An Introduction to the STL Library

Michael J. Vilot

ObjectCraft, Inc.
7 Colby Ct., # 4–321
Bedford, NH 03110–6427 USA
(603) 429–3808
`mjv@objects.mv.com`

The widespread adoption of Object-Oriented Design and C++ is having a profound impact on software engineering. While others philosophize about the "right" methodology and debate about "pure" object-oriented languages, C++ developers are busily delivering solutions to their customers' needs. Increasingly, these solutions are exploiting a quiet revolution in the practice of software development — creating programs from reusable software components.

Why are C++ development projects so much more numerous and successful than their counterparts in C, or even Smalltalk? At first glance, the odds seem against such results: C++ is a larger, more complex language than many of its rivals. But despite the esoteric language features, uneven quality of implementations, and long learning curves, C++ still offers advantages to those who use it. Perhaps the key advantage is access to an established — and growing — marketplace of well-crafted, high-quality C++ libraries. Reusing software components from these libraries offers C++ developers advantages in quality, reliability, efficiency, and time to market.

Of course, these advantages do not accrue from just any arbitrary collection of C++ classes and functions offered up as a "library." To be effective, a software component library must be clearly designed, soundly engineered, and carefully implemented. A successful library must also serve the needs of its intended users, or it will quickly find itself abandoned for one of the other library products available. Ideally, such a library will be easy to use, thus simplifying its application within a C++ project.

An example of a library meeting these criteria is the Standard Template Library (STL), developed at Hewlett-Packard by Alex Stepanov, Meng Lee, and their colleagues. STL contains general-purpose mechanisms,[1] using C++'s templates for component parameterization.[2] A notable aspect of this library is that it focuses primarily on algorithms — once thought to be the exclusive province of FORTRAN subroutine libraries.

This article focuses on using STL, rather than a comprehensive treatment of its internal technical details. The first section provides an overview of the library. The second section provides a series of C++ code examples that illustrate various aspects of using the library's "algorithm-oriented generic software" approach.[3] Finally, the last section concludes with a more detailed overview of the library. It can be used as a roadmap to the detailed reference material.

The Library

Like many of the most successful C++ component libraries, STL has been in development a long time, has a simple design at its core, and offers significant advantages to its users. In the case of STL, these advantages are flexibility, efficiency, and theoretical soundness.

Unlike GUI components or application frameworks, STL provides general-purpose components for common programming tasks. Its main value is in providing efficient and reliable implementations that eliminate the need to hand-craft low-level algorithms and data structures. As such, it can be used as the foundation for more ambitious libraries.

Background

The present library is the result of iterative development. STL evolved from several earlier libraries, developed in Scheme, Ada, and C++. Dave Musser and Alex Stepanov designed the earlier versions of the library. The key aspects of the design that emerged from these iterations included: the overall structure, semantic requirements, algorithm design, complexity analysis, and performance tuning.

In November 1993, Alex Stepanov and Meng Lee approached the ANSI/ISO C++ Standards Committee and suggested a small subset of STL might be appropriate for the Standard C++ Library. Because its size rivals that of standard iostreams, the STL proposal to X3J16 has itself been through several iterations.[4,5,6] It was accepted as part of the C++ Standard Library in July, 1994.

Quick Overview

The essential structure of STL can be represented by the OOD class category diagram[7] of Figure 1:

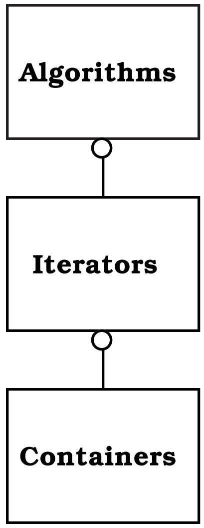

Figure 1: Key Abstractions in the STL Library

The primary focus of the library is on the components in the Algorithms category. They use *Iterators* (a generalization of pointers) and *Function Objects* (a generalization of functions) to operate upon components in the Containers category.

An important result of this design is that new algorithms and containers can be added independently, and the existing components will adapt to them unchanged. This kind of extensibility and adaptability is essential for any such library of general-purpose mechanisms.

Key Features

The essential design criteria are stated clearly in the STL proposal to X3J16:

> "The Standard Template Library provides a set of well structured generic C++ components that work together in a seamless way. Special care has been taken to ensure that all the template algorithms work not only on the data structures in the library, but also on built-in C++ data [types]. For example, all the algorithms work on regular pointers. … Another important consideration is efficiency. Much effort has been spent to verify that every template component in the library has a generic implementation that performs within a few percentage points of the efficiency of the corresponding hand coded routine."[5]

Perhaps the most interesting technical feature of the library is its ability to work equally well with pointers to built-in data types and full-blown "abstract traversal" objects.[8] The STL concept of Iterators is the key to this flexibility — they overcome a problem with treating pointers as special case template arguments.[9]

A similar flexibility applies to function pointers and "function-like" objects (known variously as Function Objects,[10] Enzymes,[11] and Functionoids[12]). In the next section, we will see how this flexibility allows a gradual transition from simple, "C like," code to relatively sophisticated use of both STL and C++.

Using the Library

This section contains 10 small C++ programs. Each program is slightly more complex than the previous one, and employs an appropriate level of abstraction to combat the creeping complexity. This serves to introduce more aspects of STL gradually into the programming task at hand.

A Simple Example

The first example is very simple. It searches a character string for the first occurence of a particular character. This task is so trivial, the program provides three solutions to it:

```
#include "stl"
#include <iostream>
#include <string.h>

int main()   // Example 1:  finding a character
{
  const char data[] = "Bjarne Stroustrup";
  const char* end = data + sizeof data;
  const char c = 'u';

  for (const char* p = data; p != end; ++p)
    if (*p == c) break;
  cout << "Found at " << (p-data) << endl;

  p = strchr(data, c);
  cout << "Found at " << (p-data) << endl;

  p = find(data, end, c);
  cout << "Found at " << (p-data) << endl;

  return 0;
}
```

The declarations establish a null-terminated character sequence to search. The `for` loop searches from the beginning to the end of the sequence, looking for an exact match on the target character. The second solution is a call to the standard `strchr()` function, which searches through the pointer. `strchr()` advances the pointer until the character is found or the pointer value becomes null. The third solution is to use the STL `find()` function. All three locate the character at index 11.

Note that `strchr()` may well be optimized to work particularly well on character sequences for the current platform. This makes it potentially more efficient than the equivalent `for` loop. It also depends critically on the null terminator convention used throughout much of the Standard C Library.

Being a function template, `find()` may also be specialized for the `char` data type. However, because it does not depend on null termination, `find()` will not wander all over memory looking for one if the string fails to contain one. This means that `find()` is not only as efficient as `strchr()`, but also more reliable.

The next example illustrates that `find()` is also more flexible. Consider using some representation for strings other than hopefully null-terminated `char` arrays:

```
#include "stl"
#include <iostream>
#include <string.h>
```

```
int main()   // Example 2: adding a new data
             // structure
{
  const char data[] = "Bjarne Stroustrup";
  const char* end = data + sizeof data;
  const char c = 'u';
  vector<char> name(data, end);

  for (const char* p = data; p != end; ++p)
    if (*p == c) break;
  cout << "Found at " << (p-data) << endl;

  p = strchr(data, c);
  cout << "Found at " << (p-data) << endl;

  p = find(data, end, c);
  cout << "Found at " << (p-data) << endl;

  p = find(name.begin(), name.end(), c);
  cout << "Found at " << (p-name.begin())
       << endl;
  return 0;
}
```

This second program introduces the use of the STL `vector` data structure component, whose constructor initializes it to the same characters. Even though we are still using `char` sequences, we cannot reuse the `strchr()` library function — there is no guarantee that vectors null-terminate their contents. Indeed, because `vectors` are parameterized by the type of item they contain, they cannot assume that a null value even exists for arbitrary types.

On the other hand, the `find()` function continues to work, unchanged. In fact, since the `begin()` and `end()` member functions return `char` pointers for `vector<char>`, the same efficient specialization of `find<char>` can be called.

Note that `find()` is not restricted to using pointers: any class with `operator*()` and `operator++()` defined will do.

Using Strings

Of course, arrays of `chars` represent a rather low level of abstraction. They are, however, quite pervasive — they form the backbone of many C and C++ programs. The reason is simple: `char` sequences are a useful representation of a fundamental abstraction, the string.

The next two example programs move away from the low-level array of `chars` representation. The `string` class used for these programs illustrates that requirements for an STL-compatible "sequence" are quite simple, as shown in Listing 1.

Listing 1: A Simple `string` Component

```
class string {
  char* _rep;
  void assign(const char*);
public:
  string();
  string(const char*);
  string(const string&);
 ~string();

  string& operator=(const string&);
  string& operator=(const char*);

  const int   size() const;
  const char* data() const;
};

int operator==(const string&, const string&);
int operator!=(const string&, const string&);
int operator <(const string&, const string&);

class istream;
istream& operator>>(istream&, string&);

class ostream;
ostream& operator<<(ostream&, const string&);
```

The main responsibility of this class is to encapsulate the details of storage management, so that callers can concentrate on more abstract operations. In particular, this `string` component provides few operations: equality, simple comparison, and iostreams extraction and insertion. Of course, in a production system, we would use a standard `string`.

First, we'll exercise the default constructor and iostreams operations:

```
#include <fstream>
#include "string.h"

int main()   // Example 3:  using arrays
{
  string a[16];  // hopefully large enough

// input:
  ifstream in("meeting.members");
  for (int i =0; in; ++i) {
    in >> a[i];
  }

// output:
  for (int j =0; j<i; ++j) {
    cout << a[j] << endl;
  }

  return 0;
}
```

This example uses a fixed-size array to hold some strings, and uses a file stream to fill the array. This program is adequate for the small file used for input:

The `meeting.members` file

```
Tom
Bjarne
Nathan
Alex
Mike
Larry
Meng
```

However, note that this program has a fatal flaw — if given more than 16 names, it merrily writes over memory past the end of the array! This is almost certainly going to cause unpleasant behavior at runtime.

The next step is to replace the low-level array with a more flexible data structure:

```
#include "stl"
#include <fstream>
#include "string.h"

int main()   // Example 4:  vectors encapsulate
{            //             memory management
  vector<string> v;

// input:
  ifstream in("meeting.members");
  while (in) {
    string s;
    in >> s;
    v.insert(v.end(),s);   // append
  }

// output:
  for (size_t i =0; i<v.size(); ++i) {
    cout << v[i] << endl;
  }
  return 0;
}
```

This fourth example program simply reads in the strings from the file and appends them to the end of the `vector`. It is functionally equivalent to the previous example, but is not limited to a fixed number of names (limited only by the amount of dynamic memory available). It is also more reliable — it doesn't scribble all over memory if we happen to read a larger file.

The `string` and `vector` components encapsulate the memory management details of character and array storage. If we replace the string component with a sophisticated copy-on-write implementation,[13] these programs do not have to change. We can also tune the `vector` component, overriding its default `allocator` template argument with a class that does special-purpose memory allocation (for example, one that manages DOS extended memory).

As it turns out, the minimal facilities defined for `strings` are all that we need to use this type with the rest of the STL library. The next example illustrates that even such a simple type is usable as the argument type of a `sort()`:

```
#include "stl"
#include <fstream>
#include "string.h"

int main()   // Example 5:  sorting strings
{
  vector<string> v;

// input:
  ifstream in("meeting.members");
  while (in) {
    string s;
    in >> s;
    v.insert(v.end(),s);   // append
  }

// process:
  sort(v.begin(),v.end());

// output:
  for (size_t i =0; i<v.size(); ++i) {
    cout << v[i] << endl;
  }

  return 0;
}
```

As noted earlier, the `begin()` and `end()` members provide pointers to the elements stored in the `vector`. `sort()` then uses these to iterate through the strings to order them, using the `operator<()` defined in Listing 1.

Notice how easily this program combined components from two substantial C++ libraries: iostreams and STL. Given the same `meeting.members` file, this fifth example program produces the output:

```
Alex
Bjarne
Larry
Meng
Mike
Nathan
Tom
```

Recall the original example of finding a character in a sequence. A similar problem is finding a string in a list. This next example illustrates that the STL `find()` component is just as effective at this new task. Note that this is exactly the same `find()` used in Examples 1 and 2:

```
#include "stl"
#include <fstream>
#include "string.h"

int main()   // Example 6:  finding Alex in a
             // lineup
{
  Vector<string> lineup;

// input:
  ifstream in("meeting.members");
  while (in) {
    string s;
    in >> s;
    lineup.insert(lineup.end(),s);  // append
  }

// process:
  const string data = "Alex";
  const string* p =
      find(lineup.begin(), lineup.end(), data);
  if (p != lineup.end()) { // sentinel
    cout << "Found at " << (p-lineup.begin())
         << endl;
  }

// output:
  for (size_t i =0; i<lineup.size(); ++i) {
    cout << i << ' ' << lineup[i] << endl;
  }

  return 0;
}
```

Given the same meeting.members input file, this sixth example program produces:

```
Found at 3
0 Tom
1 Bjarne
2 Nathan
3 Alex
4 Mike
5 Larry
6 Meng
```

Note that the STL components make minimal assumptions about the types they handle through the iterators. `find()` only requires equality comparison, and `sort()` only requires a less-than ordering relation.

In fact, we are not restricted to providing `operator==()` and `operator<()` for these operations. Overloaded versions of `find()` and `sort()` take a template argument that let us use any name for the requisite operations. As we'll see, these need not even be functions.

Handling Address Lists

Although computers were invented as number crunchers, a more common task today is manipulating structured information. This is often more string-oriented than number-oriented. A typical activity, and one often performed on PC-class machines, is address list processing. The next few examples illustrate how STL can be used in this domain of "business" programming.

For example, suppose we have files containing the names, addresses, and phone numbers of people belonging to some organization. These plain-text files have a regular, if not quite convenient, organization, such as:

The `address.list` file

```
Name         : Alex Stepanov
Address      : HP Labs
               1501 Page Mill Rd.
               Palo Alto, CA 94304
               USA
Tel          : (415) 857-8117

Name         : Bjarne Stroustrup
Address      : AT&T Bell Labs
               600 Mountain Ave
...
```

We could use our existing `string` component to read and write the individual lines in this file. However, that would not help much: for example, sorting the lines in the file would yield meaningless data.

The files contain just the text representation of the important information. Using only `string`s, we would still be left with the relatively low-level task of figuring out what they mean.

Of course, object-oriented design helps us approach this task from another direction. Another way to think about the text files is that they are simply the persistent form of a collection of objects. As Figure 2 illustrates, we can identify the abstractions that describe the objects in this collection:

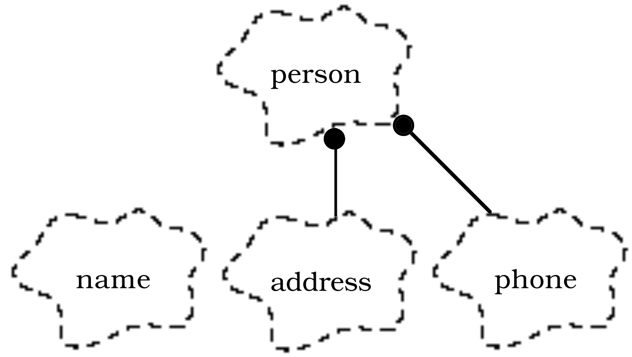

Figure 2: OOD Class Diagram for `Person`

One of the important benefits of using C++ is that it lets us represent these design ideas directly in a program's source text. Listing 2 illustrates how we might build on the `string` component to represent the key abstractions of this problem:

Listing 2: C++ Classes for `Person`

```
struct name {
  string first, middle, last;
};

int operator==(const name&, const name&);
int operator!=(const name&, const name&);
int operator <(const name&, const name&);

istream& operator>>(istream&, name&);
ostream& operator<<(ostream&, const name&);

struct address {
  string line1, line2;
  string city, state, zip;
  string country;
};

int operator==(const address&, const address&);
int operator!=(const address&, const address&);
int operator <(const address&, const address&);

istream& operator>>(istream&, address&);
ostream& operator<<(ostream&, const address&);

struct phone {
  string area, exchange, number;
  string extension;
};

int operator==(const phone&, const phone&);
int operator!=(const phone&, const phone&);
int operator <(const phone&, const phone&);

istream& operator>>(istream&, phone&);
ostream& operator<<(ostream&, const phone&);

struct person {
  name n;
  address a;
  phone p;
};

int operator==(const person&, const person&);
int operator!=(const person&, const person&);
```

```
int operator <(const person&, const person&);

istream& operator>>(istream&, person&);
ostream& operator<<(ostream&, const person&);
```

As with the `string` component, these classes encapsulate the necessary details of input/output and storage management. They are straightforward, given the supporting abstractions. Indeed, many of the operations are defined trivially in terms of their `string` counterparts.

The `name`, `address`, and `phone` extractors also encapsulate the formatting of address list entries in the file. Following the usual iostreams conventions, the `operator>>()` functions manipulate the stream state if things should go awry.[14] The example programs (and the STL components) can then rely on the usual iostreams semantics.

The next example program is much the same as the previous one. It deals with `persons` instead of `strings`, and it uses `istream` iterators to obtain its input:

```
#include "stl"
#include <fstream>
#include "names.h"

int main()   // Example 7:  input iterators
{
  vector<person> list;

// input:
  ifstream in("address.list");
  istream_iterator<person> input(in), eof(0);
  vector<person>::iterator put(list, list.end());
  copy(input, eof, put);

// process:
  sort(list.begin(), list.end());

// output:
  for (size_t i =0; i<list.size(); ++i) {
    cout << list[i].n << endl;
  }

  return 0;
}
```

Iostream iterators are the result of generalizing the notion of a "sequence." The first and second example programs used a `char` array to store a sequence of characters in memory. The subsequent example programs used a `vector` to store various sequences of things in memory. Why not treat a file as a sequence of things on secondary storage?

Applying this concept, the seventh example program copies the sequence of `persons` from the file-based sequence to the `vector`, sorts them, then displays each person's name:

```
Bruce Eckel
Thomas Keffer
Andrew Koenig
Alex Stepanov
Bjarne Stroustrup
Mike Vilot
```

Note that the STL components `copy()` and `sort()` work exactly as before. They are not affected by the changes in the type of information the programs are manipulating, nor how they obtain their information.

The next example shows that `ostream` iterators provide the complementary facility for output. It uses one to write the sorted address list back out to a file:

```
#include "stl"
#include <fstream>
#include "names.h"

int main()   // Example 8:  output iterators
{
  vector<person> list;

// input:
  ifstream in("address.list");
  istream_iterator<person> input(in), eof(0);
  vector<person>::iterator put(list, list.end());
  copy(input, eof, put);

// process:
  sort(list.begin(), list.end());

// output:
  ostream_iterator<person> out("\n\n");
   // argument is the record separator
  copy(list.begin(), list.end(), out);

  return 0;
}
```

At this point, we can see the value of STL's design using algorithms, iterators, and containers. Programs can manipulate arbitrary kinds of information, including built-in types. They can store sequences, iterating over them as necessary to use various algorithms. The sequences can be either in memory or in files on secondary store.

Obviously, we can generalize this further. For example, we could use kinds of streams that are connected to mainframe-based databases, and extractors and inserters that use remote transactions instead of local file access.

Some Advanced Uses

Note that the previous example program has an important change over the seventh example: it outputs the full name, address, and phone number of each person. Suppose we still wanted just their names? STL supports this kind of customization.

The next example program uses the `for_each()` component to iterate through the sorted list and print only the names:

```
#include "stl"
#include <fstream>
#include "names.h"

static void print_name(const person& p);

int main()   // Example 9:  transforming the
             // output
{
```

```
  vector<person> list;

// input:
  ifstream in("address.list");
  vector<person>::iterator put(list, list.end());
  istream_iterator<person> input(in), eof(0);
  copy(input, eof, put);

// process:
  sort(list.begin(), list.end());

// output:
  for_each(list.begin(), list.end(),
    print_name);

  return 0;
}
```

In this case, `for_each()` takes a pointer to the `print_name()` function and calls it once for each person on the list. The `print_name()` function is straightforward:

```
static void print_name(const person& p)
{
  cout << p.n << endl;
}
```

However, notice that this function is rather inflexible: it relies on global data (`cout`), and thus hard-codes the destination output stream into the program. Removing this restriction illustrates another kind of flexibility in STL.

Just as iterators do not have to be pointers, operations do not have to be functions. Specifically, the `for_each()` component can be instantiated with any class that has an `operator()()` defined.

For example, instead of defining a `print_name()` function, we can use a `printer` class:

```
// printer.h
#include "names.h"
#include <fstream>

class printer {          // functional object
  unsigned count;
  ostream& out;
public:
  printer(ostream& o);
  void operator()(const person& p);
};

printer::printer(ostream& o)
  : out(o), count(0) {}

void printer::operator()(const person& p)
{
  out.width(2);
  out << count << ' ' << p.n << endl;
  ++count;
}
```

This class encapsulates not only the output destination, but also some additional formatting state: it provides a sequence number next to each person's name. Obviously, we could define other semantics. But we have avoided tying the program to an inflexible design through global data.

The final example program uses an instance of the `printer` class to effect output. It also uses a second address list file for input:

```
#include "stl"
#include "printer.h"

int main()    // Example 10:  merge 2 address
              // lists
{
  vector<person> list;

// input:
  ifstream in("address.list");
  vector<person>::iterator put(list,
    list.end());
  istream_iterator<person> input(in), eof(0);
  copy(input, eof, put);

  ifstream in2("other.list");
  istream_iterator<person> input2(in2);
  copy(input2, eof, put);

// process:
  sort(list.begin(),  list.end());
  unique(list.begin(),list.end());

// output:
  printer p = cout;
  for_each(list.begin(), list.end(), p);

  return 0;
}
```

This program performs a classic "merge/purge" operation, combining mailing lists and eliminating duplicates. Once the sequence is sorted, `unique()` removes all but the first in any run of duplicated items, using `==` to compare for equality. Together with the `printer` object's formatting, it uses the equality defined for `persons` to produce output such as:

```
 0 Mike Ball
 1 Peter Becker
 2 Greg Comeau
 3 Bruce Eckel
 4 Thomas Keffer
 5 Andrew Koenig
 6 Josee Lajoie
 7 J. Lawrence Podmolik
 8 Jerry Schwarz
 9 Alex Stepanov
10 Bjarne Stroustrup
11 Mike Vilot
```

With this final example program, we can see how the simple ideas of searching and sorting can be important in useful programs. An obvious generalization would be processing a list of files given as command line arguments. Other STL components provide the extensibility for even more ambitious uses.

Other Library Elements

The STL library is comprehensive, and contains many more components than these example programs have used. Tables 1 through 3 indicate some of the other components contained in the library.

The container components implement a minimal set of data structures commonly found in many general-purpose foundation class libraries:

Table 1. Containers

Sequences	Associative	
`vector`	`set`	`multiset`
`list`	`map`	`multimap`
`deque`		
`queue`		
`priority_queue`		
`stack`		

The predefined iterators provide support for all the algorithm components:

Table 2. Iterators

`input_iterator`	`output_iterator`
`forward_iterator`	
`bidirectional_iterator`	
`random_access_iterator`	
`istream_iterator`	`ostream_iterator`

The most extensive category includes the algorithmic components:

Table 3. Algorithms

`for_each`	
`find`	`find_if`
`adjacent_find`	
`count`	`count_if`
`mismatch`	
`equal`	
`search`	
`copy`	`copy_backward`
`swap`	`swap_ranges`
`transform`	
`replace`	`replace_if`
`replace_copy`	`replace_copy_if`
`fill`	`fill_n`
`generate`	`generate_n`
`remove`	`remove_if`
`remove_copy`	`remove_copy_if`
`unique`	`unique_copy`
`reverse`	`reverse_copy`
`rotate`	`rotate_copy`
`random_shuffle`	
`partition`	`stable_partition`
`sort`	`stable_sort`
`partial_sort`	`partial_sort_copy`
`nth_element`	
`lower_bound`	`upper_bound`
`equal_range`	
`binary_search`	
`merge`	`inplace_merge`
`accumulate`	`inner_product`
`partial_sum`	`adjacent_difference`

Because the STL components are designed as an extensible foundation, the list of useful components is open-ended.

Conclusion

The HP Standard Template Library provides a collection of general-purpose software components. Like many successful C++ libraries, it provides reliable, efficient, and flexible components. Unlike most similar libraries, however, STL focuses primarily on algorithmic abstractions.

The library's implementation relies heavily on the C++ `template` mechanism for parameterization. It uses very little inheritance, and does not rely on dynamic binding through virtual functions.

As a result, the performance of STL components is at least as good as the equivalent C or C++ code. In many cases, it is better — using an inlined operation on a functional object is usually less expensive than calling a function through a pointer.

The components are reliable, having been designed with a firm theoretical foundation. A notable aspect of each component's interface is a statement of its time and space complexity requirements and guarantees.

Perhaps the most important aspect of STL is the flexibility of using the components with built-in types and pointers as easily as with arbitrary classes. As the example programs in this article have illustrated, this flexibility supports the evolutionary process that is characteristic of OOD.

Developers of C++ software component libraries will gain some valuable insights from STL's design. Its decomposition illustrates how effective parameterization can dramatically reduce the size of a library (for example, instead of providing a `search` member function for every kind of container, STL provides generic algorithms that work with all of them):

> "If software components are tabulated as a three-dimensional array, where one dimension represents different data types (e.g. `int`, `double`), the second dimension represents different containers (e.g. vector, linked-list, file), and the third dimension represents different algorithms on the containers (e.g. searching, sorting, rotation), if `i`, `j`, and `k` are the size of the dimensions, then `i * j * k` different versions of code have to be designed. By using template functions that are parameterized by a data type, we need only `j * k` versions. Further, by making our algorithms work on different containers, we need merely `j + k` versions."

This significantly simplifies the library's design. It also makes it possible to use components in the library together with user defined components in a very flexible way.

C++ developers who use STL components will gain some valuable assistance through advantages in quality, reliability, efficiency, and time to market.

References

[1] Vilot, M. and G. Booch, "Component Library Design Considerations," *C++ Report*, vol. 4, no. 5, June 1992, p. 18.

[2] Vilot and Booch, "Component Design Considerations," *C++ Report*, vol. 4, no. 7, September 1992, p. 18.

[3] Stepanov, A. and D. R. Musser, *Algorithm-Oriented Generic Software Library Development*, HPL–92–65 (R.1), Hewlett-Packard Company, Palo Alto CA, November, 1993.

[4] Stepanov, A. and M. Lee, *The Standard Template Library*, Doc. No. X3J16/94–0030 WG21/N0417, 25 January 1994.

[5] Stepanov and Lee, *The Standard Template Library*, Doc. No. X3J16/94–0030R1 WG21/N0417, 7 March 1994.

[6] Stepanov, and Lee, *The Standard Template Library*, Doc. No. X3J16/94–0095 WG21/N0482, 31 May 1994.

[7] Booch, G., *Object-Oriented Analysis and Design*, ISBN 0-8053-5340-2, Benjamin/Cummings, Redwood City CA, 1994.

[8] Dewhurst, S., "Control Abstraction," *C++ Journal*, Fall 1990, p. 23.

[9] Davidson, A., "More on the USENIX C++ Advanced Topics Workshop," *C++ Report*, vol. 3, no. 9, October 1991, p. 1.

[10] Koenig, A., "Applicators, Manipulators, and Function Objects," *C++ Journal*, Summer 1990, p. 21.

[11] Coggins, J., "Design Criteria for C++ Libraries," *USENIX C++ Conference*, San Francisco CA, 9–11 April 1990, p. 25.

[12] Jossman, p., Scheibel, E., and J. Shank, "Climbing the C++ Learning Tree," *USENIX C++ Conference*, San Francisco CA, 9–11 April 1990, p. 11.

[13] Keffer, T., "The Design and Architecture of Tools.h++," *C++ Report*, vol. 5, no. 5, June 1993, p. 28.

[14] Horstmann, C., "Understanding the iostream Library," *C++ Report*, vol. 6, no. 4, May 1994, p. 23.