to read from XML child elements using a call to `readRtti` and then calls `readArrayFromXml`.

```
protected IloXmlElement * setBoolArray(IloXmlWriter writer,
                                         const IloBoolArray Array)
```

This member function creates an XML element containing the `IloBoolArray`.

See Also

[IloXmlWriter::IntArray2String](#)

```
public IloXmlElement * setCommonArrayXml(IloXmlWriter writer,
                                         const IloExtractableArray *
extractable)
```

This member function creates a XML element with the common header for `IloExtractable` arrays.

```
public IloXmlElement * setCommonValueXml(IloXmlWriter writer,
                                         const IloRtti * exprI)
```

This member function creates an XML element with the given header for `IloRtti` from `IloSolution`.

```
public IloXmlElement * setCommonXml(IloXmlWriter writer,
                                     const IloRtti * exprI)
```

This member function creates an XML element with the common header for `IloRtti`.

```
protected IloXmlElement * setIntArray(IloXmlWriter writer,
                                       const IloIntArray Array)
```

This member function creates an XML element containing the `IloIntArray`.

See Also

[IloXmlWriter::IntArray2String](#)

```
protected IloXmlElement * setIntSet(IloXmlWriter writer,
                                     const IloIntSet Array)
```

This member function creates an XML element containing the `IloIntSet`.

See Also

[IloXmlWriter::IntSet2String](#)

```
protected IloXmlElement * setNumArray(IloXmlWriter writer,
                                       const IloNumArray Array)
```

This member function creates an XML element containing the `IloNumArray`.

See Also

[IloXmlWriter::NumArray2String](#)

```
protected IloXmlElement * setNumSet(IloXmlWriter writer,
                                     const IloNumSet Array)
```

This member function creates an XML element containing the `IloNumSet`.

See Also [IloXmlWriter::NumSet2String](#)

```
protected void setVersion(const char * version)
```

This member function sets the version of the object.

```
public void setXml(IloXmlWriter writer,  
                  IloXmlElement * element,  
                  const IloRtti * exprI)
```

This member function adds a name attribute and a ID attribute to the XML element.

```
public virtual int write(IloXmlWriter writer,  
                        const IloExtractableArray * extractable,  
                        IloXmlElement * masterElement)
```

This member function writes the given `IloExtractableArray` in XML and adds it to the XML document of `writer`. This is the method to specialize when writing a serialization class

```
public virtual IloBool write(IloXmlWriter writer,  
                             const IloRtti * exprI,  
                             IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

For backward compatibility with Concert 2.0 and the XML for `IloExtractable` objects, by default the method `writeXml` with `IloExtractableI` will be called

```
public IloBool writeExtractable(IloXmlWriter writer,  
                                IloXmlElement * element,  
                                const IloExtractable extractable,  
                                const char * attribute=0)
```

@deprecated

See Also [writeRtti](#)

```
public virtual IloBool writeRef(IloXmlWriter writer,  
                                const IloRtti * exprI,  
                                IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

For backward compatibility with Concert 2.0 and the XML for `IloExtractable` objects, by default the method `writeXmlRef` with `IloExtractableI` will be called

```
public IloBool writeRtti(IloXmlWriter writer,  
                         IloXmlElement * element,  
                         const IloRtti * rtti,  
                         const char * attribute=0)
```

This member function writes an embedded extractable. Using the `getId()` method of the extractable, it adds an attribute with the ID in the xml element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

```
// using an IloDiffI* exprI:
    writeRtti(writer, element,
              (IloRtti*)exprI->getExpr1(),
              IloXmlAttributeDef::Expr1Id);
    writeRtti(writer, element,
              (IloRtti*)exprI->getExpr2(),
              IloXmlAttributeDef::Expr2Id);
*
```

See Also

[writeRtti](#)

```
public virtual void writeSolution(IloXmlWriter writer,
                                   const IloSolution solution,
                                   const IloExtractable extractable)
```

This member function writes the specified extractable `extractable` from the `IloSolution` `solution` in XML format.

```
public void writeSolutionValue(IloXmlWriter writer,
                                const IloSolution solution,
                                IloXmlElement * element,
                                const IloRtti * rtti,
                                const char * attribute)
```

This member function writes an embedded extractable of a solution in xml. Using the `getId()` method of the extractable, it adds an attribute with the ID in the xml element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

See Also

[writeSolutionValue](#)

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                   IloXmlElement * element,
                                   IloSOS2Array array)
```

This member function writes an `IloSOS2Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS2s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                   IloXmlElement * element,
                                   IloSOS1Array array)
```

This member function writes an `IloSOS1Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS1s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
```

```
IloXmlElement * element,
IloSemiContVarArray array)
```

This member function writes an `IloSemiContVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSemiContVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloConstraintArray array)
```

This member function writes an `IloConstraintArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloConstraints` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloRangeArray array)
```

This member function writes an `IloRangeArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloRanges` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloNumVarArray array)
```

This member function writes an `IloNumVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloIntSetVarArray array)
```

This member function writes an `IloIntSetVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntSetVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloNumExprArray array)
```

This member function writes an `IloNumExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloIntExprArray array)
```

This member function writes an `IloIntExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloBoolVarArray array)
```

This member function writes an `IloBoolVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloBoolVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer,
                                IloXmlElement * element,
                                IloIntVarArray array)
```

This member function writes an `IloIntVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntVars` of `array`.

Example using `IloSOS` containing an `IloIntVarArray`:

```
// Using an IloSOS1I* exprI;
this.writeVarArray(writer,
                   element,
                   exprI->getVarArray(),
                   IloXmlAttributeDef::IdRef);
```

This sample adds an `IdRef` attribute on the `SOS` XML element, creates an XML element containing the `IloIntVarArray` with the list of `IloIntVar` IDs, and creates a list of XML elements for the `IloIntVars`.

```
public virtual IloBool writeXml(IloXmlWriter writer,
                                const IloExtractableI * exprI,
                                IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

@deprecated

```
public virtual IloBool writeXmlRef(IloXmlWriter writer,
                                   const IloExtractableI * exprI,
                                   IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

@deprecated

IloXmlReader

Category Class

InheritancePath

IloXmlReader

Definition File ilconcert/iloreader.h

Constructor Summary

public	IloXmlReader (IloEnv env, const char * fileName=0)
public	IloXmlReader (IloXmlReaderI * impl)

Method Summary

public IloBool	checkRttiOfObjectById (IloTypeIndex RTTI, IloRtti * exprI)
public IloBool	checkRttiOfObjectById (IloTypeIndex RTTI, IloInt Xml_Id)
public IloBool	checkTypeOfObjectById (IloTypeInfo type, IloInt Xml_Id)
public IloBool	checkTypeOfObjectById (IloTypeInfo type, IloRtti * exprI)
public void	deleteAllocatedMemory (const char * pointer)
public void	deleteAllocatedMemory (char * pointer)
public IloXmlElement *	findElement (IloXmlElement * root, const char * tag, const char * attribute, const char * value)
public IloXmlElement *	findElementByTag (IloXmlElement * element, const char * tag)
public IloInt	getChildrenCardinal (IloXmlElement * element)
public IloEnv	getEnv ()
public IloEnvI *	getEnvImpl ()
public IloXmlElement *	getFirstSubElement (IloXmlElement * element)
public IloBool	getIntAttribute (IloXmlElement * element, const char * attribute, IloInt & value)

public IloBool	getNumAttribute (IloXmlElement * element, const char * attribute, IloNum & value)
public IloAny	getObjectById (IloInt id)
public IloXmlElement *	getRoot ()
public IloIntArray *	getSerialized ()
public IloIntArray *	getSolutionSerialized ()
public IloBool	isSerialized (IloInt id)
public IloBool	openDocument ()
public const char *	readAttribute (IloXmlElement * element, const char * attribute)
public const char *	readCDATA (IloXmlElement * element)
public const char *	readComment (IloXmlElement * element)
public const char *	readData (IloXmlElement * element)
public const char *	readText (IloXmlElement * element)
public void	setfileName (const char * fileName)
public IloInt	string2Int (const char * str)
public IloIntArray	IloXmlReader::string2IntArray (const char * str)
public IloIntSet	string2IntSet (const char * str)
public IloNum	string2Num (const char * str)
public IloNumArray	IloXmlReader::string2NumArray (const char * str)

Description You can use an instance of `IloXmlReader` to read an `IloModel` or a `IloSolution` in XML format.

Constructors `public IloXmlReader(IloEnv env, const char * fileName=0)`

This constructor creates an `IloXmlReader` object and makes it part of the environment `env`.

The `fileName` is set to 0 by default.

`public IloXmlReader(IloXmlReaderI * impl)`

This constructor creates an XML reader from its implementation object.

Methods `public IloBool checkRttiOfObjectById(IloTypeIndex RTTI, IloRtti * exprI)`

This method checks the RTTI of the given object.

`public IloBool checkRttiOfObjectById(IloTypeIndex RTTI, IloInt Xml_Id)`

This method checks the RTTI of the object referenced by the identifier `Xml_Id` in the XML. This object must already be serialized.

```
public IloBool checkTypeOfObjectById(IloTypeInfo type,
                                       IloInt Xml_Id)
```

This method checks the TypeInfo of the object referenced by the id in the xml. This object must have been already serialized.

```
public IloBool checkTypeOfObjectById(IloTypeInfo type,
                                       IloRtti * exprI)
```

This method checks the TypeInfo of the given object

```
public void deleteAllocatedMemory(const char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the `IloXmlWriter::Int2String` member function.

```
public void deleteAllocatedMemory(char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the `IloXmlWriter::Int2String` member function.

```
public IloXmlElement * findElement(IloXmlElement * root,
                                    const char * tag,
                                    const char * attribute,
                                    const char * value)
```

This member function examines the XML element `root` to identify the XML child element denoted by `tag`, `attribute`, and `value`.

```
public IloXmlElement * findElementByTag(IloXmlElement * element,
                                          const char * tag)
```

This member function examines the XML element `element` to identify the XML child element denoted by `tag`.

```
public IloInt getChildrenCardinal(IloXmlElement * element)
```

This member function counts the number of child elements of the XML element `element`.

```
public IloEnv getEnv()
```

This member function gets the `IloEnv` of the object.

```
public IloEnvI * getEnvImpl()
```

This member function gets the implementation of the `IloEnv` of the object.

```
public IloXmlElement * getFirstSubElement(IloXmlElement * element)
```

This member function gets the first child in the XML element `element`.

```
public IloBool getIntAttribute(IloXmlElement * element,
                                const char * attribute,
```


`IloInt & value)`

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloInt`.

```
public IloBool getNumAttribute(IloXmlElement * element,
                             const char * attribute,
                             IloNum & value)
```

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloNum`.

```
public IloAny getObjectById(IloInt id)
```

This member function gets the already serialized object of the given identifier `id`.

```
IloDiff Diff(reader.getEnv(),
             IloExpr((IloNumExprI*)reader.getObjectById(IdExpr1)),
             IloExpr((IloNumExprI*)reader.getObjectById(IdExpr2)),
             reader.readAttribute(element,
                                 IloXmlAttributeDef::Name));
```

The sample code creates a `IloDiff` from a XML element referencing its two expressions with the attributes `IdRef1` and `IdRef2`.

```
public IloXmlElement * getRoot()
```

This member function gets the XML root, that is, the XML document without the header.

```
public IloIntArray * getSerialized()
```

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the model.

```
public IloIntArray * getSolutionSerialized()
```

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the solution.

```
public IloBool isSerialized(IloInt id)
```

This member function checks whether the extractable with the ID `id` in the model has already been serialized.

```
public IloBool openDocument()
```

This member function opens the XML document specified in the constructor or with the `setFileName` method.

```
public const char * readAttribute(IloXmlElement * element,
                                 const char * attribute)
```

This member function returns the value of the `attribute` in the XML element `element`.

```
public const char * readCDATA(IloXmlElement * element)
```

This member function reads the CDATA of the XML element `element`.

```
public const char * readComment(IloXmlElement * element)
```

This member function returns the value of the comment in the XML element `element`.

```
public const char * readData(IloXmlElement * element)
```

This member function reads the data of the XML element `element`.

```
public const char * readText(IloXmlElement * element)
```

This member function returns the value of the text contained in the XML element `element`, independently of its origin (data or CDATA).

```
public void setfileName(const char * fileName)
```

This member function sets `fileName` as the file from which to read the XML.

```
public IloInt string2Int(const char * str)
```

This member function converts `str` into an `IloInt`.

```
public IloIntArray string2IntArray(const char * str)
```

This member function converts `str` into an `IloIntArray`.

```
public IloIntSet string2IntSet(const char * str)
```

This member function converts `str` into an `IloIntSet`.

```
public IloNum string2Num(const char * str)
```

This member function converts `str` into an `IloNum`.

```
public IloNumArray string2NumArray(const char * str)
```

This member function converts `str` into an `IloNumArray`.

IloXmlWriter

Category Class

InheritancePath



Definition File ilconcert/ilowriter.h

Constructor Summary

public	IloXmlWriter (IloEnv env, const char * rootTag, const char * fileName=0)
public	IloXmlWriter (IloXmlWriterI * impl)

Method Summary

public void	addAttribute (IloXmlElement * element, const char * attribute, const char * value)
public void	addCDATA (IloXmlElement * element, const char * CData)
public void	addComment (IloXmlElement * element, const char * comment)
public void	addElement (IloXmlElement * element)
public void	addSubElement (IloXmlElement * element, IloXmlElement * subElement)
public void	addText (IloXmlElement * element, const char * text)
public IloXmlElement *	createElement (const char * element)
public void	deleteAllocatedMemory (const char * pointer)
public void	deleteAllocatedMemory (char * pointer)
public IloEnv	getEnv ()
public IloEnvI *	getEnvImpl ()
public const char *	getFileNames ()
public IloXmlElement *	getRoot ()
public IloIntArray *	getSerialized ()
public IloIntArray *	getSolutionSerialized ()
public const char *	IloXmlWriter::Int2String (const IloInt number)

public const char *	IloXmlWriter::IntArray2String(const IloIntArray intArray)
public const char *	IloXmlWriter::IntSet2String(const IloIntSet intSet)
public IloBool	isSerialized(IloInt id)
public IloBool	isSolutionSerialized(IloInt id)
public const char *	Num2String(const IloNum number)
public const char *	IloXmlWriter::NumArray2String(const IloNumArray numArray)
public const char *	IloXmlWriter::NumSet2String(const IloNumSet numSet)
public void	setFileName(const char * fileName)
public IloInt	string2Int(const char * str)
public void	writeDocument()

Description You can use an instance of `IloXmlWriter` to serialize an `IloModel` or an `IloSolution` in xml.

Constructors

```
public IloXmlWriter(IloEnv env,
                   const char * rootTag,
                   const char * fileName=0)
```

This constructor creates an `IloXmlWriter` object and makes it part of the environment `env`.

The `fileName` is set to 0 by default.

```
public IloXmlWriter(IloXmlWriterI * impl)
```

This constructor creates a XML writer object from its implementation object.

Methods

```
public void addAttribute(IloXmlElement * element,
                       const char * attribute,
                       const char * value)
```

This member function adds an attribute of the specified value to the XML element.

```
public void addCDATA(IloXmlElement * element,
                    const char * cData)
```

This member function adds a CDATA section to the XML element `element`.

```
public void addComment(IloXmlElement * element,
                      const char * comment)
```

This member function adds comment to the XML element `element`.

```
public void addElement(IloXmlElement * element)
```

This member function adds the XML element `element` to the end of the XML.

```
public void addSubElement(IloXmlElement * element,
                          IloXmlElement * subElement)
```

This member function adds a child element, `subElement`, to the XML element `element`.

```
public void addText(IloXmlElement * element,
                    const char * text)
```

This member function adds `text` to the specified element.

```
public IloXmlElement * createElement(const char * element)
```

This member function creates an empty element with the given tag, `element`.

```
public void deleteAllocatedMemory(const char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the `IloXmlWriter::Int2String` member function.

```
public void deleteAllocatedMemory(char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the `IloXmlWriter::Int2String` member function.

```
public IloEnv getEnv()
```

This member function gets the `IloEnv` of the object.

```
public IloEnvI * getEnvImpl()
```

This member function gets the implementation of the `IloEnv` of the object.

```
public const char * getfileName()
```

This member function returns the name of the XML file

```
public IloXmlElement * getRoot()
```

This member function gets the root XML element of the XML document.

```
public IloIntArray * getSerialized()
```

This member function gets the IDs of the serialized objects of an `IloModel`.

```
public IloIntArray * getSolutionSerialized()
```

This member function gets the IDs of the serialized objects of an `IloSolution`.

```
public const char * Int2String(const IloInt number)
```

This member function converts the `IloInt` object `number` into a string, `const char*`.

```
public const char * IntArray2String(const IloIntArray intArray)
```

This member function converts the `IloIntArray` object `intArray` into a string, `const char*`.

```
public const char * IntSet2String(const IloIntSet intSet)
```

This member function converts the IloIntSet object `intSet` into a string, `const char*`.

```
public IloBool isSerialized(IloInt id)
```

This member function checks whether an object has been serialized.

```
public IloBool isSolutionSerialized(IloInt id)
```

This member function checks whether a solution object has already been serialized.

```
public const char * Num2String(const IloNum number)
```

This member function converts the IloNum object `number` into a string, `const char*`.

```
public const char * NumArray2String(const IloNumArray numArray)
```

This member function converts the IloNumArray object `numArray` into a string, `const char*`.

```
public const char * NumSet2String(const IloNumSet numSet)
```

This member function converts the IloNumSet object `numSet` into a string, `const char*`.

```
public void setfileName(const char * fileName)
```

This member function specifies `fileName` as the name of the XML file.

```
public IloInt string2Int(const char * str)
```

This member function converts `str` into an IloInt.

```
public void writeDocument()
```

This member function outputs the XML to the file specified in the constructor or using the `setFileName` method. If null, this member function outputs on the `cout` io.

Group `optim.concert.extensions`

The ILOG Concert Extensions Library.

Classes Summary	
IloCsvLine	
IloCsvReader	
IloCsvReader::IloColumnHeaderNotFoundException	

IloCsvReader::IloCsvReaderParameterException	
IloCsvReader::IloDuplicatedTableException	
IloCsvReader::IloFieldNotFoundException	
IloCsvReader::IloFileNotFoundException	
IloCsvReader::IloIncorrectCsvReaderUseException	
IloCsvReader::IloLineNotFoundException	
IloCsvReader::IloTableNotFoundException	
IloCsvReader::LineIterator	
IloCsvReader::TableIterator	
IloCsvTableReader	
IloCsvTableReader::LineIterator	
IloIntervalList	
IloIntervalListCursor	
IloNumToAnySetStepFunction	
IloNumToAnySetStepFunctionCursor	
IloNumToNumSegmentFunction	
IloNumToNumSegmentFunctionCursor	
IloNumToNumStepFunction	
IloNumToNumStepFunctionCursor	

Global Functions Summary

IloDifference	
IloDifference	
IloIntersection	
IloMax	
IloMin	
IloUnion	
IloUnion	
operator *	
operator+	

operator-	
operator	
operator==	
operator==	
operator==	

Description**IloCsvLine**

Category Class

InheritancePath



Definition File ilconcert/ilocsvreader.h

Constructor Summary	
public	IloCsvLine()
public	IloCsvLine(IloCsvLineI * impl)
public	IloCsvLine(const IloCsvLine & csvLine)

Method Summary	
public void	copy (const IloCsvLine)
public IloBool	emptyFieldByHeader (const char * name) const
public IloBool	emptyFieldByPosition (IloInt i) const
public void	end ()
public IloNum	getFloatByHeader (const char * name) const
public IloNum	getFloatByHeaderOrDefaultValue (const char * name, IloNum defaultValue) const
public IloNum	getFloatByPosition (IloInt i) const
public IloNum	getFloatByPositionOrDefaultValue (IloInt i, IloNum defaultValue) const
public IloCsvLineI *	getImpl () const
public IloInt	getIntByHeader (const char * name) const

public IloInt	getIntByHeaderOrDefaultValue(const char * name, IloInt defaultValue) const
public IloInt	getIntByPosition(IloInt i) const
public IloInt	getIntByPositionOrDefaultValue(IloInt i, IloInt defaultValue) const
public IloInt	getLineNumber() const
public IloInt	getNumberOfFields() const
public char *	getStringByHeader(const char * name) const
public char *	getStringByHeaderOrDefaultValue(const char * name, const char * defaultValue) const
public char *	getStringByPosition(IloInt i) const
public char *	getStringByPositionOrDefaultValue(IloInt i, const char * defaultValue) const
public void	operator=(const IloCsvLine & csvLine)
public IloBool	printValueOfKeys() const

Description

An instance of `IloCsvLine` represents a line in a csv file.

Constructors

```
public IloCsvLine()
```

This constructor creates a csv line object whose handle pointer is null. This object must be assigned before it can be used.

```
public IloCsvLine(IloCsvLineI * impl)
```

This constructor creates a handle object (an instance of `IloCsvLine`) from a pointer to an implementation object (an instance of the class `IloCsvLineI`).

```
public IloCsvLine(const IloCsvLine & csvLine)
```

This copy constructor creates a handle from a reference to a csv line object. The csv line object and `csvLine` both point to the same implementation object.

Methods

```
public void copy(const IloCsvLine)
```

This member function returns the real number of the invoking csv line in the data file.

```
public IloBool emptyFieldByHeader(const char * name) const
```

This member function returns `IloTrue` if the field denoted by the string `name` in the invoking csv line is empty. Otherwise, it returns `IloFalse`.

```
public IloBool emptyFieldByPosition(IloInt i) const
```

This member function returns `IloTrue` if the field denoted by `i` in the invoking csv line is empty. Otherwise, it returns `IloFalse`.

```
public void end()
```

This member function deallocates the memory used by the csv line. If you no longer need a csv line, you can call this member function to reduce memory consumption.

```
public IloNum getFloatByHeader(const char * name) const
```

This member function returns the float contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function

`IloCsvReader::getPosition(name)` than using `getFloatByPosition` (position of `name` in the header line).

```
public IloNum getFloatByHeaderOrDefaultValue(const char * name,
                                             IloNum defaultValue) const
```

This member function returns the float contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloNum getFloatByPosition(IloInt i) const
```

This member function returns the float contained in the field `i` in the invoking csv line.

```
public IloNum getFloatByPositionOrDefaultValue(IloInt i,
                                             IloNum defaultValue) const
```

This member function returns the float contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloCsvLineI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking csv line.

```
public IloInt getIntByHeader(const char * name) const
```

This member function returns the integer contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function

`IloCsvReader::getPosition(name)` than using `getIntByPosition` (position of `name` in the header line).

```
public IloInt getIntByHeaderOrDefaultValue(const char * name,
                                             IloInt defaultValue) const
```

This member function returns the integer contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloInt getIntByPosition(IloInt i) const
```

This member function returns the integer contained in the field `i` in the invoking csv line.

```
public IloInt getIntByPositionOrDefaultValue(IloInt i,
                                             IloInt defaultValue) const
```

This member function returns the integer contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloInt getLineNumber() const
```

This member function returns the real number of the invoking csv line in the data file.

```
public IloInt getNumberOfFields() const
```

This member function returns the number of fields in the line.

```
public char * getStringByHeader(const char * name) const
```

This member function returns a reference to the string contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `getStringByPosition(position of name in the header line)`.

```
public char * getStringByHeaderOrDefaultValue(const char * name,
                                              const char * defaultValue) const
```

This member function returns the string contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public char * getStringByPosition(IloInt i) const
```

This member function returns a reference to the string contained in the field number `i` in the invoking csv line.

```
public char * getStringByPositionOrDefaultValue(IloInt i,
                                              const char * defaultValue) const
```

This member function returns the string contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public void operator=(const IloCsvLine & csvLine)
```

This operator assigns an address to the handle pointer of the invoking csv line. This address is the location of the implementation object of the argument `csvLine`.

After execution of this operator, the invoking csv line and `csvLine` both point to the same implementation object.

```
public IloBool printValueOfKeys() const
```

This member function prints the values of the keys fields in this line.

IloCsvReader

Category Class

InheritancePath

IloCsvReader

Definition File ilconcert/ilocsvreader.h

Constructor Summary

public	IloCsvReader()
public	IloCsvReader(IloCsvReaderI * impl)
public	IloCsvReader(const IloCsvReader & csv)
public	IloCsvReader(IloEnv env, const char * problem, IloBool multiTable=IloFalse, IloBool allowTableSplitting=IloFalse, const char * separator=";", const char decimalp='.', const char quote='"', const char comment='#')

Method Summary

public void	end()
public IloNum	getCsvFormat()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloNum	getFileVersion()
public IloCsvReaderI *	getImpl() const
public IloCsvLine	getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	getLineByNumber(IloInt i)
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	getNumberOfTables()
public IloInt	IloCsvReader::getPosition(const char * headingName) const
public IloCsvTableReader	getReaderForUniqueTableFile() const

public const char *	getRequiredBy()
public IloCsvTableReader	getTable()
public IloCsvTableReader	getTableByName(const char * name)
public IloCsvTableReader	getTableByNumber(IloInt i)
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvReader::IloCsvReader::IloColumnHeaderNotFoundExcep tion	
IloCsvReader::IloCsvReader::TableIterator	
IloCsvReader::IloCsvReader::LineIterator	
IloCsvReader::IloCsvReader::IloFileNotFoundException	
IloCsvReader::IloCsvReader::IloFieldNotFoundException	
IloCsvReader::IloCsvReader::IloCsvReaderParameterExcepti on	
IloCsvReader::IloCsvReader::IloDuplicatedTableException	
IloCsvReader::IloCsvReader::IloTableNotFoundException	
IloCsvReader::IloCsvReader::IloIncorrectCsvReaderUseExce ption	
IloCsvReader::IloCsvReader::IloLineNotFoundException	

Description

An instance of `IloCsvReader` reads a csv file of a specified format.

The csv file can be a multitable or a single table file. Empty lines and commented lines are allowed everywhere in the file.

Format of multitable files

The first column of the table must contain the name of the table.

Each table can begin with a line containing column headers, the first field of this line must have this format: `tableName|NAMES`

The keys can be specified in the data file by adding a line at the beginning of the table. This line is formatted as follows:

- ◆ the first field is `tableName|KEYS`
- ◆ the other fields have the value 1 if the corresponding column is a key for the table; if not they have the value 0.

If this line doesn't exist, all columns form a key. If you need to get a line having a specific value for a field, you must add the key line in which you specify that this field is a key for the table.

Any line containing '|' in its first field is ignored by the reader.

A table can be split in several parts in the file (for example, you have a part of table TA, then table TB, then the end of table TA).

Example

```
NODES|NAMES,node_type,node_name,xcoord,ycoord
NODES|KEYS,1,1,0,0
NODES,1,node1,0,1
NODES,1,node2,0,2
NODES,2,node1,0,4
```

Format of single table files

The line containing the column headers, if it exists, must have a first field of the following format: `Field|NAMES`.

Table keys can be specified by adding a line at the beginning of the table. This line must have a first field with this format: `tableName|KEYS`. If this line doesn't exist, all columns form a key.

Example

```
Field|NAMES,nodeName,xCoord,yCoord
Field|KEYS,1,0,0
node1,0,1
node2,0,2
```

Constructors

```
public IloCsvReader()
```

This constructor creates a csv reader object whose handle pointer is null. This object must be assigned before it can be used.

```
public IloCsvReader(IloCsvReaderI * impl)
```

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

```
public IloCsvReader(const IloCsvReader & csv)
```

This copy constructor creates a handle from a reference to a csv reader object. Both the csv reader object and `csv` point to the same implementation object.

```
public IloCsvReader(IloEnv env,
                    const char * problem,
                    IloBool multiTable=IloFalse,
                    IloBool allowTableSplitting=IloFalse,
                    const char * separator=";",
                    const char decimalp='.',
                    const char quote='"',
                    const char comment='#')
```

This constructor creates a csv reader object for the file `problem` in the environment `env`. If the parameter `isCached` has the value `IloTrue`, the data of the file will be stored in the memory.

The cached mode is useful only if you need to read lines by keys. It needs consequent memory consumption and takes time to load data according to the csv file size.

If the parameter `isMultiTable` has the value `IloTrue`, the file `problem` is read as a multitable file. The default value is `IloFalse`.

If the parameter `allowTableSplitting` has the value `IloFalse`, splitting the table into several parts in the file is not permitted. The default value is `IloFalse`.

The string `separator` represents the characters used as separator in the data file. The default values are `, ; .`

The character `decimal` represents the character used to write decimal numbers in the data file. The default value is `.` (period).

The character `quote` represents the character used to quote expressions.

The character `comment` represents the character used at the beginning of each commented line. The default value is `#`.

Methods

```
public void end()
```

This member function deallocates the memory used by the csv reader. If you no longer need a csv reader, you can reduce memory consumption by calling this member function.

```
public IloNum getCsvFormat()
```

This member function returns the format of the csv data file. This format is identified in the data file by `ILOG_CSV_FORMAT`.

Example

```
ILOG_CSV_FORMAT;1
```

getCsvFormat() returns 1.

Note: This member function can be used only if the parameter *isMultiTable* has the value *IloTrue*.

```
public IloCsvLine getCurrentLine() const
```

This member function returns the last line read by `getLineByKey` or `getLineByNumber`.

Note: This member function can be used only if the parameter *isMultiTable* has the value *IloFalse*.

```
public IloEnv getEnv() const
```

This member function returns the environment object corresponding to the invoking csv reader.

```
public IloNum getFileVersion()
```

This member function returns the version of the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

getFileVersion() returns 0.9.

Note: This member function can be used only if the parameter *isMultiTable* has the value *IloTrue*.

```
public IloCsvReaderI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking csv reader.


```
public IloCsvLine getLineByKey(IloInt numberOfKeys,
                               const char *,
                               ...)
```

This member function takes `numberOfKeys` as parameters; these parameters are used as one key to identify a line. It returns an instance of `IloCsvLine` representing the line having (`key1`, `key2`, ...) in the data file. If the number of keys specified is less than the number of keys in the table, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloCsvLine getLineByNumber(IloInt i)
```

This member function returns an instance of `IloCsvLine` representing the line numbered `i` in the data file. If `i` does not exist, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloInt getNumberOfColumns()
```

This member function returns the number of columns in the table. If the first column contains the name of the table it is ignored.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloInt getNumberOfItems()
```

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloInt getNumberOfKeys() const
```

This member function returns the number of keys for the table.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloInt getNumberOfTables()
```

This member function returns the number of tables in the data file.

```
public IloInt getPosition(const char * headingName) const
```

This member function returns the position (column number) of the `headingName` in the file.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public IloCsvTableReader getReaderForUniqueTableFile() const
```

This member function returns an `IloCsvTableReader` for the unique table contained in the csv data file.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public const char * getRequiredBy()
```

This member function returns the name of the project that uses the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

`getRequiredBy( )` returns PROJECTNAME.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloTrue`.*

public `IloCsvTableReader` `getTable()`

This member function returns an instance of `IloCsvTableReader` representing the unique table in the data file.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

public `IloCsvTableReader` `getTableByName`(const char * name)

This member function returns an instance of `IloCsvTableReader` representing the table named `name` in the data file.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloTrue`.*

public `IloCsvTableReader` `getTableByNumber`(`IloInt` i)

This member function returns an instance of `IloCsvTableReader` representing the table numbered `i` in the data file.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloTrue`.*

public `IloBool` `isHeadingExists`(const char * headingName) const

This member function returns `IloTrue` if the column header `headingName` exists. Otherwise, it returns `IloFalse`.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

```
public void operator=(const IloCsvReader & csv)
```

This operator assigns an address to the handle pointer of the invoking csv reader. This address is the location of the implementation object of the argument `CSV`.

After execution of this operator, both the invoking csv reader and `CSV` point to the same implementation object.

```
public IloBool printKeys() const
```

This member function prints the column header of keys if the header exists. Otherwise, it prints the column numbers of keys.

Note: *This member function can be used only if the parameter `isMultiTable` has the value `IloFalse`.*

IloCsvReader::IloColumnHeaderNotFoundException

Category Inner Class

InheritancePath

IloCsvReader::IloColumnHeaderNotFoundException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown by the member functions listed below if a header name that you use does not exist.

- ◆ [getFloatByHeader](#)
- ◆ [getIntByHeader](#)
- ◆ [getStringByHeader](#)
- ◆ [getFloatByHeaderOrDefaultValue](#)
- ◆ [getIntByHeaderOrDefaultValue](#)
- ◆ [getStringByHeaderOrDefaultValue](#)
- ◆ [IloCsvReader::getPosition](#)
- ◆ [IloCsvTableReader::getPosition](#)

IloCsvReader::IloCsvReaderParameterException

Category Inner Class

InheritancePath



IloCsvReader::IloCsvReaderParameterException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown in the constructor of the csv reader if the argument values used in the csv reader constructor are incorrect.

IloCsvReader::IloDuplicatedTableException

Category Inner Class

InheritancePath



IloCsvReader::IloDuplicatedTableException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown in the constructor of the csv reader if you read a multitable file in which two tables have the same name but table splitting has not been specified.

IloCsvReader::IloFieldNotFoundException

Category Inner Class

InheritancePath

IloCsvReader::IloFieldNotFoundException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown by the IloCsvLine methods listed below if the corresponding field does not exist.

- ◆ [getFloatByPosition](#)
- ◆ [getIntByPosition](#)
- ◆ [getStringByPosition](#)
- ◆ [getFloatByHeader](#)
- ◆ [getIntByHeader](#)
- ◆ [getStringByHeader](#)
- ◆ [getFloatByPositionOrDefaultValue](#)
- ◆ [getIntByPositionOrDefaultValue](#)
- ◆ [getStringByPositionOrDefaultValue](#)
- ◆ [getFloatByHeaderOrDefaultValue](#)
- ◆ [getIntByHeaderOrDefaultValue](#)
- ◆ [getStringByHeaderOrDefaultValue](#)

IloCsvReader::IloFileNotFoundException

Category Inner Class

InheritancePath



IloCsvReader::IloFileNotFoundException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown in the constructor of the csv reader if a specified file is not found.

IloCsvReader::IloIncorrectCsvReaderUseException

Category Inner Class

InheritancePath

IloCsvReader::IloIncorrectCsvReaderUseException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown in the following member functions if you call them from a reader built as a multitable csv reader.

- ◆ [getLineByNumber](#)
- ◆ [getLineByKey](#)
- ◆ [getNumberOfItems](#)
- ◆ [getNumberOfColumns](#)
- ◆ [getNumberOfKeys](#)
- ◆ [getReaderForUniqueTableFile](#)
- ◆ [getTable](#)
- ◆ [isHeadingExists](#)
- ◆ [printKeys](#)

This exception is throw in the following member functions if you call them from a reader built as a unique table csv reader.

- ◆ [getCsvFormat](#)
- ◆ [getFileVersion](#)
- ◆ [getTableByName](#)
- ◆ [getTableByNumber](#)
- ◆ [getRequiredBy](#)

IloCsvReader::IloLineNotFoundException

Category Inner Class

InheritancePath

IloCsvReader::IloLineNotFoundException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown by the following member functions if the line is not found.

- ◆ [IloCsvTableReader::getLineByKey](#)
- ◆ [IloCsvTableReader::getLineByNumber](#)
- ◆ [getLineByKey](#)
- ◆ [getLineByNumber](#)

IloCsvReader::IloTableNotFoundException

Category Inner Class

InheritancePath

IloCsvReader::IloTableNotFoundException

Definition File ilconcert/ilocsvreader.h

Description This exception is thrown by the constructor `IloCsvTableReader(IloCsvReaderI *, const char * name = 0)` and by the member functions listed below if the table you want to construct or to get is not found.

- ◆ [getTableByNumber](#)
- ◆ [getTableByName](#)
- ◆ [getTable](#)

IloCsvReader::LineIterator

Category	Inner Class
InheritancePath	
Definition File	ilconcert/ilocsvreader.h

Constructor Summary	
public	LineIterator()
public	LineIterator(IloCsvReader csv)

Method Summary	
public IloBool	ok() const
public IloCsvLine	operator *() const
public LineIterator &	operator++()

Description [LineIterator](#) is a nested class of the class [IloCsvReader](#). It is to be used only with csv reader objects built to read a unique-table data file.

[IloCsvReader::LineIterator](#) allows you to step through all the lines of the csv data file (except blank lines and commented lines) on which the csv reader was created.

Constructors

public [LineIterator\(\)](#)

This constructor creates an empty [LineIterator](#) object. This object must be assigned before it can be used.

public [LineIterator\(IloCsvReader csv\)](#)

This constructor creates an iterator to traverse all the lines in the csv data file on which the csv reader CSV was created.

The iterator does not traverse blank lines and commented lines.

Methods

public [IloBool](#) [ok\(\)](#) const

This member function returns [IloTrue](#) if the current position of the iterator is a valid one.

It returns [IloFalse](#) if the iterator reaches the end of the table.

```
public IloCsvLine operator *() const
```

This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

```
public LineIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

IloCsvReader::TableIterator

Category Inner Class

InheritancePath

IloCsvReader::TableIterator

Definition File ilconcert/ilocsvreader.h

Constructor Summary	
public	TableIterator (IloCsvReader csv)

Method Summary	
public IloBool	ok () const
public IloCsvTableReader	operator * () const
public TableIterator &	operator++ ()

Description [TableIterator](#) is a nested class of the class [IloCsvReader](#). It is to be used only for multitable files.

[IloCsvReader::TableIterator](#) allows you to step through all the tables of the multitable csv data file on which the csv reader was created.

Constructors public [TableIterator](#)([IloCsvReader](#) csv)
This constructor creates an iterator to traverse all the tables in the csv data file on which the csv reader CSV was created.

Methods public [IloBool](#) [ok](#)() const
This member function returns [IloTrue](#) if the current position of the iterator is a valid one.
It returns [IloFalse](#) if the iterator reaches the end of the table.
public [IloCsvTableReader](#) [operator *](#)() const

This operator returns the current instance of `IloCsvTable` (representing the current table in the csv file); the one to which the invoking iterator points.

```
public TableIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvTableReader` representing the next line in the file.

IloCsvTableReader

Category Class

InheritancePath

IloCsvTableReader

Definition File ilconcert/ilocsvreader.h

Constructor Summary

public	IloCsvTableReader()
public	IloCsvTableReader(IloCsvTableReaderI * impl)
public	IloCsvTableReader(const IloCsvTableReader & csv)
public	IloCsvTableReader(IloCsvReaderI * csvReaderImpl, const char * name=0)

Method Summary

public void	end()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloCsvTableReaderI *	getImpl() const
public IloCsvLine	IloCsvTableReader::getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	IloCsvTableReader::getLineByNumber(IloInt i)
public const char *	getNameOfTable() const
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	IloCsvTableReader::getPosition(const char *) const
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvTableReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvTableReader::IloCsvTableReader::LineIterator	

Description

An instance of `IloCsvTableReader` is used to read a csv table with a specified format.

An instance is built using a pointer to an implementation class of `IloCsvReader`, which must be created first.

Constructors

```
public IloCsvTableReader()
```

This constructor creates a table csv reader object whose handle pointer is null. This object must be assigned before it can be used.

```
public IloCsvTableReader(IloCsvTableReaderI * impl)
```

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

```
public IloCsvTableReader(const IloCsvTableReader & csv)
```

This copy constructor creates a handle from a reference to a table csv reader object.

The table csv reader object and CSV both point to the same implementation object.

```
public IloCsvTableReader(IloCsvReaderI * csvReaderImpl,
                        const char * name=0)
```

This constructor creates a table csv reader object using the implementation class of a csv reader `csvimpl`. The second parameter is the name of the table.

Methods

```
public void end()
```

This member function deallocates the memory used by the table csv reader.

If you no longer need the table csv reader, calling this member function can reduce memory consumption.

```
public IloCsvLine getCurrentLine() const
```

This member function returns the last line read using `getLineByKey` or `getLineByNumber`.

```
public IloEnv getEnv() const
```

This member function returns the environment object corresponding to the invoking table csv reader.

```
public IloCsvTableReaderI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking table csv reader.

```
public IloCsvLine getLineByKey(IloInt numberOfKeys,
                               const char *,
                               ...)
```

This member function takes numberOfKeys as parameters. These parameters are used as one key to identify a line. If the specified number of keys is less than the number of keys of the table, this member function throws an exception.

Otherwise, it returns an instance of IloCsvLine representing the line having (key1, key2, ...) in the data file.

```
public IloCsvLine getLineByNumber(IloInt i)
```

This member function returns an instance of IloCsvLine representing the line number i in the data file if it exists. Otherwise, it throws an exception.

Each time getLineByNumber or getLineByKey is called, the previous line read by one of those methods is deleted.

```
public const char * getNameOfTable() const
```

This member function returns the name of the table.

```
public IloInt getNumberOfColumns()
```

This member function returns the number of columns in the table. If the first column contains the name of the table, it is ignored.

```
public IloInt getNumberOfItems()
```

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note: This member function can be used only if the parameter *isMultiTable* has the value *IloFalse*.

```
public IloInt getNumberOfKeys() const
```

This member function returns the number of keys in the table.

```
public IloInt getPosition(const char *) const
```

This member function returns the position (column number) of headingName in the table.

```
public IloBool isHeadingExists(const char * headingName) const
```

This member function returns IloTrue if the column header named headingName exists. Otherwise, it returns IloFalse.

```
public void operator=(const IloCsvTableReader & csv)
```

This operator assigns an address to the handle pointer of the invoking table csv reader.

This address is the location of the implementation object of the argument `csv`.

After execution of this operator, the invoking table csv reader and `csv` both point to the same implementation object.

```
public IloBool printKeys() const
```

This member function prints the column headers of keys if they exist. Otherwise, it prints the column numbers of keys.

IloCsvTableReader::LineIterator

Category Inner Class

InheritancePath

IloCsvTableReader::LineIterator

Definition File ilconcert/ilocsvreader.h

Constructor Summary

public	LineIterator()
public	LineIterator(IloCsvTableReader csv)

Method Summary

public IloBool	ok() const
public IloCsvLine	operator *() const
public LineIterator &	operator++()

Description LineIterator is a nested class of the class IloCsvTableReader.

It allows you to step through all the lines of a table from a csv data file (except blank lines and commented lines) on which the table csv reader was created.

Constructors public **LineIterator**()

This constructor creates an empty LineIterator object.

This object must be assigned before it can be used.

public **LineIterator**([IloCsvTableReader](#) csv)

This constructor creates an iterator to traverse all the lines in the table csv data file on which the csv reader CSV was created.

The iterator does not traverse blank lines and commented lines.

Methods public [IloBool](#) **ok()** const

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

```
public IloCsvLine operator *() const
```

This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

```
public LineIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

IloDifference

Category Global Function

Definition File ilconcert/ilointervals.h

Synopsis

```
public IloIntervalList IloDifference(const IloIntervalList intervals1,  
                                   const IloIntervalList intervals2)
```

Description This operator creates and returns an interval list equal to the difference between the interval list `intervals1` and the interval list `intervals2`. The arguments `intervals1` and `intervals2` must be defined on the same interval. The resulting interval list is defined on the same interval as the arguments. See also: [IloIntervalList](#).

IloDifference

Category Global Function

Definition File ilconcert/ilosetfunc.h

Synopsis

```
public IloNumToAnySetStepFunction IloDifference(const  
IloNumToAnySetStepFunction f1,  
const IloNumToAnySetStepFunction f2)
```

Description This operator creates and returns a function equal to the difference between the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: [IloNumToAnySetStepFunction](#).

IloIntersection

Category	Global Function
Definition File	ilconcert/ilosetfunc.h
Synopsis	<pre>public IloNumToAnySetStepFunction IloIntersection(const IloNumToAnySetStepFunction f1, const IloNumToAnySetStepFunction f2)</pre>
Description	This operator creates and returns a function equal to the intersection between the functions f1 and f2. The argument functions f1 and f2 must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: IloNumToAnySetStepFunction.

IloIntervalList

Category Class

InheritancePath

IloIntervalList

Definition File ilconcert/ilointervals.h

Constructor Summary

public	<code>IloIntervalList(const IloEnv env, IloNum min=-IloInfinity, IloNum max=+IloInfinity, const char * name=0)</code>
public	<code>IloIntervalList(const IloEnv env, const IloNumArray times, const IloNumArray types, const char * name=0)</code>

Method Summary

public void	<code>addInterval(IloNum start, IloNum end, IloNum type=0L) const</code>
public void	<code>addPeriodicInterval(IloNum start, IloNum duration, IloNum period, IloNum end, IloNum type=0L) const</code>
public IloBool	<code>contains(const IloIntervalList intervals) const</code>
public IloIntervalList	<code>copy() const</code>
public void	<code>dilate(IloNum k) const</code>
public void	<code>empty() const</code>
public IloNum	<code>getDefinitionIntervalMax() const</code>
public IloNum	<code>getDefinitionIntervalMin() const</code>
public IloBool	<code>isEmpty() const</code>
public IloBool	<code>isKeptOpen() const</code>
public void	<code>keepOpen(IloBool val=IloTrue) const</code>
public void	<code>removeInterval(IloNum start, IloNum end) const</code>
public void	<code>removeIntervalOnDuration(IloNum start, IloNum duration) const</code>

public void	removePeriodicInterval (IloNum start, IloNum duration, IloNum period, IloNum end) const
public void	setDifference (const IloIntervalList intervals) const
public void	setPeriodic (const IloIntervalList intervals, IloNum x0, IloNum n= IloInfinity) const
public void	setUnion (const IloIntervalList intervals) const
public void	shift (IloNum dx) const

Description

An instance of the class `IloIntervalList` represents a list of non-overlapping intervals. Each interval `[timeMin, timeMax)` from the list is associated with a numerical type.

Note that if n is the number of intervals in the list, the random access to a given interval (see the member functions [addInterval](#), [contains](#), and [removeInterval](#)) has a worst-case complexity in $O(\log(n))$.

Furthermore, when two consecutive intervals of the list have the same types, these intervals are merged so that the list is always represented with the minimal number of intervals.

See Also

[IloIntervalListCursor](#), [IloUnion](#), [IloDifference](#)

Constructors

```
public IloIntervalList(const IloEnv env,
                      IloNum min=-IloInfinity,
                      IloNum max=+IloInfinity,
                      const char * name=0)
```

This constructor creates a new instance of `IloIntervalList` and adds it to the set of interval lists managed in the given environment. The arguments `min` and `max` respectively represent the origin and the horizon of the interval list. The new interval list does not contain any intervals.

```
public IloIntervalList(const IloEnv env,
                      const IloNumArray times,
                      const IloNumArray types,
                      const char * name=0)
```

This constructor creates an interval list whose intervals are defined by the two arrays `times` and `types`. More precisely, if n is the size of array `times`, then the size of array `types` must be $n-1$ and the following contiguous intervals are created on the interval list: `[times[i], times[i+1])` with type `types[i]` for all i in $[0, n-1]$.

Methods

```
public void addInterval(IloNum start,
                       IloNum end,
                       IloNum type=0L) const
```

This member function adds an interval of type `type` to the invoking interval list. The start time and end time of that newly added interval are set to `start` and `end`. By default, the type of the interval is `0`. Adding a new interval that overlaps with an already existing interval of a different type will override the existing type on the intersection.

```
public void addPeriodicInterval(IloNum start,
                                IloNum duration,
                                IloNum period,
                                IloNum end,
                                IloNum type=0L) const
```

This member function adds a set of intervals to the invoking interval list. For every $i \geq 0$ such that $start + i * period < end$, an interval of $[start + i * period, start + duration + i * period)$ is added. By default, the type of these intervals is `0`. Adding a new interval that overlaps with an already existing interval of a different type will override the existing type on the intersection.

```
public IloBool contains(const IloIntervalList intervals) const
```

This member function returns `IloTrue` if and only if each interval of `intervals` is included in an interval of the invoking interval list, regardless of interval type.

```
public IloIntervalList copy() const
```

This member function creates and returns a new interval list that is a copy of the invoking interval list.

```
public void dilate(IloNum k) const
```

This member function multiplies by `k` the scale of times for the invoking interval list. `k` must be a positive number.

```
public void empty() const
```

This member function removes all the intervals from the invoking interval list.

```
public IloNum getDefinitionIntervalMax() const
```

This member function returns the right most point (horizon) of the definition interval of the invoking interval list.

```
public IloNum getDefinitionIntervalMin() const
```

This member function returns the left most point (origin) of the definition interval of the invoking interval list.

```
public IloBool isEmpty() const
```

This member function returns `IloTrue` if and only if the invoking interval list is empty.

```
public IloBool isKeptOpen() const
```

This member function returns `IloTrue` if the interval list must be kept open. Otherwise, it returns `IloFalse`.

```
public void keepOpen(IloBool val=IloTrue) const
```

If the argument `val` is equal to `IloTrue`, this member function states that the invoking interval list must be kept open during the search for a solution to the problem. It means that additional intervals may be added during the search. Otherwise, if the argument `val` is equal to `IloFalse`, it states that all the intervals of the invoking interval list will be defined in the model before starting to solve the problem. By default, it is supposed that all the intervals of the invoking interval list are defined in the model before starting to solve the problem.

```
public void removeInterval(IloNum start,
                           IloNum end) const
```

This member function removes all intervals on the invoking interval list between `start` and `end`. If `start` is placed inside an interval `[start1, end1)`, that is, `start1 < start < end1`, this results in an interval `[start1, start)`. If `end` is placed inside an interval `[start2, end2)` this results in an interval `[end, end2)`.

```
public void removeIntervalOnDuration(IloNum start,
                                     IloNum duration) const
```

This member function removes all intervals on the invoking resource between `start` and `start+duration`.

```
public void removePeriodicInterval(IloNum start,
                                    IloNum duration,
                                    IloNum period,
                                    IloNum end) const
```

This member function removes intervals from the invoking interval list. More precisely, for every $i \geq 0$ such that $start + i * period < end$, this function removes all intervals between $start + i * period$ and $start + duration + i * period$.

```
public void setDifference(const IloIntervalList intervals) const
```

This member function removes from the invoking interval list all the intervals contained in the interval list `intervals`. The definition interval of the invoking interval list is not changed.

```
public void setPeriodic(const IloIntervalList intervals,
                        IloNum x0,
                        IloNum n=IloInfinity) const
```

This member function initializes the invoking interval list as an interval list that repeats the interval list `intervals` n times after `x0`.

```
public void setUnion(const IloIntervalList intervals) const
```

This member function sets the invoking interval list to be the union between the current interval list and the interval list `intervals`. An instance of `IloException` is thrown if two intervals with different types overlap. The definition interval of the

invoking interval list is set to the union between the current definition interval and the definition interval of `intervals`.

```
public void shift(IloNum dx) const
```

This member function shifts the intervals of the invoking interval list from `dx` to the right if `dx > 0` or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`.

IloIntervalListCursor

Category Class

InheritancePath



Definition File ilconcert/ilointervals.h

Constructor Summary	
public	IloIntervalListCursor (const IloIntervalList)
public	IloIntervalListCursor (const IloIntervalList , IloNum x)
public	IloIntervalListCursor (const IloIntervalListCursor &)

Method Summary	
public IloNum	getEnd() const
public IloNum	getStart() const
public IloNum	getType() const
public IloBool	ok() const
public void	operator++()
public void	operator--()
public void	operator=(const IloIntervalListCursor &)
public void	seek(IloNum)

Description An instance of the class [IloIntervalListCursor](#) allows you to inspect the intervals of an [IloIntervalList](#). Cursors are intended to iterate forward or backward over the intervals of an interval list.

Note: *The structure of the interval list cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the interval list, such as `addInterval`, should not be called while the cursor is being used.*

See Also

[IloIntervalList](#)

Constructors

```
public IloIntervalListCursor(const IloIntervalList)
```

This constructor creates a cursor to inspect the interval list argument. This cursor lets you iterate forward or backward over the intervals of the interval list. The cursor initially indicates the first interval of the interval list.

```
public IloIntervalListCursor(const IloIntervalList,
                             IloNum x)
```

This constructor creates a cursor to inspect the interval list `intervals`. This cursor lets you iterate forward or backward over the interval list. The cursor initially indicates the interval of the interval list that contains `x`.

Note that if n is the number of intervals of the interval list given as argument, the worst-case complexity of this constructor is $O(\log(n))$.

```
public IloIntervalListCursor(const IloIntervalListCursor &)
```

This constructor creates a new cursor that is a copy of the argument. The new cursor initially indicates the same interval and the same interval list as the argument `cursor`.

Methods

```
public IloNum getEnd() const
```

This member function returns the end point of the interval currently indicated by the cursor.

```
public IloNum getStart() const
```

This member function returns the start point of the interval currently indicated by the cursor.

```
public IloNum getType() const
```

This member function returns the type of the interval currently indicated by the cursor.

```
public IloBool ok() const
```

This member function returns `IloFalse` if the cursor does not currently indicate an interval included in the interval list. Otherwise, it returns `IloTrue`.

```
public void operator++()
```


This operator moves the cursor to the interval adjacent to the current interval (forward move).

```
public void operator--()
```

This operator moves the cursor to the interval adjacent to the current interval (backward move).

```
public void operator=(const IloIntervalListCursor &)
```

This operator assigns an address to the handle pointer of the invoking instance of **IloIntervalListCursor**. That address is the location of the implementation object of the argument **cursor**. After the execution of this operator, the invoking object and **cursor** both point to the same implementation object.

```
public void seek(IloNum)
```

This member function sets the cursor to indicate the first interval of the interval list whose end is strictly greater than x . Note that if n is the number of intervals of the interval list traversed by the invoking iterator, the worst-case complexity of this member function is $O(\log(n))$. An instance of **IloException** is thrown if x does not belong to the interval of definition of the invoking interval list.

IloMax

Category Global Function

Definition File ilconcert/ilonumfunc.h

Synopsis

```
public IloNumToNumStepFunction IloMax(const IloNumToNumStepFunction f1,
                                       const IloNumToNumStepFunction f2)
```

Description This operator creates and returns a function equal to the maximal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the $\max(f1(x), f2(x))$. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: [IloNumToNumStepFunction](#).

IloMin

Category Global Function

Definition File ilconcert/ilonumfunc.h

Synopsis

```
public IloNumToNumStepFunction IloMin(const IloNumToNumStepFunction f1,  
                                       const IloNumToNumStepFunction f2)
```

Description This operator creates and returns a function equal to the minimal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the `min(f1(x), f2(x))`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: [IloNumToNumStepFunction](#).

IloNumToAnySetStepFunction

Category Class

InheritancePath

IloNumToAnySetStepFunction

Definition File ilconcert/ilosetfunc.h

Constructor Summary

public	IloNumToAnySetStepFunction (const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)
public	IloNumToAnySetStepFunction (const IloEnv env, IloNum xmin, IloNum xmax, const IloAnySet dval, const char * name=0)

Method Summary

public void	add (const IloNumToAnySetStepFunction f) const
public void	add (IloNum xMin, IloNum xMax, IloAny elt) const
public void	add (IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const
public IloBool	alwaysContains (const IloNumToAnySetStepFunction f) const
public IloBool	alwaysContains (IloNum xMin, IloNum xMax, const IloAnySet elts) const
public IloBool	alwaysContains (IloNum xMin, IloNum xMax, IloAny elt) const
public IloBool	alwaysIntersects (const IloNumToAnySetStepFunction f) const
public IloBool	alwaysIntersects (IloNum xMin, IloNum xMax, const IloAnySet elts) const
public IloBool	contains (IloNum x, const IloAnySet elts) const
public IloBool	contains (IloNum x, IloAny elt) const
public IloNumToAnySetStepFunction	copy () const
public void	dilate (IloNum k) const

public void	<code>empty(IloNum xMin,IloNum xMax) const</code>
public <code>IloBool</code>	<code>everContains(const IloNumToAnySetStepFunction f) const</code>
public <code>IloBool</code>	<code>everContains(IloNum xMin,IloNum xMax,const IloAnySet elts) const</code>
public <code>IloBool</code>	<code>everContains(IloNum xMin,IloNum xMax,IloAny elt) const</code>
public <code>IloBool</code>	<code>everIntersects(const IloNumToAnySetStepFunction f) const</code>
public <code>IloBool</code>	<code>everIntersects(IloNum xMin,IloNum xMax,const IloAnySet elts) const</code>
public void	<code>fill(IloNum xMin,IloNum xMax) const</code>
public <code>IloAnySet</code>	<code>getComplementSet(IloNum x) const</code>
public <code>IloNum</code>	<code>getDefinitionIntervalMax() const</code>
public <code>IloNum</code>	<code>getDefinitionIntervalMin() const</code>
public <code>IloAnySet</code>	<code>getSet(IloNum x) const</code>
public <code>IloBool</code>	<code>intersects(IloNum x,const IloAnySet elts) const</code>
public <code>IloBool</code>	<code>isEmpty(IloNum x) const</code>
public <code>IloBool</code>	<code>isFull(IloNum x) const</code>
public void	<code>remove(const IloNumToAnySetStepFunction f) const</code>
public void	<code>remove(IloNum xMin,IloNum xMax,IloAny elt) const</code>
public void	<code>remove(IloNum xMin,IloNum xMax,const IloAnySet elts,IloBool complt=IloFalse) const</code>
public void	<code>set(IloNum xMin,IloNum xMax,IloAny elt) const</code>
public void	<code>set(IloNum xMin,IloNum xMax,const IloAnySet elts,IloBool complt=IloFalse) const</code>
public void	<code>setIntersection(const IloNumToAnySetStepFunction f) const</code>
public void	<code>setIntersection(IloNum xMin,IloNum xMax,IloAny elt) const</code>
public void	<code>setIntersection(IloNum xMin,IloNum xMax,const IloAnySet elts,IloBool complt=IloFalse) const</code>
public void	<code>setPeriodic(const IloNumToAnySetStepFunction f,IloNum x0,IloNum n,const IloAnySet dval) const</code>
public void	<code>shift(IloNum dx,const IloAnySet dval) const</code>
public <code>IloBool</code>	<code>usesComplementaryRepresentation(IloNum x) const</code>

Description

An instance of `IloNumToAnySetStepFunction` represents a step function that associates sets with intervals. It is defined everywhere on an interval $[xMin,xMax)$. Each interval $[x1,x2)$ on which the function has the same set is called a step.

Note that if n is the number of steps of the function, the random access to a given step (see the member functions [add](#), [alwaysIntersects](#), [contains](#), [empty](#), [everContains](#), [everIntersects](#), [fill](#), [getComplementSet](#), [getSet](#), [intersects](#), [isEmpty](#), [isFull](#), [remove](#), [set](#), and [setIntersection](#)) has a worst-case complexity of $O(\log(n))$.

Complementary Representation of Values

`IloNumToAnySetStepFunction` allows the implicit representation of infinite sets through the representation of the complement of the actual set value. This, for example, allows you to completely fill a set (using the [fill](#) member function) and then specify the elements that are not in the set. Under normal circumstances, it is not necessary to know if the value of the step function at a particular point is represented by the set or its complement: all the member functions that manipulate the step function value will correctly adapt to either representation. The only case where it is necessary to know the internal representation is if you want to directly access the set that represents a value (using the [getSet](#) or [getComplementSet](#) member functions). In that circumstance only, it is necessary to use the [usesComplementaryRepresentation](#) member function to determine the internal representation, and then use either [getSet](#) or [getComplementSet](#) depending on the return value of [usesComplementaryRepresentation](#). Note that [getSet](#) will raise an error if it is used to access a set that is represented as a complement set. [getComplementSet](#) will raise an error if it is used to access a set that is directly represented.

See Also

[IloNumToAnySetStepFunctionCursor](#)

Constructors

```
public IloNumToAnySetStepFunction(const IloEnv env,
                                IloNum xmin=-IloInfinity,
                                IloNum xmax=IloInfinity,
                                const char * name=0)
```

This constructor creates a step function defined everywhere on the interval `[xmin, xmax)` with empty set as the value.

```
public IloNumToAnySetStepFunction(const IloEnv env,
                                IloNum xmin,
                                IloNum xmax,
                                const IloAnySet dval,
                                const char * name=0)
```

This constructor creates a step function defined everywhere on the interval `[xmin, xmax)` with the same set `dval`.

Methods

```
public void add(const IloNumToAnySetStepFunction f) const
```

This member function adds the value of `f` at point `x` to the value of the invoking step function at point `x`, for all points `x` in the definition interval of the invoking step

function. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public void add(IloNum xMin,
               IloNum xMax,
               IloAny elt) const
```

This member function adds `elt` to the value of the invoking step function on the interval `[xMin, xMax)`.

```
public void add(IloNum xMin,
               IloNum xMax,
               const IloAnySet elts,
               IloBool complt=IloFalse) const
```

This member function adds the elements of `elts` to the value of the invoking step function on the interval `[xMin, xMax)`.

```
public IloBool alwaysContains(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if for all points `x` on the definition interval of the invoking step function, the value of `f` at point `x` is a subset of the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public IloBool alwaysContains(IloNum xMin,
                              IloNum xMax,
                              const IloAnySet elts) const
```

This member function returns `IloTrue` if `elts` is a subset of the value of the invoking step function at all points on the interval `[xMin, xMax)`.

```
public IloBool alwaysContains(IloNum xMin,
                              IloNum xMax,
                              IloAny elt) const
```

This member function returns `IloTrue` if at all points on the interval `[xMin, xMax)` the value of the invoking step function contains `elt`.

```
public IloBool alwaysIntersects(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if for all points `x` in the definition interval of the invoking step function, the intersection of `f` and the invoking step function is not empty. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public IloBool alwaysIntersects(IloNum xMin,
                              IloNum xMax,
                              const IloAnySet elts) const
```

This member function returns `IloTrue` if for all `x` on the interval `[xMin, xMax)` the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

```
public IloBool contains(IloNum x,
                        const IloAnySet elts) const
```

This member function returns `IloTrue` if `elts` is a subset of the value of the invoking step function at point `x`.

```
public IloBool contains(IloNum x,
                        IloAny elt) const
```

This member function returns `IloTrue` if the invoking step function contains element `elt` at point `x`.

```
public IloNumToAnySetStepFunction copy() const
```

This member function creates and returns a new function that is a copy of the invoking function.

```
public void dilate(IloNum k) const
```

This member function multiplies by `k` the scale of `x` for the invoking step function. `k` must be a non-negative numerical value. More precisely, if the invoking step function was defined over an interval `[xMin, xMax)`, it will be redefined over the interval `[k*xMin, k*xMax)` and the value at `x` will be the former value at `x/k`.

```
public void empty(IloNum xMin,
                  IloNum xMax) const
```

This member function sets the value of the invoking step function on the interval `[xMin, xMax)` to be the empty set.

```
public IloBool everContains(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if at any point `x` in the definition interval of the invoking step function, `f` at point `x` is a subset of the invoking step function at point `x`. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public IloBool everContains(IloNum xMin,
                            IloNum xMax,
                            const IloAnySet elts) const
```

This member function returns `IloTrue` if at any point on the interval `[xMin, xMax)` `elts` is a subset of the value of the invoking step function.

```
public IloBool everContains(IloNum xMin,
                            IloNum xMax,
                            IloAny elt) const
```

This member function returns `IloTrue` if at any point on the interval `[xMin, xMax)` the value of the invoking step function contains `elt`.

```
public IloBool everIntersects(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if at some point `x` in the definition interval of the invoking step function, the intersection of `f` and the invoking step function is not

empty. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public IloBool everIntersects(IloNum xMin,
                               IloNum xMax,
                               const IloAnySet elts) const
```

This member function returns `IloTrue` if at any point `x` on the interval `[xMin, xMax)` the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

```
public void fill(IloNum xMin,
                 IloNum xMax) const
```

This member function sets the value of the invoking step function on the interval `[xMin, xMax)` to be the full set.

```
public IloAnySet getComplementSet(IloNum x) const
```

This member function returns the complement of the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the invoking step function at point `x` does not use the complementary representation. See [Complementary Representation of Values](#) for more information.

```
public IloNum getDefinitionIntervalMax() const
```

This member function returns the right-most point of the definition interval of the invoking step function.

```
public IloNum getDefinitionIntervalMin() const
```

This member function returns the left-most point of the definition interval of the invoking step function.

```
public IloAnySet getSet(IloNum x) const
```

This member function returns the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the invoking step function at point `x` uses the complementary representation. See [Complementary Representation of Values](#) for more information.

```
public IloBool intersects(IloNum x,
                           const IloAnySet elts) const
```

This member function returns `IloTrue` if the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

```
public IloBool isEmpty(IloNum x) const
```

This member function returns `IloTrue` if the function is empty at point `x`. In other words, a return of `IloTrue` means that the member function `empty` has been applied to point `x` and no elements have been subsequently added to the value of the invoking step function at point `x`.

```
public IloBool isFull(IloNum x) const
```

This member function returns `IloTrue` if the function is full at point `x`. In other words, a return of `IloTrue` means that the member function `fill` has been applied to point `x` and no elements have been subsequently removed from the value of the invoking step function at point `x`.

```
public void remove(const IloNumToAnySetStepFunction f) const
```

This member function removes the value of `f` from the value of the invoking step function at all points on the definition interval of the invoking step function. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public void remove(IloNum xMin,
                  IloNum xMax,
                  IloAny elt) const
```

This member function removes `elt` from the value of the invoking step function on the interval `[xMin, xMax)`.

```
public void remove(IloNum xMin,
                  IloNum xMax,
                  const IloAnySet elts,
                  IloBool complt=IloFalse) const
```

This member function removes all the elements in `elts` from the value of the invoking step function on the interval `[xMin, xMax)`.

```
public void set(IloNum xMin,
               IloNum xMax,
               IloAny elt) const
```

This member function sets the value of the invoking step function to be `elt` on the interval `[xMin, xMax)`.

```
public void set(IloNum xMin,
               IloNum xMax,
               const IloAnySet elts,
               IloBool complt=IloFalse) const
```

This member function sets the value of the invoking step function to be `elts` on the interval `[xMin, xMax)`.

```
public void setIntersection(const IloNumToAnySetStepFunction f) const
```

This member function assigns the value of the invoking step function at all points `x` on the definition interval of the invoking step function to be the intersection of the value of `f` at point `x` and the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public void setIntersection(IloNum xMin,
                          IloNum xMax,
```

```
IloAny elt) const
```

This member function assigns the value of the invoking step function at all points x on the interval $[xMin, xMax)$ to be the intersection of the set containing `elt` and the value of the invoking set function at point x .

```
public void setIntersection(IloNum xMin,
                          IloNum xMax,
                          const IloAnySet elts,
                          IloBool complt=IloFalse) const
```

This member function assigns the value of the invoking step function at all points x on the interval $[xMin, xMax)$ to be the intersection of `elts` and the value of the invoking set function at point x .

```
public void setPeriodic(const IloNumToAnySetStepFunction f,
                      IloNum x0,
                      IloNum n,
                      const IloAnySet dval) const
```

This member function initializes the invoking step function as a function that repeats the step function `f`, `n` times after `x0`. More precisely, if `f` is defined on $[xfpMin, xfpMax)$ and if the invoking step function is defined on $[xMin, xMax)$, the value of the invoking step function will be:

- ◆ `dval` on $[xMin, x0)$,
- ◆ $f((x-x0) \% (xfpMax-xfpMin))$ for x in $[x0, \text{Min}(x0+n*(xfpMax-xfpMin), xMax))$, and
- ◆ `dval` on $[\text{Min}(x0+n*(xfpMax-xfpMin), xMax), xMax)$

```
public void shift(IloNum dx,
                 const IloAnySet dval) const
```

This member function shifts the invoking step function from `dx` to the right if `dx > 0`, or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`. More precisely, if the invoking step function is defined on $[xMin, xMax)$ and `dx > 0`, the new value of the invoking step function is:

- ◆ `dval` on the interval $[xMin, xMin+dx)$,
- ◆ for all x in $[xMin+dx, xMax)$, the former value at $x-dx$.

If `dx < 0`, the new value of the invoking step function is:

- ◆ for all x in $[xMin, xMax+dx)$, the former value at $x-dx$,
- ◆ `dval` on the interval $[xMax+dx, xMax)$.

```
public IloBool usesComplementaryRepresentation(IloNum x) const
```

This member function returns `IloTrue` if the value of the `invoking` function at point `x` is represented by a `complementary` set, rather than by directly representing the value as a set itself. See [Complementary Representation of Values](#) for more information.

IloNumToAnySetStepFunctionCursor

Category Class

InheritancePath

IloNumToAnySetStepFunctionCursor

Definition File ilconcert/ilosetfunc.h

Constructor Summary

public	IloNumToAnySetStepFunctionCursor (const IloNumToAnySetStepFunction)
public	IloNumToAnySetStepFunctionCursor (const IloNumToAnySetStepFunction , IloNum x)
public	IloNumToAnySetStepFunctionCursor (const IloNumToAnySetStepFunctionCursor &)

Method Summary

public IloAnySet	getComplementSet () const
public IloNum	getSegmentMax () const
public IloNum	getSegmentMin () const
public IloAnySet	getSet () const
public IloBool	isEmpty () const
public IloBool	isFull () const
public IloBool	ok () const
public void	operator++ ()
public void	operator-- ()
public void	operator= (const IloNumToAnySetStepFunctionCursor &)
public void	seek (IloNum)
public IloBool	usesComplementaryRepresentation () const

Description An instance of the class [IloNumToAnySetStepFunctionCursor](#) allows you to inspect the contents of an [IloNumToAnySetStepFunction](#). A step of a step function is defined as an interval [x1,x2) over which the value of the function is the

same. Cursors are intended to iterate forward or backward over the steps of a step function.

Note: *The structure of the step function cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the step function, such as `set`, should not be called while the cursor is being used.*

See Also

[IloNumToAnySetStepFunction](#)

Constructors

```
public IloNumToAnySetStepFunctionCursor(const
IloNumToAnySetStepFunction)
```

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially indicates the first step of the function.

```
public IloNumToAnySetStepFunctionCursor(const
IloNumToAnySetStepFunction,
                                     IloNum x)
```

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially indicates the step of the function that contains `x`.

Note that if n is the number of steps of the function given as argument, the worst-case complexity of this constructor is $O(\log(n))$.

```
public IloNumToAnySetStepFunctionCursor(const
IloNumToAnySetStepFunctionCursor &)
```

This constructor creates a new cursor that is a copy of the argument. The new cursor initially indicates the same step and the same function as the argument cursor.

Methods

```
public IloAnySet getComplementSet() const
```

This member function returns the set representing the complement of the value of the step currently indicated by the cursor. An instance of `IloException` is thrown if the value of the step does not use a complementary representation.

```
public IloNum getSegmentMax() const
```

This member function returns the right-most point of the step currently indicated by the cursor.

```
public IloNum getSegmentMin() const
```

This member function returns the left-most point of the step currently indicated by the cursor.

```
public IloAnySet getSet() const
```

This member function returns the value of the step currently indicated by the cursor. An instance of `IloException` is thrown if the value of the step uses a complementary representation.

```
public IloBool isEmpty() const
```

This member function returns `IloTrue` if the value of the current step is the empty set.

```
public IloBool isFull() const
```

This member function returns `IloTrue` if the value of the current step is the full set. (See also: `isFull`).

```
public IloBool ok() const
```

This member function returns `IloFalse` if the cursor does not currently indicate a step included in the definition interval of the step function. Otherwise, it returns `IloTrue`.

```
public void operator++()
```

This operator moves the cursor to the step adjacent to the current step (forward move).

```
public void operator--()
```

This operator moves the cursor to the step adjacent to the current step (backward move).

```
public void operator=(const IloNumToAnySetStepFunctionCursor &)
```

This operator assigns an address to the handle pointer of the invoking instance of `IloNumToAnySetStepFunctionCursor`. That address is the location of the implementation object of the argument `cursor`. After the execution of this operator, the invoking object and `CURSOR` both point to the same implementation object.

```
public void seek(IloNum)
```

This member function sets the cursor to indicate the step of the function that contains x . Note that if n is the number of steps of the step function traversed by the invoking iterator, the worst-case complexity of this member function is $O(\log(n))$. An instance of `IloException` is thrown if x does not belong to the definition interval of the invoking function.

```
public IloBool usesComplementaryRepresentation() const
```

This member function returns `IloTrue` if the value of the current step uses the complementary representation.

IloNumToNumSegmentFunction

Category

Class

InheritancePath

IloNumToNumSegmentFunction

Definition File

ilconcert/ilosegfunc.h

Constructor Summary

public	IloNumToNumSegmentFunction (const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)
public	IloNumToNumSegmentFunction (const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)
public	IloNumToNumSegmentFunction (const IloNumToNumStepFunction & numFunction)

Method Summary

public void	addValue (IloNum x1, IloNum x2, IloNum v) const
public IloNumToNumSegmentFunction	copy () const
public void	dilate (IloNum k) const
public IloNum	getArea (IloNum x1, IloNum x2) const
public IloNum	getDefinitionIntervalMax () const
public IloNum	getDefinitionIntervalMin () const
public IloNum	getMax (IloNum x1, IloNum x2) const
public IloNum	getMin (IloNum x1, IloNum x2) const
public IloNum	getValue (IloNum x) const
public void	operator *= (IloNum k) const
public void	operator += (const IloNumToNumSegmentFunction fct) const
public void	operator -= (const IloNumToNumSegmentFunction fct) const

public void	setMax (const IloNumToNumSegmentFunction fct) const
public void	setMax (IloNum x1, IloNum v1, IloNum x2, IloNum v2) const
public void	setMax (IloNum x1, IloNum x2, IloNum v) const
public void	setMin (const IloNumToNumSegmentFunction fct) const
public void	setMin (IloNum x1, IloNum v1, IloNum x2, IloNum v2) const
public void	setMin (IloNum x1, IloNum x2, IloNum v) const
public void	setPeriodic (const IloNumToNumSegmentFunction f, IloNum x0, IloNum n=IloInfinity, IloNum dval=0) const
public void	setPeriodicValue (IloNum x1, IloNum x2, const IloNumToNumSegmentFunction f, IloNum offset=0) const
public void	setPoints (const IloNumArray x, const IloNumArray v) const
public void	setSlope (IloNum x1, IloNum x2, IloNum v, IloNum slope) const
public void	setValue (IloNum x1, IloNum x2, IloNum v) const
public void	shift (IloNum dx, IloNum dval=0) const

Description

An instance of `IloNumToNumSegmentFunction` represents a piecewise linear function that is defined everywhere on an interval $[xMin, xMax)$. Each interval $[x1, x2)$ on which the function is linear is called a *segment*.

Note that if n is the number of segments of the function, the random access to a given segment (see the member functions [addValue](#), [getArea](#), [getValue](#), [setValue](#)) has a worst-case complexity in $O(\log(n))$.

Furthermore, when two consecutive segments of the function are co-linear, these segments are merged so that the function is always represented with the minimal number of segments.

See Also

[IloNumToNumSegmentFunctionCursor](#)

Constructors

```
public IloNumToNumSegmentFunction(const IloEnv env,
                                   IloNum xmin=-IloInfinity,
                                   IloNum xmax=IloInfinity,
                                   IloNum dval=0.0,
                                   const char * name=0)
```

This constructor creates a piecewise linear function that is constant. It is defined everywhere on the interval $[xmin, xmax)$ with the same value `dval`.

```
public IloNumToNumSegmentFunction(const IloEnv env,
```

```
const IloNumArray x,
const IloNumArray v,
IloNum xmin=-IloInfinity,
IloNum xmax=IloInfinity,
const char * name=0)
```

This constructor creates a piecewise linear function defined everywhere on the interval $[x_{\min}, x_{\max})$ whose segments are defined by the two argument arrays x and v . More precisely, the size n of array x must be equal to the size of array v and, if the created function is defined on the interval $[x_{\min}, x_{\max})$, its values will be:

- ◆ $v[0]$ on interval $[x_{\min}, x[0])$,
- ◆ $v[i] + (t - x[i]) * (v[i+1] - v[i]) / (x[i+1] - x[i])$ for t in $[x[i], x[i+1])$ for all i in $[0, n-2]$ such that $x[i-1] <> x[i]$, and
- ◆ $v[n-1]$ on interval $[x[n-1], x_{\max})$.

```
public IloNumToNumSegmentFunction(const IloNumToNumStepFunction &
numFunction)
```

This copy constructor creates a new piecewise linear function. The new piecewise linear function is a copy of the step function `numFunction`. They point to different implementation objects.

Methods

```
public void addValue(IloNum x1,
                    IloNum x2,
                    IloNum v) const
```

This member function adds v to the value of the invoking piecewise linear function everywhere on the interval $[x_1, x_2)$.

```
public IloNumToNumSegmentFunction copy() const
```

This member function creates and returns a new function that is a copy of the invoking function.

```
public void dilate(IloNum k) const
```

This member function multiplies by k the scale of x for the invoking piecewise linear function. k must be a non-negative numerical value. More precisely, if the invoking function was defined over an interval $[x_{\min}, x_{\max})$, it will be redefined over the interval $[k * x_{\min}, k * x_{\max})$ and the value at x will be the former value at x/k .

```
public IloNum getArea(IloNum x1,
                    IloNum x2) const
```

This member function returns the area of the invoking piecewise linear function on the interval $[x_1, x_2)$. An instance of `IloException` is thrown if the interval $[x_1, x_2)$ is not included in the definition interval of the invoking function.

```
public IloNum getDefinitionIntervalMax() const
```

This member function returns the right-most point of the definition interval of the invoking piecewise linear function.

```
public IloNum getDefinitionIntervalMin() const
```

This member function returns the left-most point of the definition interval of the invoking piecewise linear function.

```
public IloNum getMax(IloNum x1,
                    IloNum x2) const
```

This member function returns the maximal value of the invoking piecewise linear function on the interval $[x1, x2)$. An instance of `IloException` is thrown if the interval $[x1, x2)$ is not included in the definition interval of the invoking function.

```
public IloNum getMin(IloNum x1,
                    IloNum x2) const
```

This member function returns the minimal value of the invoking piecewise linear function on the interval $[x1, x2)$. An instance of `IloException` is thrown if the interval $[x1, x2)$ is not included in the definition interval of the invoking function.

```
public IloNum getValue(IloNum x) const
```

This member function returns the value of the function at point x .

```
public void operator *=(IloNum k) const
```

This operator multiplies by a factor k the value of the invoking piecewise linear function everywhere on the definition interval.

```
public void operator+=(const IloNumToNumSegmentFunction fct) const
```

This operator adds the argument function `fct` to the invoking piecewise linear function.

```
public void operator-=(const IloNumToNumSegmentFunction fct) const
```

This operator subtracts the argument function `fct` from the invoking piecewise linear function.

```
public void setMax(const IloNumToNumSegmentFunction fct) const
```

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The interval of definition of `fct` must be the same as that of the invoking piecewise linear function.

```
public void setMax(IloNum x1,
                  IloNum v1,
                  IloNum x2,
                  IloNum v2) const
```

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and the value of the linear function:

$x \rightarrow v1 + (x-x1)*(v2-v1)/(x2-x1)$ everywhere on the interval $[x1, x2)$.

```
public void setMax(IloNum x1,
                  IloNum x2,
                  IloNum v) const
```

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and v everywhere on the interval $[x1, x2)$.

```
public void setMin(const IloNumToNumSegmentFunction fct) const
```

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and the value of fct everywhere on the definition interval of the invoking function. The definition interval of fct must be the same as the one of the invoking piecewise linear function.

```
public void setMin(IloNum x1,
                  IloNum v1,
                  IloNum x2,
                  IloNum v2) const
```

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and the value of the linear function:

$x \rightarrow v1 + (x-x1)*(v2-v1)/(x2-x1)$ everywhere on the interval $[x1, x2)$.

```
public void setMin(IloNum x1,
                  IloNum x2,
                  IloNum v) const
```

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and v everywhere on the interval $[x1, x2)$.

```
public void setPeriodic(const IloNumToNumSegmentFunction f,
                      IloNum x0,
                      IloNum n=IloInfinity,
                      IloNum dval=0) const
```

This member function initializes the invoking function as a piecewise linear function that repeats the piecewise linear function f , n times after $x0$. More precisely, if f is defined on $[xfpMin, xfpMax)$ and if the invoking function is defined on $[xMin, xMax)$, the value of the invoking function will be:

- ◆ $dval$ on $[xMin, x0)$,
- ◆ $f((x-x0) \% (xfpMax-xfpMin))$ for x in $[x0, \text{Min}(x0+n*(xfpMax-xfpMin), xMax))$, and
- ◆ $dval$ on $[\text{Min}(x0+n*(xfpMax-xfpMin), xMax), xMax)$

```
public void setPeriodicValue(IloNum x1,
                           IloNum x2,
                           const IloNumToNumSegmentFunction f,
```

```
IloNum offset=0) const
```

This member function changes the value of the invoking function on the interval $[x_1, x_2)$. On this interval, the invoking function is set to equal a repetition of the pattern function f with an initial offset of `offset`. The invoking function is not modified outside the interval $[x_1, x_2)$. More precisely, if $[min, max)$ denotes the definition interval of f , for all t in $[x_1, x_2)$, the invoking function at t is set to equal $f(min + (offset + t - x_1) \% (max - min))$ where $\%$ denotes the modulo operator. By default, the offset is equal to 0.

```
public void setPoints(const IloNumArray x,
                    const IloNumArray v) const
```

This member function initializes the invoking function as a piecewise linear function whose segments are defined by the two parameters arrays x and v .

More precisely, the size n of array x must be equal to the size of array v , and if the created function is defined on the interval $[xmin, xmax)$, its values will be:

- ◆ $v[0]$ on interval $[xmin, x[0])$,
- ◆ $v[i] + (t - x[i]) * (v[i+1] - v[i]) / (x[i+1] - x[i])$ for t in $[x[i], x[i+1])$ for all i in $[0, n-2]$ such that $x[i-1] < x[i]$, and
- ◆ $v[n-1]$ on interval $[x[n-1], xmax)$.

```
public void setSlope(IloNum x1,
                   IloNum x2,
                   IloNum v,
                   IloNum slope) const
```

This member function sets the value of the invoking piecewise linear function equal to f , associating for each x in $[x_1, x_2) \rightarrow f(x) = v + slope * (x - x_1)$.

```
public void setValue(IloNum x1,
                   IloNum x2,
                   IloNum v) const
```

This member function sets the value of the invoking piecewise linear function to be constant and equal to v on the interval $[x_1, x_2)$.

```
public void shift(IloNum dx,
                IloNum dval=0) const
```

This member function shifts the invoking function from dx to the right if $dx > 0$ or $-dx$ to the left if $dx < 0$. It has no effect if $dx = 0$. More precisely, if the invoking function is defined on $[xMin, xMax)$ and $dx > 0$, the new value of the invoking function is:

- ◆ $dval$ on the interval $[xMin, xMin+dx)$,
- ◆ for all x in $[xMin+dx, xMax)$, the former value at $x - dx$.

If $dx < 0$, the new value of the invoking function is:

- ◆ for all x in $[xMin, xMax+dx)$, the former value at $x-dx$,
- ◆ `dval` on the interval $[xMax+dx, xMax)$.

IloNumToNumSegmentFunctionCursor

Category Class

InheritancePath

IloNumToNumSegmentFunctionCursor

Definition File ilconcert/ilosegfunc.h

Constructor Summary

public	IloNumToNumSegmentFunctionCursor (const IloNumToNumSegmentFunction , IloNum x)
public	IloNumToNumSegmentFunctionCursor (const IloNumToNumSegmentFunctionCursor &)

Method Summary

public IloNum	getSegmentMax() const
public IloNum	getSegmentMin() const
public IloNum	getValue (IloNum t) const
public IloNum	getValueLeft() const
public IloNum	getValueRight() const
public IloBool	ok() const
public void	operator++()
public void	operator--()
public void	seek (IloNum)

Description An instance of the class [IloNumToNumSegmentFunctionCursor](#) allows you to inspect the contents of an [IloNumToNumSegmentFunction](#). A segment of a piecewise linear function is defined as an interval [x1, x2) over which the function is linear. Cursors are intended to iterate forward or backward over the segments of a piecewise linear function.

Note: *The structure of the piecewise linear function cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the piecewise linear function, such as `setValue`, should not be called while the cursor is being used.*

See Also

[IloNumToNumSegmentFunction](#)

Constructors

```
public IloNumToNumSegmentFunctionCursor(const
    IloNumToNumSegmentFunction,
    IloNum x)
```

This constructor creates a cursor to inspect the piecewise linear function argument. This cursor lets you iterate forward or backward over the segments of the function. The cursor initially indicates the segment of the function that contains `x`.

Note that if n is the number of steps of the function given as argument, the worst-case complexity of this constructor is $O(\log(n))$.

```
public IloNumToNumSegmentFunctionCursor(const
    IloNumToNumSegmentFunctionCursor &)
```

This constructor creates a new cursor that is a copy of the argument `cursor`. The new cursor initially indicates the same segment and the same function as the argument `cursor`.

Methods

```
public IloNum getSegmentMax() const
```

This member function returns the right-most point of the segment currently indicated by the cursor.

```
public IloNum getSegmentMin() const
```

This member function returns the left-most point of the segment currently indicated by the cursor.

```
public IloNum getValue(IloNum t) const
```

This member function returns the value of the piecewise linear function at time `t`. `t` must be between the left-most and the right-most point of the segment currently indicated by the cursor.

```
public IloNum getValueLeft() const
```

This member function returns the value of the function at the left-most point of the segment currently indicated by the cursor.

```
public IloNum getValueRight() const
```


This member function returns the value of the function at the right-most point of the segment currently indicated by the cursor.

```
public IloBool ok() const
```

This member function returns `IloFalse` if the cursor does not currently indicate a segment included in the definition interval of the piecewise linear function. Otherwise, it returns `IloTrue`.

```
public void operator++()
```

This operator moves the cursor to the segment adjacent to the current step (forward move).

```
public void operator--()
```

This operator moves the cursor to the segment adjacent to the current step (backward move).

```
public void seek(IloNum)
```

This member function sets the cursor to indicate the segment of the function that contains x . An `IloException` is thrown if x does not belong to the definition interval of the piecewise linear function associated with the invoking cursor.

IloNumToNumStepFunction

Category Class

InheritancePath

IloNumToNumStepFunction

Definition File ilconcert/ilonumfunc.h

Constructor Summary

public	IloNumToNumStepFunction (const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)
public	IloNumToNumStepFunction (const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)

Method Summary

public void	addValue (IloNum x1, IloNum x2, IloNum v) const
public IloNumToNumStepFunction	copy () const
public void	dilate (IloNum k) const
public IloNum	getArea (IloNum x1, IloNum x2) const
public IloNum	getDefinitionIntervalMax () const
public IloNum	getDefinitionIntervalMin () const
public IloNum	getMax (IloNum x1, IloNum x2) const
public IloNum	getMin (IloNum x1, IloNum x2) const
public IloNum	getValue (IloNum x) const
public void	operator *= (IloNum k) const
public void	operator += (const IloNumToNumStepFunction fct) const
public void	operator -= (const IloNumToNumStepFunction fct) const
public void	setMax (const IloNumToNumStepFunction fct) const

public void	setMax(IloNum x1,IloNum x2,IloNum v) const
public void	setMin(const IloNumToNumStepFunction fct) const
public void	setMin(IloNum x1,IloNum x2,IloNum v) const
public void	setPeriodic(const IloNumToNumStepFunction f,IloNum x0,IloNum n=IloInfinity,IloNum dval=0) const
public void	setPeriodicValue(IloNum x1,IloNum x2,const IloNumToNumStepFunction f,IloNum offset=0) const
public void	setSteps(const IloNumArray x,const IloNumArray v) const
public void	setValue(IloNum x1,IloNum x2,IloNum v) const
public void	shift(IloNum dx,IloNum dval=0) const

Description

An instance of `IloNumToNumStepFunction` represents a step function that is defined everywhere on an interval $[xMin, xMax]$. Each interval $[x1, x2]$ on which the function has the same value is called a step.

Note that if n is the number of steps of the function, the random access to a given step (see the member functions [addValue](#), [getArea](#), [getValue](#), [setValue](#)) has a worst-case complexity in $O(\log(n))$.

Furthermore, when two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

See Also

[IloNumToNumStepFunctionCursor](#)

Constructors

```
public IloNumToNumStepFunction(const IloEnv env,
                               IloNum xmin=-IloInfinity,
                               IloNum xmax=IloInfinity,
                               IloNum dval=0.0,
                               const char * name=0)
```

This constructor creates a step function defined everywhere on the interval $[xmin, xmax]$ with the same value `dval`.

```
public IloNumToNumStepFunction(const IloEnv env,
                               const IloNumArray x,
                               const IloNumArray v,
                               IloNum xmin=-IloInfinity,
                               IloNum xmax=IloInfinity,
                               const char * name=0)
```

This constructor creates a step function defined everywhere on the interval $[xmin, xmax]$ whose steps are defined by the two argument arrays `x` and `v`. More precisely, if

n is the size of array x , size of array v must be $n+1$ and, if the created function is defined on the interval $[x_{\min}, x_{\max})$, its values will be:

- ◆ $v[0]$ on interval $[x_{\min}, x[0])$,
- ◆ $v[i]$ on interval $[x[i-1], x[i])$ for all i in $[0, n-1]$, and
- ◆ $v[n]$ on interval $[x[n-1], x_{\max})$.

The values in the array are copied, and no modification to the arrays will be taken into account once the constructor has been called.

Methods

```
public void addValue(IloNum x1,
                    IloNum x2,
                    IloNum v) const
```

This member function adds v to the value of the invoking step function everywhere on the interval $[x1, x2)$.

```
public IloNumToNumStepFunction copy() const
```

This member function creates and returns a new function that is a copy of the invoking function.

```
public void dilate(IloNum k) const
```

This member function multiplies by k the scale of x for the invoking step function. k must be a non-negative numerical value. More precisely, if the invoking function was defined over an interval $[x_{\min}, x_{\max})$, it will be redefined over the interval $[k \cdot x_{\min}, k \cdot x_{\max})$ and the value at x will be the former value at x/k .

```
public IloNum getArea(IloNum x1,
                    IloNum x2) const
```

This member function returns the sum of the invoking step function on the interval $[x1, x2)$. An instance of `IloException` is thrown if the interval $[x1, x2)$ is not included in the definition interval of the invoking function.

```
public IloNum getDefinitionIntervalMax() const
```

This member function returns the right-most point of the definition interval of the invoking step function.

```
public IloNum getDefinitionIntervalMin() const
```

This member function returns the left-most point of the definition interval of the invoking step function.

```
public IloNum getMax(IloNum x1,
                    IloNum x2) const
```

This member function returns the maximal value of the invoking step function on the interval $[x1, x2)$. An instance of `IloException` is thrown if the interval $[x1, x2)$ is not included in the definition interval of the invoking function.

```
public IloNum getMin(IloNum x1,
                    IloNum x2) const
```

This member function returns the minimal value of the invoking step function on the interval $[x1, x2)$. An instance of `IloException` is thrown if the interval $[x1, x2)$ is not included in the definition interval of the invoking function.

```
public IloNum getValue(IloNum x) const
```

This member function returns the value of the invoking step function at x . An instance of `IloException` is thrown if x does not belong to the definition interval of the invoking function.

```
public void operator *=(IloNum k) const
```

This operator multiplies by a factor k the value of the invoking step function everywhere on the definition interval.

```
public void operator+=(const IloNumToNumStepFunction fct) const
```

This operator adds the argument function `fct` to the invoking step function.

```
public void operator-=(const IloNumToNumStepFunction fct) const
```

This operator subtracts the argument function `fct` from the invoking step function.

```
public void setMax(const IloNumToNumStepFunction fct) const
```

This member function sets the value of the invoking step function to be the maximum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The interval of definition of `fct` must be the same as that of the invoking step function.

```
public void setMax(IloNum x1,
                  IloNum x2,
                  IloNum v) const
```

This member function sets the value of the invoking step function to be the maximum between the current value and v everywhere on the interval $[x1, x2)$.

```
public void setMin(const IloNumToNumStepFunction fct) const
```

This member function sets the value of the invoking step function to be the minimum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The definition interval of `fct` must be the same as the one of the invoking step function.

```
public void setMin(IloNum x1,
                  IloNum x2,
                  IloNum v) const
```

This member function sets the value of the invoking step function to be the minimum between the current value and v everywhere on the interval $[x1, x2)$.

```
public void setPeriodic(const IloNumToNumStepFunction f,
```

```

IloNum x0,
IloNum n=IloInfinity,
IloNum dval=0) const

```

This member function initializes the invoking function as a step function that repeats the step function f , n times after $x0$. More precisely, if f is defined on $[x_{fpMin}, x_{fpMax})$ and if the invoking function is defined on $[xMin, xMax)$, the value of the invoking function will be:

- ◆ $dval$ on $[xMin, x0)$,
- ◆ $fp((x-x0) \% (x_{fpMax}-x_{fpMin}))$ for x in $[x0, Min(x0+n*(x_{fpMax}-x_{fpMin}), xMax))$, and
- ◆ $dval$ on $[Min(x0+n*(x_{fpMax}-x_{fpMin}), xMax), xMax)$

```

public void setPeriodicValue(IloNum x1,
                             IloNum x2,
                             const IloNumToNumStepFunction f,
                             IloNum offset=0) const

```

This member function changes the value of the invoking function on the interval $[x1, x2)$. On this interval, the invoking function is set to equal a repetition of the pattern function f with an initial offset of $offset$. The invoking function is not modified outside the interval $[x1, x2)$. More precisely, if $[min, max)$ denotes the definition interval of f , for all t in $[x1, x2)$, the invoking function at t is set to equal $f(min + (offset+t-x1)\%(max-min))$ where $\%$ denotes the modulo operator. By default, the offset is equal to 0.

```

public void setSteps(const IloNumArray x,
                    const IloNumArray v) const

```

This member function initializes the invoking function as a step function whose steps are defined by the two arguments arrays x and v . More precisely, if n is the size of array x , size of array v must be $n+1$ and, if the invoking function is defined on the interval $[xMin, xMax)$, its values will be:

- ◆ $v[0]$ on interval $[xMin, x[0])$,
- ◆ $v[i]$ on interval $[x[i-1], x[i])$ for all i in $[0, n-1]$, and
- ◆ $v[n]$ on interval $[x[n-1], xMax)$.

```

public void setValue(IloNum x1,
                     IloNum x2,
                     IloNum v) const

```

This member function sets the value of the invoking step function to be v on the interval $[x1, x2)$.

```

public void shift(IloNum dx,
                 IloNum dval=0) const

```

This member function shifts the invoking function from dx to the right if $dx > 0$ or from $-dx$ to the left if $dx < 0$. It has no effect if $dx = 0$. More precisely, if the invoking function is defined on $[xMin, xMax)$ and $dx > 0$, the new value of the invoking function is:

- ◆ `dval` on the interval $[xMin, xMin+dx)$,
- ◆ for all x in $[xMin+dx, xMax)$, the former value at $x-dx$.

If $dx < 0$, the new value of the invoking function is:

- ◆ for all x in $[xMin, xMax+dx)$, the former value at $x-dx$,
- ◆ `dval` on the interval $[xMax+dx, xMax)$.

IloNumToNumStepFunctionCursor

Category Class

InheritancePath



Definition File ilconcert/ilonumfunc.h

Constructor Summary	
public	IloNumToNumStepFunctionCursor (const IloNumToNumStepFunction , IloNum x)
public	IloNumToNumStepFunctionCursor (const IloNumToNumStepFunctionCursor &)

Method Summary	
public IloNum	getSegmentMax() const
public IloNum	getSegmentMin() const
public IloNum	getValue() const
public IloBool	ok() const
public void	operator++()
public void	operator--()
public void	seek(IloNum)

Description An instance of the class [IloNumToNumStepFunctionCursor](#) allows you to inspect the contents of an [IloNumToNumStepFunction](#). A step of a step function is defined as an interval [x1,x2) over which the value of the function is the same. Cursors are intended to iterate forward or backward over the steps of a step function.

Note: *The structure of the step function cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the step function, such as `setValue`, should not be called while the cursor is being used.*

See Also [IloNumToNumStepFunction](#)

Constructors

```
public IloNumToNumStepFunctionCursor(const IloNumToNumStepFunction,
                                     IloNum x)
```

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially indicates the step of the function that contains `x`.

Note that if n is the number of steps of the function given as argument, the worst-case complexity of this constructor is $O(\log(n))$.

```
public IloNumToNumStepFunctionCursor(const IloNumToNumStepFunctionCursor
&)
```

This constructor creates a new cursor that is a copy of the argument `cursor`. The new cursor initially indicates the same step and the same function as the argument `cursor`.

Methods

```
public IloNum getSegmentMax() const
```

This member function returns the right-most point of the step currently indicated by the cursor.

```
public IloNum getSegmentMin() const
```

This member function returns the left-most point of the step currently indicated by the cursor.

```
public IloNum getValue() const
```

This member function returns the value of the step currently indicated by the cursor.

```
public IloBool ok() const
```

This member function returns `IloFalse` if the cursor does not currently indicate a step included in the definition interval of the step function. Otherwise, it returns `IloTrue`.

```
public void operator++()
```

This operator moves the cursor to the step adjacent to the current step (forward move).

```
public void operator--()
```

This operator moves the cursor to the step adjacent to the current step (backward move).

```
public void seek(IloNum)
```

This member function sets the cursor to indicate the step of the function that contains `x`. An `IloException` is thrown if `x` does not belong to the definition interval of the step function associated with the invoking cursor.

IloUnion

Category Global Function

Definition File ilconcert/ilointervals.h

Synopsis

```
public IloIntervalList IloUnion(const IloIntervalList intervals1,
                                const IloIntervalList intervals2)
```

Description This operator creates and returns an interval list equal to the union of the interval lists `intervals1` and `intervals2`. The arguments `intervals1` and `intervals2` must be defined on the same interval. An instance of `IloException` is thrown if two intervals with different types overlap. The resulting interval list is defined on the same interval as the arguments. See also: [IloIntervalList](#).

IloUnion

Category Global Function

Definition File ilconcert/ilosetfunc.h

Synopsis

```
public IloNumToAnySetStepFunction IloUnion(const IloNumToAnySetStepFunction
f1,
      const IloNumToAnySetStepFunction f2)
```

Description

This operator creates and returns a function equal to the union of the functions **f1** and **f2**. The argument functions **f1** and **f2** must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: [IloNumToAnySetStepFunction](#).

operator *

Category	Global Function
Definition File	ilconcert/ilonumfunc.h
Synopsis	<pre>public IloNumToNumStepFunction operator *(const IloNumToNumStepFunction f1, IloNum k) public IloNumToNumStepFunction operator *(IloNum k, const IloNumToNumStepFunction f1)</pre>
Description	These operators create and return a function equal to the function <code>f1</code> multiplied by a factor <code>k</code> everywhere on the definition interval. The resulting function is defined on the same interval as the argument function <code>f1</code>. See also: IloNumToNumStepFunction.

operator+

Category	Global Function
Definition File	ilconcert/ilonumfunc.h
Synopsis	<pre>public IloNumToNumStepFunction operator+(const IloNumToNumStepFunction f1, const IloNumToNumStepFunction f2)</pre>
Description	This operator creates and returns a function equal the sum of the functions <code>f1</code> and <code>f2</code>. The argument functions <code>f1</code> and <code>f2</code> must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: IloNumToNumStepFunction.

operator-

Category	Global Function
Definition File	ilconcert/ilonumfunc.h
Synopsis	<pre>public IloNumToNumStepFunction operator-(const IloNumToNumStepFunction f1, const IloNumToNumStepFunction f2)</pre>
Description	This operator creates and returns a function equal to the difference between functions <code>f1</code> and <code>f2</code>. The argument functions <code>f1</code> and <code>f2</code> must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: IloNumToNumStepFunction.

operator<<

Category	Global Function
Definition File	ilconcert/ilocsvreader.h
Synopsis	<pre>public ostream & operator<<(ostream & out, const IlocsvLine & line)</pre>
Description	This operator has been overloaded to treat an <code>IlocsvLine</code> object appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument <code>line</code>.

operator==

Category	Global Function
Definition File	ilconcert/ilointervals.h
Synopsis	<pre>public IloBool operator==(const IloIntervallList intervals1, const IloIntervallList intervals2)</pre>
Description	This operator returns <code>IloTrue</code> if the interval lists are the same. That is, <code>IloTrue</code> is returned if they have the same definition interval and if they contain the same intervals. Note that it compares the content of the interval lists as well as the equality of implementation pointer. See also: IloIntervallList.

operator==

Category	Global Function
Definition File	ilconcert/ilonumfunc.h
Synopsis	<pre>public IloBool operator==(const IloNumToNumStepFunction f1, const IloNumToNumStepFunction f2)</pre>
Description	This operator returns <code>IloTrue</code> if the functions <code>f1</code> and <code>f2</code> are the same. That is, <code>IloTrue</code> is returned if they have the same definition interval and if they have the same value over time. Note that it compares the content of the functions as well as the equality of implementation pointer. See also: IloNumToNumStepFunction.

operator==

Category Global Function

Definition File ilconcert/ilosetfunc.h

Synopsis

```
public IloBool operator==(const IloNumToAnySetStepFunction f1,
                          const IloNumToAnySetStepFunction f2)
```

Description This operator returns `IloTrue` if the functions are the same. That is, `IloTrue` is returned if they have the same definition interval and if they have the same value over time. Note that it compares the content of the functions as well as the equality of implementation pointer. See also: [IloNumToAnySetStepFunction](#).

Group optim.cplex.cpp

The API of ILOG CPLEX for users of C++.

Classes Summary	
IloCplex	
IloCplex::BarrierCallbackI	
IloCplex::BranchCallbackI	
IloCplex::Callback	
IloCplex::CallbackI	
IloCplex::ContinuousCallbackI	
IloCplex::ControlCallbackI	
IloCplex::CrossoverCallbackI	
IloCplex::CutCallbackI	
IloCplex::DisjunctiveCutCallbackI	
IloCplex::Exception	
IloCplex::FlowMIRCutCallbackI	
IloCplex::FractionalCutCallbackI	
IloCplex::Goal	
IloCplex::GoalI	
IloCplex::HeuristicCallbackI	
IloCplex::IncumbentCallbackI	

IloCplex::InvalidCutException	
IloCplex::LazyConstraintCallbackI	
IloCplex::MIPCallbackI	
IloCplex::MultipleConversionException	
IloCplex::MultipleObjException	
IloCplex::NetworkCallbackI	
IloCplex::NodeCallbackI	
IloCplex::NodeEvaluator	
IloCplex::NodeEvaluatorI	
IloCplex::PresolveCallbackI	
IloCplex::ProbingCallbackI	
IloCplex::SearchLimit	
IloCplex::SearchLimitI	
IloCplex::SimplexCallbackI	
IloCplex::SolveCallbackI	
IloCplex::UnknownExtractableException	
IloCplex::UserCutCallbackI	
MIPCallbackI::NodeData	

Macros Summary	
ILOBARRIERCALLBACK0	
ILOBRANCHCALLBACK0	
ILOCONTINUOUSCALLBACK0	
ILOCMPLXGOAL0	
ILOCROSSOVERCALLBACK0	
ILOCUTCALLBACK0	
ILODISJUNCTIVECUTCALLBACK0	
ILOFRACTIONALCUTCALLBACK0	
ILOHEURISTICCALLBACK0	
ILOINCUMBENTCALLBACK0	
ILOLAZYCONSTRAINTCALLBACK0	
ILOMIPCALLBACK0	
ILONETWORKCALLBACK0	
ILONODECALLBACK0	
ILOPRESOLVECALLBACK0	

ILOPROBINGCALLBACK0	
ILOSIMPLEXCALLBACK0	
ILOSOLVECALLBACK0	
ILOUSERCUTCALLBACK0	

Enumerations Summary	
BranchCallbackI::BranchType	
Callback::Type	
ControlCallbackI::IntegerFeasibility	
GoalI::BranchType	
GoalI::IntegerFeasibility	
IloCplex::Algorithm	
IloCplex::BasisStatus	
IloCplex::BoolParam	
IloCplex::BranchDirection	
IloCplex::CplexStatus	
IloCplex::DeleteMode	
IloCplex::DualPricing	
IloCplex::IISStatus	
IloCplex::IntParam	
IloCplex::MIPEmphasisType	
IloCplex::NodeSelect	
IloCplex::NumParam	
IloCplex::PWLFormulation	
IloCplex::PrimalPricing	
IloCplex::Quality	
IloCplex::StringParam	
IloCplex::VariableSelect	

Type Definitions Summary	
GoalI::IntegerFeasibilityArray	
IloCplex::BasisStatusArray	
IloCplex::BranchDirectionArray	

IloCplex::IISStatusArray	

Description**What is IloCplex?**

IloCplex is a Concert Technology class derived from **IloAlgorithm**. Instances of this class are capable of solving optimization problems of the following types:

- ◆ Linear Programs (LPs),
- ◆ Mixed Integer Linear Programs (MILPs),
- ◆ Mixed Integer Programs (MIPs),
- ◆ Quadratic Programs (QPs),
- ◆ Mixed Integer Quadratic Programs (MIQPs),
- ◆ Quadratically Constrained Programs (QCPs);
- ◆ Mixed Integer Quadratically Constrained Programs (MIQCPs).

An instance of **IloCplex** can extract and solve models consisting of the following Concert Technology extractables:

Extractable Class	Used to Model
IloNumVar	numeric variables
IloSemiContVar	semi-continuous or semi-integer variables
IloObjective	at most one objective function with linear, piecewise linear, or quadratic expressions
IloRange	range constraints with linear or piecewise linear expressions
IloConstraint	ranged constraints of the form <i>expr1relationexpr</i> , where <i>expr1</i> indicates a linear, logical, or quadratic expression and the relation less than or equal to or the relation greater than or equal to; constraints can be combined by logical operators
IloConversion	variable type conversions
IloModel	submodels
IloSOS1	special ordered sets of type 1
IloSOS2	special ordered sets of type 2
IloAnd	constraint clauses

What is special about this set of extractable classes recognized by `IloCplex` is that models consisting of these objects can be transformed into mathematical programming problems of the form:

$$\begin{aligned} \min/\max \quad & cx + \frac{1}{2} x^T Q x \\ \text{s.t.} \quad & L \leq Ax \leq U \\ & l \leq x \leq u \end{aligned}$$

When all variables are continuous and Q is zero, problems of this form are known as Linear Programs (LPs). If Q is not zero, such problems are known as Quadratic Programs (QPs). If any variables are integer, semi-continuous, or Boolean, such problems are called Mixed Integer Programs (MIPs). A MIP with a zero Q matrix is called a Mixed Integer Linear Program (MILP), and a MIP with a non-zero Q is called a Mixed Integer Quadratic Program (MIQP). If there are quadratic constraints in the problem, and its variables are continuous, it is known as a Quadratically Constrained Program (QCP). If in addition to the quadratic constraints, there are discrete variables in the problem (such as integer, Boolean, or semi-continuous variables), then it is known as MIQCP.

- ◆ Objects of the class `IloNumVar` represent modeling variables. They are defined by the lower and upper bounds of the variable, and the type of the variable. The type of the variable can be one of these:

`ILOFLOAT`, for continuous,
`ILOINT`, for integer,
`ILOBOOL`, for Boolean variables.

- ◆ Objects of the class `IloSemiContVar` represent semi-continuous variables. A semi-continuous variable may be 0 (zero) or may take a value within an interval defined by its semi-continuous lower and upper bounds. Semi-continuous variables are usually defined as continuous variables, but you can designate an instance of `IloSemiContVar` as integer by using the type indicator it inherits from `IloNumVar`.
- ◆ Objects of the class `IloObjective` represent objective functions of optimization models. `IloCplex` deals with models containing at most one objective function, and the objective function must be linear, piecewise linear, or quadratic.
- ◆ Objects of the class `IloRange` represent constraints of the form: lower bound $\leq$ expression $\leq$ upper bound. Any floating-point value or `+/- IloInfinity` can be used for the bounds.
- ◆ Objects of the class `IloConversion` change the type of a variable in a model. This class allows you to use the same variable with different types in different models.

- ◆ Objects of the class `IloModel` represent models which consist of extractable objects. They can be used to create submodels or additional models in a given environment.
- ◆ Objects of the class `IloSOS1` represent type 1 Special Ordered Sets (SOSs). A type 1 SOS specifies that at most one variable from a set of variables may take a nonzero value. Similarly, objects of the class `IloSOS2` represent type 2 SOSs. A type 2 SOS specifies that at most two variables from a set of variables may take nonzero values and that these two variables must be neighbors with respect to a specified order of the variables. SOS1 are rarely used and SOS2 are mostly used to model piecewise linear functions, for which Concert Technology provides direct support (with the class `IloPiecewiseLinear`).
- ◆ Objects of the class `IloAnd` are used in conjunction with objects of the class `IloSolution`.

Refer to the *ILOG Concert Technology Reference Manual* for more information about these classes.

IloCplex Optimizer Options

An instance of the class `IloCplex` is not really only one algorithm, but, in fact, consists of a set of highly configurable algorithms, also known as optimizer options. They include primal and dual simplex algorithms, barrier algorithm, a sifting algorithm, a network simplex algorithm, and a branch & cut algorithm for MIPs. Though in most cases `IloCplex` can be used like a black box, the optimizer options can be selected individually to provide a wealth of parameters that allow you to fine tune the algorithm to your particular model. In the case of the mixed integer optimizer, you can use your own goals or callbacks and directly control the branch & cut search carried out by `IloCplex`.

The most general kind of problem is a MIP. You might think of the LPs as a subset of MIPs: an LP is a problem in which the model is:

- ◆ without integer variables,
- ◆ without Boolean variables,
- ◆ without semi-continuous variables,
- ◆ without piecewise linear functions,
- ◆ without a quadratic component in the objective function,
- ◆ without quadratic constraints,
- ◆ and without a special ordered set (SOS).

For linear programming problems (LPs), a variety of additional solution information can be queried. These queries include dual information or, with the appropriate optimizer option, basis information. Sensitivity analysis allows you to analyze how you can

modify your model while preserving the same solution. Or, if your model is infeasible, the infeasibility finder enables you to analyze the source of the infeasibility.

ILOBARRIERCALLBACK0

Category

Macro

Synopsis

```
ILOBARRIERCALLBACK0(name)
ILOBARRIERCALLBACK1(name, type1, x1)
ILOBARRIERCALLBACK2(name, type1, x1, type2, x2)
ILOBARRIERCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOBARRIERCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOBARRIERCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOBARRIERCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOBARRIERCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{ }` following the macro invocation, like this:

```
ILOBARRIERCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::BarrierCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::BarrierCallbackI](#)

ILOBRANCHCALLBACK0

Category

Macro

Synopsis

```
ILOBRANCHCALLBACK0(name)
ILOBRANCHCALLBACK1(name, type1, x1)
ILOBRANCHCALLBACK2(name, type1, x1, type2, x2)
ILOBRANCHCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOBRANCHCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOBRANCHCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOBRANCHCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOBRANCHCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOBRANCHCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::BranchCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::BranchCallbackI](#)

ILOCONTINUOUSCALLBACK0

Category

Macro

Synopsis

```
ILOCONTINUOUSCALLBACK0(name)
ILOCONTINUOUSCALLBACK1(name, type1, x1)
ILOCONTINUOUSCALLBACK2(name, type1, x1, type2, x2)
ILOCONTINUOUSCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCONTINUOUSCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCONTINUOUSCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOCONTINUOUSCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOCONTINUOUSCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOCONTINUOUSCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::ContinuousCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::ContinuousCallbackI](#)

ILOCPLEXGOAL0

Category

Macro

Synopsis

```
ILOCPLEXGOAL0(name)
ILOCPLEXGOAL1(name, type0, var0)
ILOCPLEXGOAL2(name, type0, var0, type1, var1)
ILOCPLEXGOAL3(name, type0, var0, type1, var1, type2, var2)
ILOCPLEXGOAL4(name, t0, v0, t1, v1, t2, v2, t3, v3)
ILOCPLEXGOAL5(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4)
ILOCPLEXGOAL6(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4, t5, v5)
```

Description

This macro defines a user goal class named `nameI` and a constructor named `name` with n data members, where n is the number following `ILOCPLEXGOAL`. The first parameter of this macro is always the name of the constructor to be created. What follows are n pairs of parameters, each parameter specifying a data member of the goal. The first parameter of such a pair specifies the type of the data member and is denoted as `Ti` in the macro definition above. The second parameter of such a pair, denoted by `datai`, specifies the data member's name.

The constructor `name` created by this function will have `IloEnvenv` as its first argument, followed by n additional arguments. The constructor creates a new instance of the user-written goal class `nameI` and populates its data members with the arguments following `IloEnvenv` in the function argument list. The constructor `name` is what you should use to create new goal objects.

The call to the macro must be followed immediately by the `execute` method of the goal class. This method must be enclosed in curly brackets, as shown in the examples that follow. The macro will also generate an implementation of the method `duplicateGoal` that simply calls the default constructor for the new class `nameI`.

You are not obliged to use this macro to define goals. In particular, if your data members do not permit the use of the default constructor as an implementation of the method `duplicateGoal` or the default destructor, you must subclass `IloCplex::GoalI` directly and implement those methods appropriately.

Since the argument `name` is used to construct the name of the goal's implementation class, it is not possible to use the same name for several goal definitions.

Example

Here's how to define a goal with one data member:

```
ILOCPLEXGOAL1(PrintX, IloInt, x) {
    IloEnv env = getEnv();
    env.out() << "PrintX: a goal with one data member" <<
endl;
    env.out() << x << endl;
```

```
    return 0;
}
```

This macro generates code similar to the following lines:

```
class PrintXI : public IloCplex::GoalI {
public:
    IloInt x;
    PrintXI(IloEnv env, IloInt arg1)
    IloCplex::Goal execute();
    IloCplex::Goal duplicateGoal();
};

PrintXI::PrintXI(IloEnv env, IloInt arg1) :
    IloCplex::GoalI(env),
    x(arg1) {
}

IloCplex::Goal PrintX(IloEnv env, IloInt x) {
    return new PrintXI(env, x);
}

IloCplex::Goal PrintXI::execute() {
    IloEnv env = getEnv();
    env.out() << "PrintX: a goal with one data member" <<
endl;
    env.out() << x << endl;
    return 0;
}

IloCplex::Goal PrintXI::duplicateGoal() {
    return new PrintXI(getEnv(), x);
}
```

ILOCROSSOVERCALLBACK0

Category

Macro

Synopsis

```
ILOCROSSOVERCALLBACK0(name)
ILOCROSSOVERCALLBACK1(name, type1, x1)
ILOCROSSOVERCALLBACK2(name, type1, x1, type2, x2)
ILOCROSSOVERCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCROSSOVERCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCROSSOVERCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOCROSSOVERCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOCROSSOVERCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILOCROSSOVERCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::CrossoverCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::CallbackI](#), [IloCplex::CrossoverCallbackI](#)

ILOUTCALLBACK0

Category

Macro

Synopsis

```
ILOUTCALLBACK0(name)
ILOUTCALLBACK1(name, type1, x1)
ILOUTCALLBACK2(name, type1, x1, type2, x2)
ILOUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOUTCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::CutCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::CutCallbackI](#)

ILODISJUNCTIVECUTCALLBACK0

Category

Macro

Synopsis

```
ILODISJUNCTIVECUTCALLBACK0(name)
ILODISJUNCTIVECUTCALLBACK1(name, type1, x1)
ILODISJUNCTIVECUTCALLBACK2(name, type1, x1, type2, x2)
ILODISJUNCTIVECUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILODISJUNCTIVECUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4,
x4)
ILODISJUNCTIVECUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5)
ILODISJUNCTIVECUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5, type6, x6)
ILODISJUNCTIVECUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILODISJUNCTIVECUTCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::DisjunctiveCutCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::DisjunctiveCutCallbackI](#)

ILOFRACTIONALCUTCALLBACK0

Category

Macro

Synopsis

```
ILOFRACTIONALCUTCALLBACK0(name)
ILOFRACTIONALCUTCALLBACK1(name, type1, x1)
ILOFRACTIONALCUTCALLBACK2(name, type1, x1, type2, x2)
ILOFRACTIONALCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFRACTIONALCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFRACTIONALCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOFRACTIONALCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOFRACTIONALCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILOFRACTIONALCUTCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::FractionalCutCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::FractionalCutCallbackI](#)

ILOHEURISTICCALLBACK0

Category

Macro

Synopsis

```
ILOHEURISTICCALLBACK0(name)
ILOHEURISTICCALLBACK1(name, type1, x1)
ILOHEURISTICCALLBACK2(name, type1, x1, type2, x2)
ILOHEURISTICCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOHEURISTICCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOHEURISTICCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOHEURISTICCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOHEURISTICCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOHEURISTICCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::HeuristicCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::HeuristicCallbackI](#)

ILOINCUMBENTCALLBACK0

Category

Macro

Synopsis

```
ILOINCUMBENTCALLBACK0(name)
ILOINCUMBENTCALLBACK1(name, type1, x1)
ILOINCUMBENTCALLBACK2(name, type1, x1, type2, x2)
ILOINCUMBENTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOINCUMBENTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOINCUMBENTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOINCUMBENTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOINCUMBENTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOINCUMBENTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::IncumbentCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::IncumbentCallbackI](#)

ILOLAZYCONSTRAINTCALLBACK0

Category

Macro

Synopsis

```
ILOLAZYCONSTRAINTCALLBACK0(name)
ILOLAZYCONSTRAINTCALLBACK1(name, type1, x1)
ILOLAZYCONSTRAINTCALLBACK2(name, type1, x1, type2, x2)
ILOLAZYCONSTRAINTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOLAZYCONSTRAINTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4,
x4)
ILOLAZYCONSTRAINTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5)
ILOLAZYCONSTRAINTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5, type6, x6)
ILOLAZYCONSTRAINTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4,
x4, type5, x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined lazy constraint callback named `nameI` and a function named `name` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `makeClone` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILOLAZYCONSTRAINTCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class `IloCplex::LazyConstraintCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::LazyConstraintCallbackI](#)

ILOMIPCALLBACK0

Category

Macro

Synopsis

```
ILOMIPCALLBACK0(name)
ILOMIPCALLBACK1(name, type1, x1)
ILOMIPCALLBACK2(name, type1, x1, type2, x2)
ILOMIPCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOMIPCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOMIPCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOMIPCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOMIPCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILOMIPCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::MIPCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::MIPCallbackI](#)

ILONETWORKCALLBACK0

Category

Macro

Synopsis

```

ILONETWORKCALLBACK0(name)
ILONETWORKCALLBACK1(name, type1, x1)
ILONETWORKCALLBACK2(name, type1, x1, type2, x2)
ILONETWORKCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONETWORKCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONETWORKCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILONETWORKCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILONETWORKCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)

```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets `{}` by the user following the macro invocation, like this:

```

ILONETWORKCALLBACKn(name, ...) {

    // implementation of the callback

}

```

For the implementation of the callback, methods from class [IloCplex::NetworkCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::NetworkCallbackI](#)

ILONODECALLBACK0

Category

Macro

Synopsis

```
ILONODECALLBACK0(name)
ILONODECALLBACK1(name, type1, x1)
ILONODECALLBACK2(name, type1, x1, type2, x2)
ILONODECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONODECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONODECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILONODECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILONODECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILONODECALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::NodeCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::NodeCallbackI](#)

ILOPRESOLVECALLBACK0

Category

Macro

Synopsis

```
ILOPRESOLVECALLBACK0(name)
ILOPRESOLVECALLBACK1(name, type1, x1)
ILOPRESOLVECALLBACK2(name, type1, x1, type2, x2)
ILOPRESOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPRESOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPRESOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOPRESOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOPRESOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOPRESOLVECALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::PresolveCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::PresolveCallbackI](#)

ILOPROBINGCALLBACK0

Category

Macro

Synopsis

```
ILOPROBINGCALLBACK0(name)
ILOPROBINGCALLBACK1(name, type1, x1)
ILOPROBINGCALLBACK2(name, type1, x1, type2, x2)
ILOPROBINGCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPROBINGCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPROBINGCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOPROBINGCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOPROBINGCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOPROBINGCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from class [IloCplex::ProbingCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::ProbingCallbackI](#)

ILOSIMPLEXCALLBACK0

Category

Macro

Synopsis

```
ILOSIMPLEXCALLBACK0(name)
ILOSIMPLEXCALLBACK1(name, type1, x1)
ILOSIMPLEXCALLBACK2(name, type1, x1, type2, x2)
ILOSIMPLEXCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOSIMPLEXCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOSIMPLEXCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOSIMPLEXCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOSIMPLEXCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOSIMPLEXCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::SimplexCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::SimplexCallbackI](#)

ILOSOLVECALLBACK0

Category

Macro

Synopsis

```

ILOSOLVECALLBACK0(name)
ILOSOLVECALLBACK1(name, type1, x1)
ILOSOLVECALLBACK2(name, type1, x1, type2, x2)
ILOSOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOSOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOSOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOSOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOSOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)

```

Description

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```

ILOSOLVECALLBACKn(name, ...) {

    // implementation of the callback

}

```

For the implementation of the callback, methods from the class [IloCplex::SolveCallbackI](#) and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

[IloCplex::SolveCallbackI](#)

ILOUSERCUTCALLBACK0

Category

Macro

Synopsis

```
ILOUSERCUTCALLBACK0(name)
ILOUSERCUTCALLBACK1(name, type1, x1)
ILOUSERCUTCALLBACK2(name, type1, x1, type2, x2)
ILOUSERCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOUSERCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOUSERCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOUSERCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOUSERCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

Description

This macro creates two things: an implementation class for a user-defined user cut callback named `nameI` and a function named `name` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `makeClone` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILOUSERCUTCALLBACKn(name, ...) {

    // implementation of the callback

}
```

For the implementation of the callback, methods from class `IloCplex::UserCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

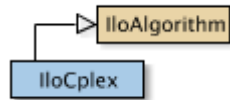
See Also

[IloCplex::UserCutCallbackI](#)

IloCplex

Category Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public	IloCplex (IloEnv env)
public	IloCplex (const IloModel model)

Method Summary

public IloConstraint	addCut (IloConstraint con)
public const IloConstraintArray	addCuts (const IloConstraintArray con)
public IloConstraint	addLazyConstraint (IloConstraint con)
public const IloConstraintArray	addLazyConstraints (const IloConstraintArray con)
public IloConstraint	addUserCut (IloConstraint con)
public const IloConstraintArray	addUserCuts (const IloConstraintArray con)
public static IloCplex::Goal	Apply (IloCplex cplex, IloCplex::Goal goal, IloCplex::NodeEvaluator eval)
public void	basicPresolve (const IloNumVarArray vars, IloNumArray redlb=0, IloNumArray redub=0, const IloRangeArray rngs=0, IloBoolArray redundant=0) const
public void	clearCuts ()
public void	clearLazyConstraints ()
public void	clearModel ()

public void	clearUserCuts()
public void	delDirection(IloNumVar var)
public void	delDirections(const IloNumVarArray var)
public void	delPriorities(const IloNumVarArray var)
public void	delPriority(IloNumVar var)
public IloNum	dualFarkas(IloConstraintArray rng,IloNumArray y)
public void	exportModel(const char * filename) const
public IloBool	feasOpt(const IloRangeArray rngs,IloNumArray rnglb,IloNumArray rngub,IloBool optimize=IloFalse)
public IloBool	feasOpt(const IloNumVarArray vars,IloNumArray varlb,IloNumArray varub,IloBool optimize=IloFalse)
public IloBool	feasOpt(const IloRangeArray rngs,IloNumArray rnglb,IloNumArray rngub,const IloNumVarArray vars,IloNumArray varlb,IloNumArray varub,IloBool optimize=IloFalse)
public void	freePresolve()
public IloCplex::Algorithm	getAlgorithm() const
public void	getAX(IloNumArray val,const IloRangeArray con) const
public IloNum	getAX(const IloRange range) const
public IloCplex::BasisStatus	getBasisStatus(const IloConstraint con) const
public IloCplex::BasisStatus	getBasisStatus(const IloIntVar var) const
public IloCplex::BasisStatus	getBasisStatus(const IloNumVar var) const
public void	getBasisStatuses(IloCplex::BasisStatusArray cstat,const IloNumVarArray var,IloCplex::BasisStatusArray rstat,const IloConstraintArray con) const
public void	getBasisStatuses(IloCplex::BasisStatusArray stat,const IloConstraintArray con) const
public void	getBasisStatuses(IloCplex::BasisStatusArray stat,const IloNumVarArray var) const
public IloNum	getBestObjValue() const
public void	getBoundSA(IloNumArray lblower,IloNumArray lbupper,IloNumArray ublower,IloNumArray ubupper,const IloNumVarArray vars) const
public IloCplex::CplexStatus	getCplexStatus() const
public IloCplex::CplexStatus	getCplexSubStatus() const
public IloNum	getCutoff() const
public IloBool	getDefault(IloCplex::BoolParam parameter) const
public IloCplex::DeleteMode	getDeleteMode() const

public IloCplex::BranchDirection	getDirection(IloNumVar var) const
public void	getDirections(IloCplex::BranchDirectionArray dir,const IloNumVarArray var) const
public IloExtractable	getDiverging() const
public IloNum	getDual(const IloRange range) const
public void	getDUALS(IloNumArray val,const IloRangeArray con) const
public IloNum	getFormulationBigValue()
public IloNum	getFormulationEpsValue()
public void	getIIS(IloCplex::IISStatusArray cstat,IloNumVarArray var,IloCplex::IISStatusArray rstat,IloConstraintArray rng) const
public IloInt	getIncumbentNode() const
public IloInt	getMax(IloCplex::IntParam parameter) const
public IloInt	getMin(IloCplex::IntParam parameter) const
public IloInt	getNbarrierIterations() const
public IloInt	getNbinVars() const
public IloInt	getNcliques() const
public IloInt	getNcols() const
public IloInt	getNcovers() const
public IloInt	getNcrossDExch() const
public IloInt	getNcrossDPush() const
public IloInt	getNcrossPExch() const
public IloInt	getNcrossPPush() const
public IloInt	getNdualSuperbasics() const
public IloInt	getNintVars() const
public IloInt	getNiterations() const
public IloInt	getNnodes() const
public IloInt	getNnodesLeft() const
public IloInt	getNNZs() const
public IloInt	getNphaseOneIterations() const
public IloInt	getNprimalSuperbasics() const
public IloInt	getNQCs() const
public IloInt	getNrows() const
public IloInt	getNsemiContVars() const
public IloInt	getNsemiIntVars() const
public IloInt	getNsiftingIterations() const
public IloInt	getNsiftingPhaseOneIterations() const
public IloInt	getNSOSs() const
public void	getObjSA(IloNumArray lower,IloNumArray upper,const IloNumVarArray vars) const

public IloBool	getParam(IloCplex::BoolParam parameter) const
public void	getPriorities(IloNumArray pri,const IloNumVarArray var) const
public IloNum	getPriority(IloNumVar var) const
public IloCplex::PWLFormulation	getPWLFormulation()
public IloNum	getQuality(IloCplex::Quality q,IloNumVar * var=0,IloConstraint * rng=0) const
public void	getRangeSA(IloNumArray lblower,IloNumArray lbupper,IloNumArray ublower,IloNumArray ubupper,const IloRangeArray con) const
public void	getRay(IloNumArray vals,IloNumVarArray vars) const
public IloNum	getReducedCost(const IloIntVar var) const
public IloNum	getReducedCost(const IloNumVar var) const
public void	getReducedCosts(IloNumArray val,const IloIntVarArray var) const
public void	getReducedCosts(IloNumArray val,const IloNumVarArray var) const
public void	getRHSSA(IloNumArray lower,IloNumArray upper,const IloRangeArray cons) const
public IloNum	getSlack(const IloRange range) const
public void	getSlacks(IloNumArray val,const IloRangeArray con) const
public IloAlgorithm::Status	getStatus() const
public IloCplex::Algorithm	getSubAlgorithm() const
public void	getValues(const IloIntVarArray var,IloNumArray val) const
public void	getValues(IloNumArray val,const IloIntVarArray var) const
public void	getValues(const IloNumVarArray var,IloNumArray val) const
public void	getValues(IloNumArray val,const IloNumVarArray var) const
public const char *	getVersion() const
public void	importModel(IloModel & m,const char * filename) const
public void	importModel(IloModel & m,const char * filename,IloObjective & obj,IloNumVarArray vars,IloRangeArray rngs,IloRangeArray lazy=0,IloRangeArray cuts=0) const
public void	importModel(IloModel & model,const char * filename,IloObjective & obj,IloNumVarArray vars,IloRangeArray rngs,IloSOS1Array sos1,IloSOS2Array sos2,IloRangeArray lazy=0,IloRangeArray cuts=0) const

public IloBool	isDualFeasible() const
public IloBool	isMIP() const
public IloBool	isPrimalFeasible() const
public IloBool	isQC() const
public IloBool	isQO() const
public static IloCplex::Goal	LimitSearch (IloCplex cplex , IloCplex::Goal goal, IloCplex::SearchLimit limit)
public void	presolve (IloCplex::Algorithm alg)
public void	protectVariables (const IloIntVarArray var)
public void	protectVariables (const IloNumVarArray var)
public void	qpIndefCertificate (IloNumVarArray var, IloNumArray x)
public void	readBasis (const char * name) const
public void	readMIPStart (const char * name) const
public void	readOrder (const char * filename) const
public void	readParam (const char * name) const
public void	readTree (const char * name) const
public void	readVectors (const char * name) const
public void	setBasisStatuses (const IloCplex::BasisStatusArray cstat,const IloNumVarArray var,const IloCplex::BasisStatusArray rstat,const IloConstraintArray con)
public void	setDefaultts ()
public void	setDeleteMode (IloCplex::DeleteMode mode)
public void	setDirection (IloNumVar var, IloCplex::BranchDirection dir)
public void	setDirections (const IloNumVarArray var,const IloCplex::BranchDirectionArray dir)
public void	setFormulationBigValue (IloNum v)
public void	setFormulationEpsValue (IloNum eps)
public void	setParam (IloCplex::BoolParam parameter, IloBool value)
public void	setPriorities (const IloNumVarArray var,const IloNumArray pri)
public void	setPriority (IloNumVar var, IloNum pri)
public void	setPWLFormulation (IloCplex::PWLFormulation f)
public void	setVectors (const IloNumArray x,const IloNumArray dj,const IloNumVarArray var,const IloNumArray slack,const IloNumArray pi,const IloRangeArray rng)
public IloBool	solve (IloCplex::Goal goal)
public IloBool	solve ()

public IloBool	solveFixed()
public IloCplex::Callback	use(IloCplex::Callback cb)
public void	writeBasis(const char * name) const
public void	writeMIPStart(const char * name) const
public void	writeOrder(const char * filename) const
public void	writeParam(const char * name) const
public void	writeTextSolution(const char * name) const
public void	writeTree(const char * name) const
public void	writeVectors(const char * name) const

Inherited methods from **IloAlgorithm**

clear, end, error, IloAlgorithm::extract, getEnv, getIntValue, getIntValues, getObjValue, getStatus, getTime, getValue, getValue, getValue, getValue, getValues, getValues, isExtracted, out, printTime, resetTime, setError, setOut, setWarning, solve, warning

Inner Enumeration

IloCplex::Algorithm	
IloCplex::BasisStatus	
IloCplex::BoolParam	
IloCplex::BranchDirection	
IloCplex::CplexStatus	
IloCplex::DeleteMode	
IloCplex::DualPricing	
IloCplex::IISStatus	
IloCplex::IntParam	
IloCplex::MIPEmphasisType	
IloCplex::NodeSelect	
IloCplex::NumParam	
IloCplex::PWLFormulation	
IloCplex::PrimalPricing	
IloCplex::Quality	
IloCplex::StringParam	
IloCplex::VariableSelect	

Inner Class	
IloCplex::IloCplex::GoalI	
IloCplex::GoalBaseI	
IloCplex::IloCplex::Goal	
IloCplex::IloCplex::CutCallbackI	
IloCplex::IloCplex::HeuristicCallbackI	
IloCplex::IloCplex::BranchCallbackI	
IloCplex::IloCplex::ControlCallbackI	
IloCplex::IloCplex::DisjunctiveCutCallbackI	
IloCplex::IloCplex::FlowMIRCutCallbackI	
IloCplex::IloCplex::FractionalCutCallbackI	
IloCplex::IloCplex::CrossoverCallbackI	
IloCplex::IloCplex::BarrierCallbackI	
IloCplex::IloCplex::ContinuousCallbackI	
IloCplex::IloCplex::CallbackI	
IloCplex::IloCplex::Callback	
IloCplex::IloCplex::Exception	
IloCplex::IloCplex::MultipleObjException	
IloCplex::IloCplex::MultipleConversionException	
IloCplex::IloCplex::UnknownExtractableException	
IloCplex::IloCplex::InvalidCutException	
IloCplex::IloCplex::PresolveCallbackI	
IloCplex::IloCplex::SimplexCallbackI	
IloCplex::IloCplex::NetworkCallbackI	

IloCplex::IloCplex::MIPCallbackI	
IloCplex::IloCplex::ProbingCallbackI	
IloCplex::IloCplex::IncumbentCallbackI	
IloCplex::IloCplex::NodeCallbackI	
IloCplex::IloCplex::SolveCallbackI	
IloCplex::IloCplex::UserCutCallbackI	
IloCplex::IloCplex::LazyConstraintCallbackI	
IloCplex::IloCplex::NodeEvaluatorI	
IloCplex::IloCplex::NodeEvaluator	
IloCplex::IloCplex::SearchLimitI	
IloCplex::IloCplex::SearchLimit	

Inner Type Def	
IloCplex::IloCplex::BasisStatusArray	
IloCplex::IloCplex::BranchDirectionArray	
IloCplex::IloCplex::IISStatusArray	
IloCplex::IloCplex::Status	An enumeration for the class IloAlgorithm .

Description

IloCplex derives from the class [IloAlgorithm](#). Use it to solve LP (linear programming), QP (programs with quadratic terms in the objective function), QCP (quadratically constrained programming), and MIP (mixed integer programming) Mathematical Programming models. More precisely, models to be solved by IloCplex should contain only [IloExtractable](#) objects from the following list:

- ◆ **variables:** objects of type [IloNumVar](#) and its extensions [IloIntVar](#) and [IloSemiContVar](#)
- ◆ **range constraints:** objects of type [IloRange](#)

- ◆ **other relational constraints:** objects of type `IloConstraint` of the form *expr1* relation *expr2*, where the relation is one of `==`, `>=`, `<=`, or `!=`
- ◆ **objective function:** one object of type `IloObjective`
- ◆ **variable type conversions:** objects of type `IloConversion`
- ◆ **special ordered sets:** objects of type `IloSOS1` or `IloSOS2`

The expressions used in the constraints and objective function handled by `IloCplex` are built from variables of those listed types and can be linear or quadratic. In addition, expressions may contain the following constructs:

- ◆ **minimum:**`IloMin`
- ◆ **maximum:**`IloMax`
- ◆ **absolute value:**`IloAbs`
- ◆ **piecewise linear functions:**`IloPiecewiseLinear`

Expressions that evaluate only to 0 (zero) and 1 (one) are referred to as Boolean expressions. Such expressions also support:

- ◆ **negation:**operator `!`
- ◆ **conjunction:**operator `&&` or, equivalently, `IloAnd`
- ◆ **disjunction:**operator `||` or, equivalently, `IloOr`

Moreover, Boolean expressions can be constructed not only from variables, but also from constraints.

`IloCplex` will automatically transform all of these constructs into an equivalent representation amenable to `IloCplex`. Such models can be represented in the following way:

```

Minimize (or Maximize)   c'x + x'Qx
subject to                L ≤ Ax ≤ U
                        a_i'x + x'Q_i x ≤ r_i, for i = 1, ..., q
                        l ≤ x ≤ u.
```

That is, in fact, the standard math programming matrix representation that `IloCplex` uses internally. A is the matrix of linear constraint coefficients, and L and U are the vectors of lower and upper bounds on the vector of variables in the array x . The Q matrix must be positive semi-definite (or negative semi-definite in the maximization case) and represents the quadratic terms of the objective function. The matrices Q_i must be positive semi-definite and represent the quadratic terms of the i -th quadratic constraint. The a_i are vectors containing the corresponding linear terms. For details

about the `Q_i`, see the chapter about quadratically constrained programs (QCP) in the *ILOG CPLEX User's Manual*.

Special ordered sets (SOS) fall outside the conventional representation in terms of `A` and `Q` matrices and are stored separately.

If the model contains integer, Boolean, or semi-continuous variables, or if the model has special ordered sets (SOSs), the model is referred to as a *mixed integer program* (MIP). You can query whether the active model is a MIP with the method `isMIP`.

A model with quadratic terms in the objective is referred to as a *mixed integer quadratic program* (MIQP) if it is also a MIP, and a *quadratic program* (QP) otherwise. You can query whether the active model has a quadratic objective by calling method `isQO`.

A model with quadratic constraints is referred to as a *mixed integer quadratically constrained program* (MIQCP) if it is also a MIP, and as a *quadratically constrained program* (QCP) otherwise. You can query whether the active model is quadratically constrained by calling the method `isQC`. A QCP may or may not have a quadratic objective; that is, a given problem may be both QP and QCP. Likewise, a MIQCP may or may not have a quadratic objective; that is, a given problem may be both MIQP and MIQCP.

If there are no quadratic terms in the objective, no integer constraints, and the problem is not quadratically constrained, and all variables are continuous it is called a *linear program* (LP).

Information related to the matrix representation of the model can be queried through these methods:

- ◆ `getNcols` for querying the number of columns of `A`,
- ◆ `getNrows` for querying the number of rows of `A`; that is, the number of linear constraints,
- ◆ `getNQCs` for querying the number of quadratic constraints,
- ◆ `getNNZs` for querying the number of nonzero elements in `A`, and
- ◆ `getNSOSs` for querying the number of special ordered sets (SOSs).

Additional information about the active model can be obtained through iterators defined on the different types of modeling objects in the extracted or active model.

`IloCplex` effectively treats all models as MIQCP models. That is, it allows the most general case, although the solution algorithms make efficient use of special cases, such as taking advantage of the absence of quadratic terms in the formulation. The method `solve` begins by solving the *root relaxation* of the MIQCP model, where all integrality constraints and SOSs are ignored. If the model has no integrality constraints or SOSs, then the optimization is complete once the root relaxation is solved. Otherwise,

`IloCplex` uses a branch and cut procedure to reintroduce the integrality constraints or SOSs. See the *ILOG CPLEX User's Manual* for more information about branch & cut.

Most users can simply call `solve` to solve their models. However, several parameters are available for users who require more control. Perhaps the most important one is `IloCplex::RootAlg`, which determines the algorithm used to solve the root relaxation. Possible settings, as defined in `IloCplex::Algorithm`, are:

- ◆ **`IloCplex::Auto`** `IloCplex` automatically selects an algorithm. This is the default setting.
- ◆ **`IloCplex::Primal`** Use the primal simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- ◆ **`IloCplex::Dual`** Use the dual simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- ◆ **`IloCplex::Network`** Use network simplex on the embedded network part of the model, followed by dual simplex on the entire model. This option is not available for quadratically constrained problems.
- ◆ **`IloCplex::Barrier`** Use the barrier algorithm.
- ◆ **`IloCplex::Sifting`** Use the sifting algorithm. This option is not available for quadratic problems. If selected nonetheless, `IloCplex` defaults to the `IloCplex::Auto` setting.
- ◆ **`IloCplex::Concurrent`** Use the several algorithms concurrently. This option is not available for quadratic problems. If selected nonetheless, `IloCplex` defaults to the `IloCplex::Auto` setting.

Numerous other parameters allow you to control algorithmic aspects of the optimizer. See the nested enumerations `IloCplex::IntParam`, `IloCplex::DoubleParam`, and `IloCplex#StringParam` for further information. Parameters are set with the method `setParam`.

Even higher levels of control can be achieved through goals (see `IloCplex::Goal`) or through callbacks (see `IloCplex::Callback` and its extensions).

Information about a Solution

The `solve` method returns an `IloBool` value indicating whether (`IloTrue`) or not (`IloFalse`) a solution (not necessarily the optimal one) has been found. Further information about the solution can be queried with the method `getStatus`. The return code of type `IloAlgorithm::Status` indicates whether the solution is feasible, bounded, or optimal, or if the model has been proven to be infeasible or unbounded.

The method `getCplexStatus` provides more detailed information about the status of the optimizer after `solve` returns. For example, it can provide information on why the optimizer terminated prematurely (time limit, iteration limit, or other similar limits).

The methods `isPrimalFeasible` and `isDualFeasible` can determine whether a primal or dual feasible solution has been found and can be queried.

The most important solution information computed by `IloCplex` are usually the solution vector and the objective function value. The method `IloCplex::getValue` queries the solution vector. The method `IloCplex::getObjValue` queries the objective function value. Most optimizers also compute additional solution information, such as dual values, reduced costs, simplex bases, and others. This additional information can also be queried through various methods of `IloCplex`. If you attempt to retrieve solution information that is not available from a particular optimizer, `IloCplex` will throw an exception.

If you are solving an LP and a basis is available, the solution can be further analyzed by performing sensitivity analysis. This information tells you how sensitive the solution is with respect to changes in variable bounds, constraint bounds, or objective coefficients. The information is computed and accessed with the methods `getBoundSA`, `getRangeSA`, `getRHSSA`, and `getObjSA`.

An important consideration when you access solution information is the numeric quality of the solution. Since `IloCplex` performs arithmetic operations using finite precision, solutions are always subject to numeric errors. For most problems, numeric errors are well within reasonable tolerances. However, for numerically difficult models, you are advised to verify the quality of the solution using the method `getQuality`, which offers a variety of quality measures.

More about Solving Problems

By default when the method `solve` is called, `IloCplex` first presolves the model; that is, it transforms the model into a smaller, yet equivalent model. This operation can be controlled with the following parameters:

- ◆ `IloCplex::PreInd`,
- ◆ `IloCplex::PreDual`,
- ◆ `IloCplex::AggInd`, and
- ◆ `IloCplex::AggFill`.

For the rare occasion when a user wants to monitor progress during presolve, the callback class `IloCplex::PresolveCallbackI` is provided.

After the presolve is completed, `IloCplex` solves the first node relaxation and (in cases of a true MIP) enters the branch & cut process. `IloCplex` provides callback classes that allow the user to monitor solution progress at each level. Callbacks derived from `IloCplex::ContinuousCallbackI` or one of its derived classes are called regularly during the solution of a node relaxation (including the root), and callbacks derived from `IloCplex::MIPCallbackI` or one of its derived callbacks are

called regularly during branch & cut search. All callbacks provide the option to abort the current optimization.

Branch Priorities and Directions

When a branch occurs at a node in the branch & cut tree, usually there is a set of fractional-valued variables available to pick from for branching. `IloCplex` has several built-in rules for making such a choice, and they can be controlled by the parameter `IloCplex::VarSel`. Also, the method `setPriority` allows the user to specify a priority order. An instance of `IloCplex` branches on variables with an assigned priority before variables without a priority. It also branches on variables with higher priority before variables with lower priority, when the variables have fractional values.

Frequently, when two new nodes have been created (controlled by the parameter `IloCplex::BtTol`), one of the two nodes is processed next. This activity is known as diving. The branch direction determines which of the branches, the up or the down branch, is used when diving. By default, `IloCplex` automatically selects the branch direction. The user can control the branch direction by the method `setDirection`.

As mentioned before, the greatest flexibility for controlling the branching during branch & cut search is provided through goals (see `IloCplex::Goal`) or through the callbacks (see `IloCplex::BranchCallbackI`). With these concepts, you can control the branching decision based on runtime information during the search, instead of statically through branch priorities and directions, but the default strategies work well on many problems.

Cuts

An instance of `IloCplex` can also generate certain cuts in order to strengthen the relaxation, that is, in order to make the relaxation a better approximation of the original MIP. Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

When solving a MIP, `IloCplex` tries to generate violated cuts to add to the problem after solving a node. After `IloCplex` adds cuts, the subproblem is re-optimized. `IloCplex` then repeats the process of adding cuts at a node and reoptimizing until it finds no further effective cuts.

An instance of `IloCplex` generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular node. After a cut has been added to the problem, it will remain in the problem to the end of the optimization. However, cuts are added only internally; that is, they will not be part of the model extracted to the `IloCplex` object after the optimization. Cuts are most frequently seen at the root node, but they may be added by an instance of `IloCplex` at other nodes as conditions warrant.

IloCplex looks for various kinds of cuts that can be controlled by the following parameters:

- ◆ IloCplex::Cliques,
- ◆ IloCplex::Covers,
- ◆ IloCplex::FlowCovers,
- ◆ IloCplex::GUBCovers,
- ◆ IloCplex::FracCuts,
- ◆ IloCplex::MIRCuts,
- ◆ IloCplex::FlowPaths,
- ◆ IloCplex::ImplBd, and
- ◆ IloCplex::DisjCuts.

During the search, you can query information about those cuts with a callback (see IloCplex::MIPCallbackI and its subclasses). For types of cuts that may take a long time to generate, callbacks are provided to monitor the progress and potentially abort the cut generation progress. In particular, those callback classes are IloCplex::FractionalCutCallbackI and IloCplex::DisjunctiveCutCallbackI. The callback class IloCplex::CutCallbackI allows you to add your own problem-specific cuts during search. This callback also allows you to generate and add local cuts, that is, cuts that are valid only within the subtree where they have been added.

Instead of using callbacks, you can use goals to add your own cuts during the optimization.

Heuristics

After a node has been processed, that is, the LP has been solved and no more cuts were generated, IloCplex may try to construct an integer feasible solution from the LP solution at that node. The parameter IloCplex::HeurFreq and other parameters provide some control over this activity. In addition, goals or the callback class IloCplex::HeuristicCallbackI make it possible to call user-written heuristics to find an integer feasible solution.

Again, instead of using callbacks, you can use goals to add inject your own heuristically constructed solution into the running optimization.

Node Selection

When IloCplex is not diving but picking an unexplored node from the tree, several options are available that can be controlled with the parameter IloCplex::NodeSel. Again, IloCplex offers a callback class, IloCplex::NodeCallbackI, to give the user full control over this selection. With

goals, objects of type `IloCplex::NodeEvaluatorI` can be used to define your own selection strategy.

See also `IloAlgorithm` in the *ILOG Concert Reference Manual*.

See also *Goals* among the Concepts in this manual. See also `goals` in the *ILOG CPLEX User's Manual*.

See Also

`IloCplex::Algorithm`, `IloCplex::BasisStatus`,
`IloCplex::BasisStatusArray`, `IloCplex::BranchDirection`,
`IloCplex::BranchDirectionArray`, `IloCplex::CallbackI`,
`IloCplex::DeleteMode`, `IloCplex::DualPricing`,
`IloCplex::Exception`, `IloCplex::IISStatus`,
`IloCplex::IISStatusArray`, `IloCplex::IntParam`,
`IloCplex::MIPEmphasisType`, `IloCplex::NodeSelect`,
`IloCplex::NumParam`, `IloCplex::PrimalPricing`,
`IloCplex::Quality`, `IloCplex::CplexStatus`,
`IloCplex::StringParam`, `IloCplex::VariableSelect`,
`IloCplex::GoalI`

Constructors

```
public IloCplex(IloEnv env)
```

This constructor creates an ILOG CPLEX algorithm. The new `IloCplex` object has no `IloModel` loaded (or extracted) to it.

```
public IloCplex(const IloModel model)
```

This constructor creates an ILOG CPLEX algorithm and extracts `model` for that algorithm.

When you create an algorithm (an instance of `IloCplex`, for example) and extract a model for it, you can write either this line:

```
IloCplex cplex(model);
```

or these two lines:

```
IloCplex cplex(env);  
cplex.extract(model);
```

Methods

```
public IloConstraint addCut(IloConstraint con)
```

This method adds `con` as a cut to the invoking `IloCplex` object. The cut is not extracted as the regular constraints in a model, but is only copied when invoking the method `addCut`. Thus, `con` may be deleted or modified after `addCut` has been called and the change will not be notified to the invoking `IloCplex` object.

When columns are deleted from the extracted model, all cuts are deleted as well and need to be reextracted if they should be considered. Cuts are not part of the root problem, but are considered on an as-needed basis. A solution computed by `IloCplex` is guaranteed to satisfy all cuts added with this method.

```
public const IloConstraintArray addCuts(const IloConstraintArray con)
```

This method adds the constraints in `con` as cuts to the invoking `IloCplex` object. Everything said for `addCut` applies equally to each of the cuts given in array `con`.

```
public IloConstraint addLazyConstraint(IloConstraint con)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds `con` as a lazy constraint to the invoking `IloCplex` object. The constraint `con` is copied into the lazy constraint pool; the `con` itself is not part of the pool, so changes to `con` after it has been copied into the lazy constraint pool will not affect the lazy constraint pool.

Lazy constraints added with `addLazyConstraint` are typically constraints of the model that are not expected to be violated when left out. The idea behind this is that the LPs that are solved when solving the MIP can be kept smaller when these constraints are not included. `IloCplex` will, however, include a lazy constraint in the LP as soon as it becomes violated. In other words, the solution computed by `IloCplex` makes sure that all the lazy constraints that have been added are satisfied.

By contrast, if the constraint does not change the feasible region of the extracted model but only strengthens the formulation, it is referred to as a user cut. While user cuts can be added to `IloCplex` with `addLazyConstraint`, it is generally preferable to do so with `addUserCuts`. It is an error, however, to add lazy constraints by means of the method `addUserCuts`.

When columns are deleted from the extracted model, all lazy constraints are deleted as well and need to be recopied into the lazy constraint pool. Use of this method in place of `addCuts` allows for further presolve reductions

This method is equivalent to `addCut`.

```
public const IloConstraintArray addLazyConstraints(const
IloConstraintArray con)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds a set of lazy constraints to the invoking `IloCplex` object. Everything said for `addLazyConstraint` applies to each of the lazy constraints given in array `con`.

This method is equivalent to `addCuts`.

```
public IloConstraint addUserCut(IloConstraint con)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds `con` as a user cut to the invoking `IloCplex` object. The constraint `con` is copied into the user cut pool; the `con` itself is not part of the pool, so changes to `con` after it has been copied into the user cut pool will not affect the user cut pool.

Cuts added with `addUserCut` must be real cuts, in that the solution of a MIP does not depend on whether the cuts are added or not. Instead, they are there only to strengthen the formulation.

When columns are deleted from the extracted model, all user cuts are deleted as well and need to be recopied into the user cut pool.

Note: It is an error to use `addUserCut` for lazy constraints, that is, constraints whose absence may potentially change the solution of the problem. Use `addLazyConstraints` or, equivalently, `addCut` when you add such a constraint.

```
public const IloConstraintArray addUserCuts(const IloConstraintArray con)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds a set of user cuts to the invoking `IloCplex` object. Everything said for `addUserCut` applies to each of the user cuts given in array `con`.

```
public static IloCplex::Goal Apply(IloCplex cplex,
                                   IloCplex::Goal goal,
                                   IloCplex::NodeEvaluator eval)
```

This method is used to create and return a goal that applies the node selection strategy defined by `eval` to the search strategy defined by `goal`. The resulting goal will use the node strategy defined by `eval` for the subtree generated by `goal`.

```
public void basicPresolve(const IloNumVarArray vars,
                          IloNumArray redlb=0,
                          IloNumArray redub=0,
                          const IloRangeArray rngs=0,
                          IloBoolArray redundant=0) const
```

This method can be used to compute tighter bounds for the model variables and to detect redundant constraints in the model extracted to the invoking `IloCplex` object. For every variable specified in parameter `vars`, it will return possibly tightened bounds in the corresponding elements of arrays `redlb` and `redub`. Similarly, for every constraint specified in parameter `rngs`, this method will return a boolean indicating whether or not it is redundant in the model in the corresponding element of array `redundant`.

```
public void clearCuts()
```

This method deletes all cuts that have previously been added to the invoking `IloCplex` object with the methods `addCut` and `addCuts`.

```
public void clearLazyConstraints()
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method deletes all lazy constraints added to the invoking `IloCplex` object with the methods [addLazyConstraint](#) and [addLazyConstraints](#).

This method is equivalent to [clearCuts](#).

```
public void clearModel()
```

This method can be used to unextract the model that is currently extracted to the invoking `IloCplex` object.

```
public void clearUserCuts()
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method deletes all user cuts that have previously been added to the invoking `IloCplex` object with the methods [addUserCut](#) and [addUserCuts](#).

```
public void delDirection(IloNumVar var)
```

This method removes any existing branching direction assignment from variable `var`.

```
public void delDirections(const IloNumVarArray var)
```

This method removes any existing branching direction assignments from all variables in the array `var`.

```
public void delPriorities(const IloNumVarArray var)
```

This method removes any existing priority order assignments from all variables in the array `var`.

```
public void delPriority(IloNumVar var)
```

This method removes any existing priority order assignment from variable `var`.

```
public IloNum dualFarkas(IloConstraintArray rng,  
                        IloNumArray y)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method returns a Farkas proof of infeasibility for the active LP model after it has been proven to be infeasible by one of the simplex optimizers. For every constraint i of the active LP this method computes a value $y[i]$ such that $y'A \geq y'b$, where A denotes the constraint matrix. For more detailed information about the Farkas proof of infeasibility, see the C routine `CPXdualfarkas`, documented in the reference manual of the Callable Library.

Parameters : ***rng**

An array of length `getNrows` where constraints corresponding to the values in y are returned.

***y**

An array of length `getNrows`.

Returns : The value of $y'b - y'A z$ for the vector z defined such that $z[j] = ub[j]$ if $y'A[j] > 0$ and $z[j] = lb[j]$ if $y'A[j] < 0$ for all variables j .

```
public void exportModel(const char * filename) const
```

This method writes the active model (that is, the model that has been extracted by the invoking algorithm) to the file `filename`. The file format is determined by the extension of the file name. The following extensions are recognized on most platforms:

- ◆ `.sav`
- ◆ `.mps`
- ◆ `.lp`
- ◆ `.sav.gz` (if `gzip` is properly installed)
- ◆ `.mps.gz` (if `gzip` is properly installed)
- ◆ `.lp.gz` (if `gzip` is properly installed)

Microsoft Windows does not support gzipped files for this API.

If no name has been assigned to a variable or range (that is, the method `getName` returns null for that variable or range), `IloCplex` uses a default name when writing the model to the file (or to the optimization log). Default names are of the form `IloXj` for variables and `IloCi`, where i and j are internal indices of `IloCplex`.

See the reference manual *ILOG CPLEX File Formats* for more detail and the *ILOG CPLEX User's Manual* for additional information about file formats.

```
public IloBool feasOpt(const IloRangeArray rngs,
                        IloNumArray rnglb,
                        IloNumArray rngub,
                        IloBool optimize=IloFalse)
```


Attempts to find a feasible relaxation of an active model by relaxing the bounds of the constraints specified in `rngs` as specified in `rnglb` and `rngub`. This is a special version of the method `feasOpt(IloRangeArray, IloNumArray, IloNumArray, IloNumVarArray, IloNumArray, IloNumArray, IloBool)`

The method returns `IloTrue` if a feasible relaxation is found along with a solution for it.

```
public IloBool feasOpt(const IloNumVarArray vars,
                      IloNumArray varlb,
                      IloNumArray varub,
                      IloBool optimize=IloFalse)
```

Attempts to find a feasible relaxation of an active model by relaxing the bounds of the variables specified in `vars` as specified in `varlb` and `varub`. This is a special version of the method `feasOpt(IloRangeArray, IloNumArray, IloNumArray, IloNumVarArray, IloNumArray, IloNumArray, IloBool)`

The method returns `IloTrue` if a feasible relaxation is found, along with a solution for it.

```
public IloBool feasOpt(const IloRangeArray rngs,
                      IloNumArray rnglb,
                      IloNumArray rngub,
                      const IloNumVarArray vars,
                      IloNumArray varlb,
                      IloNumArray varub,
                      IloBool optimize=IloFalse)
```

The method `feasOpt` computes a minimal relaxation of the bounds of variables and constraints that make the active model feasible. For each bound, the user may specify a preferred value the reciprocal of which is the penalty associated with relaxing the bound. The user may also specify an analogous preferred value for every finite upper or lower bound of a constraint. Thus, larger values indicate greater willingness to relax a bound or constraint. A negative or 0 (zero) value indicates that the corresponding bound must not be relaxed. Typically, values greater than or equal to 1 (one) should be used. The preferred value for relaxing the lower and upper bound of constraint `rngs[i]` must be provided in `rnglb[i]` and `rngub[i]`, respectively. Similarly, the preferred value for relaxing the lower and upper bound of constraint `vars[i]` must be provided in `varlb[i]` and `varub[i]`, respectively.

If enough variables or constraints were allowed to be relaxed, the function will return `IloTrue` and `IloFalse` otherwise. In case of success, the routine returns with suggested relaxed values that would make the active model feasible. These bounds are chosen in such a way that the relaxation is minimal in the sense that the sum of (relaxation amount)/(priority) over all variables and constraints is minimized.

The bounds that make the model feasible are returned in the same arrays as the priorities are passed into this method. That is, after successful termination of the method `feasOpt`, `rnglb[i]` and `rngub[i]` contain the lower and upper bound to use for the constraint `rngs[i]` in order to make the model feasible. Similarly, `varlb[i]` and `varub[i]` contain the lower and upper bound to use for variable `vars[i]` in order to make the model feasible. If bounds were not allowed to be relaxed or did not need to be relaxed in order to make the model feasible, the original bounds are returned.

The active model is not changed by this method.

If `feasOpt` successfully finds a relaxation that made the model feasible, it also installs a solution vector that would be feasible, had the model been relaxed as specified with the returned relaxed bounds. This solution can be queried with the conventional query methods.

When the `optind` parameter is `IloTrue`, this method also tries to find the optimal solution of the minimally relaxed model. That is, it finds the solution that is optimal with respect to the original objective function, but requires only minimal relaxation. With the default setting of the parameter `optind` the original objective function is not evaluated.

```
public void freePresolve()
```

This method frees the presolved problem. Under the default setting of parameter `Reduce`, the presolved problem is freed when an optimal solution is found; however, it is not freed if `Reduce` has been set to 1 (primal reductions) or to 2 (dual reductions). In these instances, the function `freePresolve` can be used when necessary to free it manually.

```
public IloCplex::Algorithm getAlgorithm() const
```

This method returns the algorithm type that was used to solve the most recent model in cases where it was not a MIP.

```
public void getAX(IloNumArray val,
                  const IloRangeArray con) const
```

For the constraints in `con`, this method places the values of the expressions, or, equivalently, the activity levels of the constraints for the current solution of the invoking `IloCplex` object into the array `val`. Array `val` is resized to the same size as array `con`, and `val[i]` will contain the slack value for constraint `con[i]`. All ranges in `con` must be part of the extracted model.

```
public IloNum getAX(const IloRange range) const
```

This method returns the value of the expression of constraint `range`, or, equivalently, its activity level, for the current solution of the invoking `IloCplex` object. The `range` must be part of the extracted model.

```
public IloCplex::BasisStatus getBasisStatus(const IloConstraint con) const
```

This method returns the basis status of the implicit slack or artificial variable created for the constraint `con`.

```
public IloCplex::BasisStatus getBasisStatus(const IloIntVar var) const
```

This method returns the basis status for the variable `var`.

```
public IloCplex::BasisStatus getBasisStatus(const IloNumVar var) const
```

This method returns the basis status for the variable `var`.

```
public void getBasisStatuses(IloCplex::BasisStatusArray cstat,  
                           const IloNumVarArray var,  
                           IloCplex::BasisStatusArray rstat,  
                           const IloConstraintArray con) const
```

This method puts the basis status of each variable in `var` into the corresponding element of the array `cstat`, and it puts the status of each row in `con` (an array of ranges or constraints) into the corresponding element of the array `rstat`. Arrays `rstat` and `cstat` are resized accordingly.

```
public void getBasisStatuses(IloCplex::BasisStatusArray stat,  
                           const IloConstraintArray con) const
```

This method puts the basis status of each constraint in `con` into the corresponding element of the array `stat`. Array `stat` is resized accordingly.

```
public void getBasisStatuses(IloCplex::BasisStatusArray stat,  
                           const IloNumVarArray var) const
```

This method puts the basis status of each variable in `var` into the corresponding element of the array `stat`. Array `stat` is resized accordingly.

```
public IloNum getBestObjValue() const
```

This method returns a bound on the optimal solution value of the problem. When a model has been solved to optimality, this value matches the optimal solution value. If a MIP optimization is terminated before optimality has been proven, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

```
public void getBoundSA(IloNumArray lblower,  
                     IloNumArray lbupper,  
                     IloNumArray ublower,  
                     IloNumArray ubupper,  
                     const IloNumVarArray vars) const
```

For the given set of variables `vars`, bound sensitivity information is computed. When the method returns, the element `lblower[j]` and `lbupper[j]` will contain the lowest and highest value the lower bound of variable `vars[j]` can assume without affecting the optimality of the solution. Likewise, `ublower[j]` and `ubupper[j]` will contain the lowest and highest value the upper bound of variable `vars[j]` can assume without affecting the optimality of the solution. The arrays `lblower`,

`lbupper`, `ublower`, and `ubupper` will be resized to the size of array `vars`. The value 0 (zero) can be passed for any of the return arrays if the information is not desired.

```
public IloCplex::CplexStatus getCplexStatus() const
```

This method returns the ILOG CPLEX status of the invoking algorithm. For possible ILOG CPLEX values, see the enumeration type `IloCplex::CplexStatus`.

```
public IloCplex::CplexStatus getCplexSubStatus() const
```

This method is used to access the solution status of the last node problem that was solved in the event of an error termination in the previous invocation of `solve`. The method `IloCplex::getCplexSubStatus` returns 0 in the event of a normal termination. If the invoking `IloCplex` object is continuous, this is equivalent to the status returned by the method `getCplexStatus`.

```
public IloNum getCutoff() const
```

This method returns the MIP cutoff value being used during the MIP optimization. In a minimization problem, all nodes are pruned that have an optimal solution value of the continuous relaxation that is larger than the current cutoff value. The cutoff is updated with the incumbent. If the invoking `IloCplex` object is an LP or QP, `+IloInfinity` or `-IloInfinity` is returned, depending on the optimization sense.

```
public IloBool getDefault(IloCplex::BoolParam parameter) const
```

This method return the default setting for the parameter `parameter`.

```
public IloCplex::DeleteMode getDeleteMode() const
```

This method returns the current delete mode of the invoking `IloCplex` object.

```
public IloCplex::BranchDirection getDirection(IloNumVar var) const
```

This method returns the branch direction previously assigned to variable `var` with method `setDirection` or `setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public void getDirections(IloCplex::BranchDirectionArray dir,
                        const IloNumVarArray var) const
```

This method returns the branch directions previously assigned to variables listed in `var` with the method `setDirection` or `setDirections`. When the function returns, `dir[i]` will contain the branch direction assigned for variables `var[i]`. If no branch direction has been assigned to `var[i]`, `dir[i]` will be set to `IloCplex::BranchGlobal`.

```
public IloExtractable getDiverging() const
```

This method returns the diverging variable or constraint, in a case where the primal Simplex algorithm has determined the problem to be infeasible. The returned extractable is either an `IloNumVar` or an `IloConstraint` object extracted to the invoking

IloCplex optimizer; it is of type `IloNumVar` if the diverging column corresponds to a variable, or of type `IloConstraint` if the diverging column corresponds to the slack variable of a constraint.

```
public IloNum getDual(const IloRange range) const
```

This method returns the dual value associated with constraint `range` in the current solution of the invoking algorithm.

```
public void getDuals(IloNumArray val,
                    const IloRangeArray con) const
```

This method puts the dual values associated with the ranges in the array `con` into the array `val`. Array `val` is resized to the same size as array `con`, and `val[i]` will contain the dual value for constraint `con[i]`.

```
public IloNum getFormulationBigValue()
```

The method `getFormulationBigValue` returns the current setting of the large value used to set a bound on otherwise unbounded expressions appearing in logical constraints. Such a value is sometimes referred to as Big M.

By default, this value is 1e6. You can change that default setting by means of the method `setFormulationBigValue`.

In order to transform a logical constraint automatically into an equivalent MIP representation, it is sometimes necessary to set a finite upper or lower bound on an otherwise unbounded term. When such a bound does not already exist, CPLEX uses the value returned by this method for that purpose. For example, consider the following disjunction (that is, or-statement):

$$(x == 1) \vee (x \leq 2)$$

It is necessary to have an upper bound on x . If the upper bound on x is `IloInfinity`, CPLEX will use the value returned by `getFormulationBigValue` in the transformed representation of logical formulations.

```
public IloNum getFormulationEpsValue()
```

This method returns the epsilon value that is used to model strict relations. In order to model a strict inequality, such as $x < b$ in a MIP, IloCplex uses $x \leq b - \text{eps}$ as a surrogate, where `eps` is the value returned by this method.

```
public void getIIS(IloCplex::IISStatusArray cstat,
                  IloNumVarArray var,
                  IloCplex::IISStatusArray rstat,
                  IloConstraintArray rng) const
```

If an LP model has been proven to be infeasible, this method can compute an irreducibly inconsistent set (IIS). The variables and range constraints in the IIS are returned in the arrays `var` and `rng`, respectively.

Note:CPLEX resizes these arrays for you.

For each variable `var[i]`, `cstat[i]` gives the IIS status for that variable in the IIS, where the meaning of `cstat[i]` is:

`IloCplex::Lower` the lower bound of the variable participates in the IIS

`IloCplex::Upper` the upper bound of the variable participates in the IIS

`IloCplex::Both` both variable bounds participate in the IIS

Similarly, for each range constraint `rng[i]`, `rstat[i]` gives the IIS status for that constraint in the IIS, where the meaning of `rstat[i]` is:

`IloCplex::Lower` the lower bound of the constraint participates in the IIS

`IloCplex::Upper` the upper bound of the constraint participates in the IIS

`IloCplex::Both` range is equality constraint and participates in the IIS.

The statuses `IloCplex::Lower` and `IloCplex::Upper` are needed only for constraints that are truly ranged, that is, where both bounds are finite but different.

By definition, if any of the bounds or constraints participating in the IIS is removed, the IIS becomes feasible. However, this does not mean that the model also becomes feasible by doing so. Instead, there may be yet another source of infeasibility in the model.

If the LP has been proven infeasible but no basis is available, `IloCplex` automatically invokes the primal simplex optimizer to compute the basis needed for the IIS-finding algorithm.

```
public IloInt getIncumbentNode() const
```

This method returns the node number where the current incumbent was found. If the invoking `IloCplex` object is an LP or a QP, 0 is returned.

```
public IloInt getMax(IloCplex::IntParam parameter) const
```

These method return the maximum allowed value for the parameter `parameter`.

```
public IloInt getMin(IloCplex::IntParam parameter) const
```

These method return the minimum allowed value for the parameter `parameter`.

```
public IloInt getNbarrierIterations() const
```

This method returns the number of barrier iterations from the last solve.

```
public IloInt getNbinVars() const
```

This method returns the number of binary variables in the matrix representation of the active model in the invoking **IloCplex** object.

```
public IloInt getNcliques() const
```

This method returns the number of clique cuts that have been added to the problem. If the invoking **IloCplex** object is not a MIP, 0 is returned.

```
public IloInt getNcols() const
```

This method returns the number of columns extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of object of type **IloNumVar** and its subclasses in the model that is extracted. This is because automatic transformation of non-linear constraints and expressions may introduce new variables.

```
public IloInt getNcovers() const
```

This method returns the number of cover cuts that have been added to the problem. If the invoking **IloCplex** object is not a MIP, 0 is returned.

```
public IloInt getNcrossDExch() const
```

This method returns the number of dual exchange operations in the crossover of the last call to method **solve** or **solveFixed**, if barrier with crossover has been used for solving an LP or QP.

```
public IloInt getNcrossDPush() const
```

This method returns the number of dual push operations in the crossover of the last call to **solve** or **solveFixed**, if barrier with crossover was used for solving an LP or QP.

```
public IloInt getNcrossPExch() const
```

This method returns the number of primal exchange operations in the crossover of the last call of method **solve** or **solveFixed**, if barrier with crossover was used for solving an LP or QP.

```
public IloInt getNcrossPPush() const
```

This method returns the number of primal push operations in the crossover of the last call of method **solve** or **solveFixed**, if barrier with crossover was used for solving an LP or QP.

```
public IloInt getNdualSuperbasics() const
```

This method returns the number of dual superbasic variables in the current solution of the invoking **IloCplex** object.

```
public IloInt getNintVars() const
```

This method returns the number of integer variables in the `matrix` representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNiterations() const
```

This method returns the number of iterations that occurred during the last call to the method `solve` in the invoking algorithm.

```
public IloInt getNnodes() const
```

This method returns the number of branch-and-cut nodes that were processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```
public IloInt getNnodesLeft() const
```

This method returns the number of branch-and-cut nodes that remain to be processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```
public IloInt getNNZs() const
```

This method returns the number of non-zeros extracted to the constraint matrix `A` of the invoking algorithm.

```
public IloInt getNphaseOneIterations() const
```

If a simplex method was used for solving continuous model, this method returns the number of iterations in phase 1 of the last call to `solve` or `solveFixed`.

```
public IloInt getNprimalSuperbasics() const
```

This method returns the number of primal superbasic variables in the current solution of the invoking `IloCplex` object.

```
public IloInt getNQCs() const
```

This method returns the number of quadratic constraints extracted from the active model for the invoking algorithm. This number may be different from the total number of constraints in the active model because linear constraints are not accounted for in this function.

See Also

[`getNrows`](#)

```
public IloInt getNrows() const
```

This method returns the number of rows extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of `IloRanges` and `IloConstraints` in the model that is extracted. This is because quadratic constraints are not accounted for by method `getNrows` and automatic transformation of non-linear constraints and expressions may introduce new constraints.

See Also

[`getNQCs`](#)

```
public IloInt getNsemiContVars() const
```


This method returns the number of semi-continuous variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNsemiIntVars() const
```

This method returns the number of semi-integer variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNsiftingIterations() const
```

This method returns the number of sifting iterations performed for solving the last LP with algorithm type `IloCplex::Sifting`, or, equivalently, the number of work LPs that have been solved for it.

```
public IloInt getNsiftingPhaseOneIterations() const
```

This method returns the number of sifting iterations performed for solving the last LP with algorithm type `IloCplex::Sifting` in order to achieve primal feasibility.

```
public IloInt getNSOSs() const
```

This method returns the number of SOSs extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of `IloNumVar`, etc. in the model that is extracted. This is because piecewise linear functions are extracted to a set of SOS.

```
public void getObjSA(IloNumArray lower,  
                    IloNumArray upper,  
                    const IloNumVarArray vars) const
```

This method performs objective sensitivity analysis for the variables specified in array `vars`. When this method returns `lower[i]` and `upper[i]` will contain the lowest and highest value the objective function coefficient for variable `vars[i]` can assume without affecting the optimality of the solution. The arrays `lower` and `upper` will be resized to the size of array `vars`. If any of the information is not requested, 0 (zero) can be passed for the corresponding array parameter.

```
public IloBool getParam(IloCplex::BoolParam parameter) const
```

This method returns the current setting of `parameter` in the invoking algorithm.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

```
public void getPriorities(IloNumArray pri,  
                        const IloNumVarArray var) const
```

This method returns query branch priorities previously assigned to variables listed in `var` with the method `setPriority` or `setPriorities`. When the function returns, `pri[i]` will contain the priority value assigned for variables `var[i]`. If no priority has been assigned to `var[i]`, `pri[i]` will contain 0 (zero).

```
public IloNum getPriority(IloNumVar var) const
```

This method returns the priority previously assigned to the variable `var` with the method `setPriority` or `setPriorities`. It returns 0 (zero) if no priority has been assigned.

```
public IloCplex::PWLFormulation getPWLFormulation()
```

This method queries the way piecewise linear expressions are represented when CPLEX automatically transforms them during extraction.

```
public IloNum getQuality(IloCplex::Quality q,
                        IloNumVar * var=0,
                        IloConstraint * rng=0) const
```

These method return the requested quality measure.

Some quality measures are related to a variable or a constraint. For example `IloCplex::MaxPrimalInfeas` is related to the variable or range constraint where the maximum infeasibility (bound violation) occurs. If this information is also requested, pointers to `IloNumVar` or `IloConstraint` objects may be passed for the optional parameters where the relevant variable or range will be written.

```
public void getRangeSA(IloNumArray lblower,
                      IloNumArray lbupper,
                      IloNumArray ublower,
                      IloNumArray ubupper,
                      const IloRangeArray con) const
```

This method performs sensitivity analysis for the upper and lower bounds of the ranged constraints passed in the array `con`. When the method returns, `lblower[i]` and `lbupper[i]` will contain, respectively, the lowest and the highest value that the lower bound of constraint `con[i]` can assume without affecting the optimality of the solution. Similarly, `ublower[i]` and `ubupper[i]` will contain, respectively, the lowest and the highest value that the upper bound of the constraint `con[i]` can assume without affecting the optimality of the solution. The arrays `lblower`, `lbupper`, `ublower`, and `ubupper` will be resized to the size of array `con`. If any of the information is not requested, 0 can be passed for the corresponding array parameter.

```
public void getRay(IloNumArray vals,
                  IloNumVarArray vars) const
```

This method returns an unbounded direction (also known as a ray) corresponding to the present basis for an LP that has been determined to be an unbounded problem. CPLEX puts the variables of the extracted model into the array `vars` and it puts the corresponding values of the unbounded direction into the array `vals`.

Note:CPLEX resizes these arrays for you.

```
public IloNum getReducedCost(const IloIntVar var) const
```

This method returns the reduced cost associated with `var` in the invoking algorithm.

```
public IloNum getReducedCost(const IloNumVar var) const
```

This method returns the reduced cost associated with `var` in the invoking algorithm.

```
public void getReducedCosts(IloNumArray val,
                           const IloIntVarArray var) const
```

This method puts the reduced costs associated with the numeric variables of the array `var` into the array `val`. The array `val` is automatically resized to the same length as array `var`, and `val[i]` will contain the reduced cost for variable `var[i]`.

```
public void getReducedCosts(IloNumArray val,
                           const IloNumVarArray var) const
```

This method puts the reduced costs associated with the variables in the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the reduced cost for variable `var[i]`.

```
public void getRHSSA(IloNumArray lower,
                    IloNumArray upper,
                    const IloRangeArray cons) const
```

This method performs righthand side sensitivity analysis for the constraints specified in array `cons`. The constraints must be of the form `cons[i]:expr[i] rel rhs[i]`. When this method returns `lower[i]` and `upper[i]` will contain the lowest and highest value `rhs[i]` can assume without affecting the optimality of the solution. The arrays `lower` and `upper` will be resized to the size of array `cons`. If any of the information is not requested, 0 (zero) can be passed for the corresponding array parameter.

```
public IloNum getSlack(const IloRange range) const
```

This method returns the slack value associated with constraint `range` in the current solution of the invoking algorithm. For a range with finite lower and upper bounds, the slack value consists of the difference between the expression of the range and its lower bound.

```
public void getSlacks(IloNumArray val,
                     const IloRangeArray con) const
```

This method puts the slack values associated with the constraints indicated by the array `con` into the array `val`. For a ranged constraint with finite lower and upper bounds, the slack value consists of the difference between the expression in the range and its lower bound. Array `val` is resized to the same size as array `con`, and `val[i]` will contain the slack value for constraint `con[i]`.

```
public IloAlgorithm::Status getStatus() const
```

This method returns the status of the invoking algorithm. For its ILOG CPLEX status, see the method `IloCplex::getCplexStatus`.

```
public IloCplex::Algorithm getSubAlgorithm() const
```

This method returns the algorithm type that was used to solve most recent node of a MIP in the case of termination because of an error during mixed integer optimization.

```
public void getValues(const IloIntVarArray var,
                    IloNumArray val) const
```

This method puts the solution values of the integer variables indicated by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(IloNumArray val,
                    const IloIntVarArray var) const
```

This method puts the solution values of the integer variables indicated by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(const IloNumVarArray var,
                    IloNumArray val) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public void getValues(IloNumArray val,
                    const IloNumVarArray var) const
```

This method puts the solution values of the numeric variables indicated by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public const char * getVersion() const
```

This method returns a string indicating the version of `IloCplex`.

```
public void importModel(IloModel & m,
                    const char * filename) const
```

This method reads a model from the file indicated by `filename` into `m`, a model. The invoking `IloCplex` object is not affected when you call this method unless `m` is its active model.

```
public void importModel(IloModel & m,
                    const char * filename,
```

```

IloObjective & obj,
IloNumVarArray vars,
IloRangeArray rngs,
IloRangeArray lazy=0,
IloRangeArray cuts=0) const

```

This method is a simplification of the `importModel` method that does not provide arrays to return SOSs. This method is easier to use in the case where you are dealing with continuous models because in such a case you already know that no SOS will be present.

```

public void importModel(IloModel & model,
                        const char * filename,
                        IloObjective & obj,
                        IloNumVarArray vars,
                        IloRangeArray rngs,
                        IloSOS1Array sos1,
                        IloSOS2Array sos2,
                        IloRangeArray lazy=0,
                        IloRangeArray cuts=0) const

```

This method reads a model from the file indicated by `filename` into `model`. If `model` refers to the current active model, then when CPLEX is reading a file, the current active model is first cleaned out and then new modeling objects, as required by the input file, are added to it. Otherwise, if `model` is **not** the current active model, then the invoking `IloCplex` object is **not** affected when you call this method.

As this method reads a model from a file, it places the objective it has read in `obj`, the variables it has read in the array `vars`, the ranges it has read in the array `rngs`; and the Special Ordered Sets it has read in the arrays `sos1` and `sos2`.

Note: *CPLEX resizes these arrays for you to accomodate the returned objects.*

Note: This note is for advanced users only. The two arrays `lazy` and `cuts` are filled with the `lazy` constraints and user cuts that may be included in the model in file `filename`.

The format of the file is indicated by the extension of the file name. The following extensions are recognized on most platforms:

- ◆ `.sav`
- ◆ `.mps`

- ◆ .lp
- ◆ .sav.gz (if gzip is properly installed)
- ◆ .mps.gz (if gzip is properly installed)
- ◆ .lp.gz (if gzip is properly installed)

Microsoft Windows does not support gzipped files for this API.

```
public IloBool isDualFeasible() const
```

This method returns **IloTrue** if a dual feasible solution is recorded in the invoking **IloCplex** object and can be queried.

```
public IloBool isMIP() const
```

This method returns **IloTrue** if the invoking algorithm has extracted a model that is a MIP (mixed-integer programming problem) and **IloFalse** otherwise. Member functions for accessing duals and reduced cost basis work only if the model is not a MIP.

```
public IloBool isPrimalFeasible() const
```

This method returns **IloTrue** if a primal feasible solution is recorded in the invoking **IloCplex** object and can be queried.

```
public IloBool isQC() const
```

This method returns **IloTrue** if the invoking algorithm has extracted a model that is quadratically constrained. Otherwise, it returns **IloFalse**. For an explanation of *quadratically constrained* see the topic QCP in the *ILOG CPLEX User's Manual*.

```
public IloBool isQO() const
```

This method returns **IloTrue** if the invoking algorithm has extracted a model that has quadratic objective function terms. Otherwise, it returns **IloFalse**.

```
public static IloCplex::Goal LimitSearch(IloCplex cplex,  
                                         IloCplex::Goal goal,  
                                         IloCplex::SearchLimit limit)
```

This method creates and returns a goal that puts the search specified by `goal` under the limit defined by `limit`. Only the subtree controlled by `goal` will be subjected to limit `limit`.

```
public void presolve(IloCplex::Algorithm alg)
```

This method performs Presolve on the model. The enumeration `alg` tells Presolve which algorithm is intended to be used on the reduced model; **NOAlg** should be specified for MIP models.

```
public void protectVariables(const IloIntVarArray var)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of methods instead.

This method specifies a set of integer variables that should not be substituted out of the problem. If presolve can fix a variable to a value, it is removed, even if it is specified in the protected list.

```
public void protectVariables(const IloNumVarArray var)
```

Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method specifies a set of numeric variables that should not be substituted out of the problem. If presolve can fix a variable to a value, it is removed, even if it is specified in the protected list.

```
public void qpIndefCertificate(IloNumVarArray var,  
                             IloNumArray x)
```

The quadratic objective terms in a QP model must form a positive semi-definite Q matrix (negative semi-definite for maximization). If IloCplex finds that this is not true, it will discontinue the optimization. In such cases, the **qpIndefCertificate** method can be used to compute assignments (returned in array *x*) to all variables (returned in array *var*) such that the quadratic term of the objective function evaluates to a negative value ($x^T Q x < 0$ in matrix terms) to prove the indefiniteness.

Note: *CPLEX resizes these arrays for you.*

```
public void readBasis(const char * name) const
```

Reads a simplex basis from the BAS file indicated by *name*, and copies that basis into the invoking IloCplex object. The parameter *AdvInd* must be set to a nonzero value

(e.g. its default setting) for the simplex basis to be used to start a subsequent optimization with one of the Simplex algorithms.

By convention, the file extension is `.bas`. The BAS file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void readMIPStart(const char * name) const
```

Reads the MST file indicated by `name` and copies the MIP start information into the invoking `IloCplex` object. The parameter `MIPStart` must be turned on in order for the MIP start information to be used to with a subsequent MIP optimization. The default value for the parameter is off.

By convention, the file extension is `.mst`. The MST file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void readOrder(const char * filename) const
```

This method reads a priority order from an file in ORD format into the invoking `IloCplex` object. The names in the ORD file must match the names in the active model. The priority order will be associated with the model. The parameter `MipOrdType` must be nonzero, for the next invocation of the method `IloCplex::solve` to take the order into account.

By convention, the file extension is `.ord`. The ORD file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void readParam(const char * name) const
```

Reads parameters and their settings from the file indicated by `name` and applies them to the invoking `IloCplex` object. Parameters not listed in the parameter file will be reset to their default setting.

By convention, the file extension is `.prm`. The PRM file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void readTree(const char * name) const
```

Reads branch-and-bound progress information from a prior run, contained in the TRE file indicated by `name`, into a CPLEX problem object. The parameter `AdvInd` must set to a nonzero value be on (e.g. its default value), in order for the tree to be used for starting a subsequent optimization.

By convention, the file extension is `.tre`. The TRE file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void readVectors(const char * name) const
```

Reads a barrier solution from the VEC file indicated by `name` and copies this information into a CPLEX problem object. This routine is typically used to initiate a crossover from the barrier solution. The parameter `AdvInd` must set to a nonzero value

(e.g. its default setting) in order for the vector file to take effect for starting a crossover at the next invocation of a simplex algorithm with method `solve`.

By convention, the file extension is `.vec`. The VEC file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void setBasisStatuses(const IloCplex::BasisStatusArray cstat,
                             const IloNumVarArray var,
                             const IloCplex::BasisStatusArray rstat,
                             const IloConstraintArray con)
```

This method uses the array `cstats` to set the basis status of the variables in the array `var`; it uses the array `rstats` to set the basis status of the ranges in the array `con`.

```
public void setDefault()
```

This method resets all CPLEX parameters to their default values.

```
public void setDeleteMode(IloCplex::DeleteMode mode)
```

This method sets the delete mode in the invoking `IloCplex` object to `mode`.

```
public void setDirection(IloNumVar var,
                          IloCplex::BranchDirection dir)
```

This method sets the preferred branching direction for variable `var` to `dir`. This setting will cause CPLEX first to explore the branch indicated by `dir` after branching on variable `var`.

```
public void setDirections(const IloNumVarArray var,
                           const IloCplex::BranchDirectionArray dir)
```

This method sets the preferred branching direction for each variable in the array `var` to the corresponding value in the array `dir`. This will cause CPLEX to first explore the branch indicated by `dir[i]` after branching on variable `var[i]`.

```
public void setFormulationBigValue(IloNum v)
```

This method sets the large value used to set a bound on any otherwise unbounded expressions appearing in logical constraints. Such a value is sometimes referred to as Big M. By default, it is set to `1e6`.

The method `getFormulationBigValue` returns the current setting of the large value.

In order to transform a logical constraint automatically into an equivalent MIP formulation, it is sometimes necessary to set a finite upper or lower bound on an otherwise unbounded term. When such a bound does not already exist, CPLEX uses the value set by this method for that purpose. For example, consider the following disjunction (that is, an or-statement):

$$(x == 1) \vee (x \leq 2)$$

It is necessary to have an upper bound on x . If the upper bound on x is `IloInfinity`, CPLEX will use the value controlled by this method in the transformed representation of logical formulations.

```
public void setFormulationEpsValue(IloNum eps)
```

This method sets the epsilon value that is used to model strict relations. In order to model a strict inequality such as $x < b$ in a MIP, IloCplex uses $x \leq b - \text{eps}$ as a surrogate, where `eps` is the value controlled by this method.

```
public void setParam(IloCplex::BoolParam parameter,
                    IloBool value)
```

This method sets `parameter` to `value` in the invoking algorithm. See the *ILOG CPLEX User's Manual* for more detailed information about parameters and for examples of their use.

```
public void setPriorities(const IloNumVarArray var,
                        const IloNumArray pri)
```

This method sets the priority order for all variables in the array `var` to the corresponding value in the array `pri`. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities. Further, variables that have priority assigned to them are given preference over variables that don't. Priorities must be positive integers.

```
public void setPriority(IloNumVar var,
                      IloNum pri)
```

This method sets the priority order for the variable `var` to `pri`. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities. Further, variables that have priority assigned to them are given preference over variables that don't. Priorities must be positive integers.

```
public void setPWLFormulation(IloCplex::PWLFormulation f)
```

This method allows you to control the way piecewise linear expressions are represented when CPLEX automatically transforms them during extraction.

```
public void setVectors(const IloNumArray x,
                    const IloNumArray dj,
                    const IloNumVarArray var,
                    const IloNumArray slack,
                    const IloNumArray pi,
                    const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the method `solve`.

For all variables in `var`, `x[i]` specifies the starting value for the variable `var[i]`. Similarly, `dj[i]` specifies the starting reduced cost for variable `var[i]`. For all

ranged constraints specified in `rng`, `slack[i]` specifies the starting slack value for `rng[i]`. Similarly, `pi[i]` specifies the starting dual value for `rng[i]`.

Zero can be passed for any individual parameter. However, `x` and `var` must be the same length. Likewise, `pi` and `rng` must be the same length.

In other words, you must provide starting values for either the primal or dual variables `x` and `pi`. If you provide values for `dj`, then you must provide the corresponding values for `x`. If you provide values for `slack`, then you must provide the corresponding values for `pi`.

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm, provided that the `AdvInd` parameter is set to a value greater than or equal to one. In particular, if the extracted model is an LP, and the root algorithm is dual or primal, the information is used to construct a starting basis for the simplex method for the original model, if `AdvInd` is set to 1 (one). If `AdvInd` is set to 2, the information is used to construct a starting basis for the presolved model.

If the extracted model is a MIP, only `x` values can be used. A value may be specified for any variable. Usually you would specify values for variables of type `ILOINT`, members of Special Ordered Sets (SOSs), and variables in piecewise linear expressions.

Cplex constructs a subMIP with the variables fixed to their specified values. It solves the subMIP with a node limit of `SubMIPNodeLim`. If it finds a solution, that solution becomes the incumbent in the next search. If it does not find a solution, then it ignores the specified start values.

The parameter `IloCplex::MIPStart` must be turned on (that is, set to `IloTrue`) for the starting point to take effect. Its default value is `IloFalse`.

```
public IloBool solve(IloCplex::Goal goal)
```

This method initializes the goal stack of the root node with `goal` before starting the branch & cut search. The search tree will be defined by the `execute` method of `goal` and its subgoals. See the concept Goals and the nested class `IloCplex::GoalI` for more information.

```
public IloBool solve()
```

This method solves the model currently extracted to the invoking `IloCplex` object. The method returns `IloTrue` if it finds a solution (not necessarily an optimal one).

```
public IloBool solveFixed()
```

After the invoking algorithm has solved the extracted MIP model to a feasible (but not necessarily optimal) solution as a MIP, this member function solves the relaxation of the model obtained by fixing all the integer variables of the extracted MIP to the values of the current solution.

```
public IloCplex::Callback use(IloCplex::Callback cb)
```

This method instructs the invoking `IloCplex` object to use `cb` as a callback. If a callback of the same type as `cb` is already being used by the invoking `IloCplex` object, the previously used callback will be overridden. If the callback object `cb` is already being used by another `IloCplex` object, a copy of `cb` will be used instead. A handle to the callback that is installed in the invoking `IloCplex` object is returned. See [IloCplex::CallbackI](#) for a discussion of how to implement callbacks.

```
public void writeBasis(const char * name) const
```

Writes the current Simplex basis to the file indicated by `name`.

By convention, the file extension is `.bas`. The BAS file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void writeMIPStart(const char * name) const
```

Writes MIP start information to a file indicated by `name`

By convention, the file extension is `.mst`. The MST file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void writeOrder(const char * filename) const
```

This method writes a priority order to the file `filename`. If a priority order has been associated with the CPLEX problem object, or the parameter `MipOrdType` is nonzero, or a MIP feasible solution exists, this method writes the priority order into the indicated file.

By convention, the file extension is `.ord`. The ORD file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void writeParam(const char * name) const
```

Writes the parameter name and its current setting into the file indicated by `name` for all the CPLEX parameters that are not currently set at their default.

By convention, the file extension is `.prm`. The PRM file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void writeTextSolution(const char * name) const
```

Writes a solution file in text format for the current problem to the file indicated by `name`.

```
public void writeTree(const char * name) const
```

Writes the branch-and-bound tree from the last optimization to a file specified by `name` in TRE format.

By convention, the file extension is `.tre`. The TRE file format is documented in the reference manual *ILOG CPLEX File Formats*.

```
public void writeVectors(const char * name) const
```

This method writes solution information from a barrier optimization (without crossover) into the file indicated by `name`. By convention, the file extension is `.vec`. The VEC file format is documented in the reference manual *ILOG CPLEX File Formats*.

IloCplex::Algorithm

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilocplex.h

Synopsis

```

Algorithm{
    NoAlg,
    AutoAlg,
    Primal,
    Dual,
    Barrier,
    Sifting,
    Concurrent,
    Network
};

```

Description The enumeration `IloCplex::Algorithm` lists the available algorithms in CPLEX to solve continuous models as controlled by the parameters `IloCplex::RootAlg` and `IloCplex::NodeAlg`.

See Also [IloCplex](#), [IloCplex::IntParam](#), [getAlgorithm](#), [getSubAlgorithm](#), [RootAlg](#), [NodeAlg](#)

Fields

```

NoAlg
    = CPX_ALG_NONE

AutoAlg
    = CPX_ALG_AUTOMATIC

Primal
    = CPX_ALG_PRIMAL

Dual
    = CPX_ALG_DUAL

Barrier
    = CPX_ALG_BARRIER

Sifting
    = CPX_ALG_SIFTING

Concurrent
    = CPX_ALG_CONCURRENT

```

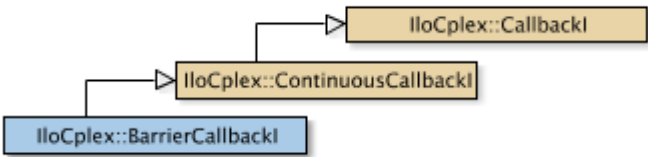
Network

= CPX_ALG_NET

IloCplex::BarrierCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	BarrierCallbackI()

Method Summary	
protected IloNum	getDualObjValue() const

Inherited methods from IloCplex::ContinuousCallbackI
ContinuousCallbackI::getDualInfeasibility , ContinuousCallbackI::getInfeasibility , ContinuousCallbackI::getNiterations , ContinuousCallbackI::getObjValue , ContinuousCallbackI::isDualFeasible , ContinuousCallbackI::isFeasible

Inherited methods from IloCplex::CallbackI


```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQC,
CallbackI::getNrows, CallbackI::main
```

Description

An instance of the class `IloCplex::BarrierCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the barrier optimizer. `IloCplex` calls the user-written callback after each iteration during optimization with the `barrier` method. If an attempt is made to access information not available to an instance of this class, an exception is thrown.

The constructor and methods of this class are for use in deriving a user-written callback class and in implementing the `main` method there.

For more information about the barrier optimizer, see the *ILOG CPLEX User's Manual*.

See Also

`ILOBARRIERCALLBACK0`, `IloCplex`, `IloCplex::Callback`,
`IloCplex::CallbackI`, `IloCplex::ContinuousCallbackI`

Constructors

protected `BarrierCallbackI()`

This constructor creates a callback for use in an application of the barrier optimizer.

Methods

protected `IloNum getDualObjValue() const`

This method returns the current dual objective value of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

IloCplex::BasisStatus

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```
BasisStatus{  
    NotABasicStatus,  
    Basic,  
    AtLower,  
    AtUpper,  
    FreeOrSuperbasic  
};
```

Description The enumeration `IloCplex::BasisStatus` lists values that the status of variables or range constraints may assume in a basis. `NotABasicStatus` is not a valid status for a variable. A basis containing such a status does not constitute a valid basis. The basis status of a ranged constraint corresponds to the basis status of the corresponding slack or artificial variable that `IloCplex` manages for it. `FreeOrSuperbasic` indicates that the variable is nonbasic, but not at a bound.

See Also [IloCplex](#), [IloCplex::BasisStatusArray](#)

Fields

```
NotABasicStatus  
Basic  
AtLower  
AtUpper  
FreeOrSuperbasic
```

IloCplex::BasisStatusArray

Category	Inner Type Definition
Definition File	ilcplex/ilocplexi.h
Include Files	ilocplex.h
Synopsis	<code>IloArray< BasisStatus > BasisStatusArray</code>
Description	This type defines an array-type for <code>IloCplex::BasisStatus</code> . The fully qualified name of a basis status array is <code>IloCplex::BasisStatusArray</code> .
See Also	IloCplex , IloCplex::BasisStatus

IloCplex::BoolParam

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

BoolParam{
    PreInd,
    MIPStart,
    ReverseInd,
    XXXInd,
    MIPOrdInd,
    RelaxPreInd,
    PerInd,
    PreLinear,
    DataCheck,
    BarOOC,
    Symmetry,
    QPmakePSDInd,
    FinalFactor
};

```

Description

The enumeration `IloCplex::BoolParam` lists the parameters of CPLEX that require Boolean values. Boolean values are also known in certain contexts as binary values or as zero-one (0-1) values. Use these values with the methods that accept Boolean parameters: [IloCplex::getParam](#) and [IloCplex::setParam](#).

See the reference manual *ILOG CPLEX Parameters* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

PreInd
= CPX_PARAM_PREIND

MIPStart
= CPX_PARAM_MIPSTART

ReverseInd
= CPX_PARAM_REVERSEIND

XXXInd
= CPX_PARAM_XXXIND

MIPOrdInd

= CPX_PARAM_MIPORDIND

RelaxPreInd

= CPX_PARAM_RELAXPREIND

PerInd

= CPX_PARAM_PERIND

PreLinear

= CPX_PARAM_PRELINEAR

DataCheck

= CPX_PARAM_DATACHECK

BarOOC

= CPX_PARAM_BAROOC

Symmetry

= CPX_PARAM_SYMMETRY

QPmakePSDInd

= CPX_PARAM_QPMAKEPSDIND

FinalFactor

= CPX_PARAM_FINALFACTOR

IloCplex::BranchCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	BranchCallbackI()

Method Summary

protected IloNum	getBranch(IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const
protected BranchCallbackI::BranchType	getBranchType() const
protected IloInt	getNbranches() const
protected NodeId	getNodeId() const
protected IloBool	isIntegerFeasible() const
protected NodeId	makeBranch(const IloConstraintArray cons, const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
protected NodeId	makeBranch(const IloConstraintArray cons, const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
protected NodeId	makeBranch(const IloConstraint con, IloNum objestimate, NodeData * data=0)
protected NodeId	makeBranch(const IloConstraintArray cons, IloNum objestimate, NodeData * data=0)
protected NodeId	makeBranch(const IloIntVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)

protected NodeId	<code>makeBranch(const IloNumVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)</code>
protected void	<code>prune()</code>

Inherited methods from **IloCplex::ControlCallbackI**

`ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,`
`ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,`
`ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,`
`ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,`
`ControlCallbackI::getLB, ControlCallbackI::getLB,`
`ControlCallbackI::getLBs, ControlCallbackI::getLBs,`
`ControlCallbackI::getNodeData, ControlCallbackI::getObjValue,`
`ControlCallbackI::getSlack, ControlCallbackI::getSlacks,`
`ControlCallbackI::getUB, ControlCallbackI::getUB,`
`ControlCallbackI::getUBs, ControlCallbackI::getUBs,`
`ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,`
`ControlCallbackI::getValue, ControlCallbackI::getValue,`
`ControlCallbackI::getValue, ControlCallbackI::getValues,`
`ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,`
`ControlCallbackI::isSOSFeasible`

Inherited methods from **IloCplex::MIPCallbackI**

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Inner Enumeration

```
BranchCallbackI::BranchType
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class **IloCplex::BranchCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer program (MIP). The user-written callback is called prior to branching at a node in the branch & cut tree during the optimization of a MIP. It allows you to query how the invoking instance of **IloCplex** is about to create subnodes at the current node and

gives you the option to override the selection made by the invoking instance of [IloCplex](#). You can create zero, one, or two branches.

- ◆ The method [prune](#) removes the current node from the search tree. No subnodes from the current node will be added to the search tree.
- ◆ The method [makeBranch](#) tells an instance of [IloCplex](#) how to create a subproblem. You may call this method zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call).
- ◆ If you call neither [IloCplex::BranchCallbackI::prune](#) nor [IloCplex::BranchCallbackI::makeBranch](#), the instance of [IloCplex](#) proceeds with its own selection.
- ◆ Calling both [IloCplex::BranchCallbackI::prune](#) and [IloCplex::BranchCallbackI::makeBranch](#) in one invocation of a branch callback is an error and results in unspecified behavior.

The methods of this class are for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[ILOBRANCHCALLBACK0](#), [IloCplex::BranchDirection](#),
[IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::MIPCallbackI](#), [IloCplex::ControlCallbackI](#)

Constructors

protected **BranchCallbackI()**

This constructor creates a branch callback, that is, a control callback for splitting a node into two branches.

Methods

protected **IloNum getBranch**([IloNumVarArray](#) vars,
[IloNumArray](#) bounds,
[IloCplex::BranchDirectionArray](#) dirs,
[IloInt](#) i) const

This method accesses branching information for the *i*-th branch that the invoking instance of [IloCplex](#) is about to create. The parameter *i* must be between 0 (zero) and (`getNbranches` - 1); that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

protected `BranchCallbackI::BranchType` **getBranchType()** const

This method returns the type of branching `IloCplex` is going to do for the current node.

protected `IloInt` **getNbranches()** const

This method returns the number of branches `IloCplex` is going to create at the current node.

protected `NodeId` **getNodeId()** const

Returns the `NodeId` of the current node.

protected `IloBool` **isIntegerFeasible()** const

This method returns `IloTrue` if the solution of the current node is integer feasible.

protected `NodeId` **makeBranch**(const `IloConstraintArray` cons,
 const `IloIntVarArray` vars,
 const `IloNumArray` bounds,
 const `IloCplex::BranchDirectionArray` dirs,
 `IloNum` objestimate,
 `NodeData * data=0`)

This method offers the same facilities as the other methods

`IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

protected `NodeId` **makeBranch**(const `IloConstraintArray` cons,
 const `IloNumVarArray` vars,
 const `IloNumArray` bounds,
 const `IloCplex::BranchDirectionArray` dirs,
 `IloNum` objestimate,
 `NodeData * data=0`)

This method offers the same facilities as the other methods

`IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

```
protected NodeId makeBranch(const IloConstraint con,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method offers the same facilities for a branch specified by only one constraint as `IloCplex::BranchCallbackI::makeBranch` does for a branch specified by a set of constraints.

```
protected NodeId makeBranch(const IloConstraintArray cons,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`, by specifying a branch on constraints. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

- ◆ The parameter `cons` specifies an array of constraints that are to be added for the subnode being created.
- ◆ The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such `data` objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
protected NodeId makeBranch(const IloIntVar var,
                             IloNum bound,
                             IloCplex::BranchDirection dir,
                             IloNum objestimate,
                             NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
protected NodeId makeBranch(const IloNumVar var,
                             IloNum bound,
                             IloCplex::BranchDirection dir,
                             IloNum objestimate,
                             NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
protected NodeId makeBranch(const IloIntVarArray vars,
                             const IloNumArray bounds,
                             const IloCplex::BranchDirectionArray dirs,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- ◆ The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
protected NodeId makeBranch(const IloNumVarArray vars,
                             const IloNumArray bounds,
                             const IloCplex::BranchDirectionArray dirs,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- ◆ The parameter `objectestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

protected void **prune()**

By calling this method, you instruct the CPLEX branch & cut search not to create any child nodes from the current node, or, in other words, to discard nodes below the current node; it does not revisit the discarded nodes below the current node. In short, it creates no branches. It is an error to call both `prune` and `makeBranch` in one invocation of a callback.

BranchCallbackI::BranchType

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

    BranchType{
        BranchOnVariable,
        BranchOnSOS1,
        BranchOnSOS2,
        BranchOnAny,
        UserBranch
    };

```

Description `IloCplex::BranchCallbackI::BranchType` is an enumeration limited in scope to the class `IloCplex::BranchCallbackI`. This enumeration is used by the method `IloCplex::BranchCallbackI::getBranchType` to tell what kind of branch `IloCplex` is about to do:

- ◆ `BranchOnVariable` indicates branching on a single variable.
- ◆ `BranchOnAny` indicates multiple bound changes and constraints will be used for branching.
- ◆ `BranchOnSOS1` indicates branching on an SOS of type 1.
- ◆ `BranchOnSOS2` indicates branching on an SOS of type 2.

See Also [IloCplex::BranchCallbackI](#)

Fields

```

    BranchOnVariable
    = CPX_TYPE_VAR

    BranchOnSOS1
    = CPX_TYPE_SOS1

    BranchOnSOS2
    = CPX_TYPE_SOS2

    BranchOnAny
    = CPX_TYPE_ANY

    UserBranch

```

IloCplex::BranchDirection

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

    BranchDirection{
        BranchGlobal,
        BranchDown,
        BranchUp
    };

```

Description The enumeration `IloCplex::BranchDirection` lists values that can be used for specifying branch directions either with the branch direction parameter `IloCplex::BrDir` or with the methods [IloCplex::setDirection](#) and [IloCplex::setDirections](#). The branch direction specifies which direction to explore first after branching on one variable.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#), [IloCplex::BranchDirectionArray](#)

Fields

```

    BranchGlobal
    = CPX_BRANCH_GLOBAL

    BranchDown
    = CPX_BRANCH_DOWN

    BranchUp
    = CPX_BRANCH_UP

```

IloCplex::BranchDirectionArray

Category	Inner Type Definition
Definition File	ilcplex/ilocplexi.h
Include Files	ilocplex.h
Synopsis	<code>IloArray< BranchDirection > BranchDirectionArray</code>
Description	This type defines an array-type for <code>IloCplex::BranchDirection</code> . The fully qualified name of a branch direction array is <code>IloCplex::BranchDirectionArray</code> .
See Also	IloCplex , IloCplex::BranchDirection

IloCplex::Callback

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
public	Callback(IloCplex::CallbackI * impl=0)

Method Summary	
public void	end()
public IloCplex::CallbackI *	getImpl() const
public Callback::Type	getType() const

Inner Enumeration	
Callback::Type	

Description This class is the handle class for all callback implementation classes available for IloCplex. Callback implementation classes are user-defined classes derived from a subclass of [IloCplex::CallbackI](#).

See Also [IloCplex](#), [IloCplex::CallbackI](#)

Constructors public **Callback**([IloCplex::CallbackI](#) * impl=0)
This constructor creates a callback handle object and initializes it to the implementation object passed as the argument.

Methods

```
public void end()
```

This method deletes the implementation object pointed at by the invoking handle and sets the pointer to 0.

```
public IloCplex::CallbackI * getImpl() const
```

This method returns a pointer to the implementation object of the invoking handle.

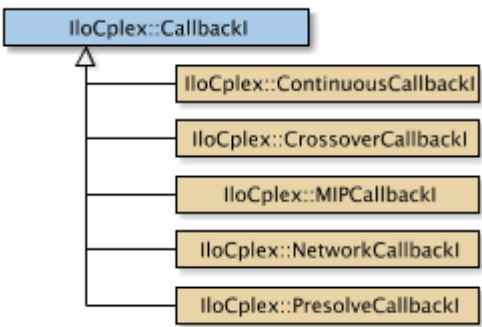
```
public Callback::Type getType() const
```

This method returns the type of the callback implementation object referenced by the invoking handle.

IloCplex::Callback

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected void	CallbackI::abort()
protected virtual CallbackI *	CallbackI::duplicateCallback() const
protected IloEnv	CallbackI::getEnv() const
protected IloModel	CallbackI::getModel() const
protected IloInt	CallbackI::getNcols() const
protected IloInt	CallbackI::getNQCcs() const
protected IloInt	CallbackI::getNrows() const
protected virtual void	CallbackI::main()

Description This is the abstract base class for user-written callback classes. It provides their common application programming interface (API). Callbacks may be called repeatedly at various points during an optimization; for each place a callback is called, ILOG CPLEX provides a separate callback class (derived from this class). Such a callback class provides the specific API as a protected method to use for the particular implementation.

You do not create instances of this class; rather, you use one of its child classes to implement your own callback. In order to implement your user-written callbacks with an instance of `IloCplex`, you should follow these steps:

- ◆ Determine which kind of callback you want to write, and choose the appropriate class for it. The class hierarchy in `Tree` may give you some ideas here.
- ◆ Derive your own subclass, `MyCallbackI`, say, from the appropriate predefined callback class.
- ◆ In your subclass of the callback class, use the protected API defined in the base class to implement the `main` routine of your user-written callback. (All constructors of predefined callback classes are protected; they can be called only from user-written derived subclasses.)
- ◆ In your subclass, implement the method `duplicateCallback`.
- ◆ Write a function `myCallback`, say, that creates an instance of your implementation class in the Concert Technology environment and returns it as an `IloCplex::Callback` handle.
- ◆ Create an instance of your callback class and pass it to the member function `use`.

Note: Macros `ILOXXXCALLBACKn` (for `n` from 0 to 7) are available to facilitate steps 2 through 5, where `XXX` stands for the particular callback under construction and `n` stands for the number of parameters that the function written in step 5 is to receive in addition to the environment parameter.

You can use one instance of a callback with only one instance of `IloCplex`. When you use a callback with a second instance of `IloCplex`, a copy will be automatically created using the method `duplicateCallback`, and that copy will be used instead.

Also, an instance of `IloCplex` takes account of only one instance of a particular callback at any given time. That is, if you call `use` more than once with the same class of callback, the last call overrides any previous one. For example, you can use only one primal simplex callback at a time, or you can use only one network callback at a time; and so forth.

There are two varieties of callbacks:

- ◆ Query callbacks enable your application to retrieve information about the current solution in an instance of `IloCplex`. The information available depends on the algorithm (primal simplex, dual simplex, barrier, mixed integer, or network) that you are using. For example, a query callback can return the current objective value, the number of simplex iterations that have been completed, and other details. Query

callbacks can also be called from presolve, probing, fractional cuts, and disjunctive cuts.

- ◆ Control callbacks enable you to direct the search when you are solving a MIP in an instance of `IloCplex`. For example, control callbacks enable you to select the next node to process or to control the creation of subnodes (among other possibilities).

Existing extractables should never be modified within a callback. Temporary extractables, such as arrays, expressions, and range constraints, can be created and modified. Temporary extractables are often useful, for example, for computing cuts.

See Also

`ILOBARRIERCALLBACK0`, `ILOBRANCHCALLBACK0`, `IloCplex`,
`IloCplex::BarrierCallbackI`, `IloCplex::BranchCallbackI`,
`IloCplex::Callback`, `IloCplex::ControlCallbackI`,
`IloCplex::CrossoverCallbackI`, `IloCplex::CutCallbackI`,
`IloCplex::DisjunctiveCutCallbackI`,
`IloCplex::SimplexCallbackI`,
`IloCplex::FractionalCutCallbackI`,
`IloCplex::HeuristicCallbackI`, `IloCplex::IncumbentCallbackI`,
`IloCplex::ContinuousCallbackI`, `IloCplex::MIPCallbackI`,
`IloCplex::NetworkCallbackI`, `IloCplex::NodeCallbackI`,
`IloCplex::PresolveCallbackI`, `IloCplex::ProbingCallbackI`,
`IloCplex::SolveCallbackI`, `ILOCROSSOVERCALLBACK0`,
`ILOCUTCALLBACK0`, `ILOBRANCHCALLBACK0`,
`ILODISJUNCTIVECUTCALLBACK0`, `ILOFRACTIONALCUTCALLBACK0`,
`ILOHEURISTICCALLBACK0`, `ILOINCUMBENTCALLBACK0`,
`ILOCONTINUOUSCALLBACK0`, `ILOMIPCALLBACK0`, `ILONETWORKCALLBACK0`,
`ILONODECALLBACK0`, `ILOPRESOLVECALLBACK0`, `ILOPROBINGCALLBACK0`,
`ILOSIMPLEXCALLBACK0`, `ILOSOLVECALLBACK0`

Methods

protected void **abort**()

This method instructs CPLEX to stop the current optimization after the user callback finishes. Note that executing additional `IloCplex` callback methods in the callback can lead to unpredictable behavior. For example, callback methods such as `IloCplex::SolveCallbackI::solve` or `IloCplex::BranchCallbackI::makeBranch` can overwrite the callback status and thus enable the optimization to continue. Therefore, to abort an optimization effectively, a user should exit the callback by one of the following ways:

- ◆ Call `return` immediately after the call of `abort`.
- ◆ Structure the callback so that it calls no additional methods of `IloCplex::CallbackI` and its subclasses after `abort`.

protected virtual CallbackI * **duplicateCallback**() const

This virtual method must be implemented to create a copy of the invoking callback object on the same environment. Typically the following implementation will work for a callback class called `MyCallbackI`:

```
IloCplex::CallbackI* MyCallbackI::duplicateCallback() const {
    return (new (getEnv()) MyCallbackI(*this));
}
```

This method is called by an `IloCplex` object in two cases:

- ◆ When `cplex.use(cb)` is called for a callback object `cb` that is already used by another instance of `IloCplex`, a copy of the implementation object of `cb` is created by calling `duplicateCallback` and used in its place. The method `use` will return a handle to that copy.
- ◆ When a parallel optimizer is used, `IloCplex` creates copies of every callback that it uses by calling `duplicateCallback`. One copy of a callback is created for each additional thread being used in the parallel optimizer. During the optimization, each thread will invoke the copy corresponding to the thread number. The methods provided by the callback APIs are guaranteed to be threadsafe. However, when accessing parameters passed to callbacks or members stored in a callback, it is up to the user to make sure of thread safety by synchronizing access or creating distinct copies of the data in the implementation of `duplicateCallback`.

```
protected IloEnv getEnv() const
```

This method returns the environment belonging to the `IloCplex` object that invoked the method `main`.

```
protected IloModel getModel() const
```

This method returns the model currently extracted for the instance of `IloCplex` where the invoking callback executed.

```
protected IloInt getNcols() const
```

This method returns the number of columns in the model currently being optimized.

```
protected IloInt getNQCs() const
```

This method returns the number of quadratic constraints in the model currently being optimized.

```
protected IloInt getNrows() const
```

This method returns the number of rows in the model currently being optimized.

```
protected virtual void main()
```

This virtual method is to be implemented by the user in a derived callback class to define the functionality of the callback. When an instance of `IloCplex` uses a callback (an instance of `IloCplex::CallbackI` or one of its derived subclasses), `IloCplex`

calls this virtual method `main` at the point during the optimization at which the callback is executed.

Callback::Type

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

Type{
Continuous,
Presolve,
Simplex,
Barrier,
Crossover,
Network,
MIP,
Probing,
FractionalCut,
DisjunctiveCut,
Branch,
Cut,
Node,
Heuristic,
Incumbent,
Solve,
FlowMIRCut,
_Number
};

```

Description This enumeration type is used to identify the type of a callback implementation object referred to by an `IloCplex::Callback` handle.

See Also [IloCplex::Callback](#)

Fields

```

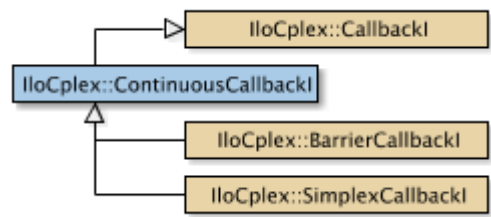
Continuous
Presolve
Simplex
Barrier
Crossover
Network
MIP
Probing
FractionalCut
DisjunctiveCut
Branch
Cut
Node
Heuristic
Incumbent
Solve
FlowMIRCut
_Number

```


IloCplex::ContinuousCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	ContinuousCallbackI()

Method Summary	
protected IloNum	ContinuousCallbackI::getDualInfeasibility() const
protected IloNum	ContinuousCallbackI::getInfeasibility() const
protected IloInt	ContinuousCallbackI::getNiterations() const
protected IloNum	ContinuousCallbackI::getObjValue() const
protected IloBool	ContinuousCallbackI::isDualFeasible() const
protected IloBool	ContinuousCallbackI::isFeasible() const

Inherited methods from IloCplex::CallbackI

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQC,
CallbackI::getNrows, CallbackI::main
```

Description

An instance of a class derived from `IloCplex::ContinuousCallbackI` represents a user-written callback in an ILOG CPLEX application that uses an instance of `IloCplex` with the primal simplex, dual simplex, or barrier optimizer. `IloCplex` calls the user-written callback after each iteration during an optimization of a problem solved at a node. This class offers methods for use within the callbacks you write. In particular, there are methods in this class to access primal and dual feasibility, number of iterations, and objective value.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

Note: There are special callbacks for simplex and barrier, that is, `IloCplex::SimplexCallbackI` and `IloCplex::BarrierCallbackI`, respectively. Using a continuous callback sets this callback in both of these algorithms. If a special callback was already set for one of these algorithms, (for example, simplex) it is replaced by the general continuous callback.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [ILOCONTINUOUSCALLBACK0](#)

Constructors

protected **ContinuousCallbackI()**

This constructor creates a callback for use in an application that solves continuous models.

Methods

protected **IloNum getDualInfeasibility()** const

This method returns the current dual infeasibility measure of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

protected **IloNum getInfeasibility()** const

This method returns the current primal infeasibility measure of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

protected **IloInt getNiterations()** const

This method returns the number of iterations completed so far by an instance of `IloCplex` at the invoking callback is executed.

protected `IloNum` **getObjValue()** const

This method returns the current objective value of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

protected `IloBool` **isDualFeasible()** const

This method returns `IloTrue` if the current solution is dual feasible.

protected `IloBool` **isFeasible()** const

This method returns `IloTrue` if the current solution is primal feasible.

IloCplex::ControlCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected IloNum	ControlCallbackI::getDownPseudoCost(const IloIntVar var) const
protected IloNum	ControlCallbackI::getDownPseudoCost(const IloNumVar var) const
protected void	ControlCallbackI::getFeasibilities(IntegerFeasibilityArray stat,const IloIntArray var) const
protected void	ControlCallbackI::getFeasibilities(IntegerFeasibilityArray stat,const IloNumVarArray var) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloIntVar var) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloNumVar var) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloSOS2 sos) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloSOS1 sos) const
protected IloNum	ControlCallbackI::getLB(const IloIntVar var) const
protected IloNum	ControlCallbackI::getLB(const IloNumVar var) const
protected void	ControlCallbackI::getLBs(IloNumArray val,const IloIntArray vars) const
protected void	ControlCallbackI::getLBs(IloNumArray val,const IloNumVarArray vars) const
protected NodeData *	ControlCallbackI::getNodeData() const
protected IloNum	ControlCallbackI::getObjValue() const

protected IloNum	ControlCallbackI::getSlack(const IloRange rng) const
protected void	ControlCallbackI::getSlacks(IloNumArray val,const IloRangeArray con) const
protected IloNum	ControlCallbackI::getUB(const IloIntVar var) const
protected IloNum	ControlCallbackI::getUB(const IloNumVar var) const
protected void	ControlCallbackI::getUBs(IloNumArray val,const IloIntVarArray vars) const
protected void	ControlCallbackI::getUBs(IloNumArray val,const IloNumVarArray vars) const
protected IloNum	ControlCallbackI::getUpPseudoCost(const IloIntVar var) const
protected IloNum	ControlCallbackI::getUpPseudoCost(const IloNumVar var) const
protected IloNum	ControlCallbackI::getValue(const IloIntVar var) const
protected IloNum	ControlCallbackI::getValue(const IloNumVar var) const
protected IloNum	ControlCallbackI::getValue(const IloExprArg expr) const
protected void	ControlCallbackI::getValues(IloNumArray val,const IloIntVarArray vars) const
protected void	ControlCallbackI::getValues(IloNumArray val,const IloNumVarArray vars) const
protected IloBool	ControlCallbackI::isSOSFeasible(const IloSOS2 sos2) const
protected IloBool	ControlCallbackI::isSOSFeasible(const IloSOS1 sos1) const

Inherited methods from IloCplex::MIPCallbackI

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff, MIPCallbackI::getDirection, MIPCallbackI::getDirection, MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues, MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum, MIPCallbackI::getNcliques, MIPCallbackI::getNcovers, MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers, MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts, MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds, MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs, MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes, MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs, MIPCallbackI::getPriority, MIPCallbackI::getPriority, MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

Inherited methods from **IloCplex::CallbackI**

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv, CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs, CallbackI::getNrows, CallbackI::main

Inner Enumeration

ControlCallbackI::IntegerFeasibility	

Inner Class

ControlCallbackI::PresolvedVariableException	

Inner Type Def

ControlCallbackI::IntegerFeasibilityArray	

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

This class defines the common application programming interface (API) for the following classes that allow you to control the MIP search:

- ◆ `IloCplex::SolveCallbackI`
- ◆ `IloCplex::CutCallbackI`
- ◆ `IloCplex::HeuristicCallbackI`
- ◆ `IloCplex::BranchCallbackI`

An instance of one of these classes represents a user-written callback that intervenes in the search for a solution at a given node in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). Control callbacks are tied to a node. They are called at each node during `IloCplex` branch & cut search. The user never subclasses the `IloCplex::ControlCallbackI` class directly; it only defines the common interface of those listed callbacks.

In particular, `SolveCallbackI` is called before solving the node relaxation and optionally allows substitution of its solution. `IloCplex` does this by default. After the node relaxation has been solved, either by an instance of `SolveCallbackI` or by `IloCplex`, the other control callbacks are called in the following order:

- ◆ `IloCplex::CutCallbackI`
- ◆ `IloCplex::HeuristicCallbackI`
- ◆ `IloCplex::BranchCallbackI`

If the cut callback added new cuts to the node relaxation, the node relaxation will be solved again using the solve callback, if used. The same is true if `IloCplex` generated its own cuts.

The methods of this class are protected and its constructor is private; you cannot directly subclass this class; you must derive from its subclasses.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[ControlCallbackI::IntegerFeasibility](#),
[IntegerFeasibilityArray](#), [IloCplex::MIPCallbackI](#)

Methods

protected [IloNum](#) **getDownPseudoCost**(const [IloIntVar](#) var) const

This method returns the current pseudo cost for branching downward on the variable `var`.

protected [IloNum](#) **getDownPseudoCost**(const [IloNumVar](#) var) const

This method returns the current pseudo cost for branching downward on the variable `var`.

protected void **getFeasibilities**([IntegerFeasibilityArray](#) stat,
const [IloIntVarArray](#) var) const

This method indicates whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array `stats`.

protected void **getFeasibilities**([IntegerFeasibilityArray](#) stat,
const [IloNumVarArray](#) var) const

This method indicates whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array `stats`.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloIntVar](#) var) const

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloNumVar](#) var) const

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloSOS2](#) sos) const

This method indicates whether the Special Ordered Set `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloSOS1](#) sos) const

This method indicates whether the Special Ordered Set `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [IloNum](#) **getLB**(const [IloIntVar](#) var) const

This method returns the lower bound of **var** at the current node. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes. The corresponding solution value from **getValue** may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of **var** at the current node. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes. The corresponding solution value from **getValue** may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected void getLBs(IloNumArray val,
                     const IloIntVarArray vars) const
```

For each element of the array **vars**, this method puts the lower bound at the current node into the corresponding element of the array **vals**. These bounds are likely to be different from the bounds in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes. The corresponding solution values from **getValues** may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected void getLBs(IloNumArray val,
                     const IloNumVarArray vars) const
```

This method puts the lower bound at the current node of each element of the array **vars** into the corresponding element of the array **vals**. These bounds are likely to be different from the bounds in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes. The corresponding solution values from **getValues** may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected NodeData * getNodeData() const
```

This method retrieves the **NodeData** object that may have previously been assigned to the current node by the user with the method **IloCplex::BranchCallbackI::makeBranch**. If no data object has been assigned to the current node, 0 (zero) will be returned.

```
protected IloNum getObjValue() const
```

This method returns the objective value of the solution of the relaxation at the current node.

```
protected IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint indicated by `rng` in the solution of the relaxation at the current node.

```
protected void getSlacks(IloNumArray val,
                        const IloRangeArray con) const
```

For each of the constraints in the array of ranges `rngs`, this method puts the slack value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected void getUBs(IloNumArray val,
                      const IloIntVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected void getUBs(IloNumArray val,
                      const IloNumVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
protected IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
protected IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
protected IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
protected IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the expression `expr` in the solution of the relaxation at the current node.

```
protected void getValues(IloNumArray val,
                        const IloIntVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected void getValues(IloNumArray val,
                        const IloNumVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a parameter to this method may be of type 2. See the *ILOG CPLEX User's Manual* for more explanation of types of special ordered sets.

```
protected IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a parameter to this method may be of type 1. See the *ILOG CPLEX User's Manual* for more explanation about these types of special ordered sets.

ControlCallbackI::IntegerFeasibility

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

IntegerFeasibility{
    ImpliedInfeasible,
    Feasible,
    Infeasible,
    ImpliedFeasible
};

```

Description The enumeration `IloCplex::ControlCallbackI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::ControlCallbackI`. This enumeration is used by `IloCplex::ControlCallbackI::getFeasibility` to represent the integer feasibility of a variable or SOS in the current node solution:

- ◆ `Feasible` indicates the variable or SOS is integer feasible.
- ◆ `ImpliedFeasible` indicates the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- ◆ `Infeasible` indicates the variable or SOS is integer infeasible.

See Also [IloCplex](#), [ControlCallbackI::IntegerFeasibilityArray](#)

Fields

```

ImpliedInfeasible
Feasible
    = CPX_INTEGER_FEASIBLE

Infeasible
    = CPX_INTEGER_INFEASIBLE

ImpliedFeasible
    = CPX IMPLIED_INTEGER_FEASIBLE

```

IloCplex::CplexStatus

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

CplexStatus{
    Unknown,
    Optimal,
    Unbounded,
    Infeasible,
    InforUnbd,
    OptimalInfeas,
    NumBest,
    FeasibleRelaxed,
    OptimalRelaxed,
    AbortObjLim,
    AbortPrimObjLim,
    AbortDualObjLim,
    AbortItLim,
    AbortTimeLim,
    AbortUser,
    OptimalFaceUnbounded,
    OptimalTol,
    SolLim,
    NodeLimFeas,
    NodeLimInfeas,
    FailFeas,
    FailInfeas,
    MemLimFeas,
    MemLimInfeas,
    FailFeasNoTree,
    FailInfeasNoTree
};

```

Description The enumeration `IloCplex::CplexStatus` lists values that the status of an `IloCplex` algorithm can assume. These values can be accessed via the method [getCplexStatus](#) and [getCplexSubStatus](#) and provide information about what the algorithm learned about the active model in the most recent invocation of the method `solve` or `feasOpt`. It may also tell why the algorithm terminated.

See the group `optim.cplex.solutionstatus` in the *Callable Library Reference Manual*, where they are listed in alphabetic order, or the topic *Interpreting Solution Status Codes* in the Overview of the APIs, where they are listed in numeric order, for more information about these values. Also see the *ILOG CPLEX User's Manual* for examples of their use.

See also the enumeration `IloAlgorithm::Status` in the *ILOG Concert Technology Reference Manual*.

See Also

[IloCplex](#)

Fields

Unknown
Optimal

= CPX_STAT_OPTIMAL

Unbounded

= CPX_STAT_UNBOUNDED

Infeasible

= CPX_STAT_INFEASIBLE

InfOrUnbd

= CPX_STAT_INForUNBD

OptimalInfeas

= CPX_STAT_OPTIMAL_INFEAS

NumBest

= CPX_STAT_NUM_BEST

FeasibleRelaxed

= CPX_STAT_FEASIBLE_RELAXED

OptimalRelaxed

= CPX_STAT_OPTIMAL_RELAXED

AbortObjLim

= CPX_STAT_ABORT_OBJ_LIM

AbortPrimObjLim

= CPX_STAT_ABORT_PRIM_OBJ_LIM

AbortDualObjLim

= CPX_STAT_ABORT_DUAL_OBJ_LIM

AbortItLim

= CPX_STAT_ABORT_IT_LIM

AbortTimeLim

= CPX_STAT_ABORT_TIME_LIM

AbortUser

= CPX_STAT_ABORT_USER

OptimalFaceUnbounded

= CPX_STAT_OPTIMAL_FACE_UNBOUNDED

OptimalTol

= CPXMIP_OPTIMAL_TOL

SolLim

= CPXMIP_SOL_LIM

NodeLimFeas

= CPXMIP_NODE_LIM_FEAS

NodeLimInfeas

= CPXMIP_NODE_LIM_INFEAS

FailFeas

= CPXMIP_FAIL_FEAS

FailInfeas

= CPXMIP_FAIL_INFEAS

MemLimFeas

= CPXMIP_MEM_LIM_FEAS

MemLimInfeas

= CPXMIP_MEM_LIM_INFEAS

FailFeasNoTree

= CPXMIP_FAIL_FEAS_NO_TREE

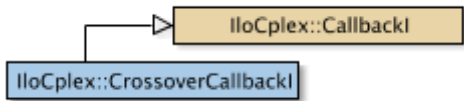
FailInfeasNoTree

= CPXMIP_FAIL_INFEAS_NO_TREE

IloCplex::CrossoverCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	CrossoverCallbackI()

Method Summary	
protected IloInt	getNdualExchanges() const
protected IloInt	getNdualPushes() const
protected IloInt	getNprimalExchanges() const
protected IloInt	getNprimalPushes() const
protected IloInt	getNsuperbasics() const

Inherited methods from IloCplex::CallbackI
CallbackI::abort , CallbackI::duplicateCallback , CallbackI::getEnv , CallbackI::getModel , CallbackI::getNcols , CallbackI::getNQCs , CallbackI::getNrows , CallbackI::main

Description An instance of the class `IloCplex::CrossoverCallbackI` represents a user-written callback in an application that uses an instance of [IloCplex](#) to solve a problem by means of the barrier optimizer with the crossover option. An instance of [IloCplex](#)

calls this callback regularly during crossover. For details about the crossover option, see the ILOG CPLEX User's Manual.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also [IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [ILOCROSSOVERCALLBACK0](#)

Constructors protected **CrossoverCallbackI()**

This constructor creates a callback for use in an application with the crossover option of the barrier optimizer.

Methods protected **IloInt getNdualExchanges()** const

This method returns the number of dual exchange operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

protected **IloInt getNdualPushes()** const

This method returns the number of dual push operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

protected **IloInt getNprimalExchanges()** const

This method returns the number of primal exchange operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

protected **IloInt getNprimalPushes()** const

This method returns the number of primal push operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

protected **IloInt getNsuperbasics()** const

This method returns the number of super basics currently present in the basis being generated with crossover by the instance of `IloCplex` that executes the invoking callback.

IloCplex::CutCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	CutCallbackI()

Method Summary	
protected IloConstraint	add(IloConstraint con)
protected IloConstraint	addLocal(IloConstraint con)

Inherited methods from IloCplex::ControlCallbackI
ControlCallbackI::getDownPseudoCost , ControlCallbackI::getDownPseudoCost , ControlCallbackI::getFeasibilities , ControlCallbackI::getFeasibilities , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getLB , ControlCallbackI::getLB , ControlCallbackI::getLBs , ControlCallbackI::getLBs , ControlCallbackI::getNodeData , ControlCallbackI::getObjValue , ControlCallbackI::getSlack , ControlCallbackI::getSlacks , ControlCallbackI::getUB , ControlCallbackI::getUB , ControlCallbackI::getUBs , ControlCallbackI::getUBs , ControlCallbackI::getUpPseudoCost , ControlCallbackI::getUpPseudoCost , ControlCallbackI::getValue , ControlCallbackI::getValue , ControlCallbackI::getValue , ControlCallbackI::getValues , ControlCallbackI::getValues , ControlCallbackI::isSOSFeasible , ControlCallbackI::isSOSFeasible

Inherited methods from **IloCplex::MIPCallbackI**

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from **IloCplex::CallbackI**

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class **IloCplex::CutCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer programming problem (a MIP). This class offers a method to add a local or global cut to the current node LP subproblem from a user-written callback. More than one cut can be added in this callback by calling the method **add** or **addLocal** multiple times. All added cuts must be linear.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the **main** method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::MIPCallbackI](#), [IloCplex::ControlCallbackI](#),
[ILOUTCALLBACK0](#)

Constructors

protected **CutCallbackI**()

This constructor creates a callback for use in an application with a user-defined cut to solve a MIP.

Methods

protected [IloConstraint](#) **add**([IloConstraint](#) con)

This method adds a cut to the current node LP subproblem for the constraint indicated by con. This cut must be globally valid; it will not be removed by backtracking or any other means during the search. The added cut must be linear.

protected [IloConstraint](#) **addLocal**([IloConstraint](#) con)

This method adds a local cut to the current node LP subproblem for the constraint indicated by con. [IloCplex](#) will manage the local cut in such a way that it will be active only when processing nodes of this subtree. The added cut must be linear.

IloCplex::DeleteMode

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

DeleteMode{
    LeaveBasis,
    FixBasis
};

```

Description This enumeration lists the possible settings for the delete mode of `IloCplex` as controlled by the method [setDeleteMode](#) and queried by the method [getDeleteMode](#).

◆ `IloCplex::LeaveBasis`

With the default setting `IloCplex::LeaveBasis`, an existing basis will remain unchanged if variables or constraints are removed from the loaded LP model. This choice generally renders the basis unusable for a restart when CPLEX is solving the modified LP and the advance indicator (parameter `IloCplex::AdvInd`) is set to `IloTrue`.

◆ `IloCplex::FixBasis`

In contrast, with delete mode set to `IloCplex::FixBasis`, the invoking object will do basis pivots in order to maintain a valid basis when variables or constraints are removed. This choice makes the delete operation more computation-intensive, but may give a better starting point for reoptimization after modification of the extracted model.

If no basis is present in the invoking object, the setting of the delete mode has no effect.

See Also [IloCplex](#)

Fields

```

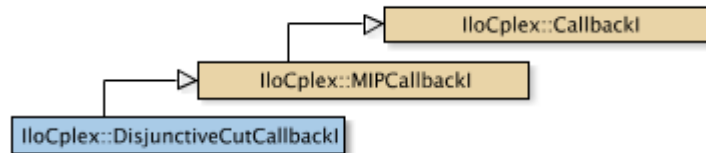
LeaveBasis
FixBasis

```

IloCplex::DisjunctiveCutCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	DisjunctiveCutCallbackI()

Method Summary

protected	IloNum	getProgress() const

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

An instance of the class **IloCplex::DisjunctiveCutCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of disjunctive cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, **IloCplex::Callback**, **IloCplex::CallbackI**,
IloCplex::MIPCallbackI

Constructors

protected **DisjunctiveCutCallbackI()**

This constructor creates a callback for use in an application where disjunctive cuts are generated.

Methods

protected **IloNum** **getProgress()** const

This method returns the fraction of completion of the disjunctive cut generation pass.

IloCplex::DualPricing

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```
DualPricing{
    DPriIndAuto,
    DPriIndFull,
    DPriIndSteep,
    DPriIndFullSteep,
    DPriIndSteepQStart,
    DPriIndDevex
};
```

Description The enumeration `IloCplex::DualPricing` lists values that the dual pricing parameter `IloCplex::DPriInd` can assume in `IloCplex` for use with the dual simplex algorithm. Use these values with the method `IloCplex::setParam(IloCplex::DPriInd, value)` when you set the dual pricing indicator.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

```
DPriIndAuto
    = CPX_DPRIIND_AUTO

DPriIndFull
    = CPX_DPRIIND_FULL

DPriIndSteep
    = CPX_DPRIIND_STEEP

DPriIndFullSteep
    = CPX_DPRIIND_FULLSTEEP

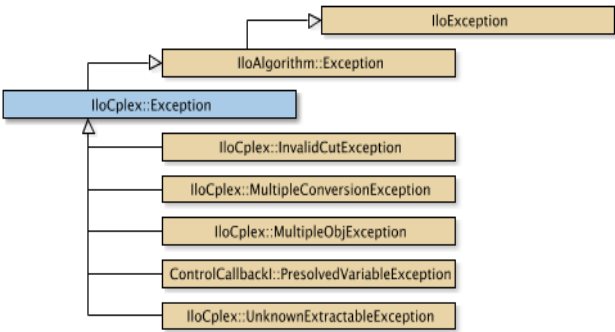
DPriIndSteepQStart
    = CPX_DPRIIND_STEEPQSTART

DPriIndDevex
    = CPX_DPRIIND_DEVEX
```


IloCplex::Exception

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
public IloInt	getStatus() const

Inherited methods from IloException
IloException::end, IloException::getMessage

Description The class IloCplex::Exception, derived from the nested class IloAlgorithm::Exception, is the base class of exceptions thrown by classes derived from IloCplex.

Methods public IloInt **getStatus()** const

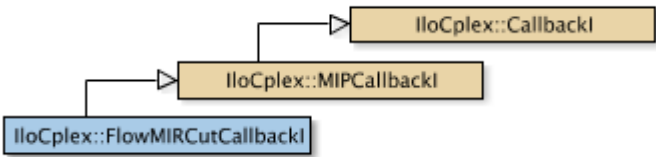
This method returns the ILOG CPLEX error code of an exception thrown by a member of IloCplex. These error codes are detailed in the reference manual as the group optim.cplex.errorcodes.

This method may also return negative values for subclasses of `IloCplex::Exception`, which are not listed in the reference manual. The exceptions listed in the reference manual are always thrown as instances of `IloCplex::Exception` and not as an instance of one of its derived classes.

IloCplex::FlowMIRCutCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilocplex.h

Constructor Summary	
protected	FlowMIRCutCallbackI()

Method Summary	
protected IloNum	getProgress() const

Inherited methods from IloCplex::MIPCallbackI

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

An instance of the class `IloCplex::FlowMIRCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a member function to check on the progress of the generation of Flow MIR cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

`IloCplex`, `IloCplex::Callback`, `IloCplex::CallbackI`,
`IloCplex::MIPCallbackI`

Constructors

protected **`FlowMIRCutCallbackI()`**

This constructor creates a callback for use in an application where flow MIR cuts are generated.

Methods

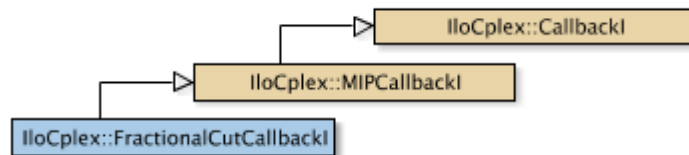
protected `IloNum` **`getProgress()`** const

This method returns the fraction of completion of the cut generation pass for FlowMIR cuts.

IloCplex::FractionalCutCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	FractionalCutCallbackI()

Method Summary

protected	IloNum	getProgress() const

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

An instance of the class **IloCplex::FractionalCutCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of fractional cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the **main** method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, **IloCplex::Callback**, **IloCplex::CallbackI**,
IloCplex::MIPCallbackI

Constructors

protected **FractionalCutCallbackI()**

This constructor creates a callback for use in an application where fractional cuts are generated.

Methods

protected **IloNum** **getProgress()** const

This method returns the fraction of completion of the fractional cut generation pass.

IloCplex::Goal

Category	Inner Class
InheritancePath	
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Constructor Summary	
public	Goal(IloCplex::GoalBaseI * goalI)
public	Goal(const Goal & goal)
public	Goal()
public	Goal(IloConstraint cut)
public	Goal(IloConstraintArray cut)

Method Summary	
public Goal	operator=(const Goal & goal)

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch & cut search in [IloCplex](#). Goals are implemented in the class [IloCplex::GoalI](#). This is the handle class for CPLEX goals.

Goal objects are reference-counted. This means every instance of [IloCplex::GoalI](#) keeps track about how many handle objects refer to it. If this number drops to 0 the [IloCplex::GoalI](#) object is automatically deleted. As a consequence, whenever you deal with a goal, you must keep a handle object around rather than only a pointer to the

implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See *Goals* among the Concepts in this manual. See also goals in the *ILOG CPLEX User's Manual*.

Constructors

```
public Goal(IloCplex::GoalBaseI * goalI)
```

Creates a new goal from a pointer to the implementation object.

```
public Goal(const Goal & goal)
```

This is the copy constructor of the goal.

```
public Goal()
```

Creates a 0 goal handle, that is, a goal with a 0 implementation object pointer. This is also referred to as an empty goal.

```
public Goal(IloConstraint cut)
```

Creates a new goal that will add the constraint `cut` as a local cut to the node where the goal is executed. As a local cut, the constraint will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraint passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, the method `end` must not be called for constraints passed to this constructor explicitly.

```
public Goal(IloConstraintArray cut)
```

Creates a new goal that adds the constraints given in the array `cut` as local cuts to the node where the goal is executed. As local cuts, the constraints will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraints and the array passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, method `end` must not be called for the constraints and the array passed to this constructor explicitly.

Methods

```
public Goal operator=(const Goal & goal)
```

This is the assignment operator. It increases the reference count of the implementation object of `goal`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If thereby the reference count becomes 0, the implementation object is deleted.

IloCplex::Goal

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public	GoalI (IloEnv env)

Method Summary

public void	abort ()
public static IloCplex::Goal	AndGoal (IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal	BranchAsCplexGoal (IloEnv env)
public virtual IloCplex::Goal	duplicateGoal ()
public virtual IloCplex::Goal	execute ()
public static IloCplex::Goal	FailGoal (IloEnv env)
public IloNum	getBestObjValue () const
public IloNum	getBranch (IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const
public GoalI::BranchType	getBranchType () const
public IloNum	getCutoff () const
public IloCplex::BranchDirection	getDirection (const IloIntVar var)
public IloCplex::BranchDirection	getDirection (const IloNumVar var)
public IloNum	getDownPseudoCost (const IloIntVar var) const
public IloNum	getDownPseudoCost (const IloNumVar var) const
public IloEnv	getEnv () const
public void	getFeasibilities (GoalI::IntegerFeasibilityArray stats,const IloIntVarArray vars) const

public void	getFeasibilities(GoalI::IntegerFeasibilityArray stats,const IloNumVarArray vars) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS2 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS1 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloIntVar var) const
public GoalI::IntegerFeasibility	getFeasibility(const IloNumVar var) const
public IloNum	getIncumbentObjValue() const
public IloNum	getIncumbentValue(const IloIntVar var) const
public IloNum	getIncumbentValue(const IloNumVar var) const
public void	getIncumbentValues(IloNumArray val,const IloIntVarArray vars) const
public void	getIncumbentValues(IloNumArray val,const IloNumVarArray vars) const
public IloNum	getLB(const IloIntVar var) const
public IloNum	getLB(const IloNumVar var) const
public void	getLBs(IloNumArray vals,const IloIntVarArray vars) const
public void	getLBs(IloNumArray vals,const IloNumVarArray vars) const
public IloModel	getModel() const
public IloInt	getMyThreadNum() const
public IloInt	getNbranches() const
public IloInt	getNcliques() const
public IloInt	getNcols() const
public IloInt	getNcovers() const
public IloInt	getNdisjunctiveCuts() const
public IloInt	getNflowCovers() const
public IloInt	getNflowPaths() const
public IloInt	getNfractionalCuts() const
public IloInt	getNGUBcovers() const
public IloInt	getNimpliedBounds() const
public IloInt	getNiterations() const
public IloInt	getNMIRs() const
public IloInt	getNnodes() const
public IloInt	getNremainingNodes() const
public IloInt	getNrows() const
public IloNum	getObjCoef(const IloIntVar var) const
public IloNum	getObjCoef(const IloNumVar var) const
public void	getObjCoefs(IloNumArray vals,const IloIntVarArray vars) const

public void	getObjCoefs(IloNumArray vals,const IloNumVarArray vars) const
public IloNum	getObjValue() const
public IloNum	getPriority(const IloIntVar var) const
public IloNum	getPriority(const IloNumVar var) const
public IloNum	getSlack(const IloRange rng) const
public void	getSlacks(IloNumArray vals,const IloRangeArray rngs) const
public IloNum	getUB(const IloIntVar var) const
public IloNum	getUB(const IloNumVar var) const
public void	getUBs(IloNumArray vals,const IloIntVarArray vars) const
public void	getUBs(IloNumArray vals,const IloNumVarArray vars) const
public IloNum	getUpPseudoCost(const IloIntVar var) const
public IloNum	getUpPseudoCost(const IloNumVar var) const
public IloInt	getUserThreads() const
public IloNum	getValue(const IloIntVar var) const
public IloNum	getValue(const IloNumVar var) const
public IloNum	getValue(const IloExpr expr) const
public void	getValues(IloNumArray vals,const IloIntVarArray vars) const
public void	getValues(IloNumArray vals,const IloNumVarArray vars) const
public static IloCplex::Goal	GlobalCutGoal(IloConstraintArray con)
public static IloCplex::Goal	GlobalCutGoal(IloConstraint con)
public IloBool	hasIncumbent() const
public IloBool	isIntegerFeasible() const
public IloBool	isSOSFeasible(const IloSOS2 sos2) const
public IloBool	isSOSFeasible(const IloSOS1 sos1) const
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1,IloCplex::Goal goal2)
public static IloCplex::Goal	SolutionGoal(const IloIntVarArray vars,const IloNumArray vals)
public static IloCplex::Goal	SolutionGoal(const IloNumVarArray vars,const IloNumArray vals)

Inner Enumeration

GoalI::BranchType	
GoalI::IntegerFeasibility	

Inner Type Def	
GoalI::GoalI::IntegerFeasibilityArray	

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch & cut search in [IloCplex](#). Goals are implemented in subclasses of the class `IloCplex::GoalI`. This is the base class for user-written implementation classes of CPLEX goals.

To implement your own goal you need to create a subclass of `IloCplex::GoalI` and implement its pure virtual methods `execute` and `duplicateGoal`. You may use one of the `ILOCPLEXGOAL0` macros to assist you in doing so. After implementing your goal class, you use it by passing it to the `solve` method when solving the model.

The method `duplicateGoal` may be called by [IloCplex](#) to create copies of a goal when needed for parallel branch & cut search. Thus the implementation of this method must create and return an exact copy of the invoked object itself.

The method `execute` controls the branch & cut search of [IloCplex](#) by the goal it returns. When [IloCplex](#) processes a node, it pops the top goal from the node's goal stack and calls method `execute` of that goal. It continues executing the top goal from the stack until the node is deactivated or the goal stack is empty. If the goal stack is empty, [IloCplex](#) proceeds with the built-in search strategy for the subtree rooted at the current node.

The class `IloCplex::GoalI` provides several methods for querying information about the current node. The method `execute` controls how to proceed with the branch & cut search via the goal it returns. The returned goal, unless it is the 0 goal, is pushed on the goal stack and will thus be executed next.

See also the chapter about goals in the *ILOG CPLEX User's Manual*.

Constructors

```
public GoalI(IloEnv env)
```

The goal constructor. It requires an instance of the same `IloEnv` as the `IloCplex` object with which to use the goal. The environment can later be queried by calling method `getEnv`.

Methods

```
public void abort()
```

Abort the optimization, that is, stop the execution of method `solve` currently in process.

```
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1,  
                                     IloCplex::Goal goal2)
```

The static methods `AndGoal` all return a goal that pushes the goals passed as parameters onto the goal stack in reverse order. As a consequence, the goals will be executed in the order they are passed as parameters to the `AndGoal` function.

```
public static IloCplex::Goal BranchAsCplexGoal(IloEnv env)
```

This static function returns a goal that creates the same branches as the currently selected built-in branch strategy of `IloCplex` would choose at the current node. This goal allows you to proceed with the `IloCplex` search strategy, but keeps the search under goal control, thereby giving you the option to intervene at any point.

This goal is also important when you use node evaluators while you use a built-in branching strategy.

For example, consider the `execute` method of a goal starting like this:

```
if (!isIntegerFeasible())  
    return AndGoal(BranchAsCplexGoal(getEnv()), this);  
// do something
```

It would do something only when `IloCplex` found a solution it considers to be a candidate for a new incumbent. Note the test of integer feasibility before returning the `BranchAsCplexGoal`. Without it, `BranchAsCplex` would be executed for a solution `IloCplex` considers to be feasible, but `IloCplex` would not know how to branch on it. An endless loop would result.

```
public virtual IloCplex::Goal duplicateGoal()
```

This virtual method must be implemented by the user. It must return a copy of the invoking goal object. This method may be called by `IloCplex` when doing parallel branch & cut search.

```
public virtual IloCplex::Goal execute()
```

This virtual method must be implemented by the user to specify the logic of the goal. The instance of `IloCplex::Goal` returned by this method will be added to the goal stack of the node where the invoking goal is being executed for further execution.

```
public static IloCplex::Goal FailGoal(IloEnv env)
```

This static method creates a goal that fails. That means that the branch where the goal is executed will be pruned or, equivalently, the search is discontinued at that node and the node is discarded.

```
public IloNum getBestObjValue() const
```

This method returns the currently best known bound on the optimal solution value of the problem at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. When a model has been solved to optimality, this value matches the optimal solution value. Otherwise, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

```
public IloNum getBranch(IloNumVarArray vars,
                       IloNumArray bounds,
                       IloCplex::BranchDirectionArray dirs,
                       IloInt i) const
```

This method accesses branching information for the `i`-th branch that the invoking instance of `IloCplex` is about to create. The parameter `i` must be between 0 (zero) and `getNbranches` - 1; that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

```
public GoalI::BranchType getBranchType() const
```

This method returns the type of branching `IloCplex` is going to do for the current node.

```
public IloNum getCutoff() const
```

The method returns the current cutoff value. An instance of `IloCplex` uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

```
public IloCplex::BranchDirection getDirection(const IloIntVar var)
```

Returns the solution value of `var` in the incumbent solution at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloCplex::BranchDirection getDirection(const IloNumVar var)
```

Returns the solution value of `var` in the incumbent solution at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloNum getDownPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloNum getDownPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloEnv getEnv() const
```

Returns the instance of `IloEnv` passed to the constructor of the goal.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats,
                           const IloIntVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array `stats`.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats,
                           const IloNumVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array `stats`.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS2 sos) const
```

This method indicates whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS1 sos) const
```

This method indicates whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloIntVar var) const
```

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloNumVar var) const
```

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public IloNum getIncumbentObjValue() const
```

This method returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far). If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloIntVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloNumVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val,
                             const IloIntVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val,
                             const IloNumVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public IloNum getLB(const IloIntVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.


```
public void getLBs(IloNumArray vals,
                  const IloIntVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public void getLBs(IloNumArray vals,
                  const IloNumVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public IloModel getModel() const
```

This method returns the model currently extracted for the instance of `IloCplex` where the invoking goal applies.

```
public IloInt getMyThreadNum() const
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads()` - 1.

```
public IloInt getNbranches() const
```

This method returns the number of branches `IloCplex` is going to create at the current node.

```
public IloInt getNcliques() const
```

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

```
public IloInt getNcols() const
```

This method returns the number of columns in the current optimization model.

```
public IloInt getNcovers() const
```

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNdisjunctiveCuts() const
```

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowCovers() const
```

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowPaths() const
```

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

```
public IloInt getNfractionalCuts() const
```

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```
public IloInt getNGUBcovers() const
```

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNimpliedBounds() const
```

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

```
public IloInt getNiterations() const
```

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

```
public IloInt getNMIRs() const
```

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

```
public IloInt getNodeCount() const
```

This method returns the number of nodes already processed in the current optimization.

```
public IloInt getNodeCountLeft() const
```

This method returns the number of nodes left to explore in the current optimization.

```
public IloInt getRowCount() const
```

This method returns the number of rows in the current optimization model.

```
public IloNum getObjCoef(const IloIntVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public IloNum getObjCoef(const IloNumVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public void getObjCoefs(IloNumArray vals,  
                        const IloIntVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
public void getObjCoefs(IloNumArray vals,  
                        const IloNumVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array **vars** into the corresponding element of the array **vals**.

```
public IloNum getObjValue() const
```

This method returns the objective value of the solution of the current node.

```
public IloNum getPriority(const IloIntVar var) const
```

Returns the branch priority used for variable **var** in the current optimization.

```
public IloNum getPriority(const IloNumVar var) const
```

Returns the branch priority used for variable **var** in the current optimization.

```
public IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint indicated by **rng** in the solution of the current node relaxation.

```
public void getSlacks(IloNumArray vals,
                    const IloRangeArray rngs) const
```

This method puts the slack value in the solution of the current node relaxation of each of the constraints in the array of ranges **rngs** into the corresponding element of the array **vals**.

```
public IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable **var** in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable **var** in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public void getUBs(IloNumArray vals,
                  const IloIntVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array **vars** into the corresponding element of the array **vals**. These bounds are likely to be different from the bounds in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public void getUBs(IloNumArray vals,
                  const IloNumVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array **vars** into the corresponding element of the array **vals**. These bounds are likely to be different from the bounds in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloInt getUserThreads() const
```

This method returns the total number of parallel threads currently running.

```
public IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloExpr expr) const
```

This method returns the value of the expression `expr` in the solution of the current node relaxation.

```
public void getValues(IloNumArray vals,
                    const IloIntVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public void getValues(IloNumArray vals,
                    const IloNumVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraintArray con)
```

This method creates a goal that when executed adds the constraints (provided in the parameter array `con`) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory management for the cuts passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself or the constraints in the array `con` passed to method `GlobalCutGoal` or the array itself is an error and must be avoided.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraint con)
```

This method creates a goal that when executed adds the constraint `con` (provided as a parameter) as global cuts to the model. These global cuts must be valid for the entire

model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory management for the cut passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself for the constraint passed to method `GlobalCutGoal` is an error and must be avoided.

```
public IloBool hasIncumbent() const
```

This method returns `IloTrue` if an integer feasible solution has been found.

```
public IloBool isIntegerFeasible() const
```

This method returns `IloTrue` if the solution of the current node is integer feasible.

```
public IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 2; the equivalent method for an SOS of type 1 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 1; the equivalent method for an SOS of type 2 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1,
                                   IloCplex::Goal goal2)
```

The static methods `OrGoal` all return a goal that creates as many branches (or, equivalently, subproblems) as there are parameters. Each of the subnodes will be initialized with the remaining goal stack of the current node. In addition, the goal parameter will be pushed on the goal stack of the corresponding subgoal. If more than six branches need to be created, instances of `OrGoal` can be combined.

```
public static IloCplex::Goal SolutionGoal(const IloIntArray vars,
                                          const IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array `vars` to the corresponding values listed in the array `vals`.

`IloCplex` will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, `IloCplex` will reject any solution that is not compatible with the branching that has been done so far.

`IloCplex` takes over memory management for arrays `vars` and `vals` passed to `SolutionGoal`. Thus `IloCplex` will call method `end` for these arrays as soon as they can be discarded. Calling `end` for the arrays passed to `SolutionGoal` is an error and must be avoided.

```
public static IloCplex::Goal SolutionGoal(const IloNumVarArray vars,  
                                         const IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array `vars` to the corresponding values listed in the array `vals`.

`IloCplex` will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, `IloCplex` will reject any solution that is not compatible with the branching that has been done so far.

`IloCplex` takes over memory management for arrays `vars` and `vals` passed to `SolutionGoal`. Thus `IloCplex` will call method `end` for these arrays as soon as they can be discarded. Calling `end` for the arrays passed to `SolutionGoal` is an error and must be avoided.

Goal::BranchType

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

BranchType{
    BranchOnVariable,
    BranchOnSOS1,
    BranchOnSOS2,
    BranchOnAny,
    UserBranch
};

```

Description `IloCplex::GoalI::BranchType` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by the method `IloCplex::GoalI::getBranchType` to tell what kind of branch `IloCplex` is about to make:

- ◆ `BranchOnVariable` indicates branching on a single variable.
- ◆ `BranchOnAny` indicates multiple bound changes and constraints will be used for branching.
- ◆ `BranchOnSOS1` indicates branching on an SOS of type 1.
- ◆ `BranchOnSOS2` indicates branching on an SOS of type 2.

See Also [IloCplex::GoalI](#)

Fields

```

BranchOnVariable
    = CPX_TYPE_VAR

BranchOnSOS1
    = CPX_TYPE_SOS1

BranchOnSOS2
    = CPX_TYPE_SOS2

BranchOnAny
    = CPX_TYPE_ANY

UserBranch

```

GoalI::IntegerFeasibility

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

IntegerFeasibility{
    ImpliedInfeasible,
    Feasible,
    Infeasible,
    ImpliedFeasible
};

```

Description The enumeration `IloCplex::GoalI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by `IloCplex::GoalI::getFeasibility` to access the integer feasibility of a variable or SOS in the current node solution:

- ◆ `Feasible` indicates the variable or SOS is integer feasible.
- ◆ `ImpliedFeasible` indicates the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- ◆ `Infeasible` indicates the variable or SOS is integer infeasible.

See Also [IloCplex](#), [GoalI::IntegerFeasibilityArray](#), [ControlCallbackI::IntegerFeasibility](#)

Fields

```

ImpliedInfeasible
Feasible
    = CPX_INTEGER_FEASIBLE

Infeasible
    = CPX_INTEGER_INFEASIBLE

ImpliedFeasible
    = CPX IMPLIED_INTEGER_FEASIBLE

```


GoalI::IntegerFeasibilityArray

Category	Inner Type Definition
Definition File	ilcplex/ilocplexi.h
Include Files	ilocplex.h
Synopsis	<code>IloArray< IntegerFeasibility > IntegerFeasibilityArray</code>
Description	This type defines an array type for <code>IloCplex::GoalI::IntegerFeasibility</code> . The fully qualified name of an integer feasibility array is <code>IloCplex::GoalI::IntegerFeasibilityArray</code> .
See Also	IloCplex , GoalI::IntegerFeasibility

IloCplex::HeuristicCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected IloCplex::CplexStatus	getCplexStatus() const
protected IloAlgorithm::Status	getStatus() const
protected IloBool	isDualFeasible() const
protected IloBool	isPrimalFeasible() const
protected void	setBounds (const IloIntVarArray var,const IloNumArray lb,const IloNumArray ub)
protected void	setBounds (const IloNumVarArray var,const IloNumArray lb,const IloNumArray ub)
protected void	setBounds (const IloIntVar var, IloNum lb, IloNum ub)
protected void	setBounds (const IloNumVar var, IloNum lb, IloNum ub)
protected void	setSolution (const IloIntVarArray vars,const IloNumArray vals, IloNum obj)
protected void	setSolution (const IloIntVarArray vars,const IloNumArray vals)
protected void	setSolution (const IloNumVarArray vars,const IloNumArray vals, IloNum obj)
protected void	setSolution (const IloNumVarArray vars,const IloNumArray vals)
protected IloBool	solve (IloCplex::Algorithm alg=Dual)

Inherited methods from [IloCplex::ControlCallbackI](#)

```
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,
ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getLB, ControlCallbackI::getLB,
ControlCallbackI::getLBs, ControlCallbackI::getLBs,
ControlCallbackI::getNodeData, ControlCallbackI::getObjValue,
ControlCallbackI::getSlack, ControlCallbackI::getSlacks,
ControlCallbackI::getUB, ControlCallbackI::getUB,
ControlCallbackI::getUBs, ControlCallbackI::getUBs,
ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,
ControlCallbackI::getValue, ControlCallbackI::getValue,
ControlCallbackI::getValue, ControlCallbackI::getValues,
ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,
ControlCallbackI::isSOSFeasible
```

Inherited methods from **IloCplex::MIPCallbackI**

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from **IloCplex::CallbackI**

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::HeuristicCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). When you derive a user-defined class of callbacks, this class offers protected methods for you to:

- ◆ give the instance of `IloCplex` a potential new incumbent solution;
- ◆ query the instance of `IloCplex` about the solution status for the current node;
- ◆ query the instance of `IloCplex` about the variable bounds at the current node;
- ◆ change bounds temporarily on a variable or group of variables at the current node;
- ◆ re-solve the problem at the node with the changed bounds;
- ◆ use all the query functions inherited from parent classes.

During branching, the heuristic callback is called after each node subproblem has been solved, including any cuts that may have been newly generated. Before branching, at the root node, the heuristic callback is also called before each round of cuts is added to the problem and re-solved.

In short, this callback allows you to attempt to construct an integer feasible solution at a node and pass it to the invoking instance of `IloCplex` to use as its new incumbent. The API supports you in finding such a solution by allowing you iteratively to change bounds of the variables and re-solve the node relaxation. Changing the bounds in the heuristic callback has no effect on the search beyond the termination of the callback.

If an attempt is made to access information not available at the node for the invoking instance of `IloCplex`, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::ControlCallbackI](#), [IloCplex::MIPCallbackI](#),
[ILOHEURISTICCALLBACK0](#)

Methods

protected `IloCplex::CplexStatus` **getCplexStatus()** const

This method returns the ILOG CPLEX status of the instance of `IloCplex` at the current node (that is, the state of the optimizer at the node) during the last call to [solve](#)

(which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloCplex::CplexStatus` lists the possible status values.

```
protected IloAlgorithm::Status getStatus() const
```

This method returns the status of the solution found by the instance of `IloCplex` at the current node during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloAlgorithm::Status` lists the possible status values.

```
protected IloBool isDualFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
protected IloBool isPrimalFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is primal feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
protected void setBounds(const IloIntVarArray var,
                        const IloNumArray lb,
                        const IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloNumVarArray var,
                        const IloNumArray lb,
                        const IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloIntVar var,
                          IloNum lb,
                          IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloNumVar var,
                          IloNum lb,
                          IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setSolution(const IloIntVarArray vars,
                          const IloNumArray vals,
                          IloNum obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses

the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

```
protected void setSolution(const IloIntVarArray vars,
                           const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

```
protected void setSolution(const IloNumVarArray vars,
                           const IloNumArray vals,
                           IloNum obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

```
protected void setSolution(const IloNumVarArray vars,
                           const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as

the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

```
protected IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method can be used to solve the current node relaxation, usually after some bounds have been changed using `setBounds`. By default it uses the dual simplex algorithm, but this can be overwritten using the optional parameter `alg`. See the enumeration `IloCplex::Algorithm` for a list of the available optimizers.

IloCplex::IISStatus

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h
Synopsis	<pre>IISStatus{ Lower, Both, Upper };</pre>
Description	The enumeration <code>IloCplex::IISStatus</code> lists values that indicate the status that variables and constraints can assume in an irreducibly inconsistent set (IIS) in <code>IloCplex</code> .
See Also	IloCplex , IloCplex::IISStatusArray
Fields	Lower = CPXIIS_AT_LOWER Both = CPXIIS_FIXED Upper = CPXIIS_AT_UPPER

IloCplex::IISStatusArray

Category	Inner Type Definition
Definition File	ilcplex/ilocplexi.h
Include Files	ilocplex.h
Synopsis	<code>IloArray< IISStatus > IISStatusArray</code>
Description	This type defines an array-type for <code>IloCplex::IISStatus</code> . The fully qualified name of a status array for a set of irreconcilibly inconsistent constraints (IIS) is <code>IloCplex::IISStatusArray</code> .
See Also	IloCplex , IloCplex::IISStatus

IloCplex::IncumbentCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected NodeData *	getNodeData() const
protected NodeId	getNodeId() const
protected IloNum	getObjValue() const
protected IloNum	getSlack(const IloRange rng) const
protected void	getSlacks(IloNumArray val,const IloRangeArray con) const
protected IloNum	getValue(const IloIntVar var) const
protected IloNum	getValue(const IloNumVar var) const
protected IloNum	getValue(const IloExprArg expr) const
protected void	getValues(IloNumArray val,const IloIntVarArray vars) const
protected void	getValues(IloNumArray val,const IloNumVarArray vars) const
protected void	reject()

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of this class represents a callback that is called whenever a new potential incumbent is found during branch & cut searches. It allows you to analyze the proposed incumbent and optionally reject it. In this case, CPLEX will continue the branch & cut search. This callback is thus typically combined with a branch callback that instructs CPLEX how to branch on a node after it has found a potential incumbent and thus considered the node solution to be integer feasible.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::MIPCallbackI](#)

Methods

```
protected NodeData * getNodeData() const
```

This method retrieves the `NodeData` object that may have previously been assigned to the current node by the user with method

`IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the current node, `0` will be returned.

```
protected NodeId getNodeId() const
```

This method returns the `NodeId` of the current node.

```
protected IloNum getObjValue() const
```

This method returns the query objective value of the potential incumbent.

```
protected IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the range indicated by `rng` for the potential incumbent.

```
protected void getSlacks(IloNumArray val,  
                        const IloRangeArray con) const
```

This method puts the slack value for each range in the array of ranges `con` into the corresponding element of the array `val` for the potential incumbent. For this CPLEX resizes array `val` to match the size of array `con`.

```
protected IloNum getValue(const IloIntVar var) const
```

This method returns the query value of the variable `var` in the potential incumbent solution.

```
protected IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the potential incumbent solution.

```
protected IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the `expr` for the potential incumbent solution.

```
protected void getValues(IloNumArray val,  
                        const IloIntVarArray vars) const
```

This method returns the query values of the variables in the array `vars` in the potential incumbent solution and copies them to `val`. CPLEX automatically resizes the array `val` to match the size of the array `vars`.

```
protected void getValues(IloNumArray val,  
                        const IloNumVarArray vars) const
```

This method returns the query values of the variables in the array `vars` in the potential incumbent solution and copies them to `val`. CPLEX automatically resizes the array `val` to match the length of the array `vars`.

```
protected void reject()
```

This method rejects the proposed incumbent.

IloCplex::IntParam

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

IntParam{
    AdvInd,
    RootAlg,
    NodeAlg,
    MIPEmphasis,
    AggFill,
    AggInd,
    BasInterval,
    ClockType,
    CraInd,
    DepInd,
    PreDual,
    PrePass,
    PreCompress,
    DPriInd,
    PriceLim,
    IISInd,
    SimDisplay,
    ItLim,
    NetFind,
    PerLim,
    PPriInd,
    ReInv,
    ScaInd,
    Threads,
    SingLim,
    RowGrowth,
    ColGrowth,
    NzGrowth,
    Reduce,
    QPNzGrowth,
    NzReadLim,
    ColReadLim,
    RowReadLim,
    QPNzReadLim,
    SiftDisplay,
    SiftAlg,
    SiftItLim,
    BrDir,
    Cliques,
    CoeRedInd,
    Covers,

```

```

MIPDisplay,
MIPInterval,
MIPThreads,
IntSolLim,
NodeFileInd,
NodeLim,
NodeSel,
VarSel,
BndStrenInd,
HeurFreq,
RINSHeur,
LBHeur,
SubMIPNodeLim,
MIPOrdType,
BBInterval,
FlowCovers,
ImplBd,
Probe,
GUBCovers,
StrongCandLim,
StrongItLim,
StrongThreadLim,
FracCand,
FracCuts,
FracPass,
PreslvNd,
FlowPaths,
MIRCuts,
DisjCuts,
AggCutLim,
CutPass,
DiveType,
BarAlg,
BarColNz,
BarDisplay,
BarItLim,
BarMaxCor,
BarOrder,
BarThreads,
BarCrossAlg,
BarStartAlg,
NetItLim,
NetPPriInd,
NetDisplay,
ExplainAlg
};

```

Description

[IloCplex](#) is the class for the CPLEX algorithms in ILOG CPLEX. The enumeration [IloCplex::IntParam](#) lists the parameters of CPLEX that require integer values. Use these values with the methods [IloCplex::getParam](#) and [IloCplex::setParam](#).

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also

[IloCplex](#)

Fields

AdvInd

= CPX_PARAM_ADVIND

RootAlg

= CPX_PARAM_LPMETHOD, CPX_PARAM_STARTALG,
CPX_PARAM_QPMETHOD

NodeAlg

= CPX_PARAM_SUBALG

MIPEmphasis

= CPX_PARAM_MIPEMPHASIS

AggFill

= CPX_PARAM_AGGFILL

AggInd

= CPX_PARAM_AGGIND

BasInterval

= CPX_PARAM_BASINTERVAL

ClockType

= CPX_PARAM_CLOCKTYPE

CraInd

= CPX_PARAM_CRAIND

DepInd

= CPX_PARAM_DEPIND

PreDual

= CPX_PARAM_PREDUAL

PrePass

= CPX_PARAM_PREPASS

PreCompress

= CPX_PARAM_PRECOMPRESS

DPriInd

= CPX_PARAM_DPRIIND

PriceLim
= CPX_PARAM_PRICELIM

IISInd
= CPX_PARAM_IISIND

SimDisplay
= CPX_PARAM_SIMDISPLAY

ItLim
= CPX_PARAM_ITLIM

NetFind
= CPX_PARAM_NETFIND

PerLim
= CPX_PARAM_PERLIM

PPriInd
= CPX_PARAM_PPRIIND

ReInv
= CPX_PARAM_REINV

ScaInd
= CPX_PARAM_SCAIND

Threads
= CPX_PARAM_THREADS

SingLim
= CPX_PARAM_SINGLIM

RowGrowth
= CPX_PARAM_ROWGROWTH

ColGrowth
= CPX_PARAM_COLGROWTH

NzGrowth
= CPX_PARAM_NZGROWTH

Reduce
= CPX_PARAM_REDUCE

QPNzGrowth
= CPX_PARAM_QPNZGROWTH

NzReadLim

```

= CPX_PARAM_NZREADLIM
ColReadLim
= CPX_PARAM_COLREADLIM
RowReadLim
= CPX_PARAM_ROWREADLIM
QPNzReadLim
= CPX_PARAM_QPNZREADLIM
SiftDisplay
= CPX_PARAM_SIFTDISPLAY
SiftAlg
= CPX_PARAM_SIFTALG
SiftItLim
= CPX_PARAM_SIFTITLIM
BrDir
= CPX_PARAM_BRDIR
Cliques
= CPX_PARAM_CLIQUES
CoeRedInd
= CPX_PARAM_COEREDIND
Covers
= CPX_PARAM_COVERS
MIPDisplay
= CPX_PARAM_MIPDISPLAY
MIPInterval
= CPX_PARAM_MIPINTERVAL
MIPThreads
= CPX_PARAM_MIPTHREADS
IntSolLim
= CPX_PARAM_INTSOLLIM
NodeFileInd
= CPX_PARAM_NODEFILEIND
NodeLim
= CPX_PARAM_NODELIM

```

NodeSel
= CPX_PARAM_NODESEL

VarSel
= CPX_PARAM_VARSEL

BndStrenInd
= CPX_PARAM_BNDSTRENIND

HeurFreq
= CPX_PARAM_HEURFREQ

RINSHeur
= CPX_PARAM_RINSHEUR

LBHeur
= CPX_PARAM_LBHEUR

SubMIPNodeLim
= CPX_PARAM_SUBMIPNODELIM

MIPOrdType
= CPX_PARAM_MIPORDTYPE

BBInterval
= CPX_PARAM_BBINTERVAL

FlowCovers
= CPX_PARAM_FLOWCOVERS

ImplBd
= CPX_PARAM_IMPLBD

Probe
= CPX_PARAM_PROBE

GUBCovers
= CPX_PARAM_GUBCOVERS

StrongCandLim
= CPX_PARAM_STRONGCANDLIM

StrongItLim
= CPX_PARAM_STRONGITLIM

StrongThreadLim
= CPX_PARAM_STRONGTHREADLIM

FracCand

= CPX_PARAM_FRACCAND

FracCuts

= CPX_PARAM_FRACCUTS

FracPass

= CPX_PARAM_FRACPASS

PreslvNd

= CPX_PARAM_PRESLVND

FlowPaths

= CPX_PARAM_FLOWPATHS

MIRCuts

= CPX_PARAM_MIRCUTS

DisjCuts

= CPX_PARAM_DISJCUTS

AggCutLim

= CPX_PARAM_AGGCUTLIM

CutPass

= CPX_PARAM_CUTPASS

DiveType

= CPX_PARAM_DIVETYPE

BarAlg

= CPX_PARAM_BARALG

BarColNz

= CPX_PARAM_BARCOLNZ

BarDisplay

= CPX_PARAM_BARDISPLAY

BarItLim

= CPX_PARAM_BARITLIM

BarMaxCor

= CPX_PARAM_BARMAXCOR

BarOrder

= CPX_PARAM_BARORDER

BarThreads

= CPX_PARAM_BARTHREADS

BarCrossAlg
= CPX_PARAM_BARCROSSALG

BarStartAlg
= CPX_PARAM_BARSTARTALG

NetItLim
= CPX_PARAM_NETITLIM

NetPPriInd
= CPX_PARAM_NETPPRIIND

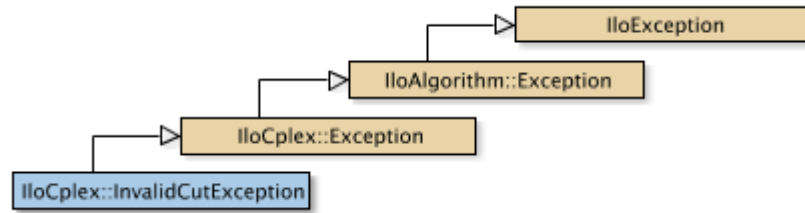
NetDisplay
= CPX_PARAM_NETDISPLAY

ExplainAlg
= CPX_PARAM_EXPLAIN_ALG

IloCplex::InvalidCutException

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Method Summary	
public IloConstraint	getCut() const

Inherited methods from IloCplex::Exception
getStatus

Inherited methods from IloException
IloException::end , IloException::getMessage

Description An instance of this exception is thrown by `IloCplex` when an attempt is made to add a malformed cut. An example of a malformed cut is one that uses variables that have not been extracted or a cut that is defined with an expression that is not linear.

Methods public [IloConstraint](#) **getCut()** const
Returns the invalid cut that triggered the invoking exception.

IloCplex::LazyConstraintCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Inherited methods from IloCplex::CutCallbackI
add , addLocal

Inherited methods from IloCplex::ControlCallbackI
ControlCallbackI::getDownPseudoCost , ControlCallbackI::getDownPseudoCost , ControlCallbackI::getFeasibilities , ControlCallbackI::getFeasibilities , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getFeasibility , ControlCallbackI::getLB , ControlCallbackI::getLB , ControlCallbackI::getLBs , ControlCallbackI::getLBs , ControlCallbackI::getNodeData , ControlCallbackI::getObjValue , ControlCallbackI::getSlack , ControlCallbackI::getSlacks , ControlCallbackI::getUB , ControlCallbackI::getUB , ControlCallbackI::getUBs , ControlCallbackI::getUBs , ControlCallbackI::getUpPseudoCost , ControlCallbackI::getUpPseudoCost , ControlCallbackI::getValue , ControlCallbackI::getValue , ControlCallbackI::getValue , ControlCallbackI::getValues , ControlCallbackI::getValues , ControlCallbackI::isSOSFeasible , ControlCallbackI::isSOSFeasible

Inherited methods from IloCplex::MIPCallbackI

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

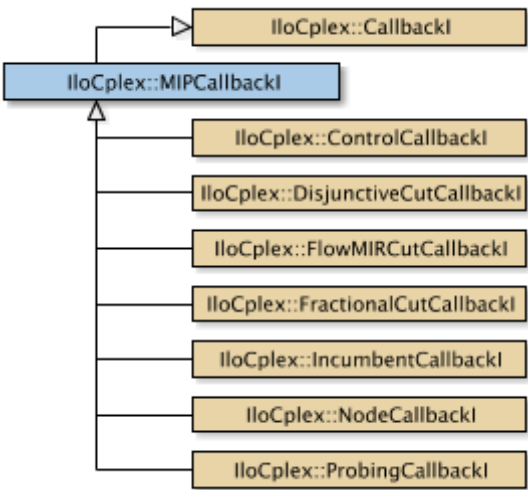
Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::LazyConstraintCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating lazy constraints. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. In fact, this callback is exactly equivalent to `IloCplex::CutCallbackI` but offers a name more consistently pointing out the difference between lazy constraints and user cuts.

IloCplex::MIPCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	MIPCallbackI()

Method Summary	
protected IloNum	MIPCallbackI::getBestObjValue() const
protected IloNum	MIPCallbackI::getCutoff() const
protected IloCplex::BranchDirection	MIPCallbackI::getDirection(const IloIntVar var) const
protected IloCplex::BranchDirection	MIPCallbackI::getDirection(const IloNumVar var) const

protected IloNum	MIPCallbackI::getIncumbentObjValue() const
protected IloNum	MIPCallbackI::getIncumbentValue(const IloIntVar var) const
protected IloNum	MIPCallbackI::getIncumbentValue(const IloNumVar var) const
protected void	MIPCallbackI::getIncumbentValues(IloNumArray val,const IloIntVarArray vars) const
protected void	MIPCallbackI::getIncumbentValues(IloNumArray val,const IloNumVarArray vars) const
protected IloInt	MIPCallbackI::getMyThreadNum() const
protected IloInt	MIPCallbackI::getNcliques() const
protected IloInt	MIPCallbackI::getNcovers() const
protected IloInt	MIPCallbackI::getNdisjunctiveCuts() const
protected IloInt	MIPCallbackI::getNflowCovers() const
protected IloInt	MIPCallbackI::getNflowPaths() const
protected IloInt	MIPCallbackI::getNfractionalCuts() const
protected IloInt	MIPCallbackI::getNGUBcovers() const
protected IloInt	MIPCallbackI::getNimpliedBounds() const
protected IloInt	MIPCallbackI::getNiterations() const
protected IloInt	MIPCallbackI::getNMIRS() const
protected IloInt	MIPCallbackI::getNnodes() const
protected IloInt	MIPCallbackI::getNremainingNodes() const
protected IloNum	MIPCallbackI::getObjCoef(const IloIntVar var) const
protected IloNum	MIPCallbackI::getObjCoef(const IloNumVar var) const
protected void	MIPCallbackI::getObjCoefs(IloNumArray val,const IloIntVarArray vars) const
protected void	MIPCallbackI::getObjCoefs(IloNumArray val,const IloNumVarArray vars) const
protected IloNum	MIPCallbackI::getPriority(const IloIntVar sos) const
protected IloNum	MIPCallbackI::getPriority(const IloNumVar sos) const
protected IloInt	MIPCallbackI::getUserThreads() const
protected IloBool	MIPCallbackI::hasIncumbent() const

Inherited methods from **IloCplex::CallbackI**

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
 CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
 CallbackI::getNrows, CallbackI::main

Inner Class	
MIPCallbackI::MIPCallbackI::NodeData	

Description

An instance of the class `IloCplex::MIPCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). `IloCplex` calls the user-written callback prior to solving each node in branch & cut search. This class offers member functions for accessing an incumbent solution and its objective value from a user-written callback. It also offers methods for accessing priority orders and statistical information such as the number of cuts. Methods are also available to query the number of generated cuts for each type of cut CPLEX generates. See the *ILOG CPLEX User's Manual* for more information about cuts.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown. For example, if there is no incumbent, the methods that query about incumbents will throw an exception.

This class also provides the common application programming interface (API) for these callback classes:

- ◆ `IloCplex::NodeCallbackI`
- ◆ `IloCplex::IncumbentCallbackI`
- ◆ `IloCplex::DisjunctiveCutCallbackI`
- ◆ `IloCplex::FractionalCutCallbackI`
- ◆ `IloCplex::ProbingCallbackI`
- ◆ `IloCplex::CutCallbackI`
- ◆ `IloCplex::BranchCallbackI`
- ◆ `IloCplex::HeuristicCallbackI`
- ◆ `IloCplex::SolveCallbackI`

See Also

`IloCplex`, `IloCplex::BranchCallbackI`, `IloCplex::Callback`,
`IloCplex::CallbackI`, `IloCplex::CutCallbackI`,
`IloCplex::DisjunctiveCutCallbackI`,
`IloCplex::FractionalCutCallbackI`,
`IloCplex::HeuristicCallbackI`, `IloCplex::IncumbentCallbackI`,

[IloCplex::NodeCallbackI](#), [IloCplex::ProbingCallbackI](#),
[IloCplex::SolveCallbackI](#), [ILOMIPCALLBACK0](#)

Constructors

protected **MIPCallbackI**()

This constructor creates a callback for use in an application that uses an instance of **IloCplex** to solve a mixed integer program (MIP).

Methods

protected [IloNum](#) **getBestObjValue**() const

This method returns the currently best known bound on the optimal solution value of the problem at the time the invoking callback is called by an instance of **IloCplex** while solving a MIP. When a model has been solved to optimality, this value matches the optimal solution value. Otherwise, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

protected [IloNum](#) **getCutoff**() const

Returns the current cutoff value.

An instance of **IloCplex** uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

protected [IloCplex::BranchDirection](#) **getDirection**(const [IloIntVar](#) var) const

Returns the **IloCplex::BranchDirection** used for variable **var** in the current optimization.

protected [IloCplex::BranchDirection](#) **getDirection**(const [IloNumVar](#) var) const

Returns the **IloCplex::BranchDirection** used for variable **var** in the current optimization.

protected [IloNum](#) **getIncumbentObjValue**() const

Returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far) at the time the invoking callback is called by an instance of **IloCplex** while solving a MIP. If there is no incumbent, this method throws an exception.

protected [IloNum](#) **getIncumbentValue**(const [IloIntVar](#) var) const

Returns the solution value of **var** in the incumbent solution at the time the invoking callback is called by an instance of **IloCplex** while solving a MIP. If there is no incumbent, this method throws an exception.

protected [IloNum](#) **getIncumbentValue**(const [IloNumVar](#) var) const

Returns the solution value of `var` in the incumbent solution at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
protected void getIncumbentValues(IloNumArray val,
                                   const IloIntVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
protected void getIncumbentValues(IloNumArray val,
                                   const IloNumVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
protected IloInt getMyThreadNum() const
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads()` - 1.

```
protected IloInt getNcliques() const
```

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNcovers() const
```

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNdisjunctiveCuts() const
```

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNflowCovers() const
```

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNflowPaths() const
```

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNfractionalCuts() const
```

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNGUBcovers() const
```

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNimpliedBounds() const
```

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNiterations() const
```

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

```
protected IloInt getNMIRs() const
```

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

```
protected IloInt getNodeNodes() const
```

Returns the number of nodes already processed in the current optimization.

```
protected IloInt getNodeRemainingNodes() const
```

Returns the number of nodes left to explore in the current optimization.

```
protected IloNum getObjectCoef(const IloIntVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
protected IloNum getObjectCoef(const IloNumVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
protected void getObjectCoefs(IloNumArray val,  
                               const IloIntVarArray vars) const
```

Puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
protected void getObjectCoefs(IloNumArray val,  
                               const IloNumVarArray vars) const
```

Puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
protected IloNum getPriority(const IloIntVar sos) const
```

Returns the branch priority used for variable `var` in the current optimization.

```
protected IloNum getPriority(const IloNumVar sos) const
```

Returns the branch priority used for variable `var` in the current optimization.

```
protected IloInt getUserThreads() const
```

Returns the total number of parallel threads currently running.

```
protected IloBool hasIncumbent() const
```

Returns `IloTrue` if an integer feasible solution has been found, or, equivalently, if an incumbent solution is available at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP.

MIPCallbackI::NodeData

Category Inner Class

InheritancePath

MIPCallbackI::NodeData

Definition File ilcplex/ilocplexi.h

Include Files ilocplex.h

Method Summary

public virtual IloAny	getDataType() const

Description Objects of (a subclass of) this class can be attached to created nodes in a branch callback with one of the `IloCplex::BranchCallbackI::makeBranch` methods. This allows the user to associate arbitrary data with the nodes. The destructor must be implemented to delete all such data. It will typically be called by `IloCplex` when a node is discarded, either because it has been processed or because it is pruned.

See Also [IloCplex::MIPCallbackI](#), [ILOBRANCHCALLBACK0](#)

Methods public virtual [IloAny](#) `getDataType()` const

`IloCplex` does not use this method. It is provided as a convenience for the user to help manage different types of `NodeData` subclasses.

IloCplex::MIPEmphasisType

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

MIPEmphasisType{
    MIPEmphasisBalanced,
    MIPEmphasisOptimality,
    MIPEmphasisFeasibility,
    MIPEmphasisBestBound,
    MIPEmphasisHiddenFeas
};

```

Description The enumeration `IloCplex::MIPEmphasisType` lists the values that the MIP emphasis parameter `IloCplex::MIPEmphasis` can assume in an instance of `IloCplex` for use when it is solving MIP problems. Use these values with the method `IloCplex::setParam(IloCplex::MIPEmphasis, value)` when you set MIP emphasis.

With the default setting of `IloCplex::MIPEmphasisBalance`, `IloCplex` tries to compute the branch & cut algorithm in such a way as to find a proven optimal solution quickly. For a discussion about various settings, refer to the *ILOG CPLEX User's Manual*.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

```

MIPEmphasisBalanced
    = CPX_MIPEMPHASIS_BALANCED

MIPEmphasisOptimality
    = CPX_MIPEMPHASIS_OPTIMALITY

MIPEmphasisFeasibility
    = CPX_MIPEMPHASIS_FEASIBILITY

MIPEmphasisBestBound
    = CPX_MIPEMPHASIS_BESTBOUND

MIPEmphasisHiddenFeas

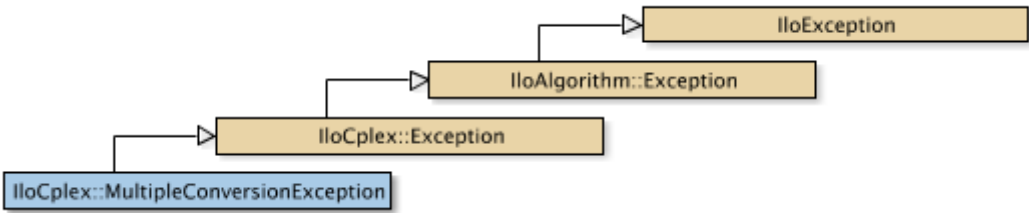
```

= CPX_MIPEMPHASIS_HIDDENFEAS

IloCplex::MultipleConversionException

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Method Summary	
public IloConversion	getConversion() const
public const IloNumVarArray	getVariables() const

Inherited methods from IloCplex::Exception
getStatus

Inherited methods from IloException
IloException::end, IloException::getMessage

Description An instance of this exception is thrown by IloCplex when there is an attempt to convert the type of a variable with more than one IloConversion object at a time, while it is being extracted by IloCplex.

Methods

```
public IloConversion getConversion() const
```

This method returns the offending IloConversion object.

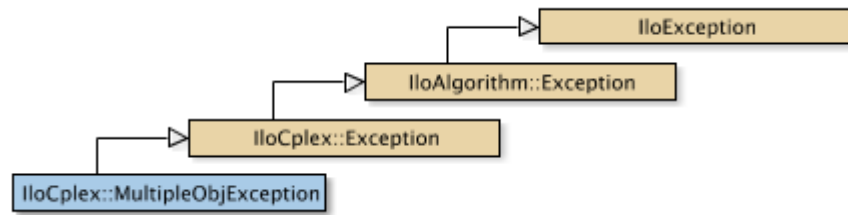
```
public const IloNumVarArray getVariables() const
```

This method returns an array of variables to which too many type conversions have been applied.

IloCplex::MultipleObjException

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Inherited methods from [IloCplex::Exception](#)

[getStatus](#)

Inherited methods from [IloException](#)

[IloException::end](#), [IloException::getMessage](#)

Description An instance of this exception is thrown by `IloCplex` when there is an attempt to use more than one objective function in a model extracted by `IloCplex`.

IloCplex::NetworkCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	NetworkCallbackI()

Method Summary	
protected IloNum	getInfeasibility() const
protected IloInt	getNiterations() const
protected IloNum	getObjValue() const
protected IloBool	isFeasible() const

Inherited methods from IloCplex::CallbackI
CallbackI::abort , CallbackI::duplicateCallback , CallbackI::getEnv , CallbackI::getModel , CallbackI::getNcols , CallbackI::getNQC s, CallbackI::getNrows , CallbackI::main

Description An instance of the class `IloCplex::NetCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` with the network optimizer. The callback is executed each time the network optimizer issues a log file message.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [ILONETWORKCALLBACK0](#)

Constructors

protected **NetworkCallbackI()**

This constructor creates a callback for use with the network optimizer.

Methods

protected **IloNum getInfeasibility()** const

This method returns the current primal infeasibility measure of the network solution in the instance of `IloCplex` at the time the invoking callback is executed.

protected **IloInt getNiterations()** const

This method returns the number of network simplex iterations completed so far by an instance of `IloCplex` at the invoking callback is executed.

protected **IloNum getObjValue()** const

This method returns the current objective value of the network solution in the instance of `IloCplex` at the time the invoking callback is executed.

protected **IloBool isFeasible()** const

This method returns `IloTrue` if the current network solution is primal feasible.

IloCplex::NodeCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	NodeCallbackI()

Method Summary

protected IloNumVar	getBranchVar (NodeId nodeid) const
protected IloNumVar	getBranchVar (int node) const
protected IloInt	getDepth (NodeId nodeid) const
protected IloInt	getDepth (int node) const
protected IloNum	getEstimatedObjValue (NodeId nodeid) const
protected IloNum	getEstimatedObjValue (int node) const
protected IloNum	getInfeasibilitySum (NodeId nodeid) const
protected IloNum	getInfeasibilitySum (int node) const
protected IloInt	getNinfeasibilities (NodeId nodeid) const
protected IloInt	getNinfeasibilities (int node) const
protected NodeData *	getNodeData (NodeId nodeid) const
protected NodeData *	getNodeData (int node) const
protected NodeId	getNodeId (int node) const
protected IloInt	getNodeNumber (NodeId nodeid) const
protected IloNum	getObjValue (NodeId nodeid) const
protected IloNum	getObjValue (int node) const
protected void	selectNode (NodeId nodeid)
protected void	selectNode (int node)

Inherited methods from [IloCplex::MIPCallbackI](#)


```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class **IloCplex::NodeCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer programming problem (a MIP). The methods of this class enable you (from a user-derived callback class) to query the instance of **IloCplex** about the next node that it plans to select in the branch & cut search, and optionally to override this selection by specifying a different node.

When an instance of this callback executes, the invoking instance of **IloCplex** still has $n = \text{getNremainingNodes}$ (inherited from **IloCplex::MIPCallbackI**) nodes left to process. These remaining nodes are numbered from 0 (zero) to $(n - 1)$. For that reason, the same node may have a different number each time an instance of **NodeCallbackI** is called. To identify a node uniquely, an instance of **IloCplex** also assigns a unique **NodeId** to each node. That unique identifier remains unchanged

throughout the search. The method `getNodeId(int i)` allows you to access the `NodeId` for each of the remaining nodes (0 to n-1). Such a query allows you to associate data with individual nodes.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also [IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [IloCplex::MIPCallbackI](#), [ILONODECALLBACK0](#)

Constructors `protected NodeCallbackI()`

This constructor creates a callback for use in an application with user-defined node selection inquiry during branch & cut searches.

Methods `protected IloNumVar getBranchVar(NodeId nodeid) const`

This method returns the variable that was branched on last when CPLEX created the node with the identifier `nodeid`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

`protected IloNumVar getBranchVar(int node) const`

Returns the variable that was branched on last when creating the node indicated by the index number `node`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

`protected IloInt getDepth(NodeId nodeid) const`

This method returns the depth of the node in the search tree for the node with the identifier `nodeid`. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

`protected IloInt getDepth(int node) const`

This method returns the depth of the node in the search tree. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

`protected IloNum getEstimatedObjValue(NodeId nodeid) const`

This method returns the estimated objective value of the node with the identifier `node`.

`protected IloNum getEstimatedObjValue(int node) const`

This method returns the estimated objective value of the node indicated by the index number `node`.

```
protected IloNum getInfeasibilitySum(NodeId nodeid) const
```

This method returns the sum of infeasibility measures at the node with the identifier `nodeid`.

```
protected IloNum getInfeasibilitySum(int node) const
```

This method returns the sum of infeasibility measures at the node indicated by the index number `node`.

```
protected IloInt getNinfeasibilities(NodeId nodeid) const
```

This method returns the number of infeasibilities at the node with the identifier `nodeid`.

```
protected IloInt getNinfeasibilities(int node) const
```

This method returns the number of infeasibilities at the node indicated by the index number `node`.

```
protected NodeData * getNodeData(NodeId nodeid) const
```

This method retrieves the `NodeData` object that may have previously been assigned by the user to the node with the identifier `nodeid` with one of the methods `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the that node, 0 (zero) will be returned.

```
protected NodeData * getNodeData(int node) const
```

This method retrieves the `NodeData` object that may have previously been assigned to the node with index `node` by the user with the method `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the specified node, 0 (zero) will be returned.

```
protected NodeId getNodeId(int node) const
```

This method returns the node identifier of the node indicated by the index number `node`. During branch & cut, an instance of `IloCplex` assigns node identifiers sequentially from 0 (zero) to (`getNodeSize` - 1) as it creates nodes. Within a search, these node identifiers are unique throughout the duration of that search. However, at any point, the remaining nodes, (that is, the nodes that have not yet been processed) are stored in an array in an arbitrary order. This method returns the identifier of the node stored at position `node` in that array.

```
protected IloInt getNodeNumber(NodeId nodeid) const
```

Returns the current index number of the node indicated by the node identifier `nodeid`.

```
protected IloNum getObjValue(NodeId nodeid) const
```

This method returns the objective value of the node with the identifier `node`.

```
protected IloNum getObjValue(int node) const
```

This method returns the objective value of the node indicated by the index number `node`.

```
protected void selectNode(NodeId nodeid)
```

This method selects the node with the identifier `nodeid` and sets it as the next node to process in the branch & cut search. The invoking instance of `IloCplex` uses the specified node as the next node to process.

```
protected void selectNode(int node)
```

This method selects the node indicated by its index number `node` and sets it as the next node to process in the branch & cut search. The parameter `node` must be an integer between 0 (zero) and (`getNremainingNodes` - 1).

The invoking instance of `IloCplex` uses the specified node as the next node to process.

IloCplex::NodeEvaluator

Category Inner Class

InheritancePath

IloCplex::NodeEvaluator

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public	NodeEvaluator()
public	NodeEvaluator(IloCplex::NodeEvaluatorI * impl)
public	NodeEvaluator(const NodeEvaluator & eval)

Method Summary

public	getImpl() const
IloCplex::NodeEvaluatorI *	
public NodeEvaluator	operator=(const NodeEvaluator & eval)

Description @proof-read

Node evaluators can be used to control the node selection strategy during goal-controlled search, that is, the order in which nodes are processed during branch & cut search using `IloCplex` goals. This is the handle class for objects of type `IloCplex::NodeEvaluatorI`, which allow you to control the node selection scheme.

`IloCplex::NodeEvaluatorI` objects are reference-counted. This means that every instance of `IloCplex::NodeEvaluatorI` keeps track of how many handle objects refer to it. When this number drops to 0, the `IloCplex::NodeEvaluatorI` object is automatically deleted. As a consequence, whenever you deal with node evaluators, you must maintain a handle object rather than just a pointer to the

implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See Also [IloCplex](#), [IloCplex::NodeEvaluatorI](#)

Constructors `public NodeEvaluator()`

The empty constructor creates a new evaluator containing no pointers to an implementation object.

`public NodeEvaluator(IloCplex::NodeEvaluatorI * impl)`

This constructor creates a new evaluator with a pointer to an implementation. It increases the reference count of `impl` by one.

`public NodeEvaluator(const NodeEvaluator & eval)`

This copy constructor increments the reference count of the implementation object referenced by `eval` by one.

Methods `public IloCplex::NodeEvaluatorI * getImpl() const`

Queries the implementation object.

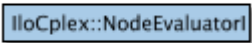
`public NodeEvaluator operator=(const NodeEvaluator & eval)`

The assignment operator increases the reference count of the implementation object of `eval`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this decrement reduces the reference count to 0 (zero), the implementation object is deleted.

IloCplex::NodeEvaluatorI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
public	NodeEvaluatorI()

Method Summary	
public virtual NodeEvaluatorI *	duplicateEvaluator()
public virtual IloNum	evaluate() const
protected IloNumVar	getBranchVar() const
protected IloNum	getDepth() const
protected IloNum	getEstimatedObjValue() const
protected IloNum	getInfeasibilitySum() const
protected IloInt	getNinfeasibilities() const
protected NodeId	getNodeId() const
protected IloNum	getObjValue() const
public virtual void	init()
public virtual IloBool	subsume(IloNum evalBest, IloNum evalCurrent) const

Description `IloCplex::NodeEvaluatorI` is the base class for implementing node evaluators. Node evaluators allow you to control the node selection strategy for a subtree by assigning values to the nodes. By default, `IloCplex` selects the node with the lowest value when choosing the next node to process during branch & cut search. This behavior can be altered by overwriting method `subsume`.

To implement your own node evaluator, you need to create a subclass of `IloCplex::NodeEvaluatorI` and implement methods `evaluate` and `duplicateEvaluator`. The method `evaluate` must be implemented to compute and return a value for a given node. The protected methods of class `IloCplex::NodeEvaluatorI` can be called to query information about the node in order to compute this value. Each node is evaluated only once, after which the value is attached to the node until the node is processed or pruned.

The `duplicateEvaluator` method is called by `IloCplex` when a copy of the evaluator must be created for use in parallel branch & cut search. Thus the implementation must simply create and return a copy of the evaluator itself—calling the copy constructor will work in most circumstances.

Node evaluators are applied to a search defined by a goal with the method `IloCplex::Apply`. The node selection strategy will be applied only to the subtree defined by the goal passed to `Apply`. Using `IloCplex::Apply`, you can assign different node selection strategies to different subtrees. You can also assign multiple node selection strategies to subtrees. In this case, node selection strategies applied first have precedence over those assigned later.

If no node evaluators are added, `IloCplex` uses the node selection strategy as controlled by the `NodeSel` parameter.

See Also

[IloCplex](#), [IloCplex::NodeEvaluator](#)

Constructors

```
public NodeEvaluatorI()
```

This constructor creates a node selector for use in an application with a user-defined node selection strategy to solve a MIP.

Methods

```
public virtual NodeEvaluatorI * duplicateEvaluator()
```

This method must be implemented by the user to return a copy of the invoking object. It is called internally to duplicate the current node evaluator for parallel branch & cut search. This method is not called for a particular node, so the `get` methods cannot be used.

```
public virtual IloNum evaluate() const
```

This method must be implemented by the user to return a value for a given node. When this method is called, the node evaluator is initialized to the node for which to compute the value. Information about this node can be obtained by the `get` methods of `IloCplex::NodeEvaluatorI`. Returning `IloInfinity` instructs `IloCplex` to discard the node being evaluated.

```
protected IloNumVar getBranchVar() const
```

This method returns the variable that `IloCplex` branched on when creating the node being evaluated from its parent. If the node has been generated with a more complex

branch, 0 (zero) will be returned instead. This method can be called only from the methods `init` and `evaluate`.

```
protected IloNum getDepth() const
```

This method returns the depth in the search tree of the node currently being evaluated. The root node is depth 0 (zero); the depth of the current node is its distance from the root, or equivalently, the number of branches taken to get from the root node to the current node. This member function can be called only from the methods `init` and `evaluate`.

```
protected IloNum getEstimatedObjValue() const
```

This method returns the estimated objective value for the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected IloNum getInfeasibilitySum() const
```

This method returns the sum of infeasibility measures at the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected IloInt getNinfeasibilities() const
```

This method returns the number of infeasibilities at the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected NodeId getNodeId() const
```

This method returns the node identifier of the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected IloNum getObjValue() const
```

This method returns the objective value of the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
public virtual void init()
```

This method is called by `IloCplex` immediately before the first time `evaluate` is called for a node, allowing you to initialize the evaluator based on that node. Information about the current node can be queried by calling the `get` methods of `IloCplex::NodeEvaluatorI`.

```
public virtual IloBool subsume(IloNum evalBest,
                               IloNum evalCurrent) const
```

`IloCplex` maintains a candidate node for selection as the next node to process. When choosing the next node, it compares the candidate to all other nodes. If a given node and the candidate node are governed by the same evaluator, `IloCplex` calls `subsume` to determine whether the node should become the new candidate. The arguments passed to the `subsume` call are:

- ◆ the value previously assigned by the method `evaluate` to the candidate node as parameter `evalBest`, and

- ◆ the value previously assigned by the method `evaluate` to the node under investigation as parameter `evalCurrent`.

By default, this method returns `IloTrue` if `evalCurrent > evalBest`.
Overwriting this function allows you to change this selection scheme.

IloCplex::NodeSelect

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

NodeSelect{
    DFS,
    BestBound,
    BestEst,
    BestEstAlt
};

```

Description The enumeration `IloCplex::NodeSelect` lists values that the parameter `IloCplex::NodeSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::NodeSel, value)`.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

```

DFS
    = CPX_NODESEL_DFS

BestBound
    = CPX_NODESEL_BESTBOUND

BestEst
    = CPX_NODESEL_BESTEST

BestEstAlt
    = CPX_NODESEL_BESTEST_ALT

```

IloCplex::NumParam

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis

```

NumParam{
    EpMrk,
    EpOpt,
    EpPer,
    EpRHS,
    NetEpOpt,
    NetEpRHS,
    TiLim,
    BtTol,
    CutLo,
    CutUp,
    EpGap,
    EpInt,
    EpAGap,
    ObjDif,
    ObjLLim,
    ObjULim,
    RelObjDif,
    CutsFactor,
    TreLim,
    WorkMem,
    BarEpComp,
    BarQCPEpComp,
    BarGrowth,
    BarObjRng
};

```

Description The enumeration `IloCplex::NumParam` lists the parameters of CPLEX that require numeric values. Use these values with the member functions: [IloCplex::getParam](#) and [IloCplex::setParam](#).

See the reference manual *ILOG CPLEX Parameters* information about these parameters. Also see the *ILOG CPLEX User's Manual* for more examples of their use.

See Also [IloCplex](#)

Fields

```

EpMrk
    = CPX_PARAM_EPMRK

EpOpt

```

= CPX_PARAM_EPOPT
EpPer
= CPX_PARAM_EPPER
EpRHS
= CPX_PARAM_EPRHS
NetEpOpt
= CPX_PARAM_NETEPOPT
NetEpRHS
= CPX_PARAM_NETEPRHS
TiLim
= CPX_PARAM_TILIM
BtTol
= CPX_PARAM_BTTOL
CutLo
= CPX_PARAM_CUTLO
CutUp
= CPX_PARAM_CUTUP
EpGap
= CPX_PARAM_EPGAP
EpInt
= CPX_PARAM_EPINT
EpAGap
= CPX_PARAM_EPAGAP
ObjDif
= CPX_PARAM_OBJDIF
ObjLLim
= CPX_PARAM_OBJLLIM
ObjULim
= CPX_PARAM_OBJULIM
RelObjDif
= CPX_PARAM_RELOBJDIF
CutsFactor
= CPX_PARAM_CUTSFACTOR

TreLim

= CPX_PARAM_TRELIM

WorkMem

= CPX_PARAM_WORKMEM

BarEpComp

= CPX_PARAM_BAREPCOMP

BarQCPEpComp

= CPX_PARAM_BARQCPEPCOMP

BarGrowth

= CPX_PARAM_BARGROWTH

BarObjRng

= CPX_PARAM_BAROBJRNG

IloCplex::PWLFormulation

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilocplex.h

Synopsis

```
PWLFormulation{
    LambdaSOS,
    LambdaBin,
    Disjunctive,
    Delta
};
```

Description The enumeration `PWLFormulation` defines symbolic constants that represent various ways of representing piecewise linear expressions in a MIP. Which of these ways to use is controlled by the method `setPWLFormulation`. All of these formulations have the same theoretical strength. However, in practice, the effectiveness for solving a particular model may vary dramatically, depending on the formulation being used. These are the various formulations:

- ◆ **Disjunctive**, the default, represents a tight formulation of the logical disjunction underlying the function that is being made piecewise linear.
- ◆ **LambdaSOS** represents a transformation of a piecewise linear expression in which a variable is created for each breakpoint, and a special ordered set of type 2 (SOS2) is added for those variables.
- ◆ **LambdaBin** represents a transformation of a piecewise linear expression similar to **LambdaSOS** in which, however, the SOS2 constraint is replaced by a set of linear constraints defined over additional binary variables.
- ◆ **Delta** indicates a cumulative formulation.

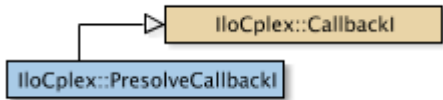
Fields

```
LambdaSOS
LambdaBin
Disjunctive
Delta
```

IloCplex::PresolveCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	PresolveCallbackI()

Method Summary	
protected IloInt	getNaggregations() const
protected IloInt	getNmodifiedCoeffs() const
protected IloInt	getNremovedCols() const
protected IloInt	getNremovedRows() const

Inherited methods from IloCplex::CallbackI
CallbackI::abort , CallbackI::duplicateCallback , CallbackI::getEnv , CallbackI::getModel , CallbackI::getNcols , CallbackI::getNQCs , CallbackI::getNrows , CallbackI::main

Description An instance of a class derivedf from PresolveCallbackI represents a user-written callback in an application that uses an instance of [IloCplex](#). The callback is called periodically during presolve. This class enables you to access information about the effects of presolve on the model extracted for the instance of [IloCplex](#). For example, there are member functions that return the number of rows or columns removed from the

model, the number of variables that have been aggregated, and the number of coefficients that have changed as a result of presolve.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also [IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [ILOPRESOLVECALLBACK0](#)

Constructors protected **PresolveCallbackI()**

This constructor creates a callback for use in presolve.

Methods protected **IloInt getNaggregations()** const

This method returns the number of aggregations performed by presolve at the time the callback is executed.

protected **IloInt getNmodifiedCoeffs()** const

This method returns the number of coefficients modified by presolve at the time the callback is executed.

protected **IloInt getNremovedCols()** const

This method returns the number of columns removed by presolve at the time the callback is executed.

protected **IloInt getNremovedRows()** const

This method returns the number of rows removed by presolve at the time the callback is executed.

IloCplex::PrimalPricing

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

PrimalPricing{
    PPriIndPartial,
    PPriIndAuto,
    PPriIndDevex,
    PPriIndSteep,
    PPriIndSteepQStart,
    PPriIndFull
};

```

Description The enumeration `IloCplex::PrimalPricing` lists values that the primal pricing parameter `IloCplex::PPriInd` can assume in `IloCplex` for use with the primal simplex algorithm. Use these values with the method [IloCplex::setParam\(IloCplex::PPriInd, value\)](#) when setting the primal pricing indicator.

See the reference manual *ILGO CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

```

PPriIndPartial
    = CPX_PPRIIND_PARTIAL

PPriIndAuto
    = CPX_PPRIIND_AUTO

PPriIndDevex
    = CPX_PPRIIND_DEVEX

PPriIndSteep
    = CPX_PPRIIND_STEEP

PPriIndSteepQStart
    = CPX_PPRIIND_STEEPQSTART

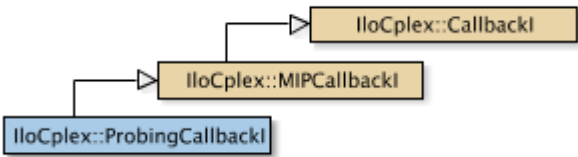
PPriIndFull
    = CPX_PPRIIND_FULL

```

IloCplex::ProbingCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	ProbingCallbackI()

Method Summary	
protected IloInt	getPhase() const
protected IloNum	getProgress() const

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

An instance of the class **IloCplex::ProbingCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of probing operation.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the **main** method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, **IloCplex::Callback**, **IloCplex::CallbackI**,
IloCplex::MIPCallbackI, **ILOPROBINGCALLBACK0**

Constructors

protected **ProbingCallbackI()**

This constructor creates a callback for use in an application when probing.

Methods

protected **IloInt getPhase()** const

This method returns the current phase of probing.

protected **IloNum getProgress()** const

This method returns the fraction of completion of the current probing phase.

IloCplex::Quality

Category	Inner Enumeration
Definition File	ilcplex/ilocplexi.h
Include Files	ilcplex/ilocplex.h

Synopsis	Quality { MaxPrimalInfeas, MaxScaledPrimalInfeas, SumPrimalInfeas, SumScaledPrimalInfeas, MaxDualInfeas, MaxScaledDualInfeas, SumDualInfeas, SumScaledDualInfeas, MaxIntInfeas, SumIntInfeas, MaxPrimalResidual, MaxScaledPrimalResidual, SumPrimalResidual, SumScaledPrimalResidual, MaxDualResidual, MaxScaledDualResidual, SumDualResidual, SumScaledDualResidual, MaxCompSlack, SumCompSlack, MaxX, MaxScaledX, MaxPi, MaxScaledPi, MaxSlack, MaxScaledSlack, MaxRedCost, MaxScaledRedCost, SumX, SumScaledX, SumPi, SumScaledPi, SumSlack, SumScaledSlack, SumRedCost, SumScaledRedCost, Kappa, Objgap, ObjGap, DualobjJ, DualObj, }

```

    PrimalObj,
    PrimalObj
};

```

Description

The enumeration `IloCplex::Quality` lists types of quality measures that can be queried for a solution with method [IloCplex::getQuality](#).

See the group `optim.cplex.solutionquality` in the *ILOG CPLEX Callable Library Reference Manual* for more information about these values. Also see the *ILOG CPLEX User's Manual* for examples of their use.

See Also

[IloCplex](#)

Fields

```

MaxPrimalInfeas
    = CPX_MAX_PRIMAL_INFEAS

MaxScaledPrimalInfeas
    = CPX_MAX_SCALED_PRIMAL_INFEAS

SumPrimalInfeas
    = CPX_SUM_PRIMAL_INFEAS

SumScaledPrimalInfeas
    = CPX_SUM_SCALED_PRIMAL_INFEAS

MaxDualInfeas
    = CPX_MAX_DUAL_INFEAS

MaxScaledDualInfeas
    = CPX_MAX_SCALED_DUAL_INFEAS

SumDualInfeas
    = CPX_SUM_DUAL_INFEAS

SumScaledDualInfeas
    = CPX_SUM_SCALED_DUAL_INFEAS

MaxIntInfeas
    = CPX_MAX_INT_INFEAS

SumIntInfeas
    = CPX_SUM_INT_INFEAS

MaxPrimalResidual
    = CPX_MAX_PRIMAL_RESIDUAL

MaxScaledPrimalResidual
    = CPX_MAX_SCALED_PRIMAL_RESIDUAL

```

SumPrimalResidual
= CPX_SUM_PRIMAL_RESIDUAL

SumScaledPrimalResidual
= CPX_SUM_SCALED_PRIMAL_RESIDUAL

MaxDualResidual
= CPX_MAX_DUAL_RESIDUAL

MaxScaledDualResidual
= CPX_MAX_SCALED_DUAL_RESIDUAL

SumDualResidual
= CPX_SUM_DUAL_RESIDUAL

SumScaledDualResidual
= CPX_SUM_SCALED_DUAL_RESIDUAL

MaxCompSlack
= CPX_MAX_COMP_SLACK

SumCompSlack
= CPX_SUM_COMP_SLACK

MaxX
= CPX_MAX_X

MaxScaledX
= CPX_MAX_SCALED_X

MaxPi
= CPX_MAX_PI

MaxScaledPi
= CPX_MAX_SCALED_PI

MaxSlack
= CPX_MAX_SLACK

MaxScaledSlack
= CPX_MAX_SCALED_SLACK

MaxRedCost
= CPX_MAX_RED_COST

MaxScaledRedCost
= CPX_MAX_SCALED_RED_COST

SumX

= CPX_SUM_X

SumScaledX
= CPX_SUM_SCALED_X

SumPi
= CPX_SUM_PI

SumScaledPi
= CPX_SUM_SCALED_PI

SumSlack
= CPX_SUM_SLACK

SumScaledSlack
= CPX_SUM_SCALED_SLACK

SumRedCost
= CPX_SUM_RED_COST

SumScaledRedCost
= CPX_SUM_SCALED_RED_COST

Kappa
= CPX_KAPPA

Objgap
DEPRECATED use ObjGap instead

ObjGap
= CPX_OBJ_GAP

DualobjJ
DEPRECATED use DualObj instead

DualObj
= CPX_DUAL_OBJ

Primalobj
DEPRECATED use PrimalObj instead

PrimalObj
= CPX_PRIMAL_OBJ

IloCplex::SearchLimit

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
public	SearchLimit()
public	SearchLimit(IloCplex::SearchLimitI * impl)
public	SearchLimit(const SearchLimit & limit)

Method Summary	
public IloCplex::SearchLimitI *	getImpl() const
public SearchLimit	operator=(const SearchLimit & limit)

Description Search limits can be used to impose limits on the exploration of certain subtrees during branch & cut search. Search limits are implemented in the class `IloCplex::SearchLimitI`. This is the handle class for CPLEX search limits.

The search limit objects are reference-counted. This means an instance of `IloCplex::SearchLimitI` keeps track of how many handle objects refer to it. If this number drops to 0, the `IloCplex::SearchLimitI` object is automatically deleted. As a consequence, whenever you deal with a search limit, you must maintain a handle object rather than only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See Also [IloCplex](#), [IloCplex::SearchLimitI](#)

Constructors public `SearchLimit()`

The default constructor creates a new search limit with 0 implementation object pointer.

```
public SearchLimit(IloCplex::SearchLimitI * impl)
```

This constructor creates a new search limit with a pointer to an implementation. It increases the reference count of `impl` by one.

```
public SearchLimit(const SearchLimit & limit)
```

This copy constructor increments the reference count of the implementation object referenced by `limit` by one.

Methods

```
public IloCplex::SearchLimitI * getImpl() const
```

Queries the implementation object of the invoking search limit.

```
public SearchLimit operator=(const SearchLimit & limit)
```

The assignment operator increases the reference count of the implementation object of `limit`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this reduces the reference count to 0, the implementation object is deleted.

IloCplex::SearchLimitI

Category Inner Class

InheritancePath

IloCplex::SearchLimitI

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public	SearchLimitI()

Method Summary

public virtual IloBool	check()
public virtual SearchLimitI *	duplicateLimit()
public virtual void	init()

Description

`IloCplex::SearchLimitI` is the base class for implementing user-defined search limits. To do so, you must subclass `IloCplex::SearchLimitI` and implement methods `check` and `duplicateLimit`. You may optionally implement method `init`. The method `check` must return `IloTrue` when the limit is reached and `IloFalse` otherwise. The method `duplicateLimit` must return a copy of the invoking object to be used in parallel search.

Whenever method `check` is called by `IloCplex`, the search limit object is first initialized to a node, referred to as the current node. Information about the current node can be queried by calling the `get` methods of class `IloCplex::SearchLimitI`.

Search limits are applied to subtrees defined by goals with method `IloCplex::LimitSearch`. For example:

```
IloGoal limitGoal = IloCplex::LimitSearch(cplex, goal1, limit);
```

creates a goal `limitGoal` which branches as specified by `goal1` until the limit specified by `limit` is reached. Only the nodes created by `goal1` (or any of the goals created by it later) are subjected to the search limit. For example, if you created two branches with the goal

```
OrGoal(limitGoal, goal2);
```

only the subtree defined by `goal1` is subject to the search limit `limit`; the subtree defined by `goal2` is not.

The ability to specify search limits for subtrees means that it is possible for certain branches to be subject to more than one search limit. Nodes with multiple search limits attached to them are processed only if none of the search limits has been reached, or, in other words, if all the search limits return `IloFalse` when method `check` is called by `IloCplex`.

Each time CPLEX uses a search limit, it is duplicated first. If you use the same instance of your limit in different branches, it will be duplicated first, the copy will be passed to the corresponding node, and `init` method will be called on the copy.

See Also

[IloCplex](#), [IloCplex::SearchLimit](#)

Constructors

```
public SearchLimitI()
```

The default constructor creates a new instance of `SearchLimitI`.

Methods

```
public virtual IloBool check()
```

This method is called for every node subjected to the invoking search limit before evaluating the node. If it returns `IloTrue`, the node is pruned, or, equivalently, the search below that node is discontinued. Thus, users implementing search limits must implement this method to return `IloTrue` if the search limit has been reached and `IloFalse` otherwise.

```
public virtual SearchLimitI * duplicateLimit()
```

This method is called internally to duplicate the current search limit. Users must implement it in a subclass to return a copy of the invoking object.

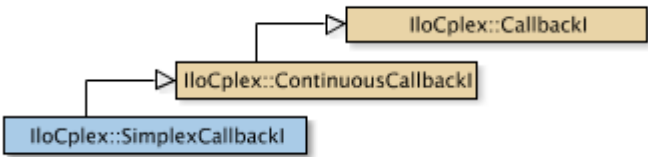
```
public virtual void init()
```

This method is called by `IloCplex` right before the first time `check` is called for a node and allows you to initialize the limit based on that node.

IloCplex::SimplexCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	SimplexCallbackI()

Inherited methods from IloCplex::ContinuousCallbackI
ContinuousCallbackI::getDualInfeasibility , ContinuousCallbackI::getInfeasibility , ContinuousCallbackI::getNiterations , ContinuousCallbackI::getObjValue , ContinuousCallbackI::isDualFeasible , ContinuousCallbackI::isFeasible

Inherited methods from IloCplex::CallbackI
CallbackI::abort , CallbackI::duplicateCallback , CallbackI::getEnv , CallbackI::getModel , CallbackI::getNcols , CallbackI::getNQCs , CallbackI::getNrows , CallbackI::main

Description An instance of the class `IloCplex::SimplexCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the simplex optimizer. For more information on the simplex optimizer, see the

ILOG CPLEX User's Manual. [IloCplex](#) calls the user-written callback after each iteration during optimization with the simplex algorithm.

The constructor of this class is protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::ContinuousCallbackI](#), [ILOSIMPLEXCALLBACK0](#)

Constructors

protected **SimplexCallbackI**()

This constructor creates a callback for use in an application of the simplex optimizer.

IloCplex::SolveCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	SolveCallbackI()

Method Summary

protected IloCplex::CplexStatus	getCplexStatus() const
protected IloAlgorithm::Status	getStatus() const
protected IloBool	isDualFeasible() const
protected IloBool	isPrimalFeasible() const
protected void	setVectors (const IloNumArray x, const IloIntVarArray var, const IloNumArray pi, const IloRangeArray rng)
protected void	setVectors (const IloNumArray x, const IloNumVarArray var, const IloNumArray pi, const IloRangeArray rng)
protected IloBool	solve (IloCplex::Algorithm alg=Dual)
protected void	useSolution()

Inherited methods from [IloCplex::ControlCallbackI](#)


```
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,
ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getLB, ControlCallbackI::getLB,
ControlCallbackI::getLBs, ControlCallbackI::getLBs,
ControlCallbackI::getNodeData, ControlCallbackI::getObjValue,
ControlCallbackI::getSlack, ControlCallbackI::getSlacks,
ControlCallbackI::getUB, ControlCallbackI::getUB,
ControlCallbackI::getUBs, ControlCallbackI::getUBs,
ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,
ControlCallbackI::getValue, ControlCallbackI::getValue,
ControlCallbackI::getValue, ControlCallbackI::getValues,
ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,
ControlCallbackI::isSOSFeasible
```

Inherited methods from **IloCplex::MIPCallbackI**

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from **IloCplex::CallbackI**

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::SolveCallbackI` can be used to solve node relaxations during branch & cut search. It allows setting a starting point for the solve or selecting the algorithm on a per-node basis.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [IloCplex::ControlCallbackI](#), [ILOSOLVECALLBACK0](#)

Constructors

protected `SolveCallbackI()`

This constructor creates a callback for use in an application for solving the node LPs during branch & cut searches.

Methods

protected `IloCplex::CplexStatus getCplexStatus() const`

This method returns the ILOG CPLEX status of the instance of `IloCplex` at the current node (that is, the state of the optimizer at the node) during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloCplex::CplexStatus` lists the possible status values.

protected `IloAlgorithm::Status getStatus() const`

This method returns the status of the solution found by the instance of `IloCplex` at the current node during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloAlgorithm::Status` lists the possible status values.

protected `IloBool isDualFeasible() const`

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the

solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
protected IloBool isPrimalFeasible() const
```

This method returns **IloTrue** if the solution provided by the last **solve** call is primal feasible. Note that an **IloFalse** return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
protected void setVectors(const IloNumArray x,  
                           const IloIntVarArray var,  
                           const IloNumArray pi,  
                           const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the **solve** method in a solve callback. Zero can be passed for any of the parameters. However, if **x** is not zero, then **var** must not be zero either. Similarly, if **pi** is not zero, then **rng** must not be zero either.

For all variables in **var**, **x[i]** specifies the starting value for the variable **var[i]**. Similarly, for all ranged constraints specified in **rng**, **pi[i]** specifies the starting dual value for **rng[i]**.

This information is exploited at the next call to **solve**, to construct a starting point for the algorithm.

```
protected void setVectors(const IloNumArray x,  
                           const IloNumVarArray var,  
                           const IloNumArray pi,  
                           const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the **solve** method in a solve callback. Zero can be passed for any of the parameters. However, if **x** is not zero, then **var** must not be zero either. Similarly, if **pi** is not zero, then **rng** must not be zero either.

For all variables in **var**, **x[i]** specifies the starting value for the variable **var[i]**. Similarly, for all ranged constraints specified in **rng**, **pi[i]** specifies the starting dual value for **rng[i]**.

This information is exploited at the next call to **solve**, to construct a starting point for the algorithm.

```
protected IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method uses the algorithm **alg** to solve the current node LP. See [**IloCplex::Algorithm**](#) for a choice of algorithms to use.

```
protected void useSolution()
```

A call to this method instructs `IloCplex` to use the solution generated with this callback.

If `useSolution` is not called, `IloCplex` uses the algorithm selected with the parameters `IloCplex::RootAlg` for the solution of the root, or `IloCplex::NodeAlg` to solve the node.

IloCplex::StringParam

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```
StringParam{  
    WorkDir  
};
```

Description The enumeration `IloCplex::StringParam` lists the parameters of CPLEX that require a character string as a value. Use these values with the methods [IloCplex::getParam](#) and [IloCplex::setParam](#).

See the reference manual *ILOG CPLEX Parameter Table* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

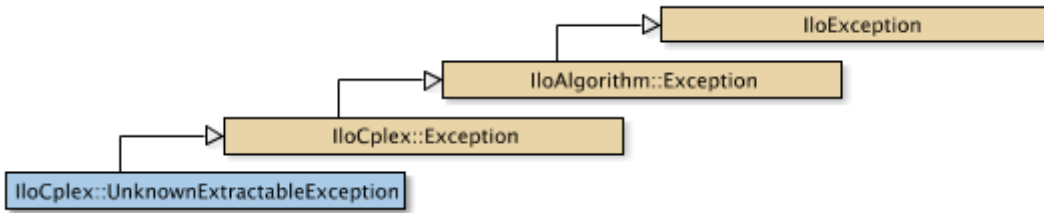
Fields

```
WorkDir  
= CPX_PARAM_WORKDIR
```

IloCplex::UnknownExtractableException

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Method Summary	
public IloExtractable	getExtractable() const

Inherited methods from IloCplex::Exception
getStatus

Inherited methods from IloException
IloException::end, IloException::getMessage

Description An instance of this exception is thrown by IloCplex when an operation is attempted using an extractable that has not been extracted.

Methods public IloExtractable **getExtractable()** const
This method returns the offending extractable object.

IloCplex::UserCutCallbackI

Category Inner Class

InheritancePath

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Inherited methods from [IloCplex::CutCallbackI](#)

[add](#), [addLocal](#)

Inherited methods from [IloCplex::ControlCallbackI](#)

[ControlCallbackI::getDownPseudoCost](#), [ControlCallbackI::getDownPseudoCost](#),
[ControlCallbackI::getFeasibilities](#), [ControlCallbackI::getFeasibilities](#),
[ControlCallbackI::getFeasibility](#), [ControlCallbackI::getFeasibility](#),
[ControlCallbackI::getFeasibility](#), [ControlCallbackI::getFeasibility](#),
[ControlCallbackI::getLB](#), [ControlCallbackI::getLB](#),
[ControlCallbackI::getLBs](#), [ControlCallbackI::getLBs](#),
[ControlCallbackI::getNodeData](#), [ControlCallbackI::getNodeData](#),
[ControlCallbackI::getSlack](#), [ControlCallbackI::getSlacks](#),
[ControlCallbackI::getUB](#), [ControlCallbackI::getUB](#),
[ControlCallbackI::getUBs](#), [ControlCallbackI::getUBs](#),
[ControlCallbackI::getUpPseudoCost](#), [ControlCallbackI::getUpPseudoCost](#),
[ControlCallbackI::getValue](#), [ControlCallbackI::getValue](#),
[ControlCallbackI::getValue](#), [ControlCallbackI::getValues](#),
[ControlCallbackI::getValues](#), [ControlCallbackI::isSOSFeasible](#),
[ControlCallbackI::isSOSFeasible](#)

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class **IloCplex::UserCutCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a MIP while generating user cuts to tighten the LP relaxation. **IloCplex** calls the user-written callback after solving each node LP exactly like **IloCplex::CutCallbackI**. It differs from **IloCplex::CutCallbackI** only in that constraints added in a **UserCutCallbackI** must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.

IloCplex::VariableSelect

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

```

VariableSelect{
    MinInfeas,
    DefaultVarSel,
    MaxInfeas,
    Pseudo,
    Strong,
    PseudoReduced
};

```

Description The enumeration `IloCplex::VariableSelect` lists values that the parameter `IloCplex::VarSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::VarSel, value)`.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also [IloCplex](#)

Fields

```

MinInfeas
    = CPX_VARSEL_MININFEAS

DefaultVarSel
    = CPX_VARSEL_DEFAULT

MaxInfeas
    = CPX_VARSEL_MAXINFEAS

Pseudo
    = CPX_VARSEL_PSEUDO

Strong
    = CPX_VARSEL_STRONG

PseudoReduced
    = CPX_VARSEL_PSEUDOREDUCED

```

Group optim.cplex.cpp.piecewiselinear

The API of ILOG CPLEX Piecewise Linear Formulations for users of C++.

Enumerations Summary	
IloCplex::PWLFormulation	

Description

Piecewise Linearity in ILOG CPLEX

Some problems are most naturally represented by constraints over functions that are not purely linear but consist of linear segments. Such functions are sometimes known as *piecewise linear*.

How to Define a Piecewise Linear Function

To define a piecewise linear function in Concert Technology, you need these components:

- ◆ the variable of the piecewise linear function;
- ◆ the breakpoints of the piecewise linear function;
- ◆ the slope of each segment (that is, the rate of increase or decrease of the function between two breakpoints);
- ◆ the geometric coordinates of at least one point of the function.

In other words, for a piecewise linear function of *n* breakpoints, you need to know *n*+1 slopes. Typically, the breakpoints of a piecewise linear function are specified as an array of numeric values. The slopes of its segments are indicated as an array of numeric values as well. The geometric coordinates of at least one point of the function must also be specified. Then in Concert Technology, those details are brought together in an instance of the class `IloPiecewiseLinear`.

Another way to specify a piecewise linear function is to give the slope of the first segment, two arrays for the coordinates of the breakpoints, and the slope of the last segment.

For examples of these ways of defining a piecewise linear function, see this topic in the *ILOG CPLEX User's Manual*.

Discontinuous Piecewise Linear Functions

Intuitively, in a continuous piecewise linear function, the endpoint of one segment has the same coordinates as the initial point of the next segment. There are piecewise linear functions, however, where two consecutive breakpoints may have the same *x*

coordinate but differ in the value of $f(x)$. Such a difference is known as a *step* in the piecewise linear function, and such a function is known as *discontinuous*.

Syntactically, a step is represented in this way:

- ◆ The value of the first point of a step in the array of slopes is the height of the step.
- ◆ The value of the second point of the step in the array of slopes is the slope of the function after the step.

By convention, a breakpoint belongs to the segment that starts at that breakpoint.

In Concert Technology, a discontinuous piecewise linear function is represented as an instance of the class `IloPiecewiseLinear` (the same class as used for continuous piecewise linear functions).

For examples of discontinuous piecewise linear functions, see this topic in the *ILOG CPLEX User's Manual*.

Using `IloPiecewiseLinear`

Whether it represents a continuous or a discontinuous piecewise linear function, an instance of `IloPiecewiseLinear` behaves like a floating-point expression. That is, you may use it in a term of a linear expression or in a constraint added to a model (an instance of `IloModel`).

IloCplex::PWLFormulation

Category Inner Enumeration

Definition File ilcplex/ilocplexi.h

Include Files ilocplex.h

Synopsis

```
PWLFormulation{
    LambdaSOS,
    LambdaBin,
    Disjunctive,
    Delta
};
```

Description The enumeration `PWLFormulation` defines symbolic constants that represent various ways of representing piecewise linear expressions in a MIP. Which of these ways to use is controlled by the method `setPWLFormulation`. All of these formulations have the same theoretical strength. However, in practice, the effectiveness for solving a particular model may vary dramatically, depending on the formulation being used. These are the various formulations:

- ◆ **Disjunctive**, the default, represents a tight formulation of the logical disjunction underlying the function that is being made piecewise linear.
- ◆ **LambdaSOS** represents a transformation of a piecewise linear expression in which a variable is created for each breakpoint, and a special ordered set of type 2 (SOS2) is added for those variables.
- ◆ **LambdaBin** represents a transformation of a piecewise linear expression similar to **LambdaSOS** in which, however, the SOS2 constraint is replaced by a set of linear constraints defined over additional binary variables.
- ◆ **Delta** indicates a cumulative formulation.

Fields

```
LambdaSOS
LambdaBin
Disjunctive
Delta
```

Group optim.cplex.cpp.logical

The API of ILOG CPLEX Logical Formulations for users of C++.

Description What Are Logical Constraints?

For ILOG CPLEX, a logical constraint combines linear constraints by means of logical operators, such as logical and, logical or, negation (that is, not), conditional statements (that is, if ... then ...) to express complex relations between linear constraints. ILOG CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of variables. Another such logical expression is the absolute value of a variable.

In C++ applications, the class `IloCplex` can extract modeling objects to solve a wide variety of MIPs and LPs. Under some conditions, a problem expressed in terms of logical constraints may be equivalent to a continuous LP, rather than a MIP. In such a case, there is no need for branching during the search for a solution. Whether a problem (or parts of a problem) represented by logical terms can be modeled and solved by LP depends on the shape of those logical terms. In this context, shape means convex or concave in the formal, mathematical sense.

For more about convexity, see that topic in the *ILOG CPLEX User's Manual*.

In fact, the class `IloCplex` can extract logical constraints as well as some logical expressions. The logical constraints that `IloCplex` can extract are these:

- ◆ `IloAnd` which can also be represented by the overloaded operator `&&`;
- ◆ `IloOr` which can also be represented by the overloaded operator `||`;
- ◆ negation as represented by the overloaded operator `!`;
- ◆ `IloIfThen`
- ◆ `==` (that is, the equivalence relation)
- ◆ `!=` (that is, the exclusive-or relation)

For examples of logical constraints in ILOG CPLEX, see the *ILOG CPLEX User's Manual*.

Group `optim.cplex.advanced.cpp`

The advanced classes and methods of the API of ILOG CPLEX for users of C++.

Classes Summary	
IloCplex::BranchCallbackI	
IloCplex::ControlCallbackI	
IloCplex::CutCallbackI	
IloCplex::Goal	
IloCplex::GoalI	

IloCplex::HeuristicCallbackI	
IloCplex::IncumbentCallbackI	
IloCplex::LazyConstraintCallbackI	
IloCplex::NodeCallbackI	
IloCplex::SolveCallbackI	
IloCplex::UserCutCallbackI	

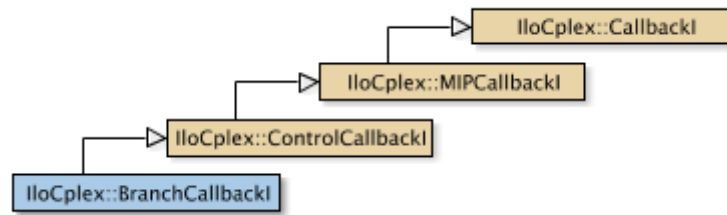
Description

These are advanced classes and methods. Advanced classes and methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other features instead.

IloCplex::BranchCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	BranchCallbackI()

Method Summary

protected IloNum	getBranch(IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const
protected BranchCallbackI::BranchType	getBranchType() const
protected IloInt	getNbranches() const
protected NodeId	getNodeId() const
protected IloBool	isIntegerFeasible() const
protected NodeId	makeBranch(const IloConstraintArray cons, const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)

protected NodeId	<code>makeBranch(const IloConstraintArray cons, const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloConstraint con, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloConstraintArray cons, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloIntVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloNumVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)</code>
protected NodeId	<code>makeBranch(const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)</code>
protected void	<code>prune()</code>

Inherited methods from **IloCplex::ControlCallbackI**

`ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,`
`ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,`
`ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,`
`ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,`
`ControlCallbackI::getLB, ControlCallbackI::getLB,`
`ControlCallbackI::getLBs, ControlCallbackI::getLBs,`
`ControlCallbackI::getNodeData, ControlCallbackI::getObjValue,`
`ControlCallbackI::getSlack, ControlCallbackI::getSlacks,`
`ControlCallbackI::getUB, ControlCallbackI::getUB,`
`ControlCallbackI::getUBs, ControlCallbackI::getUBs,`
`ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,`
`ControlCallbackI::getValue, ControlCallbackI::getValue,`
`ControlCallbackI::getValue, ControlCallbackI::getValues,`
`ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,`
`ControlCallbackI::isSOSFeasible`

Inherited methods from **IloCplex::MIPCallbackI**

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
 MIPCallbackI::getDirection, MIPCallbackI::getDirection,
 MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
 MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
 MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
 MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
 MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
 MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
 MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
 MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
 MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
 MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
 MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
 MIPCallbackI::getPriority, MIPCallbackI::getPriority,
 MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

Inherited methods from **IloCplex::CallbackI**

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
 CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
 CallbackI::getNrows, CallbackI::main

Inner Enumeration

BranchCallbackI::BranchType

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class **IloCplex::BranchCallbackI** represents a user-written callback in an application that uses an instance of **IloCplex** to solve a mixed integer program (MIP). The user-written callback is called prior to branching at a node in the

branch & cut tree during the optimization of a MIP. It allows you to query how the invoking instance of `IloCplex` is about to create subnodes at the current node and gives you the option to override the selection made by the invoking instance of `IloCplex`. You can create zero, one, or two branches.

- ◆ The method `prune` removes the current node from the search tree. No subnodes from the current node will be added to the search tree.
- ◆ The method `makeBranch` tells an instance of `IloCplex` how to create a subproblem. You may call this method zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call).
- ◆ If you call neither `IloCplex::BranchCallbackI::prune` nor `IloCplex::BranchCallbackI::makeBranch`, the instance of `IloCplex` proceeds with its own selection.
- ◆ Calling both `IloCplex::BranchCallbackI::prune` and `IloCplex::BranchCallbackI::makeBranch` in one invocation of a branch callback is an error and results in unspecified behavior.

The methods of this class are for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

`ILOBRANCHCALLBACK0`, `IloCplex::BranchDirection`,
`IloCplex::Callback`, `IloCplex::CallbackI`,
`IloCplex::MIPCallbackI`, `IloCplex::ControlCallbackI`

Constructors

protected `BranchCallbackI()`

This constructor creates a branch callback, that is, a control callback for splitting a node into two branches.

Methods

protected `IloNum getBranch(IloNumVarArray vars,
 IloNumArray bounds,
 IloCplex::BranchDirectionArray dirs,
 IloInt i) const`

This method accesses branching information for the *i*-th branch that the invoking instance of `IloCplex` is about to create. The parameter *i* must be between 0 (zero) and (`getNbranches` - 1); that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

protected `BranchCallbackI::BranchType` **getBranchType()** const

This method returns the type of branching `IloCplex` is going to do for the current node.

protected `IloInt` **getNbranches()** const

This method returns the number of branches `IloCplex` is going to create at the current node.

protected `NodeId` **getNodeId()** const

Returns the `NodeId` of the current node.

protected `IloBool` **isIntegerFeasible()** const

This method returns `IloTrue` if the solution of the current node is integer feasible.

protected `NodeId` **makeBranch**(const `IloConstraintArray` cons,
 const `IloIntVarArray` vars,
 const `IloNumArray` bounds,
 const `IloCplex::BranchDirectionArray` dirs,
 `IloNum` objestimate,
 `NodeData * data=0`)

This method offers the same facilities as the other methods

`IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

protected `NodeId` **makeBranch**(const `IloConstraintArray` cons,
 const `IloNumVarArray` vars,
 const `IloNumArray` bounds,
 const `IloCplex::BranchDirectionArray` dirs,
 `IloNum` objestimate,
 `NodeData * data=0`)

This method offers the same facilities as the other methods

`IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

```
protected NodeId makeBranch(const IloConstraint con,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method offers the same facilities for a branch specified by only one constraint as `IloCplex::BranchCallbackI::makeBranch` does for a branch specified by a set of constraints.

```
protected NodeId makeBranch(const IloConstraintArray cons,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`, by specifying a branch on constraints. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

- ◆ The parameter `cons` specifies an array of constraints that are to be added for the subnode being created.
- ◆ The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
protected NodeId makeBranch(const IloIntVar var,
                             IloNum bound,
                             IloCplex::BranchDirection dir,
                             IloNum objestimate,
                             NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
protected NodeId makeBranch(const IloNumVar var,
                             IloNum bound,
                             IloCplex::BranchDirection dir,
                             IloNum objestimate,
                             NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
protected NodeId makeBranch(const IloIntVarArray vars,
                             const IloNumArray bounds,
                             const IloCplex::BranchDirectionArray dirs,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- ◆ The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
protected NodeId makeBranch(const IloNumVarArray vars,
                             const IloNumArray bounds,
                             const IloCplex::BranchDirectionArray dirs,
                             IloNum objestimate,
                             NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

`dir[j] == IloCplex::BranchUp`

means that `bounds[j]` specifies a lower bound for `vars[j]`.

`dirs[j] == IloCplex::BranchDown`

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- ◆ The parameter `objectestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- ◆ The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

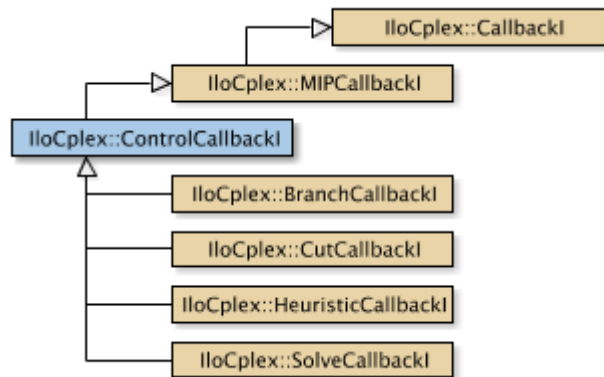
protected void **prune()**

By calling this method, you instruct the CPLEX branch & cut search not to create any child nodes from the current node, or, in other words, to discard nodes below the current node; it does not revisit the discarded nodes below the current node. In short, it creates no branches. It is an error to call both `prune` and `makeBranch` in one invocation of a callback.

IloCplex::ControlCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected IloNum	ControlCallbackI::getDownPseudoCost(const IloIntVar var) const
protected IloNum	ControlCallbackI::getDownPseudoCost(const IloNumVar var) const
protected void	ControlCallbackI::getFeasibilities(IntegerFeasibilityArray stat,const IloIntVarArray var) const
protected void	ControlCallbackI::getFeasibilities(IntegerFeasibilityArray stat,const IloNumVarArray var) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloIntVar var) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility(const IloNumVar var) const

protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility (const IloSOS2 sos) const
protected ControlCallbackI::IntegerFeasibility	ControlCallbackI::getFeasibility (const IloSOS1 sos) const
protected IloNum	ControlCallbackI::getLB (const IloIntVar var) const
protected IloNum	ControlCallbackI::getLB (const IloNumVar var) const
protected void	ControlCallbackI::getLBs (IloNumArray val,const IloIntVarArray vars) const
protected void	ControlCallbackI::getLBs (IloNumArray val,const IloNumVarArray vars) const
protected NodeData *	ControlCallbackI::getNodeData () const
protected IloNum	ControlCallbackI::getObjValue () const
protected IloNum	ControlCallbackI::getSlack (const IloRange rng) const
protected void	ControlCallbackI::getSlacks (IloNumArray val,const IloRangeArray con) const
protected IloNum	ControlCallbackI::getUB (const IloIntVar var) const
protected IloNum	ControlCallbackI::getUB (const IloNumVar var) const
protected void	ControlCallbackI::getUBs (IloNumArray val,const IloIntVarArray vars) const
protected void	ControlCallbackI::getUBs (IloNumArray val,const IloNumVarArray vars) const
protected IloNum	ControlCallbackI::getUpPseudoCost (const IloIntVar var) const
protected IloNum	ControlCallbackI::getUpPseudoCost (const IloNumVar var) const
protected IloNum	ControlCallbackI::getValue (const IloIntVar var) const
protected IloNum	ControlCallbackI::getValue (const IloNumVar var) const
protected IloNum	ControlCallbackI::getValue (const IloExprArg expr) const
protected void	ControlCallbackI::getValues (IloNumArray val,const IloIntVarArray vars) const
protected void	ControlCallbackI::getValues (IloNumArray val,const IloNumVarArray vars) const
protected IloBool	ControlCallbackI::isSOSFeasible (const IloSOS2 sos2) const
protected IloBool	ControlCallbackI::isSOSFeasible (const IloSOS1 sos1) const

Inherited methods from **IloCplex::MIPCallbackI**

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
 MIPCallbackI::getDirection, MIPCallbackI::getDirection,
 MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
 MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
 MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
 MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
 MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
 MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
 MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
 MIPCallbackI::getNiterations, MIPCallbackI::getNMIRS,
 MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
 MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
 MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
 MIPCallbackI::getPriority, MIPCallbackI::getPriority,
 MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

Inherited methods from **IloCplex::CallbackI**

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
 CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
 CallbackI::getNrows, CallbackI::main

Inner Enumeration

ControlCallbackI::IntegerFeasibility

Inner Class

ControlCallbackI::PresolvedVariableException

Inner Type Def

ControlCallbackI::IntegerFeasibilityArray	

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

This class defines the common application programming interface (API) for the following classes that allow you to control the MIP search:

- ◆ `IloCplex::SolveCallbackI`
- ◆ `IloCplex::CutCallbackI`
- ◆ `IloCplex::HeuristicCallbackI`
- ◆ `IloCplex::BranchCallbackI`

An instance of one of these classes represents a user-written callback that intervenes in the search for a solution at a given node in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). Control callbacks are tied to a node. They are called at each node during `IloCplex` branch & cut search. The user never subclasses the `IloCplex::ControlCallbackI` class directly; it only defines the common interface of those listed callbacks.

In particular, `SolveCallbackI` is called before solving the node relaxation and optionally allows substitution of its solution. `IloCplex` does this by default. After the node relaxation has been solved, either by an instance of `SolveCallbackI` or by `IloCplex`, the other control callbacks are called in the following order:

- ◆ `IloCplex::CutCallbackI`
- ◆ `IloCplex::HeuristicCallbackI`
- ◆ `IloCplex::BranchCallbackI`

If the cut callback added new cuts to the node relaxation, the node relaxation will be solved again using the solve callback, if used. The same is true if `IloCplex` generated its own cuts.

The methods of this class are protected and its constructor is private; you cannot directly subclass this class; you must derive from its subclasses.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[ControlCallbackI::IntegerFeasibility](#),
[IntegerFeasibilityArray](#), [IloCplex::MIPCallbackI](#)

Methods

protected [IloNum](#) **getDownPseudoCost**(const [IloIntVar](#) var) const

This method returns the current pseudo cost for branching downward on the variable `var`.

protected [IloNum](#) **getDownPseudoCost**(const [IloNumVar](#) var) const

This method returns the current pseudo cost for branching downward on the variable `var`.

protected void **getFeasibilities**([IntegerFeasibilityArray](#) stat,
const [IloIntVarArray](#) var) const

This method indicates whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array `stats`.

protected void **getFeasibilities**([IntegerFeasibilityArray](#) stat,
const [IloNumVarArray](#) var) const

This method indicates whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array `stats`.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloIntVar](#) var) const

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloNumVar](#) var) const

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloSOS2](#) sos) const

This method indicates whether the Special Ordered Set `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected [ControlCallbackI::IntegerFeasibility](#) **getFeasibility**(const
[IloSOS1](#) sos) const

This method indicates whether the Special Ordered Set `SOS` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
protected IloNum getLB(const IloIntVar var) const
```

This method returns the lower bound of `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected void getLBs(IloNumArray val,
                      const IloIntVarArray vars) const
```

For each element of the array `vars`, this method puts the lower bound at the current node into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected void getLBs(IloNumArray val,
                      const IloNumVarArray vars) const
```

This method puts the lower bound at the current node of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected NodeData * getNodeData() const
```

This method retrieves the `NodeData` object that may have previously been assigned to the current node by the user with the method

`IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the current node, 0 (zero) will be returned.

```
protected IloNum getObjValue() const
```

This method returns the objective value of the solution of the relaxation at the current node.

```
protected IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint indicated by `rng` in the solution of the relaxation at the current node.

```
protected void getSlacks(IloNumArray val,
                        const IloRangeArray con) const
```

For each of the constraints in the array of ranges `rngs`, this method puts the slack value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

```
protected void getUBs(IloNumArray val,
                     const IloIntVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected void getUBs(IloNumArray val,
                     const IloNumVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it

branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

```
protected IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
protected IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
protected IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
protected IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
protected IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the expression `expr` in the solution of the relaxation at the current node.

```
protected void getValues(IloNumArray val,
                        const IloIntVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected void getValues(IloNumArray val,
                        const IloNumVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
protected IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a parameter to this method may be of type 2. See the *ILOG CPLEX User's Manual* for more explanation of types of special ordered sets.

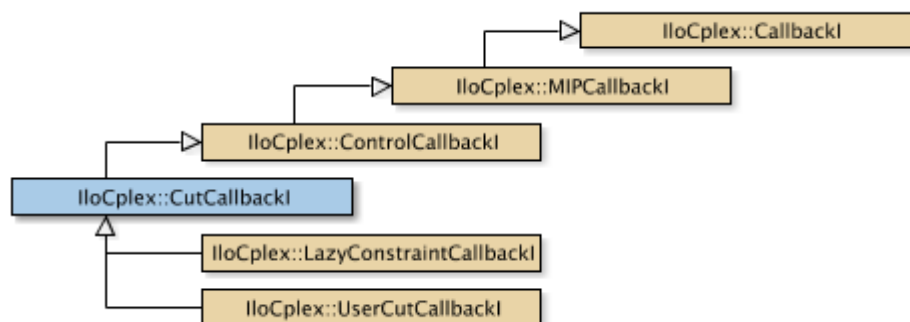
```
protected IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a parameter to this method may be of type 1. See the *ILOG CPLEX User's Manual* for more explanation about these types of special ordered sets.

IloCplex::CutCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	CutCallbackI()

Method Summary

protected	IloConstraint	add(IloConstraint con)
protected	IloConstraint	addLocal(IloConstraint con)

Inherited methods from [IloCplex::ControlCallbackI](#)

```
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,
ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getLB, ControlCallbackI::getLB,
ControlCallbackI::getLBs, ControlCallbackI::getLBs,
ControlCallbackI::getNodeData, ControlCallbackI::getObjValue,
ControlCallbackI::getSlack, ControlCallbackI::getSlacks,
ControlCallbackI::getUB, ControlCallbackI::getUB,
ControlCallbackI::getUBs, ControlCallbackI::getUBs,
ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,
ControlCallbackI::getValue, ControlCallbackI::getValue,
ControlCallbackI::getValue, ControlCallbackI::getValues,
ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,
ControlCallbackI::isSOSFeasible
```

Inherited methods from **IloCplex::MIPCallbackI**

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from **IloCplex::CallbackI**

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```


Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::CutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to add a local or global cut to the current node LP subproblem from a user-written callback. More than one cut can be added in this callback by calling the method `add` or `addLocal` multiple times. All added cuts must be linear.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

`IloCplex`, `IloCplex::Callback`, `IloCplex::CallbackI`, `IloCplex::MIPCallbackI`, `IloCplex::ControlCallbackI`, `ILOCUTCALLBACK0`

Constructors

protected `CutCallbackI()`

This constructor creates a callback for use in an application with a user-defined cut to solve a MIP.

Methods

protected `IloConstraint add(IloConstraint con)`

This method adds a cut to the current node LP subproblem for the constraint indicated by `con`. This cut must be globally valid; it will not be removed by backtracking or any other means during the search. The added cut must be linear.

protected `IloConstraint addLocal(IloConstraint con)`

This method adds a local cut to the current node LP subproblem for the constraint indicated by `con`. `IloCplex` will manage the local cut in such a way that it will be active only when processing nodes of this subtree. The added cut must be linear.

IloCplex::Goal

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
public	Goal(IloCplex::GoalBaseI * goalI)
public	Goal(const Goal & goal)
public	Goal()
public	Goal(IloConstraint cut)
public	Goal(IloConstraintArray cut)

Method Summary	
public Goal	operator=(const Goal & goal)

Description

Note:This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch & cut search in [IloCplex](#). Goals are implemented in the class [IloCplex::GoalI](#). This is the handle class for CPLEX goals.

Goal objects are reference-counted. This means every instance of [IloCplex::GoalI](#) keeps track about how many handle objects refer to it. If this number drops to 0 the

`IloCplex::GoalI` object is automatically deleted. As a consequence, whenever you deal with a goal, you must keep a handle object around rather than only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See *Goals* among the Concepts in this manual. See also goals in the *ILOG CPLEX User's Manual*.

Constructors

```
public Goal(IloCplex::GoalBaseI * goalI)
```

Creates a new goal from a pointer to the implementation object.

```
public Goal(const Goal & goal)
```

This is the copy constructor of the goal.

```
public Goal()
```

Creates a 0 goal handle, that is, a goal with a 0 implementation object pointer. This is also referred to as an empty goal.

```
public Goal(IloConstraint cut)
```

Creates a new goal that will add the constraint `cut` as a local cut to the node where the goal is executed. As a local cut, the constraint will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraint passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, the method `end` must not be called for constraints passed to this constructor explicitly.

```
public Goal(IloConstraintArray cut)
```

Creates a new goal that adds the constraints given in the array `cut` as local cuts to the node where the goal is executed. As local cuts, the constraints will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraints and the array passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, method `end` must not be called for the constraints and the array passed to this constructor explicitly.

Methods

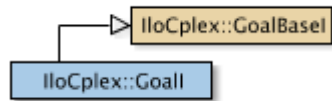
```
public Goal operator=(const Goal & goal)
```

This is the assignment operator. It increases the reference count of the implementation object of `goal`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If thereby the reference count becomes 0, the implementation object is deleted.

IloCplex::Goal

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public	GoalI (IloEnv env)

Method Summary

public void	abort ()
public static IloCplex::Goal	AndGoal (IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal	BranchAsCplexGoal (IloEnv env)
public virtual IloCplex::Goal	duplicateGoal ()
public virtual IloCplex::Goal	execute ()
public static IloCplex::Goal	FailGoal (IloEnv env)
public IloNum	getBestObjValue () const
public IloNum	getBranch (IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const
public GoalI::BranchType	getBranchType () const
public IloNum	getCutoff () const
public IloCplex::BranchDirection	getDirection (const IloIntVar var)
public IloCplex::BranchDirection	getDirection (const IloNumVar var)

public IloNum	getDownPseudoCost(const IloIntVar var) const
public IloNum	getDownPseudoCost(const IloNumVar var) const
public IloEnv	getEnv() const
public void	getFeasibilities(GoalI::IntegerFeasibilityArray stats,const IloIntVarArray vars) const
public void	getFeasibilities(GoalI::IntegerFeasibilityArray stats,const IloNumVarArray vars) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS2 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS1 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloIntVar var) const
public GoalI::IntegerFeasibility	getFeasibility(const IloNumVar var) const
public IloNum	getIncumbentObjValue() const
public IloNum	getIncumbentValue(const IloIntVar var) const
public IloNum	getIncumbentValue(const IloNumVar var) const
public void	getIncumbentValues(IloNumArray val,const IloIntVarArray vars) const
public void	getIncumbentValues(IloNumArray val,const IloNumVarArray vars) const
public IloNum	getLB(const IloIntVar var) const
public IloNum	getLB(const IloNumVar var) const
public void	getLBs(IloNumArray vals,const IloIntVarArray vars) const
public void	getLBs(IloNumArray vals,const IloNumVarArray vars) const
public IloModel	getModel() const
public IloInt	getMyThreadNum() const
public IloInt	getNbranches() const
public IloInt	getNcliques() const
public IloInt	getNcols() const
public IloInt	getNcovers() const
public IloInt	getNdisjunctiveCuts() const
public IloInt	getNflowCovers() const
public IloInt	getNflowPaths() const
public IloInt	getNfractionalCuts() const
public IloInt	getNGUBcovers() const
public IloInt	getNimpliedBounds() const
public IloInt	getNiterations() const
public IloInt	getNMIRs() const
public IloInt	getNnodes() const
public IloInt	getNremainingNodes() const

public IloInt	getNrows() const
public IloNum	getObjCoef(const IloIntVar var) const
public IloNum	getObjCoef(const IloNumVar var) const
public void	getObjCoefs(IloNumArray vals,const IloIntVarArray vars) const
public void	getObjCoefs(IloNumArray vals,const IloNumVarArray vars) const
public IloNum	getObjValue() const
public IloNum	getPriority(const IloIntVar var) const
public IloNum	getPriority(const IloNumVar var) const
public IloNum	getSlack(const IloRange rng) const
public void	getSlacks(IloNumArray vals,const IloRangeArray rngs) const
public IloNum	getUB(const IloIntVar var) const
public IloNum	getUB(const IloNumVar var) const
public void	getUBs(IloNumArray vals,const IloIntVarArray vars) const
public void	getUBs(IloNumArray vals,const IloNumVarArray vars) const
public IloNum	getUpPseudoCost(const IloIntVar var) const
public IloNum	getUpPseudoCost(const IloNumVar var) const
public IloInt	getUserThreads() const
public IloNum	getValue(const IloIntVar var) const
public IloNum	getValue(const IloNumVar var) const
public IloNum	getValue(const IloExpr expr) const
public void	getValues(IloNumArray vals,const IloIntVarArray vars) const
public void	getValues(IloNumArray vals,const IloNumVarArray vars) const
public static IloCplex::Goal	GlobalCutGoal(IloConstraintArray con)
public static IloCplex::Goal	GlobalCutGoal(IloConstraint con)
public IloBool	hasIncumbent() const
public IloBool	isIntegerFeasible() const
public IloBool	isSOSFeasible(const IloSOS2 sos2) const
public IloBool	isSOSFeasible(const IloSOS1 sos1) const
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1,IloCplex::Goal goal2)
public static IloCplex::Goal	SolutionGoal(const IloIntVarArray vars,const IloNumArray vals)
public static IloCplex::Goal	SolutionGoal(const IloNumVarArray vars,const IloNumArray vals)

Inner Enumeration	
GoalI::BranchType	
GoalI::IntegerFeasibility	

Inner Type Def	
GoalI::GoalI::IntegerFeasibilityArray	

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch & cut search in [IloCplex](#). Goals are implemented in subclasses of the class `IloCplex::GoalI`. This is the base class for user-written implementation classes of CPLEX goals.

To implement your own goal you need to create a subclass of `IloCplex::GoalI` and implement its pure virtual methods `execute` and `duplicateGoal`. You may use one of the `ILOCPLEXGOAL0` macros to assist you in doing so. After implementing your goal class, you use it by passing it to the `solve` method when solving the model.

The method `duplicateGoal` may be called by [IloCplex](#) to create copies of a goal when needed for parallel branch & cut search. Thus the implementation of this method must create and return an exact copy of the invoked object itself.

The method `execute` controls the branch & cut search of [IloCplex](#) by the goal it returns. When [IloCplex](#) processes a node, it pops the top goal from the node's goal stack and calls method `execute` of that goal. It continues executing the top goal from the stack until the node is deactivated or the goal stack is empty. If the goal stack is empty, [IloCplex](#) proceeds with the built-in search strategy for the subtree rooted at the current node.

The class `IloCplex::GoalI` provides several methods for querying information about the current node. The method `execute` controls how to proceed with the branch

& cut search via the goal it returns. The returned goal, unless it is the 0 goal, is pushed on the goal stack and will thus be executed next.

See also the chapter about goals in the *ILOG CPLEX User's Manual*.

Constructors

```
public GoalI(IloEnv env)
```

The goal constructor. It requires an instance of the same `IloEnv` as the `IloCplex` object with which to use the goal. The environment can later be queried by calling method `getEnv`.

Methods

```
public void abort()
```

Abort the optimization, that is, stop the execution of method `solve` currently in process.

```
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1,
                                     IloCplex::Goal goal2)
```

The static methods `AndGoal` all return a goal that pushes the goals passed as parameters onto the goal stack in reverse order. As a consequence, the goals will be executed in the order they are passed as parameters to the `AndGoal` function.

```
public static IloCplex::Goal BranchAsCplexGoal(IloEnv env)
```

This static function returns a goal that creates the same branches as the currently selected built-in branch strategy of `IloCplex` would choose at the current node. This goal allows you to proceed with the `IloCplex` search strategy, but keeps the search under goal control, thereby giving you the option to intervene at any point.

This goal is also important when you use node evaluators while you use a built-in branching strategy.

For example, consider the `execute` method of a goal starting like this:

```
if (!isIntegerFeasible())
    return AndGoal(BranchAsCplexGoal(getEnv()), this);
// do something
```

It would do something only when `IloCplex` found a solution it considers to be a candidate for a new incumbent. Note the test of integer feasibility before returning the `BranchAsCplexGoal`. Without it, `BranchAsCplex` would be executed for a solution `IloCplex` considers to be feasible, but `IloCplex` would not know how to branch on it. An endless loop would result.

```
public virtual IloCplex::Goal duplicateGoal()
```


This virtual method must be implemented by the user. It must return a copy of the invoking goal object. This method may be called by `IloCplex` when doing parallel branch & cut search.

```
public virtual IloCplex::Goal execute()
```

This virtual method must be implemented by the user to specify the logic of the goal. The instance of `IloCplex::Goal` returned by this method will be added to the goal stack of the node where the invoking goal is being executed for further execution.

```
public static IloCplex::Goal FailGoal(IloEnv env)
```

This static method creates a goal that fails. That means that the branch where the goal is executed will be pruned or, equivalently, the search is discontinued at that node and the node is discarded.

```
public IloNum getBestObjValue() const
```

This method returns the currently best known bound on the optimal solution value of the problem at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. When a model has been solved to optimality, this value matches the optimal solution value. Otherwise, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

```
public IloNum getBranch(IloNumVarArray vars,
                       IloNumArray bounds,
                       IloCplex::BranchDirectionArray dirs,
                       IloInt i) const
```

This method accesses branching information for the `i`-th branch that the invoking instance of `IloCplex` is about to create. The parameter `i` must be between 0 (zero) and `getNbranches - 1`; that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- ◆ The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- ◆ The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- ◆ The parameter `dirs` indicates the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

```
public GoalI::BranchType getBranchType() const
```

This method returns the type of branching `IloCplex` is going to do for the current node.

```
public IloNum getCutoff() const
```

The method returns the current cutoff value. An instance of `IloCplex` uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

```
public IloCplex::BranchDirection getDirection(const IloIntVar var)
```

Returns the solution value of `var` in the incumbent solution at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloCplex::BranchDirection getDirection(const IloNumVar var)
```

Returns the solution value of `var` in the incumbent solution at the time the invoking goal is executed by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloNum getDownPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloNum getDownPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloEnv getEnv() const
```

Returns the instance of `IloEnv` passed to the constructor of the goal.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats,
                           const IloIntVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array `stats`.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats,
                           const IloNumVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array `stats`.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS2 sos) const
```

This method indicates whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS1 sos) const
```

This method indicates whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloIntVar var) const
```

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloNumVar var) const
```

This method indicates whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public IloNum getIncumbentObjValue() const
```

This method returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far). If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloIntVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloNumVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val,
                             const IloIntVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val,
                             const IloNumVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public IloNum getLB(const IloIntVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public void getLBs(IloNumArray vals,
                  const IloIntVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public void getLBs(IloNumArray vals,
                  const IloNumVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public IloModel getModel() const
```

This method returns the model currently extracted for the instance of `IloCplex` where the invoking goal applies.

```
public IloInt getMyThreadNum() const
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads()` - 1.

```
public IloInt getNbranches() const
```

This method returns the number of branches `IloCplex` is going to create at the current node.

```
public IloInt getNcliques() const
```

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

```
public IloInt getNcols() const
```

This method returns the number of columns in the current optimization model.

```
public IloInt getNcovers() const
```

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNdisjunctiveCuts() const
```

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowCovers() const
```

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowPaths() const
```

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

```
public IloInt getNfractionalCuts() const
```

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```
public IloInt getNGUBcovers() const
```

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNimpliedBounds() const
```

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

```
public IloInt getNiterations() const
```

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

```
public IloInt getNMIRs() const
```

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

```
public IloInt getNodeNodes() const
```

This method returns the number of nodes already processed in the current optimization.

```
public IloInt getNremainingNodes() const
```

This method returns the number of nodes left to explore in the current optimization.

```
public IloInt getNrows() const
```

This method returns the number of rows in the current optimization model.

```
public IloNum getObjCoef(const IloIntVar var) const
```

Returns the linear objective coefficient for **var** in the model currently being solved.

```
public IloNum getObjCoef(const IloNumVar var) const
```

Returns the linear objective coefficient for **var** in the model currently being solved.

```
public void getObjCoefs(IloNumArray vals,  
                        const IloIntVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array **vars** into the corresponding element of the array **vals**.

```
public void getObjCoefs(IloNumArray vals,
                        const IloNumVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array **vars** into the corresponding element of the array **vals**.

```
public IloNum getObjValue() const
```

This method returns the objective value of the solution of the current node.

```
public IloNum getPriority(const IloIntVar var) const
```

Returns the branch priority used for variable **var** in the current optimization.

```
public IloNum getPriority(const IloNumVar var) const
```

Returns the branch priority used for variable **var** in the current optimization.

```
public IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint indicated by **rng** in the solution of the current node relaxation.

```
public void getSlacks(IloNumArray vals,
                      const IloRangeArray rngs) const
```

This method puts the slack value in the solution of the current node relaxation of each of the constraints in the array of ranges **rngs** into the corresponding element of the array **vals**.

```
public IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable **var** in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable **var** in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public void getUBs(IloNumArray vals,
                   const IloIntVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array **vars** into the corresponding element of the array **vals**. These bounds are likely to be different from the bounds in the original model because an instance of **IloCplex** tightens bounds when it branches from a node to its subnodes.

```
public void getUBs(IloNumArray vals,
                   const IloNumVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```
public IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloInt getUserThreads() const
```

This method returns the total number of parallel threads currently running.

```
public IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloExpr expr) const
```

This method returns the value of the expression `expr` in the solution of the current node relaxation.

```
public void getValues(IloNumArray vals,
                    const IloIntVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public void getValues(IloNumArray vals,
                    const IloNumVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraintArray con)
```

This method creates a goal that when executed adds the constraints (provided in the parameter array `con`) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory management for the cuts passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself or the constraints in the array

`con` passed to method `GlobalCutGoal` or the array itself is an error and must be avoided.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraint con)
```

This method creates a goal that when executed adds the constraint `con` (provided as a parameter) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory management for the cut passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself for the constraint passed to method `GlobalCutGoal` is an error and must be avoided.

```
public IloBool hasIncumbent() const
```

This method returns `IloTrue` if an integer feasible solution has been found.

```
public IloBool isIntegerFeasible() const
```

This method returns `IloTrue` if the solution of the current node is integer feasible.

```
public IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 2; the equivalent method for an SOS of type 1 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 1; the equivalent method for an SOS of type 2 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1,
                                   IloCplex::Goal goal2)
```

The static methods `OrGoal` all return a goal that creates as many branches (or, equivalently, subproblems) as there are parameters. Each of the subnodes will be initialized with the remaining goal stack of the current node. In addition, the goal parameter will be pushed on the goal stack of the corresponding subgoal. If more than six branches need to be created, instances of `OrGoal` can be combined.

```
public static IloCplex::Goal SolutionGoal(const IloIntArray vars,
                                          const IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array `vars` to the corresponding values listed in the array `vals`.

IloCplex will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, **IloCplex** will reject any solution that is not compatible with the branching that has been done so far.

IloCplex takes over memory management for arrays **vars** and **vals** passed to **SolutionGoal**. Thus **IloCplex** will call method **end** for these arrays as soon as they can be discarded. Calling **end** for the arrays passed to **SolutionGoal** is an error and must be avoided.

```
public static IloCplex::Goal SolutionGoal(const IloNumVarArray vars,
                                         const IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array **vars** to the corresponding values listed in the array **vals**.

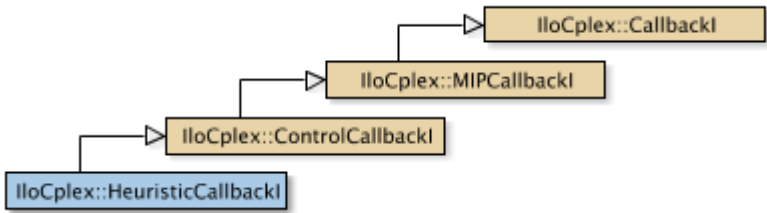
IloCplex will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, **IloCplex** will reject any solution that is not compatible with the branching that has been done so far.

IloCplex takes over memory management for arrays **vars** and **vals** passed to **SolutionGoal**. Thus **IloCplex** will call method **end** for these arrays as soon as they can be discarded. Calling **end** for the arrays passed to **SolutionGoal** is an error and must be avoided.

IloCplex::HeuristicCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected IloCplex::CplexStatus	getCplexStatus() const
protected IloAlgorithm::Status	getStatus() const
protected IloBool	isDualFeasible() const
protected IloBool	isPrimalFeasible() const
protected void	setBounds (const IloIntVarArray var,const IloNumArray lb,const IloNumArray ub)
protected void	setBounds (const IloNumVarArray var,const IloNumArray lb,const IloNumArray ub)
protected void	setBounds (const IloIntVar var, IloNum lb, IloNum ub)
protected void	setBounds (const IloNumVar var, IloNum lb, IloNum ub)
protected void	setSolution (const IloIntVarArray vars,const IloNumArray vals, IloNum obj)
protected void	setSolution (const IloIntVarArray vars,const IloNumArray vals)
protected void	setSolution (const IloNumVarArray vars,const IloNumArray vals, IloNum obj)
protected void	setSolution (const IloNumVarArray vars,const IloNumArray vals)
protected IloBool	solve (IloCplex::Algorithm alg= Dual)

Inherited methods from IloCplex::ControlCallbackI

```
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,
ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getLB, ControlCallbackI::getLB,
ControlCallbackI::getLBs, ControlCallbackI::getLBs,
ControlCallbackI::getNodeData, ControlCallbackI::getNodeData,
ControlCallbackI::getSlack, ControlCallbackI::getSlacks,
ControlCallbackI::getUB, ControlCallbackI::getUB,
ControlCallbackI::getUBs, ControlCallbackI::getUBs,
ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,
ControlCallbackI::getValue, ControlCallbackI::getValue,
ControlCallbackI::getValue, ControlCallbackI::getValues,
ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,
ControlCallbackI::isSOSFeasible
```

Inherited methods from IloCplex::MIPCallbackI

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNUGBCovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRS,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from IloCplex::CallbackI

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::HeuristicCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). When you derive a user-defined class of callbacks, this class offers protected methods for you to:

- ◆ give the instance of `IloCplex` a potential new incumbent solution;
- ◆ query the instance of `IloCplex` about the solution status for the current node;
- ◆ query the instance of `IloCplex` about the variable bounds at the current node;
- ◆ change bounds temporarily on a variable or group of variables at the current node;
- ◆ re-solve the problem at the node with the changed bounds;
- ◆ use all the query functions inherited from parent classes.

During branching, the heuristic callback is called after each node subproblem has been solved, including any cuts that may have been newly generated. Before branching, at the root node, the heuristic callback is also called before each round of cuts is added to the problem and re-solved.

In short, this callback allows you to attempt to construct an integer feasible solution at a node and pass it to the invoking instance of `IloCplex` to use as its new incumbent. The API supports you in finding such a solution by allowing you iteratively to change bounds of the variables and re-solve the node relaxation. Changing the bounds in the heuristic callback has no effect on the search beyond the termination of the callback.

If an attempt is made to access information not available at the node for the invoking instance of `IloCplex`, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [IloCplex::ControlCallbackI](#), [IloCplex::MIPCallbackI](#), [ILOHEURISTICCALLBACK0](#)

Methods

protected `IloCplex::CplexStatus` **getCplexStatus()** const

This method returns the ILOG CPLEX status of the instance of `IloCplex` at the current node (that is, the state of the optimizer at the node) during the last call to [solve](#)

(which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloCplex::CplexStatus` lists the possible status values.

```
protected IloAlgorithm::Status getStatus() const
```

This method returns the status of the solution found by the instance of `IloCplex` at the current node during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloAlgorithm::Status` lists the possible status values.

```
protected IloBool isDualFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
protected IloBool isPrimalFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is primal feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
protected void setBounds(const IloIntVarArray var,
                        const IloNumArray lb,
                        const IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloNumVarArray var,
                        const IloNumArray lb,
                        const IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloIntVar var,
                          IloNum lb,
                          IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setBounds(const IloNumVar var,
                          IloNum lb,
                          IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

Use this method only when presolve is turned off. Otherwise, you risk attempting to change the bounds of a variable that has been removed during the presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call

`IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
protected void setSolution(const IloIntArray vars,
                          const IloNumArray vals,
                          IloNum obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses

the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

```
protected void setSolution(const IloIntVarArray vars,
                           const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

```
protected void setSolution(const IloNumVarArray vars,
                           const IloNumArray vals,
                           IloNum obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

```
protected void setSolution(const IloNumVarArray vars,
                           const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as

the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

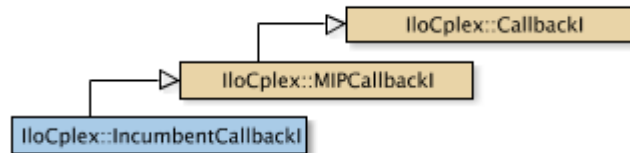
```
protected IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method can be used to solve the current node relaxation, usually after some bounds have been changed using `setBounds`. By default it uses the dual simplex algorithm, but this can be overwritten using the optional parameter `alg`. See the enumeration `IloCplex::Algorithm` for a list of the available optimizers.

IloCplex::IncumbentCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary	
protected NodeData *	getNodeData() const
protected NodeId	getNodeId() const
protected IloNum	getObjValue() const
protected IloNum	getSlack(const IloRange rng) const
protected void	getSlacks(IloNumArray val,const IloRangeArray con) const
protected IloNum	getValue(const IloIntVar var) const
protected IloNum	getValue(const IloNumVar var) const
protected IloNum	getValue(const IloExprArg expr) const
protected void	getValues(IloNumArray val,const IloIntVarArray vars) const
protected void	getValues(IloNumArray val,const IloNumVarArray vars) const
protected void	reject()

Inherited methods from [IloCplex::MIPCallbackI](#)

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from **IloCplex::CallbackI**

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of this class represents a callback that is called whenever a new potential incumbent is found during branch & cut searches. It allows you to analyze the proposed incumbent and optionally reject it. In this case, CPLEX will continue the branch & cut search. This callback is thus typically combined with a branch callback that instructs CPLEX how to branch on a node after it has found a potential incumbent and thus considered the node solution to be integer feasible.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#),
[IloCplex::MIPCallbackI](#)

Methods

```
protected NodeData * getNodeData() const
```

This method retrieves the **NodeData** object that may have previously been assigned to the current node by the user with method

IloCplex::BranchCallbackI::makeBranch. If no data object has been assigned to the current node, 0 will be returned.

```
protected NodeId getNodeId() const
```

This method returns the **NodeId** of the current node.

```
protected IloNum getObjValue() const
```

This method returns the query objective value of the potential incumbent.

```
protected IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the range indicated by **rng** for the potential incumbent.

```
protected void getSlacks(IloNumArray val,  
                        const IloRangeArray con) const
```

This method puts the slack value for each range in the array of ranges **con** into the corresponding element of the array **val** for the potential incumbent. For this CPLEX resizes array **val** to match the size of array **con**.

```
protected IloNum getValue(const IloIntVar var) const
```

This method returns the query value of the variable **var** in the potential incumbent solution.

```
protected IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable **var** in the potential incumbent solution.

```
protected IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the **expr** for the potential incumbent solution.

```
protected void getValues(IloNumArray val,  
                        const IloIntVarArray vars) const
```

This method returns the query values of the variables in the array **vars** in the potential incumbent solution and copies them to **val**. CPLEX automatically resizes the array **val** to match the size of the array **vars**.

```
protected void getValues(IloNumArray val,  
                        const IloNumVarArray vars) const
```

This method returns the query values of the variables in the array **vars** in the potential incumbent solution and copies them to **val**. CPLEX automatically resizes the array **val** to match the length of the array **vars**.

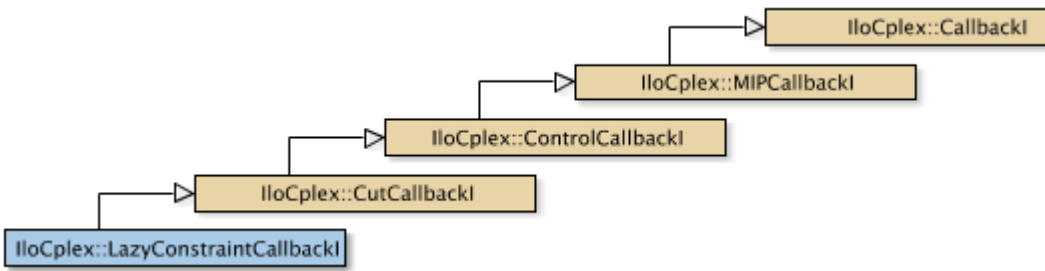
```
protected void reject()
```

This method rejects the proposed incumbent.

IloCplex::LazyConstraintCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Inherited methods from IloCplex::CutCallbackI
add, addLocal

Inherited methods from IloCplex::ControlCallbackI
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost, ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getLB, ControlCallbackI::getLB, ControlCallbackI::getLBs, ControlCallbackI::getLBs, ControlCallbackI::getNodeData, ControlCallbackI::getObjValue, ControlCallbackI::getSlack, ControlCallbackI::getSlacks, ControlCallbackI::getUB, ControlCallbackI::getUB, ControlCallbackI::getUBs, ControlCallbackI::getUBs, ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost, ControlCallbackI::getValue, ControlCallbackI::getValue, ControlCallbackI::getValue, ControlCallbackI::getValues, ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible, ControlCallbackI::isSOSFeasible

Inherited methods from IloCplex::MIPCallbackI

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRS,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from IloCplex::CallbackI

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

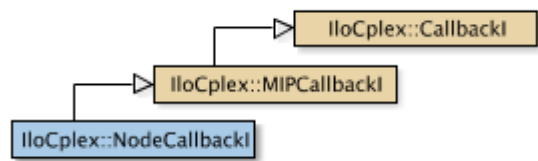
Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::LazyConstraintCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating lazy constraints. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. In fact, this callback is exactly equivalent to `IloCplex::CutCallbackI` but offers a name more consistently pointing out the difference between lazy constraints and user cuts.

IloCplex::NodeCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary	
protected	NodeCallbackI()

Method Summary	
protected IloNumVar	getBranchVar (NodeId nodeid) const
protected IloNumVar	getBranchVar (int node) const
protected IloInt	getDepth (NodeId nodeid) const
protected IloInt	getDepth (int node) const
protected IloNum	getEstimatedObjValue (NodeId nodeid) const
protected IloNum	getEstimatedObjValue (int node) const
protected IloNum	getInfeasibilitySum (NodeId nodeid) const
protected IloNum	getInfeasibilitySum (int node) const
protected IloInt	getNinfeasibilities (NodeId nodeid) const
protected IloInt	getNinfeasibilities (int node) const
protected NodeData *	getNodeData (NodeId nodeid) const
protected NodeData *	getNodeData (int node) const
protected NodeId	getNodeId (int node) const
protected IloInt	getNodeNumber (NodeId nodeid) const
protected IloNum	getObjValue (NodeId nodeid) const
protected IloNum	getObjValue (int node) const

protected void	selectNode (NodeId nodeid)
protected void	selectNode (int node)

Inherited methods from [IloCplex::MIPCallbackI](#)

[MIPCallbackI::getBestObjValue](#), [MIPCallbackI::getCutoff](#),
[MIPCallbackI::getDirection](#), [MIPCallbackI::getDirection](#),
[MIPCallbackI::getIncumbentObjValue](#), [MIPCallbackI::getIncumbentValue](#),
[MIPCallbackI::getIncumbentValue](#), [MIPCallbackI::getIncumbentValues](#),
[MIPCallbackI::getIncumbentValues](#), [MIPCallbackI::getMyThreadNum](#),
[MIPCallbackI::getNcliques](#), [MIPCallbackI::getNcovers](#),
[MIPCallbackI::getNdisjunctiveCuts](#), [MIPCallbackI::getNflowCovers](#),
[MIPCallbackI::getNflowPaths](#), [MIPCallbackI::getNfractionalCuts](#),
[MIPCallbackI::getNGUBcovers](#), [MIPCallbackI::getNimpliedBounds](#),
[MIPCallbackI::getNiterations](#), [MIPCallbackI::getNMIRs](#),
[MIPCallbackI::getNnodes](#), [MIPCallbackI::getNremainingNodes](#),
[MIPCallbackI::getObjCoef](#), [MIPCallbackI::getObjCoef](#),
[MIPCallbackI::getObjCoefs](#), [MIPCallbackI::getObjCoefs](#),
[MIPCallbackI::getPriority](#), [MIPCallbackI::getPriority](#),
[MIPCallbackI::getUserThreads](#), [MIPCallbackI::hasIncumbent](#)

Inherited methods from [IloCplex::CallbackI](#)

[CallbackI::abort](#), [CallbackI::duplicateCallback](#), [CallbackI::getEnv](#),
[CallbackI::getModel](#), [CallbackI::getNcols](#), [CallbackI::getNQCs](#),
[CallbackI::getNrows](#), [CallbackI::main](#)

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class [IloCplex::NodeCallbackI](#) represents a user-written callback in an application that uses an instance of [IloCplex](#) to solve a mixed integer programming problem (a MIP). The methods of this class enable you (from a user-derived callback class) to query the instance of [IloCplex](#) about the next node that it

plans to select in the branch & cut search, and optionally to override this selection by specifying a different node.

When an instance of this callback executes, the invoking instance of `IloCplex` still has `n = getNremainingNodes` (inherited from `IloCplex::MIPCallbackI`) nodes left to process. These remaining nodes are numbered from 0 (zero) to $(n - 1)$. For that reason, the same node may have a different number each time an instance of `NodeCallbackI` is called. To identify a node uniquely, an instance of `IloCplex` also assigns a unique `NodeId` to each node. That unique identifier remains unchanged throughout the search. The method `getNodeId(int i)` allows you to access the `NodeId` for each of the remaining nodes (0 to $n-1$). Such a query allows you to associate data with individual nodes.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [IloCplex::MIPCallbackI](#), [ILONODECALLBACK0](#)

Constructors

protected `NodeCallbackI()`

This constructor creates a callback for use in an application with user-defined node selection inquiry during branch & cut searches.

Methods

protected `IloNumVar getBranchVar(NodeId nodeid) const`

This method returns the variable that was branched on last when CPLEX created the node with the identifier `nodeid`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

protected `IloNumVar getBranchVar(int node) const`

Returns the variable that was branched on last when creating the node indicated by the index number `node`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

protected `IloInt getDepth(NodeId nodeid) const`

This method returns the depth of the node in the search tree for the node with the identifier `nodeid`. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

protected `IloInt getDepth(int node) const`

This method returns the depth of the node in the search tree. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node

plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

```
protected IloNum getEstimatedObjValue(NodeId nodeid) const
```

This method returns the estimated objective value of the node with the identifier `node`.

```
protected IloNum getEstimatedObjValue(int node) const
```

This method returns the estimated objective value of the node indicated by the index number `node`.

```
protected IloNum getInfeasibilitySum(NodeId nodeid) const
```

This method returns the sum of infeasibility measures at the node with the identifier `nodeid`.

```
protected IloNum getInfeasibilitySum(int node) const
```

This method returns the sum of infeasibility measures at the node indicated by the index number `node`.

```
protected IloInt getNinfeasibilities(NodeId nodeid) const
```

This method returns the number of infeasibilities at the node with the identifier `nodeid`.

```
protected IloInt getNinfeasibilities(int node) const
```

This method returns the number of infeasibilities at the node indicated by the index number `node`.

```
protected NodeData * getNodeData(NodeId nodeid) const
```

This method retrieves the `NodeData` object that may have previously been assigned by the user to the node with the identifier `nodeid` with one of the methods `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the that node, 0 (zero) will be returned.

```
protected NodeData * getNodeData(int node) const
```

This method retrieves the `NodeData` object that may have previously been assigned to the node with index `node` by the user with the method `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the specified node, 0 (zero) will be returned.

```
protected NodeId getNodeId(int node) const
```

This method returns the node identifier of the node indicated by the index number `node`. During branch & cut, an instance of `IloCplex` assigns node identifiers sequentially from 0 (zero) to (`getNodeSize` - 1) as it creates nodes. Within a search, these node identifiers are unique throughout the duration of that search. However, at any point, the remaining nodes, (that is, the nodes that have not yet been processed) are

stored in an array in an arbitrary order. This method returns the identifier of the node stored at position `node` in that array.

```
protected IloInt getNodeNumber(NodeId nodeid) const
```

Returns the current index number of the node indicated by the node identifier `nodeid`.

```
protected IloNum getObjValue(NodeId nodeid) const
```

This method returns the objective value of the node with the identifier `node`.

```
protected IloNum getObjValue(int node) const
```

This method returns the objective value of the node indicated by the index number `node`.

```
protected void selectNode(NodeId nodeid)
```

This method selects the node with the identifier `nodeid` and sets it as the next node to process in the branch & cut search. The invoking instance of `IloCplex` uses the specified node as the next node to process.

```
protected void selectNode(int node)
```

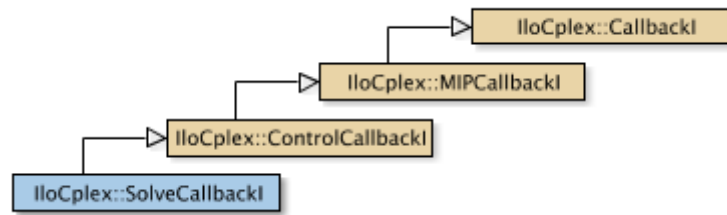
This method selects the node indicated by its index number `node` and sets it as the next node to process in the branch & cut search. The parameter `node` must be an integer between 0 (zero) and (`getNremainingNodes` - 1).

The invoking instance of `IloCplex` uses the specified node as the next node to process.

IloCplex::SolveCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

protected	SolveCallbackI()

Method Summary

protected IloCplex::CplexStatus	getCplexStatus() const
protected IloAlgorithm::Status	getStatus() const
protected IloBool	isDualFeasible() const
protected IloBool	isPrimalFeasible() const
protected void	setVectors (const IloNumArray x,const IloIntVarArray var,const IloNumArray pi,const IloRangeArray rng)
protected void	setVectors (const IloNumArray x,const IloNumVarArray var,const IloNumArray pi,const IloRangeArray rng)
protected IloBool	solve (IloCplex::Algorithm alg=Dual)
protected void	useSolution()

Inherited methods from IloCplex::ControlCallbackI

```
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost,
ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility,
ControlCallbackI::getLB, ControlCallbackI::getLB,
ControlCallbackI::getLBs, ControlCallbackI::getLBs,
ControlCallbackI::getNodeData, ControlCallbackI::getNodeData,
ControlCallbackI::getSlack, ControlCallbackI::getSlacks,
ControlCallbackI::getUB, ControlCallbackI::getUB,
ControlCallbackI::getUBs, ControlCallbackI::getUBs,
ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost,
ControlCallbackI::getValue, ControlCallbackI::getValue,
ControlCallbackI::getValue, ControlCallbackI::getValues,
ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible,
ControlCallbackI::isSOSFeasible
```

Inherited methods from IloCplex::MIPCallbackI

```
MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRS,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent
```

Inherited methods from IloCplex::CallbackI

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect or unexpected behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::SolveCallbackI` can be used to solve node relaxations during branch & cut search. It allows setting a starting point for the solve or selecting the algorithm on a per-node basis.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

[IloCplex](#), [IloCplex::Callback](#), [IloCplex::CallbackI](#), [IloCplex::ControlCallbackI](#), [ILOSOLVECALLBACK0](#)

Constructors

protected `SolveCallbackI()`

This constructor creates a callback for use in an application for solving the node LPs during branch & cut searches.

Methods

protected `IloCplex::CplexStatus getCplexStatus() const`

This method returns the ILOG CPLEX status of the instance of `IloCplex` at the current node (that is, the state of the optimizer at the node) during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloCplex::CplexStatus` lists the possible status values.

protected `IloAlgorithm::Status getStatus() const`

This method returns the status of the solution found by the instance of `IloCplex` at the current node during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloAlgorithm::Status` lists the possible status values.

protected `IloBool isDualFeasible() const`

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the

solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
protected IloBool isPrimalFeasible() const
```

This method returns **IloTrue** if the solution provided by the last **solve** call is primal feasible. Note that an **IloFalse** return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
protected void setVectors(const IloNumArray x,  
                           const IloIntVarArray var,  
                           const IloNumArray pi,  
                           const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the **solve** method in a solve callback. Zero can be passed for any of the parameters. However, if **x** is not zero, then **var** must not be zero either. Similarly, if **pi** is not zero, then **rng** must not be zero either.

For all variables in **var**, **x[i]** specifies the starting value for the variable **var[i]**. Similarly, for all ranged constraints specified in **rng**, **pi[i]** specifies the starting dual value for **rng[i]**.

This information is exploited at the next call to **solve**, to construct a starting point for the algorithm.

```
protected void setVectors(const IloNumArray x,  
                           const IloNumVarArray var,  
                           const IloNumArray pi,  
                           const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the **solve** method in a solve callback. Zero can be passed for any of the parameters. However, if **x** is not zero, then **var** must not be zero either. Similarly, if **pi** is not zero, then **rng** must not be zero either.

For all variables in **var**, **x[i]** specifies the starting value for the variable **var[i]**. Similarly, for all ranged constraints specified in **rng**, **pi[i]** specifies the starting dual value for **rng[i]**.

This information is exploited at the next call to **solve**, to construct a starting point for the algorithm.

```
protected IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method uses the algorithm **alg** to solve the current node LP. See [**IloCplex::Algorithm**](#) for a choice of algorithms to use.

```
protected void useSolution()
```

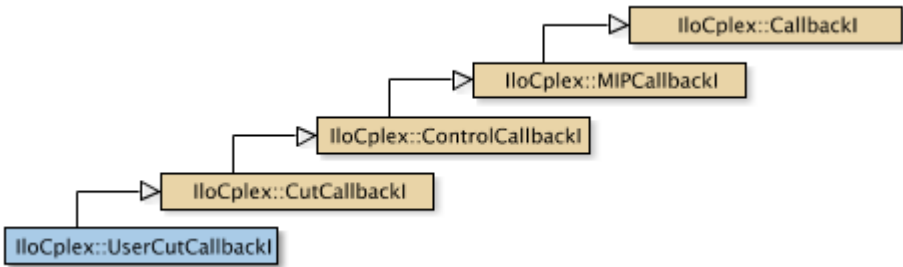
A call to this method instructs `IloCplex` to use the solution generated with this callback.

If `useSolution` is not called, `IloCplex` uses the algorithm selected with the parameters `IloCplex::RootAlg` for the solution of the root, or `IloCplex::NodeAlg` to solve the node.

IloCplex::UserCutCallbackI

Category Inner Class

InheritancePath



Definition File ilcplex/ilocplexi.h

Include Files ilcplex/ilocplex.h

Inherited methods from IloCplex::CutCallbackI
add, addLocal

Inherited methods from IloCplex::ControlCallbackI
ControlCallbackI::getDownPseudoCost, ControlCallbackI::getDownPseudoCost, ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibilities, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getFeasibility, ControlCallbackI::getLB, ControlCallbackI::getLB, ControlCallbackI::getLBs, ControlCallbackI::getLBs, ControlCallbackI::getNodeData, ControlCallbackI::getObjValue, ControlCallbackI::getSlack, ControlCallbackI::getSlacks, ControlCallbackI::getUB, ControlCallbackI::getUB, ControlCallbackI::getUBs, ControlCallbackI::getUBs, ControlCallbackI::getUpPseudoCost, ControlCallbackI::getUpPseudoCost, ControlCallbackI::getValue, ControlCallbackI::getValue, ControlCallbackI::getValue, ControlCallbackI::getValues, ControlCallbackI::getValues, ControlCallbackI::isSOSFeasible, ControlCallbackI::isSOSFeasible

Inherited methods from `IloCplex::MIPCallbackI`

```

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff,
MIPCallbackI::getDirection, MIPCallbackI::getDirection,
MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue,
MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues,
MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum,
MIPCallbackI::getNcliques, MIPCallbackI::getNcovers,
MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers,
MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts,
MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds,
MIPCallbackI::getNiterations, MIPCallbackI::getNMIRS,
MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes,
MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef,
MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs,
MIPCallbackI::getPriority, MIPCallbackI::getPriority,
MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

```

Inherited methods from `IloCplex::CallbackI`

```

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main

```

Description

Note: This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::UserCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating user cuts to tighten the LP relaxation. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. It differs from `IloCplex::CutCallbackI` only in that constraints added in a `UserCutCallbackI` must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.

Index

A

Algorithm **606**

B

BarrierCallbackI **608**
 BasisStatus **610**
 BasisStatusArray **611**
 BoolParam **612**
 BranchCallbackI **614, 767**
 BranchDirection **623**
 BranchDirectionArray **624**
 BranchType **622, 679**

C

Callback **625**
 CallbackI **627**
 CannotExtractException **56**
 CannotRemoveException **58**
 ContinuousCallbackI **633**
 ControlCallbackI **636, 775**
 CplexStatus **645**
 CrossoverCallbackI **648**
 CutCallbackI **650, 783**

D

DeleteMode **653**

DisjunctiveCutCallbackI **654**
 DualPricing **656**

E

Exception **60, 657**

F

FlowMIRCutCallbackI **659**
 FractionalCutCallbackI **661**

G

Goal **663, 786**
 GoalI **665, 788**

H

HeuristicCallbackI **682, 802**

I

IISStatus **689**
 IISStatusArray **690**
 ILO_NO_MEMORY_MANAGER **44**
 IloAbs **45**
 IloAbstraction **46**
 IloAdd **48**
 IloAlgorithm **49**

- IloAllDiff 65**
- IloAllMinDistance 68**
- IloAllNullIntersect 71**
- IloAnd 72**
- IloAny 76**
- IloAnyArray 77**
- IloAnyBinaryPredicate 79**
- ILOANYBINARYPREDICATE0 40**
- IloAnySet 81**
- IloAnySetVar 86**
- IloAnySetVarArray 90**
- IloAnyTernaryPredicate 93**
- ILOANYTERNARYPREDICATE0 41**
- IloAnyTupleSet 95**
- IloAnyTupleSetIterator 98**
- IloAnyVar 99**
- IloAnyVarArray 101**
- IloArcCos 103**
- IloArray 104**
- IloBarrier 108**
- ILOBARRIERCALLBACK0 545**
- IloBaseEnvMutex 111**
- IloBool 112**
- IloBoolAbstraction 113**
- IloBoolArray 115**
- IloBoolVar 118**
- IloBoolVarArray 121**
- IloBox 124**
- ILOBRANCHCALLBACK0 546**
- IloCard 128**
- IloCeil 129**
- IloColumnHeaderNotFoundException 469**
- IloCondition 130**
- IloConstraint 133**
- IloConstraintArray 136**
- ILOCONTINUOUSCALLBACK0 547**
- IloConversion 139**
- IloCplex 565**
- ILOCPLEXGOAL0 548**
- ILOCROSSOVERCALLBACK0 550**
- IloCsvLine 456**
- IloCsvReader 460**
- IloCsvReaderParameterException 470**
- IloCsvTableReader 481**
- ILOCUTCALLBACK0 551**
- IloDeleterMode 143**
- IloDiff 144**
- IloDifference 487, 488**
- IloDisableNANDetection 147**
- ILODISJUNCTIVECUTCALLBACK0 552**
- IloDistribute 148**
- IloDiv 152**
- IloDuplicatedTableException 471**
- IloEmptyHandleException 153**
- IloEnableNANDetection 154**
- IloEndMT 155**
- IloEnv 156**
- IloEnvironmentMismatch 161**
- IloEqIntersection 162**
- IloEqPartition 163**
- IloEqUnion 164**
- IloException 165**
- IloExponent 167**
- IloExpr 168**
- IloExprArray 175**
- IloExtractable 179**
- IloExtractableArray 182**
- IloFastMutex 184**
- IloFieldNotFoundException 472**
- IloFileNotFoundException 473**
- IloFloatArray 188**
- IloFloatVar 189**
- IloFloatVarArray 190**
- IloFloor 191**
- ILOFRACTIONALCUTCALLBACK0 553**
- IloFunction 192**
- IloGetClone 193**
- IloHalfPi 194**
- ILOHEURISTICCALLBACK0 554**
- IloIfThen 195**
- IloIncorrectCsvReaderUseException 474**
- ILOINCUMBENTCALLBACK0 555**
- IloInitMT 198**
- IloInt 199**
- IloIntArray 200**
- IloIntBinaryPredicate 204**
- ILOINTBINARYPREDICATE0 42**
- IloIntersection 489**
- IloIntervalList 490**
- IloIntervalListCursor 495**

- IloIntExpr **206**
- IloIntExprArg **209**
- IloIntExprArray **211**
- IloIntSet **214**
- IloIntSetVar **219**
- IloIntSetVarArray **224**
- IloIntTernaryPredicate **226**
- ILOINTTERNARYPREDICATE0 **43**
- IloIntTupleSet **228**
- IloIntTupleSetIterator **231**
- IloIntVar **232**
- IloIntVarArray **238**
- IloInverse **242**
- IloIsNAN **245**
- IloIterator **246**
- ILOLAZYCONSTRAINTCALLBACK0 **556**
- IloLinearTimeConverter **248**
- IloLineNotFoundException **475**
- IloLog **249**
- IloMax **250, 498**
- IloMaximize **251**
- IloMember **252**
- IloMin **253, 499**
- IloMinimize **254**
- ILOMIPCALLBACK0 **557**
- IloModel **255**
- IloMonotonicDecreasingNumExpr **260**
- IloMonotonicIncreasingNumExpr **261**
- IloMutexDeadlock **262**
- IloMutexNotOwner **263**
- IloMutexProblem **264**
- ILONETWORKCALLBACK0 **558**
- ILONODECALLBACK0 **559**
- IloNot **266**
- IloNotMember **268**
- IloNullIntersect **269**
- IloNum **270**
- IloNumArray **271**
- IloNumColumn **274**
- IloNumColumnArray **277**
- IloNumExpr **279**
- IloNumExprArg **282**
- IloNumExprArray **284**
- IloNumFunction **288**
- IloNumSet **289**
- IloNumToAnySetStepFunction **500**
- IloNumToAnySetStepFunctionCursor **509**
- IloNumToNumSegmentFunction **512**
- IloNumToNumSegmentFunctionCursor **519**
- IloNumToNumStepFunction **522**
- IloNumToNumStepFunctionCursor **528**
- IloNumVar **293**
- IloNumVarArray **299**
- IloObjective **305**
- IloOr **312**
- IloPack **316**
- IloPartition **321**
- IloPathLength **322**
- IloPathTransitFunction **327**
- IloPathTransitI **328**
- IloPi **330**
- IloPiecewiseLinear **331**
- IloPower **332**
- ILOPRESOLVECALLBACK0 **560**
- ILOPROBINGCALLBACK0 **561**
- IloQuarterPi **333**
- IloRandom **334**
- IloRange **337**
- IloRangeArray **344**
- IloRound **351**
- IloScalProd **364, 365, 366, 367**
- IloSemaphore **368**
- IloSemiContVar **371**
- IloSemiContVarArray **375**
- IloSequence **378**
- ILOSIMPLEXCALLBACK0 **562**
- IloSolution **381**
- IloSolutionArray **392**
- IloSolutionIterator **395**
- IloSolutionManip **397**
- ILOSOLVECALLBACK0 **563**
- IloSOS1 **352**
- IloSOS1Array **355**
- IloSOS2 **358**
- IloSOS2Array **361**
- IloSquare **398**
- IloSubset **399**
- IloSubsetEq **400**
- IloSum **401**
- IloTableConstraint **402**

IloTableNotFoundException **476**
 IloThreeHalfPi **404**
 IloTimer **405**
 IloTwoPi **407**
 IloTypedList **408**
 IloTypedListManagerI **409**
 IloUnion **530, 531**
 ILOUSERCUTCALLBACK0 **564**
 IloXmlContext **428**
 IloXmlInfo **435**
 IloXmlReader **446**
 IloXmlWriter **451**
 IncumbentCallbackI **691, 809**
 IntegerFeasibility **644, 680**
 IntegerFeasibilityArray **681**
 IntParam **694**
 InvalidCutException **702**
 Iterator **85, 218, 258, 393**

L

LazyConstraintCallbackI **703, 812**
 LinearIterator **177**
 LineIterator **477, 485**

M

MIPCallbackI **705**
 MIPEmphasisType **713**
 MultipleConversionException **715**
 MultipleObjException **717**

N

NetworkCallbackI **718**
 NodeCallbackI **720, 814**
 NodeData **712**
 NodeEvaluator **725**
 NodeEvaluatorI **727**
 NodeSelect **731**
 NonLinearExpression **287**
 NotExtractedException **61**
 NumParam **732**

O

operator **421, 422, 423, 535**
 operator- **419, 534**
 operator && **410**
 operator * **411, 532**
 operator new **412**
 operator! **413**
 operator!= **414**
 operator% **415, 416, 417**
 operator+ **418, 533**
 operator/ **420**
 operator== **424, 536, 537, 538**
 operator> **425**
 operator>= **426**
 operator>> **427**
 operator|| **428**
 optim.concert **33**
 optim.concert.extensions **454**
 optim.concert.xml **428**
 optim.cplex.advanced.cpp **765**
 optim.cplex.cpp **538**
 optim.cplex.cpp.logical **764**
 optim.cplex.cpp.piecelinear **762**

P

PresolveCallbackI **736**
 PrimalPricing **738**
 ProbingCallbackI **739**
 PWLFormulation **735, 764**

Q

Quality **742**

S

SearchLimit **746**
 SearchLimitI **748**
 Sense **311**
 SimplexCallbackI **750**
 SolveCallbackI **752, 819**
 Status **63, 142**
 StringParam **757**

T

TableIterator **479**

Type **304, 632**

U

UnknownExtractableException **758**

UserCutCallbackI **759, 824**

V

VariableSelect **761**