Learning doxygen for source code documentation

Arpan Sen

July 29, 2008

Maintaining and adding new features to legacy systems developed using `C/C++` is a daunting task. Fortunately, doxygen—a documentation system for the `C/C++`, Java™, Python, and other programming languages—can help. Discover the features of doxygen in the context of projects using `C/C++` as well as how to document code using doxygen-defined tags.

Maintaining and adding new features to legacy systems developed using `C/C++` is a daunting task. There are several facets to the problem—understanding the existing class hierarchy and global variables, the different user-defined types, and function call graph analysis, to name a few. This article discusses several features of doxygen, with examples in the context of projects using `C/C++`. However, doxygen is flexible enough to be used for software projects developed using the Python, Java, PHP, and other languages, as well. The primary motivation of this article is to help extract information from `C/C++` sources, but it also briefly describes how to document code using doxygen-defined tags.

Installing doxygen

You have two choices for acquiring doxygen. You can download it as a pre-compiled executable file, or you can check out sources from the SVN repository and build it. Listing 1 shows the latter process.

Listing 1. Install and build doxygen sources

```
bash-2.05$ svn co https://doxygen.svn.sourceforge.net/svnroot/doxygen/trunk doxygen-svn

bash-2.05$ cd doxygen-svn
bash-2.05$ ./configure –prefix=/home/user1/bin
bash-2.05$ make

bash-2.05$ make install
```

Note that the configure script is tailored to dump the compiled sources in /home/user1/bin (add this directory to the PATH variable after the build), as not every UNIX® user has permission to write to the /usr folder. Also, you need the `svn` utility to check out sources.

Generating documentation using doxygen

To use doxygen to generate documentation of the sources, you perform three steps.

Generate the configuration file

At a shell prompt, type the command **`doxygen -g`**. This command generates a text-editable configuration file called *Doxyfile* in the current directory. You can choose to override this file name, in which case the invocation should be `doxygen -g <`*`user-specified file name`*`>`, as shown in Listing 2.

Listing 2. Generate the default configuration file

```
bash-2.05b$ doxygen -g
Configuration file 'Doxyfile' created.
Now edit the configuration file and enter
  doxygen Doxyfile
to generate the documentation for your project
bash-2.05b$ ls Doxyfile
Doxyfile
```

Edit the configuration file

The configuration file is structured as `<TAGNAME>` = `<VALUE>`, similar to the Make file format. Here are the most important tags:

- **`<OUTPUT_DIRECTORY>`:** You must provide a directory name here—for example, /home/user1/documentation—for the directory in which the generated documentation files will reside. If you provide a nonexistent directory name, doxygen creates the directory subject to proper user permissions.
- **`<INPUT>`:** This tag creates a space-separated list of all the directories in which the `C/C++` source and header files reside whose documentation is to be generated. For example, consider the following snippet:

  ```
  INPUT = /home/user1/project/kernel /home/user1/project/memory
  ```

  In this case, doxygen would read in the `C/C++` sources from these two directories. If your project has a single source root directory with multiple sub-directories, specify that folder and make the `<RECURSIVE>` tag **Yes**.
- **`<FILE_PATTERNS>`:** By default, doxygen searches for files with typical `C/C++` extensions such as *.c, .cc, .cpp, .h,* and *.hpp.* This happens when the `<FILE_PATTERNS>` tag has no value associated with it. If the sources use different naming conventions, update this tag accordingly. For example, if a project convention is to use *.c86* as a `C` file extension, add this to the `<FILE_PATTERNS>` tag.
- **`<RECURSIVE>`:** Set this tag to **Yes** if the source hierarchy is nested and you need to generate documentation for `C/C++` files at all hierarchy levels. For example, consider the root-level source hierarchy /home/user1/project/kernel, which has multiple sub-directories such as /home/user1/project/kernel/vmm and /home/user1/project/kernel/asm. If this tag is set to *Yes*, doxygen recursively traverses the hierarchy, extracting information.
- **`<EXTRACT_ALL>`:** This tag is an indicator to doxygen to extract documentation even when the individual classes or functions are undocumented. You must set this tag to **Yes**.
- **`<EXTRACT_PRIVATE>`:** Set this tag to **Yes**. Otherwise, private data members of a class would not be included in the documentation.

- **`<EXTRACT_STATIC>`:** Set this tag to **Yes**. Otherwise, static members of a file (both functions and variables) would not be included in the documentation.

Listing 3 shows an example of a Doxyfile.

Listing 3. Sample doxyfile with user-provided tag values

```
OUTPUT_DIRECTORY = /home/user1/docs
EXTRACT_ALL = yes
EXTRACT_PRIVATE = yes
EXTRACT_STATIC = yes
INPUT = /home/user1/project/kernel
#Do not add anything here unless you need to. Doxygen already covers all
#common formats like .c/.cc/.cxx/.c++/.cpp/.inl/.h/.hpp
FILE_PATTERNS =
RECURSIVE = yes
```

Run doxygen

Run doxygen in the shell prompt as `doxygen Doxyfile` (or with whatever file name you've chosen for the configuration file). Doxygen issues several messages before it finally produces the documentation in Hypertext Markup Language (HTML) and Latex formats (the default). In the folder that the `<OUTPUT_DIRECTORY>` tag specifies, two sub-folders named *html* and *latex* are created as part of the documentation-generation process. Listing 4 shows a sample doxygen run log.

Listing 4. Sample log output from doxygen

```
Searching for include files...
Searching for example files...
Searching for images...
Searching for dot files...
Searching for files to exclude
Reading input files...
Reading and parsing tag files
Preprocessing /home/user1/project/kernel/kernel.h
…
Read 12489207 bytes
Parsing input...
Parsing file /project/user1/project/kernel/epico.cxx
…
Freeing input...
Building group list...
..
Generating docs for compound MemoryManager::ProcessSpec
…
Generating docs for namespace std
Generating group index...
Generating example index...
Generating file member index...
Generating namespace member index...
Generating page index...
Generating graph info page...
Generating search index...
Generating style sheet...
```

Documentation output formats

Doxygen can generate documentation in several output formats other than HTML. You can configure doxygen to produce documentation in the following formats:

- **UNIX man pages:** Set the `<GENERATE_MAN>` tag to **Yes**. By default, a sub-folder named *man* is created within the directory provided using `<OUTPUT_DIRECTORY>`, and the documentation is generated inside the folder. You must add this folder to the MANPATH environment variable.
- **Rich Text Format (RTF):** Set the `<GENERATE_RTF>` tag to **Yes**. Set the `<RTF_OUTPUT>` to wherever you want the .rtf files to be generated—by default, the documentation is within a sub-folder named *rtf* within the OUTPUT_DIRECTORY. For browsing across documents, set the `<RTF_HYPERLINKS>` tag to **Yes**. If set, the generated .rtf files contain links for cross-browsing.
- **Latex:** By default, doxygen generates documentation in Latex and HTML formats. The `<GENERATE_LATEX>` tag is set to **Yes** in the default Doxyfile. Also, the `<LATEX_OUTPUT>` tag is set to Latex, which implies that a folder named *latex* would be generated inside OUTPUT_DIRECTORY, where the Latex files would reside.
- **Microsoft® Compiled HTML Help (CHM) format:** Set the `<GENERATE_HTMLHELP>` tag to **Yes**. Because this format is not supported on UNIX platforms, doxygen would only generate a file named *index.hhp* in the same folder in which it keeps the HTML files. You must feed this file to the HTML help compiler for actual generation of the .chm file.
- **Extensible Markup Language (XML) format:** Set the `<GENERATE_XML>` tag to **Yes**. (Note that the XML output is still a work in progress for the doxygen team.)

Listing 5 provides an example of a Doxyfile that generates documentation in all the formats discussed.

Listing 5. Doxyfile with tags for generating documentation in several formats

```
#for HTML
GENERATE_HTML = YES
HTML_FILE_EXTENSION = .htm

#for CHM files
GENERATE_HTMLHELP = YES

#for Latex output
GENERATE_LATEX = YES
LATEX_OUTPUT = latex

#for RTF
GENERATE_RTF = YES
RTF_OUTPUT = rtf
RTF_HYPERLINKS = YES

#for MAN pages
GENERATE_MAN = YES
MAN_OUTPUT = man
#for XML
GENERATE_XML = YES
```

Special tags in doxygen

Doxygen contains a couple of special tags.

Preprocessing C/C++ code

First, doxygen must preprocess `C/C++` code to extract information. By default, however, it does only partial preprocessing -- conditional compilation statements (`#if…#endif`) are evaluated, but macro expansions are not performed. Consider the code in Listing 6.

Listing 6. Sample C++ code that makes use of macros

```
#include <cstring>
#include <rope>

#define USE_ROPE

#ifdef USE_ROPE
  #define STRING std::rope
#else
  #define STRING std::string
#endif

static STRING name;
```

With `<USE_ROPE>` defined in sources, generated documentation from doxygen looks like this:

```
Defines
    #define USE_ROPE
    #define STRING std::rope

Variables
    static STRING name
```

Here, you see that doxygen has performed a conditional compilation but has not done a macro expansion of `STRING`. The `<ENABLE_PREPROCESSING>` tag in the Doxyfile is set by default to *Yes*. To allow for macro expansions, also set the `<MACRO_EXPANSION>` tag to **Yes**. Doing so produces this output from doxygen:

```
Defines
   #define USE_ROPE
    #define STRING std::string

Variables
    static std::rope name
```

If you set the `<ENABLE_PREPROCESSING>` tag to *No*, the output from doxygen for the earlier sources looks like this:

```
Variables
    static STRING name
```

Note that the documentation now has no definitions, and it is not possible to deduce the type of `STRING`. It thus makes sense always to set the `<ENABLE_PREPROCESSING>` tag to **Yes**.

As part of the documentation, it might be desirable to expand only specific macros. For such purposes, along setting `<ENABLE_PREPROCESSING>` and `<MACRO_EXPANSION>` to **Yes**, you must set the `<EXPAND_ONLY_PREDEF>` tag to **Yes** (this tag is set to *No* by default) and provide the macro details as part of the `<PREDEFINED>` or `<EXPAND_AS_DEFINED>` tag. Consider the code in Listing 7, where only the macro `CONTAINER` would be expanded.

Listing 7. C++ source with multiple macros

```
#ifdef USE_ROPE
  #define STRING std::rope
#else
  #define STRING std::string
#endif

#if ALLOW_RANDOM_ACCESS == 1
  #define CONTAINER std::vector
#else
  #define CONTAINER std::list
#endif

static STRING name;
static CONTAINER gList;
```

Listing 8 shows the configuration file.

Listing 8. Doxyfile set to allow select macro expansions

```
ENABLE_PREPROCESSING = YES
MACRO_EXPANSION = YES
EXPAND_ONLY_PREDEF = YES
EXPAND_AS_DEFINED = CONTAINER
…
```

Here's the doxygen output with only `CONTAINER` expanded:

```
Defines
#define STRING   std::string
#define CONTAINER   std::list

Variables
static STRING name
static std::list gList
```

Notice that only the `CONTAINER` macro has been expanded. Subject to `<MACRO_EXPANSION>` and `<EXPAND_AS_DEFINED>` both being *Yes*, the `<EXPAND_AS_DEFINED>` tag selectively expands only those macros listed on the right-hand side of the equality operator.

As part of preprocessing, the final tag to note is `<PREDEFINED>`. Much like the same way you use the `-D` switch to pass the G++ compiler preprocessor definitions, you use this tag to define macros. Consider the Doxyfile in Listing 9.

Listing 9. Doxyfile with macro expansion tags defined

```
ENABLE_PREPROCESSING = YES
MACRO_EXPANSION = YES
EXPAND_ONLY_PREDEF = YES
EXPAND_AS_DEFINED =
PREDEFINED = USE_ROPE= \
                             ALLOW_RANDOM_ACCESS=1
```

Here's the doxygen-generated output:

```
Defines
#define USE_CROPE
#define STRING   std::rope
#define CONTAINER   std::vector

Variables
static std::rope name
static std::vector gList
```

When used with the `<PREDEFINED>` tag, macros should be defined as `<`*`macro name`*`>=<`*`value`*`>`. If no value is provided—as in the case of simple `#define`—just using `<`*`macro name`*`>=<`*`spaces`*`>` suffices. Separate multiple macro definitions by spaces or a backslash (`\`).

Excluding specific files or directories from the documentation process

In the `<EXCLUDE>` tag in the Doxyfile, add the names of the files and directories for which documentation should not be generated separated by spaces. This comes in handy when the root of the source hierarchy is provided and some sub-directories must be skipped. For example, if the root of the hierarchy is src_root and you want to skip the examples/ and test/memoryleaks folders from the documentation process, the Doxyfile should look like Listing 10.

Listing 10. Using the EXCLUDE tag as part of the Doxyfile

```
INPUT = /home/user1/src_root
EXCLUDE = /home/user1/src_root/examples /home/user1/src_root/test/memoryleaks
…
```

Generating graphs and diagrams

By default, the Doxyfile has the `<CLASS_DIAGRAMS>` tag set to **Yes**. This tag is used for generation of class hierarchy diagrams. The following tags in the Doxyfile deal with generating diagrams:

- **`<CLASS_DIAGRAMS>`:** The default tag is set to **Yes** in the Doxyfile. If the tag is set to **No**, diagrams for inheritance hierarchy would not be generated.
- **`<HAVE_DOT>`:** If this tag is set to **Yes**, doxygen uses the dot tool to generate more powerful graphs, such as collaboration diagrams that help you understand individual class members and their data structures. Note that if this tag is set to **Yes**, the effect of the `<CLASS_DIAGRAMS>` tag is nullified.
- **`<CLASS_GRAPH>`:** If the `<HAVE_DOT>` tag is set to **Yes** along with this tag, the inheritance hierarchy diagrams are generated using the `dot` tool and have a richer look and feel than what you'd get by using only `<CLASS_DIAGRAMS>`.
- **`<COLLABORATION_GRAPH>`:** If the `<HAVE_DOT>` tag is set to **Yes** along with this tag, doxygen generates a collaboration diagram (apart from an inheritance diagram) that shows the individual class members (that is, containment) and their inheritance hierarchy.

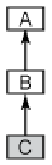
Listing 11 provides an example using a few data structures. Note that the `<HAVE_DOT>`, `<CLASS_GRAPH>`, and `<COLLABORATION_GRAPH>` tags are all set to *Yes* in the configuration file.

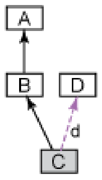
Listing 11. Interacting C++ classes and structures

```
struct D {
  int d;
};

class A {
  int a;
};

class B : public A {
  int b;
};

class C : public B {
  int c;
  D d;
};
```

Figure 1 shows the output from doxygen.

Figure 1. The Class inheritance graph and collaboration graph generated using the dot tool





Code documentation style

So far, you've used doxygen to extract information from code that is otherwise undocumented. However, doxygen also advocates documentation style and syntax, which helps it generate more detailed documentation. This section discusses some of the more common tags doxygen advocates using as part of `C/C++` code. For further details, see Related topics.

Every code item has two kinds of descriptions: one brief and one detailed. Brief descriptions are typically single lines. Functions and class methods have a third kind of description known as the *in-body description,* which is a concatenation of all comment blocks found within the function body. Some of the more common doxygen tags and styles of commenting are:

- **Brief description:** Use a single-line `C++` comment, or use the `<\brief>` tag.
- **Detailed description:** Use JavaDoc-style commenting `/** … test … */` (note the two asterisks [`*`] in the beginning) or the Qt-style `/*! … text … */`.
- **In-body description:** Individual `C++` elements like classes, structures, unions, and namespaces have their own tags, such as `<\class>`, `<\struct>`, `<\union>`, and `<\namespace>`.

To document global functions, variables, and enum types, the corresponding file must first be documented using the `<\file>` tag. Listing 12 provides an example that discusses item 4 with a function tag (`<\fn>`), a function argument tag (`<\param>`), a variable name tag (`<\var>`), a tag for `#define` (`<\def>`), and a tag to indicate some specific issues related to a code snippet (`<\warning>`).

Listing 12. Typical doxygen tags and their use

```
/*! \file globaldecls.h
      \brief Place to look for global variables, enums, functions
           and macro definitions
  */

/** \var const int fileSize
      \brief Default size of the file on disk
  */
const int fileSize = 1048576;

/** \def SHIFT(value, length)
      \brief Left shift value by length in bits
  */
#define SHIFT(value, length) ((value) << (length))

/** \fn bool check_for_io_errors(FILE* fp)
      \brief Checks if a file is corrupted or not
      \param fp Pointer to an already opened file
      \warning Not thread safe!
  */
bool check_for_io_errors(FILE* fp);
```

Here's how the generated documentation looks:

```
Defines
#define SHIFT(value, length)   ((value) << (length))
             Left shift value by length in bits.

Functions
bool check_for_io_errors (FILE *fp)
        Checks if a file is corrupted or not.

Variables
const int fileSize = 1048576;
Function Documentation
bool check_for_io_errors (FILE* fp)
Checks if a file is corrupted or not.

Parameters
              fp: Pointer to an already opened file

Warning
Not thread safe!
```

Conclusion

This article discusses how doxygen can extract a lot of relevant information from legacy `C/C++` code. If the code is documented using doxygen tags, doxygen generates output in an easy-to-read format. Put to good use, doxygen is a ripe candidate in any developer's arsenal for maintaining and managing legacy systems.

Related topics

- The doxygen site site contains a very informative manual and several articles about doxygen.
- Download doxygen.
- IBM trial software: Build your next development project with software for download directly from developerWorks.