DOC++

A Documentation System for C, C++, IDL and Java

Contents

Welcome to the wonderful world of DOC++

DOC++ is a documentation system for C, C++, IDL and Java generating both, TeX output for high quality hardcopies and HTML output for sophisticated online browsing of your documentation. The documentation is extracted directly from the C/C++/IDL header/source files or Java class files.

Here is a list of highlights:

- hierarchically structured documentation
- automatic class graph generation (as Java applets for HTML)
- cross references
- high end formatting support including typesetting of equations

For an introduction to the philosophy of DOC++ go on reading the Introduction. If you want to jump right into using DOC++ on your sources go to section Quickstart (→ 4.3.1.1, *page* 24). Section Reference Manual (→ 4.3.1.1, *page* 24) provides you a complete manual on the features provided DOC++. If you have problems using DOC++ you should consult our Frequently Asked Questions. If you first want to look at some output of DOC++, don't do anything at all, you are just reading an example. An example for some dummy C++ classes can be found in Example (→ 4.3.1.1, *page* 24).

Enjoy!

1 Introduction

The idea of DOC++ is to provide a tool that supports the *programmer* for writing *high quality* documentation while keeping concentration on the program development. In order to do so, it is important that the programmer can add the documentation right into the source code he/she developes. Only with such an approach, a programmer would really write some documentation to his/her classes, methods etc. and keep them up to date with upcoming changes of code. At the same time it must be possible to directly compile the code without need to a previous filter phase as needed when using e.g. 'cweb'. Hence, the only place where to put documentation are comments.

This is exactly what DOC++ uses for generating documentation. However, there are two types of comments the programmer wants to distinguish. One are comments he/she does for remembering some implementational issues, while the others are comments for documenting classes, functions etc. such that he/she or someone else would be able to use the code later on. In DOC++ this distinction is done via different types of comments. Similar to 'JavaDoc', documentation comments are of the following format

- `/** ... */`
- `/// ...`

where the documentation is given in "`...`". Such comments are referred to as *DOC++ comments*. Each DOC++ comment generates a manual entry for the *next* declaration in the source code. Trailing comments can be used to generate manual entries too while being in Quantel mode.

Now, let's consider what "high quality" documentation means. Many programmers like to view the documentation online simply by klicking their mouse buttons. A standard for such documents is HTML for which good viewers are available on almost every machine. Hence, DOC++ has been designed to produce HTML output in a structured way.

But have you ever printed a HTML page? Doesn't it look ugly, compared to what one is used to? This is not a problem for DOC++ since it also provides TeX output for generating high quality hardcopies.

For both ouput formats, it is important that the documentation is well structured. DOC++ provides hierarchies, that are reflected as sections/subsections etc., or HTML page hierarchies, respectively. Also an index is generated that allows the user to easily find what he/she looks for.

As C++ and Java are (somewhat) object-oriented languages, another type of hierarchy is introduced, namely class hierarchies. The best way to read such a hierarchy is by looking at a picture of it. Indeed, DOC++ automatically draws a picture for each class derivation hierarchy or shows it with a Java applet in the HTML output.

An additional goody of DOC++ is its ability to generate a TeX typeset source code listing for C and C++ code.

2 Quickstart

If you want to jump straight into using DOC++, add a line like

```
/// ...
```

before each function, variable, class, define, etc. you wish to document. For "..." you may choose to add a short documentation string for the entry. You will typically want to do so for your header files only. If you intend to write more than one line of documentation, succeed this line with a comment like

```
/**
   ...
 */
```

and put the long documentation text in place of "...". A source file containing such comments is said to be *docified*. You may call

```
> docify <original> <docfile>
```

from your shell to create a docified copy <docfile> form your <original> file. The ">" indicates the shell prompt and must not to be typed.

Now run DOC++ by typing:

```
> doc++ --dir html <files>
```

for HTML output or

```
> doc++ --tex --output doc.tex <files>
```

for TeX output in you shell, where <files> is the list of docified files.

Each '///' -comment yields one manual entry. If you need to group manual entries, you may do so with the construction:

```
/**@name <name for the group>
 *  <documentation for the group>
 */
//@{
   <other manual entries>
//@}
```

This will create an entry with the specified name, that contains all <other manual entries> as subentries. Note, however, that class members are automatically set as subentries of the class's manual entry. You also may include other files using the comment:

```
//@Include: <file(s)>
```

3

Reference Manual

Names

DOC++ follows the approach of maintaining one source code that contains both, the C/C++ or Java program itself along with the documentation in order to avoid incompatibilities between the program and its documentation. Unlike other approaches such as 'WEB', sources documented with DOC++ are ready to be compiled without need of any preprocessing (like 'tangle'). We feel that this is of great advantage for intensive programming and debugging activities.

This documentation is organized as follows. Section Usage (→ 4.3.1.1, *page* 24) describes how to generate your documentation for readily docified sources, hence explains the comand line options of DOC++. Section Manual Entries (→ 4.3.1.1, *page* 24) discusses, how the manual entries generated for DOC++ comments are built up and section Structure how to structure your documentation hierarchically. Finally section Text Formatting (→ 4.3.1.1, *page* 24) describes all the features provided by DOC++ to format the documentation text (such as bold face typesetting etc.).

3.1

Usage

Names

In addition to command line / configuration file options, the TeX output can be customized by editing the style file "`docxx.sty`" (sorry, there is no documentation on how to do this).

In addition to command line / configuration file options, the HTML output can be customized by means of the following 6 files:

indexHeader.inc Header for index HTML pages

indexFooter.inc Footer for index HTML pages

hierHeader.inc Header for class hierarchy HTML pages

hierFooter.inc Footer for class hierarchy HTML pages

classHeader.inc Header for all other HTML pages

classFooter.inc Footer for all other HTML pages

If one or more of these files are found in the current directory, the corresponding part of a HTML page is substituted by the contents of the file. The '`indexHeader.inc`' and '`hierHeader.inc`' files should start with "`<HTML><TITLE> ...`". File '`classHeader.inc`' should start with "`<BODY> ...`", since for such pages DOC++ sets up the title.

The HTML page header and footer may contain one or more template strings, which will be substituted by DOC++ at documentation generating time:

%file entry's file name

%fullname entry's full name (includes the inheritance)

%name entry's name

%type entry's return type

As an example, the TeX version of this document has been generated with

```
doc++ --tex --output doc.tex --package a4wide doc.dxx
```

while the HTML version has been created using

```
doc++ --dir html doc.dxx
```

As you can see, this documentation itself is written using DOC++ in order to gain the benefits of having just one documentation source and two different output possibilities.

3.1.1 Command Line Options

Calling DOC++ with '`-h`' or '`--help`' option will give you a long screen with one-line descriptions of the command line options provided by DOC++. However, we'll now attempt to provide a more detailed description suitable for you to understand how to call DOC++ with your docified sources.

At the command line DOC++ may be called with a sequence of options and a list of files or directories. No option may be passed after the first filename. All files passed to DOC++ are parsed in the order they are specified for generating documentation from them. All directories are traversed recursively and all files `*.h*` or `*.java` (depending on the '`-J`' or '`--java`' command line option) are parsed. However, it is good practice to control the input files with one main input file and use the '`@Include:`' directive (the way this documentation was written).

Options consists either of a leading character '`-`', followed by one or two characters, or a leading '`--`', followed by the long option name, and optionally a space-separated argument.

Note that options that are set in the configuration file overrides command line options.

Command line options come in three different flavours. The first type of options control parameters that are independent of the chosen output, the second type when generating HTML output (the default) and the third for TeX output (selected with '`-t`' or '`--tex`' option). These are:

-A -all Instructs DOC++ to generate manual entries for every declaration it finds, no matter if it is documented with a DOC++ comment or not.

-c –c-comments Instructs DOC++ to use the C/C++ comments as DOC++ comments.

-C –config `FILE` Read options from the configuration file `FILE`.

-h –help Don't do anything, just print a one-line description of all options to the standard output.

-H –html Instructs DOC++ to parse HTML as formatting language instead of TeX.

-I –input `FILE` Instructs DOC++ to read the list of input files from `FILE` instead of command line.

-J –java Sets DOC++ into Java mode, i.e. instructs DOC++ to parse Java instead of C/C++ (the default).

-nd –no-define Instructs DOC++ to ignore the '#define' macros.

-ng –no-class-graph Suppress the class graph generation.

-p –private Instructs DOC++ to include private class members in the documentation. If not specified no private member will show up in the documentation (even if they are docified).

-q –quick Turn DOC++ into a quick operating mode, which increase the generated documentation size.

-Q –quantel Parse Quantel extensions.

-R –internal-doc Generate internal documentation too.

-t –tex Instructs DOC++ to produce TeX output rather than HTML.

-u –upwards-arrows Draw arrows from derived class to the base class when generating class graphs.

-v –verbose Sets DOC++ into verbose mode making it operate more noisy. This may be helpful when debugging your documentation.

-V –version Don't do anything, just output version information.

-y –scan-includes Scan '#include'ed header files

-Y –idl Sets DOC++ into IDL mode, i.e. instructs DOC++ to parse IDL instead of C/C++ (the default).

-z –php Sets DOC++ into PHP mode, i.e. instructs DOC++ to parse PHP instead of C/C++ (the default).

-Z –docbook Instructs DOC++ to produce DocBook SGML instead of HTML.

The following command line options are only active when HTML output is selected, i.e. no '`-t`' or '`--tex`' option is passed:

-a –tables When this option is specified, DOC++ will use HTML tables for listing the members of a class. This yields all member names to be aligned.

-b –tables-border Same as "`--tables`" except that a bordered table will be used.

-B –footer `FILE` Use `FILE` as the footer for every HTML page generated by DOC++. This is how to get rid of DOC++ logos and customize the output for your needs.

-d –dir `NAME` This specifies the directory where the HTML files and GIFs are to be written. If not specified, the current directory will be used. If the specified directory does not exist, it will be created by DOC++.

-f –filenames Instructs DOC++ to write on each HTML page the file of the source code, where this manual entry has been declared.

-F –filenames-path Same as "`--filenames`" except the complete path of the source file is shown.

-g –no-gifs Instructs DOC++ not to generate GIFs for equations and '`\TEX{}`' text in the documentation. This may reduce execution time when calling DOC++, but note that DOC++ keeps a database of already generated GIFs, such that GIFs are not recreated if they already exists. However, if you do not have 'latex', 'dvips', 'ghostscript' and the 'ppmtools' installed on your system, you *must* use this option, since then DOC++ will fail setting up the GIFs.

-G –gifs This instructs DOC++ to reconstruct all GIFs, even if they already exists. This may be useful if the database is corrupted for some reason.

-i –no-inherited Instructs DOC++ not to show inherited members in the generated HTML documentation.

-j –no-java-graphs Suppresses the generation of Java applets for drawing class graphs.

-k –trivial-graphs Generate class graphs for classes with neither base class nor child classes.

-m –no-members Don't show the members with zero-length documentation in DOC section.

-M –full-toc Show members in HTML TOC.

-P –no-general Discard general stuff.

-S –sort Instructs DOC++ to sort documentation entries alphabetically.

-T –header `FILE` Use `FILE` as header for every HTML page generated by DOC++. This is how to get rid of DOC++ logos and customize the output for your needs.

-w –before-group Print the groups' documentation before groups.

-W –before-class Print the classes' documentation before classes.

-x –suffix `SUFFIX` Use `SUFFIX` as suffix for every generated HTML page, instead of ".html".

-K –stylesheet `FILE` Use `FILE` as style sheet for every generated HTML page.

Finally, this set of command line options provides some control for the TeX output of DOC++:

-ec –class-graph Only generates the class graph.

-ef –env `FILE` Reads the TeX environment from `FILE`.

-ei –index Only generates the index.

-eo –style `OPTION` Adds `OPTION` to TeX's '`\documentclass`'.

-ep –package `PACKAGE` Adds '`\usepackage{package}`' to the TeX environment.

-et –title `FILE` Uses the contents of `FILE` as TeX title page.

-D –depth `DEPTH` Sets the minimum depth (number of levels) in TOC.

-l –no-env Switches off generation of the TeX environment. This should be used if you intend to include the documentation in some TeX document.

-o –output `FILE` Sets the output file name. If not specified, the output is printed to standard output.

-s –source Instead of generating a manual from the manual entries, DOC++ will generate a source code listing. This listing contains all *normal* C or C++ comments typeset in TeX quality. Every line is preceeded with its line number.

-X –hide-index Turn off generation of index at beggining of every section.

3.1.2 Configuration File

The configuration file is a simple ASCII text file. It consists in a list of assignment statements. Each statement consists of a token name and the token's value. The value is separated from the name either by a '`=`' sign or by an arbitrary number of spaces or tabs. If the value consists in a list of values these values must be separated by a space or a comma. The options are the same with the command line ones, except they have slightly different names.

By default, DOC++ looks in the current running directory for a configuration file named "`doc++.conf`". You may tell DOC++ where to find that file using the '`-C`' or '`--config`' command line option. Note that options that are set in the configuration file overrides command line options.

Comments are allowed and may be placed anywhere within the file. They begins with the '#' character and ends at the end of the line. Options independent of the output type:

documentAll Instructs DOC++ to generate manual entries for every declaration it finds, no matter if it is documented with a DOC++ comment or not. The default value is `false`.

useNormalComments Instructs DOC++ to use the C/C++ comments as DOC++ comments. The default value is `false`.

HTMLSyntax Instructs DOC++ to parse HTML as formatting language instead of TeX. The default value is `false`.

fileList Instructs DOC++ to read the list of input files from `FILE` instead of command line. By default it's not set.

parseJava Sets DOC++ into Java mode, i.e. instructs DOC++ to parse Java instead of C/C++ (the default). The default value is `false`.

ignoreDefines Instructs DOC++ to ignore the '#define' macros. The default value is `false`.

noClassGraph Suppress the class graph generation. The default value is `false`.

documentPrivateMembers Instructs DOC++ to include private class members in the documentation. If set to `false` no private member will show up in the documentation (even if they are docified). The default value is `false`.

optimizeForSpeed Turn DOC++ into a quick operating mode, which increase the generated documentation size. The default value is `false`.

quantelExtensions Parse Quantel extensions. The default value is `false`.

internalDoc Generate internal documentation too. The default value is `false`.

doTeX Instructs DOC++ to produce TeX output rather than HTML. The default value is `false`.

upwardsArrows Draw arrows from derived class to the base class when generating class graphs. The default value is `false`.

verboseOperation Sets DOC++ into verbose mode making it operate more noisy. This may be helpful when debugging your documentation. The default value is `false`.

scanIncludes Scan '#include'ed header files. The default value is `false`.

parseIDL Sets DOC++ into IDL mode, i.e. instructs DOC++ to parse IDL instead of C/C++ (the default). The default value is `false`.

parsePHP Sets DOC++ into PHP mode, i.e. instructs DOC++ to parse PHP instead of C/C++ (the default). The default value is `false`.

doDOCBOOK Instructs DOC++ to produce DocBook SGML instead of HTML. The default value is `false`.

Options valid only for HTML output:

useTables When this option is specified, DOC++ will use HTML tables for listing the members of a class. This yields all member names to be aligned. The default value is `false`.

useTablesWithBorders Same as "`useTables`" except that a bordered table will be used. The default value is `false`.

footer Use the specified file as the footer for every HTML page generated by DOC++. This is how to get rid of DOC++ logos and customize the output for your needs. By default it's not set.

outputDir This specifies the directory where the HTML files and GIFs are to be written. If not set, the current directory will be used. If the specified directory does not exist, it will be created by DOC++. By default is not set.

showFilenames Instructs DOC++ to write on each HTML page the file of the source code, where this manual entry has been declared. The default value is `false`.

showFilenamesWithPath Same as "`showFilenames`" except the complete path of the source file is shown. The default value is `false`.

noGifs Instructs DOC++ not to generate GIFs for equations and '`\TEX{}`' text in the documentation. This may reduce execution time when calling DOC++, but note that DOC++ keeps a database of already generated GIFs, such that GIFs are not recreated if they already exists. However, if you do not have 'latex', 'dvips', 'ghostscript' and the 'ppmtools' installed on your system, you *must* use this option, since then DOC++ will fail setting up the GIFs. The default value is `false`.

forceGifs This instructs DOC++ to reconstruct all GIFs, even if they already exists. This may be useful if the database is corrupted for some reason. The default value is `false`.

noInheritedMembers Instructs DOC++ not to show inherited members in the generated HTML documentation. The default value is `false`.

noJavaGraphs Suppresses the generation of Java applets for drawing class graphs. The default value is `false`.

trivialGraphs Generate class graphs for classes with neither base class nor child classes. The default value is `false`.

noMembers Don't show the members with zero-length documentation in DOC section. The default value is `false`.

showMembersInTOC Show members in HTML TOC. The default value is `false`.

discardGeneral Discard general stuff. The default value is `false`.

sortEntries Instructs DOC++ to sort documentation entries alphabetically. The default value is `false`.

header Use specified file as the header to every HTML page generated by DOC++. This is how to get rid of DOC++ logos and customize the output for your needs. By default it's not set.

groupBeforeGroup Print the groups' documentation before groups. The default value is `false`.

classBeforeGroup Print the classes' documentation before classes. The default value is `false`.

htmlSuffix Use the specified suffix as the suffix to every generated HTML page. The default value is "`.html`". Note that the suffix have to begin with a dot.

htmlStyleSheet Use the specified file as style sheet for every generated HTML page.

Options valid only for TeX output:

onlyClassGraph Only generates the class graph. The default value is `false`.

environment Reads the TeX environment from the specified file. By default it's not set.

generateIndex Only generates the index. The default value is `false`.

style Adds the specified option to TeX's '`\documentclass`'. By default it's not set.

usePackage Adds '`\usepackage{package}`' to the TeX environment. By default it's not set.

title Uses the contents of the specified file as TeX title page. By default it's not set.

minimumDepth Sets the minimum depth (number of levels) in TOC. The default value is `1`.

noEnvironment Switches off generation of the TeX environment. This should be used if you intend to include the documentation in some TeX document. The default value is `false`.

outputFilename Sets the output file name. If not set, the output is printed to standard output. By default it's not set.

generateSourceListing Instead of generating a manual from the manual entries, DOC++ will generate a source code listing. This listing contains all *normal* C or C++ comments typeset in TeX quality. Every line is preceeded with its line number. The default value is `false`.

hideIndex Do not print the index when starting a new section. The default value is `false`.

3.2 Manual Entries

Names

Just like in JavaDoc, the documentation for DOC++ is contained in special versions of Java, C or C++ comments. These are comments with the format:

- `/** ... */`
- `/// ...`

Note that DOC++ comments are only those with a double asterisk '`/**`' or '`//`' respectively. We shall refer to such a comment as a *DOC++ comment*. Each DOC++ comment is used to specify the documentation for the *subsequent* declaration (of a variable, class, etc.).

Every DOC++ comment defines a *manual entry*. A manual entry consists in documentation provided in the DOC++ comment and some information from the subsequent declaration, if available.

Trailing comments can be used to define manual entries too by turning on the Quantel extensions. This is done by using the '`--quantel`' (or '`-Q`') command line option.

Manual entries are structured into various *fields*. Some of them are automatically filled in by DOC++ while the others may be specified by the documentation writer. Here is the list of the fields of manual entries:

Field name	provider	description
`args`	DOC++	depends on source code
`author`	user	author
`deprecated`	user	doc for deprecated functions
`doc`	user	long documentation
`exception`	user	doc for exceptions thrown by a function
`field`	user	doc for fields documentation
`friends`	DOC++	doc for entry's friends
`invariant`	user	doc for invariants
`memo`	user	short documentation
`name`	both	depends on source code
`param`	user	doc of parameters of a function
`postcondition`	user	doc for postconditions
`precondition`	user	doc for preconditions
`return`	user	doc of return value of a function
`see`	user	cross reference
`since`	user	version since the function exists
`type`	DOC++	depends on source code
`version`	user	version

Except for explicit manual entries, the first three fields will generally be filled automatically by DOC++. How they are filled depends on the *category* of a manual entry, which is determined by the source code following a DOC++ comment. Generally they contain the entire signature of the subsequent declaration. The following table lists all categories of manual entries and how the fields are filled:

Category	`@type`	`@name`	`@args`
macro	#define	name	[argument list]
variable	Type	name	-
function/method	Return type	name	arguments list [exceptions]
union/enum	union/enum	name	-
class/struct	class/struct	name	[derived classes]
interface	interface	name	[extended interfaces]

In any case '`@name`' contains the name of the declaration to be documented. It will be included in the table of contents.

The remaining fields are filled from the text in the DOC++ comment. Except for the '`@doc`' and '`@memo`' field, the text for a field must be preceeded by the field name in the beginning of a line of the DOC++ comment. The subsequent text up to the next occurrence of a field name is used for the field. Field '`@name`' is an exception in that only the remaining text in the same line is used to fill the field. As an example:

```
@author Snoopy
```

is used to fill the '`@author`' field with the text "Snoopy".

Text that is not preceeded by a field name is used for the '`@doc`' field. The very first text in a DOC++ comment up to the first occurrence of character '.' is also copied to the '`@memo`' field. This may be overridden by explicitly specifying a '`@memo`' field. In this case also characters '.' are allowed.

The '`@type`', '`@args`' and '`@doc`' fields may not be filled explicitly.

3.2.1 Manual Entry Fields

All fields names start with a at-sign (`@`) or with a backslash (\).

author Author

deprecated Warning for deprecated functions

doc Entry's documentation

exception Documentation for exception thrown by a function

invariant Documentation for invariants

memo Short description of the entry

param Documentation of parameter of a function. Multiple `param` fields are allowed.

postcondition Documentation for postconditions

precondition Documentation for preconditions

return Documentation of return value of a function. Multiple `return` fields are allowed.

see "See also" references. Multiple `see` fields are allowed.

since Version since the function exists

version Version

3.3 Structure

DOC++ automatically imposes a hierarchical stucture to the manual entries for classes, structs, unions, enums and interfaces, in that it organizes members of such as sub-entries.

Additionally DOC++ provides means for manually creating subentries to a manual entry. This is done via *documentation scopes*. A documentation scope is defined using a pair of brackets:

```
//@{
    ...
//@}
```

just like variable scopes in C, C++ or Java. Instead of "`//@{`" and "`//@}`" one can also use "`/*@{*/`" and "`/*@}*/`". All the manual entries within a documentation scope are organized as subentries of the manual entry preceeding the opening bracket of the scope, but only if this is an explicit manual entry. Otherwise a dummy explicit manual entry is created.

In addition to this, Java allows the programmer to organize classes hierarchically by means of "`packages`". Packages are directly represented in the manual entry hierarchy generated by DOC++. When a DOC++ comment is found before a '`package`' statement, the documentation is added to the package's manual entry. This functionality as well as documentation scopes are extensions to the features of JavaDoc.

Similar to Java's packages, C++ comes with the "`namespace`" concept. The idea is to group various class, functions, etc. declarations into different universes. DOC++ deals with namespaces in the same way it does with packages.

3.4 File Inclusion

There is one more special type of comments for DOC++, namely "`//@Include: <files>`" and "`/*@Include: <files>*/`". When any of such comments is parsed, DOC++ will read the specified files in the order they are given. Also wildcards using "`*`" are allowed. It is good practice to use one input file only and include all documented files using such comments, especially when explicit manual entries are used for structuring the documentation. This text is a good example for such a documentation.

3.5 Tags

#foo# Corresponds to the TeX "\verb!foo!", i.e. outputs "foo" verbatim.

@filename foo Force the manual entry to go to the specified file.

{@link foo name} Make a reference to a manual entry with name "foo". The link name is "name". The second parameter is optional.

\Date Insert the current date and time (according to the current locale settings) into the documentation.

\IMG{filename} Insert an image into the documentation.

\IMG[HTML parameters]{filename} Insert an image into the documentation.

\IMG[HTML parameters][TeX parameters]{filename} Insert an image into the documentation. Note that if you are using TeX output you'll have to tell DOC++ to include the 'graphicx' TeX package.

\Label{foo} Make a label with the name "name".

\Ref{foo} Make a reference to a manual entry with name "foo".

\URL{foo} Make a link to the WWW page "foo".

\URL[name]{foo} Make a link to the WWW page "foo", with the name "name".

\TEX{foo} Include the "foo" TeX code into your document. For HTML output DOC++ will run TeX to process it, produce GIFs out of it and includes them into the HTML document. NOTE: this requires 'latex', 'dvips', 'ghostscript' and 'ppmtools' to be correctly installed on your system!

\includegraphics Same as '\IMG'.

\today Same as '\Date'.

3.6

Text Formatting

Names

DOC++ provides both HTML and TeX output. Both languages have formatting macros which are more or less powerful. The idea of DOC++ is to be able to generate both ouput formats from a single source. Hence, it is not possible to rely on the full functionality of either formatting macros. Instead, DOC++ supports a subset of each set of macros, that has proved to suffice for most applications. However, in one run of DOC++ the user must decide for the formating macros to use. The subset of each macro packet is listed in the following subsections. If one uses only one of the subsets, good looking output can be expected for both formats.

3.6.1

supported TeX macros

This is the subset of TeX formatting instructions provided by DOC++:

`$...$` math mode for inline equations

`\[...\]` display math mode

\# to output character "#"

_ to output character "_"

**\ ** to output character " "

\em only to be used as "`{\em ...}`" or "`\emph{...}`"

\bf only to be used as "`{\bf ...}`" or "`\textbf{...}`"

\it only to be used as "`{\it ...}`" or "`\textit{...}`"

\tt only to be used as "`{\tt ...}`" or "`\texttt{...}`"

\tiny only to be used as "`{\tiny ...}`"

\scriptsize only to be used as "`{\scriptsize ...}`"

\footnotesize only to be used as "`{\footnotesize ...}`"

\small only to be used as "`{\small ...}`"

\large only to be used as "`{\large ...}`"

\Large only to be used as "`{\Large ...}`"

\LARGE only to be used as "{\LARGE ...}"

\huge only to be used as "{\huge ...}"

\Huge only to be used as "{\Huge ...}"

\HUGE only to be used as "{\HUGE ...}"

\hline only to be used as "\hline"

center i.e. "\begin{center} ... \end{center}"

flushleft i.e. "\begin{flushleft} ... \end{flushleft}"

flushright i.e. "\begin{flushright} ... \end{flushright}"

verbatim i.e. "\begin{verbatim} ... \end{verbatim}"

tabular i.e. "\begin{tabular}{lll} ...&...\\ ... \end{tabular}"

array i.e. "\begin{array}{lll} ...&...\\ ... \end{array}"

itemize i.e. "\begin{itemize} \item ... ... \end{itemize}"

enumerate i.e. "\begin{enumerate} \item ... ... \end{enumerate}"

description i.e. "\begin{description} \item[...] ... ... \end{description}"

equation i.e. "\begin{equation} ... \end{equation}"

equation array i.e. "\begin{eqnarray} ... \end{eqnarray}"

When writing your documentation using only this, you can be sure to get reasonable TeX **and** HTML documentation for your code.

3.6.2

supported HTML macros

This is the subset of HTML formatting instructions provided by DOC++:

**
** new line

<P> paragraph

<EM> ... </EM> emphasize

<I> ... </I> italic

<B> ... </B> bold face

<STRONG> ... </STRONG> bold face

<TT> ... </TT>

<PRE> ... </PRE> verbatim

<CODE> ... </CODE> verbatim

<OL> ... </OL> enumerations

<DL> ... </DL> description

<DT>

<DD>

<UL> ... </UL> itemize

<LL> ... </LL> enumerations

<LI>

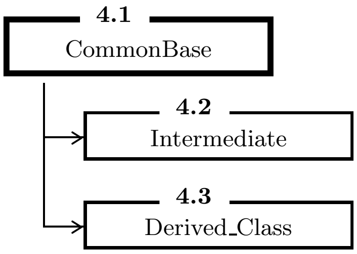
4 Example

Names

4.1 class **CommonBase**

Common base class.

Inheritance



Public Members

Protected Members

Common base class. This could be a long documentation for the common base class. Note that protected members are displayed after public ones, even if they appear in another order in the source code.

This is how this documentation has been generated:

```
/** Common base class.
    This could be a long documentation for the common base class.
    Note that protected members are displayed after public ones, even if they
    appear in another order in the source code.
    This is how this documentation has been generated:
* /

class CommonBase {
 private:
  /// this member shows up only if you call DOC++ with `--private' option
  int privateMember();

 protected:
  /// a protected member variable
  double variable;

 public:
  /// a public member function showing links to argument and type classes
  const Derived_Class& getB(const Intermediate& c) const;
};
```

4.1.3

const Derived_Class& **getB** (constIntermediate& c) const

a public member function showing links to argument and type classes

a public member function showing links to argument and type classes

4.1.2

double **variable**

a protected member variable

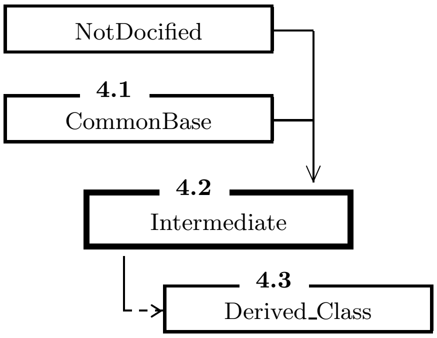
a protected member variable

4.2

class **Intermediate** : public CommonBase, public NotDocified

Just to make the class graph look more interesting.

Inheritance



Just to make the class graph look more interesting. Here we show multiple inheritance from one docified class and a nondocified one.

This is how this documentation has been generated:

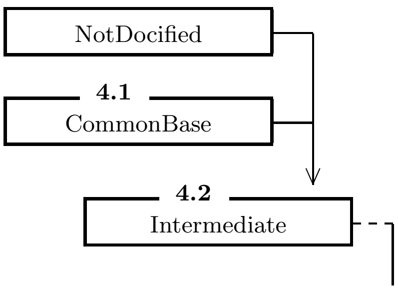
```
/** Just to make the class graph look more interesting.
    Here we show multiple inheritance from one docified class and a nondocified
    one.

    This is how this documentation has been generated:
* /

class Intermediate : public CommonBase, public NotDocified {
  };
```

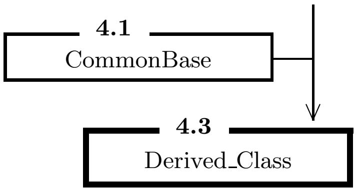
4.3

class **Derived_Class** : public CommonBase, protected Intermediate

A derived class.

Inheritance





Public Members

A derived class. Here we show multiple inheritance from two docified classes. This example shows how to structure the members of a class, if desired.

This is how this documentation has been generated:

```
/** A derived class.
    Here we show multiple inheritance from two docified classes.
    This example shows how to structure the members of a class, if desired.

    This is how this documentation has been generated:
* /

class Derived_Class : public CommonBase, protected Intermediate {
 public:
  /**@name parameters * /
  //@{
  /// the first parameter
  double a;
  /// a second parameter
  int b;
  //@}

  /**@name methods * /
  //@{
  /// constructor
  /** This constructor takes two arguments, just for the sake of
      demonstrating how documented members are displayed by DOC++.
      @param a this is good for many things
      @param b this is good for nothing
  * /
  DerivedClass(double a, int b);
  /// destructor
  ~DerivedClass();
  //@}
};
```

4.3.1

parameters

Names

4.3.1.1

double **a**

the first parameter

the first parameter

4.3.1.2

int **b**

a second parameter

a second parameter

4.3.2

methods

Names

4.3.2.1

Derived_Class (double a, int b)

Constructor.

Constructor. This constructor takes two arguments, just for the sake of demonstrating how documented members are displayed by DOC++.

Parameters: `a` this is good for many things
`b` this is good for nothing

4.3.2.2

~Derived_Class ()

destructor

destructor

4.4

int **function** (const DerivedClass& c)

A global function.

A global function. As promised, not only classes and members can be documented with DOC++. This is an example for how to document global scope functions. You'll notice that there is no technical difference to documenting member functions. The same applies to variables or macros.

This is how this documentation has been generated:

```
/** A global function.
    As promised, not only classes and members can be documented with DOC++.
    This is an example for how to document global scope functions.
    You'll notice that there is no technical difference to documenting
    member functions. The same applies to variables or macros.

    @param c reference to input data object
    @return whatever
    @author Snoopy
    @version 3.3.12
    @see Derived_Class
 * /

    int function(const DerivedClass& c);
```

Return Value:	`whatever`
Parameters:	`c` reference to input data object
See Also:	Derived_Class (→4.3, *page 22*)
Author:	Snoopy
Version:	3.3.12

5

General Informations

DOC++ is originally written by Roland Wunderling and Malte Zoeckler, further improved by Michael Meeks, currently maintained by Dragos Acostachioaie <dragos@iname.com>.

DOC++ is free software, but is protected by the GNU General Public License. Please see http://www.gnu.org/copyleft/gpl.html for details. The files output by DOC++ are not covered by this license.

You may want to consult the DOC++'s Web page: http://docpp.sourceforge.net. The DOC++ source tarball and pre-compiled binaries can be downloaded from http://docpp.sourceforge.net/download.html.

If you are using DOC++, you may want to subscribe to the DOC++ mailing list. To subscribe, send an empty mail to docpp-subscribe@egroups.com. Please send bug reports, suggestions, feedback, patches, or anything else that you think, to docpp@egroups.com.

6 Installation Instructions

The steps in order to compile this package are:

1. ‘cd’ to the directory containing the package’s source code and type ‘./configure’ to configure the package for your system. If you’re using ‘csh’ on an old version of System V, you might need to type ‘sh ./configure’ instead to prevent ‘csh’ from trying to execute ‘configure’ itself.

Running ‘configure’ takes awhile. While running, it prints some messages telling which features it is checking for.

The ‘configure’ script is generated from ‘configure.in’ by GNU’s autoconf and it attempts to guess correct values for various system-dependent variables used during compilation. It also creates the Makefiles needed to compile the package and a ‘.h’ file containing system-dependent definitions;

2. Type ‘make’ to compile the package;

3. Type ‘make install’ to install the package. This operation should be done logged on as root;

4. You can remove the object files from the source directory by typing ‘make clean’. To also remove the files that ‘configure’ created (so you can compile the package for a different kind of system), type ‘make distclean’;

5. Type ‘make uninstall’ to remove the package from the destination directories.

Installation directories

By default, ‘make install’ will install the package’s binaries in ‘/usr/bin’. You can specify an installation prefix other than ‘/usr’ by giving ‘configure’ the option ‘–prefix=PATH’.

Other features and options

–enable-debug to enable generation of debugging informations; for other options, type ‘./configure –help’.

7 Frequently Asked Questions

Q: How can I group a number of entries?

A: Right as in this example:

```
/**@name comparison operators * /
//@{
  /// equal
  bool operator==(const Date& cmpDate);
  ///
  bool operator!=(const Date& cmpDate);
  /// less
  bool operator<(const Date& cmpDate);
  /// greater
  bool operator>(const Date& cmpDate);
//@}
```

Q: How can I influence the order of the entries?

A: The order of class members is the same as in the class declaration. The order of the entries in the table of contents is the order in which DOC++ reads the classes. Hence, typing "doc++ *" yields an alphabetically ordered list. You may also use "`//@Include:`" to read your files in the desired order.

Q: How can I change fonts/borders/whatever in TeX output?

A: Edit the file 'docxx.sty' (there is no documentation about how to do this, sorry :-( ).

Q: What do the blue and grey balls in the HTML-output mean?

A: Entries that have a doc-string (not only memo) have a blue ball. Clicking on this ball gets you to the documentation.

Q: How can I avoid scrolling all the way down to the class' documentation?

A: Click on the class name to jump there.

Q: How can I get other paper formats for the TeX output?

A: Try the '-e' options. E.g.: with "-eo a4paper", the 'a4paper' option will be set for the documentstyle; with "-ep a4wide" a "`\usepackage{a4wide}`" will be inserted before "`\begin{document}`". Finally, one can provide a completely own TeX environment setup using the '-ef' option.

Q: I have the following:

```
///
    class A { ... } a;
```

Why do I get scrambled results ?

A: DOC++ does not know what you intend to document, the class A or the variable a. Solution: Split up class and variable declarations like this:

```
    ///
class A { ... };
    ///
A a;
```

Q: I have the following old C typedef:
`/** ... */`
`typedef struct a { ... } a_t ;`
Why do I get scrambled results?

A: This is the same problem as above. The solution is also equivalent:
`/** ... */`
`struct a { ... };`
`/** ... */`
`typedef struct a a_t ;`

Q: Is there a way to make the equation font larger in the HTML output?

A: Sure, more than one. You may use "`\large`" or so within the equations. Or you may use the '-eo 12pt' option to render all GIFs in 12pt instead of 10pt. Or you may use you own TeX environment with '-ef' option to setup all fonts as desired.

Q: Why does DOC++ fail to build GIFs for my formulae?

A: There are two typical kinds of failure. One is, that you don't have setup your path to find the 'ppmtools', 'gs' or 'latex'. The other is that 'latex' fails to process your formulae. Check the file 'dxxgifs.tex' in your html directory to see what LaTeX tries to process.

Class Graph