Documenting the code

This chapter covers two topics:

1. How to put comments in your code such that doxygen incorporates them in the documentation it generates. This is further detailed in the **next section**.
2. Ways to structure the contents of a comment block such that the output looks good, as explained in section **Anatomy of a comment block**.

Special comment blocks

A special comment block is a C or C++ style comment block with some additional markings, so doxygen knows it is a piece of structured text that needs to end up in the generated documentation. The **next** section presents the various styles supported by doxygen.

For Python, VHDL, Fortran, and Tcl code there are different commenting conventions, which can be found in sections **Comment blocks in Python**, **Comment blocks in VHDL**, **Comment blocks in Fortran**, and **Comment blocks in Tcl** respectively.

Comment blocks for C-like languages (C/C++/C#/Objective-C/PHP/Java)

For each entity in the code there are two (or in some cases three) types of descriptions, which together form the documentation for that entity; a *brief* description and *detailed* description, both are optional. For methods and functions there is also a third type of description, the so called *in body* description, which consists of the concatenation of all comment blocks found within the body of the method or function.

Having more than one brief or detailed description is allowed (but not recommended, as the order in which the descriptions will appear is not specified).

As the name suggest, a brief description is a short one-liner, whereas the detailed description provides longer, more detailed documentation. An "in body" description can also act as a detailed description or can describe a collection of implementation details. For the HTML output brief descriptions are also used to provide tooltips at places where an item is referenced.

There are several ways to mark a comment block as a detailed description:

1. You can use the JavaDoc style, which consist of a C-style comment block starting with two *'s, like this:

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

2. or you can use the Qt style and add an exclamation mark (!) after the opening of a C-style comment block, as shown in this example:

```
/*!
 * ... text ...
 */
```

In both cases the intermediate *'s are optional, so

```
/*!
 ... text ...
*/
```

is also valid.

3. A third alternative is to use a block of *at least two* C++ comment lines, where each line starts with an additional slash or an exclamation mark. Here are examples of the two cases:

```
///
/// ... text ...
///
```

or

```
//!
//!... text ...
//!
```

Note that a blank line ends a documentation block in this case.

4. Some people like to make their comment blocks more visible in the documentation. For this purpose you can use the following:

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

(note the 2 slashes to end the normal comment block and start a special comment block).

or

```
/////////////////////////////////////////////
/// ... text ...
/////////////////////////////////////////////
```

For the brief description there are also several possibilities:

1. One could use the **\brief** command with one of the above comment blocks. This command ends at the end of a paragraph, so the detailed description follows after an empty line.

    Here is an example:

    ```
    /*! \brief Brief description.
     *         Brief description continued.
     *
     *  Detailed description starts here.
     */
    ```

2. If **JAVADOC_AUTOBRIEF** is set to YES in the configuration file, then using JavaDoc style comment blocks will automatically start a brief description which ends at the first dot followed by a space or new line. Here is an example:

    ```
    /** Brief description which ends at this dot. Details follow
     *  here.
     */
    ```

    The option has the same effect for multi-line special C++ comments:

    ```
    /// Brief description which ends at this dot. Details follow
    /// here.
    ```

3. A third option is to use a special C++ style comment which does not span more than one line. Here are two examples:

    ```
    /// Brief description.
    /** Detailed description. */
    ```

    or

    ```
    //! Brief description.

    //! Detailed description
    //! starts here.
    ```

    Note the blank line in the last example, which is required to separate the brief description from the block containing the detailed description. The **JAVADOC_AUTOBRIEF** should also be set to NO for this case.

As you can see doxygen is quite flexible. If you have multiple detailed descriptions, like in the following example:

```
//! Brief description, which is
//! really a detailed description since it spans multiple lines.
/*! Another detailed description!
 */
```

They will be joined. Note that this is also the case if the descriptions are at different places in the code! In this case the order will depend on the order in which doxygen parses the code.

Unlike most other documentation systems, doxygen also allows you to put the documentation of members (including global functions) in front of the *definition*. This way the documentation can be placed in the source file instead of the header file. This keeps the header file compact, and allows the implementer of the members more direct access to the documentation. As a compromise the brief description could be placed before the declaration and the detailed description before the member definition.

Putting documentation after members

If you want to document the members of a file, struct, union, class, or enum, it is sometimes desired to place the documentation block after the member instead of before. For this purpose you have to put an additional < marker in the comment block. Note that this also works for the parameters of a function.

Here are some examples:

```
int var; /*!< Detailed description after the member */
```

This block can be used to put a Qt style detailed documentation block *after* a member. Other ways to do the same are:

```
int var; /**< Detailed description after the member */
```

or

```
int var; //!< Detailed description after the member
        //!<
```

or

```
int var; ///< Detailed description after the member
         ///<
```

Most often one only wants to put a brief description after a member. This is done as follows:

```
int var; //!< Brief description after the member
```

or

```
int var; ///< Brief description after the member
```

For functions one can use the **@param** command to document the parameters and then use [in], [out], [in,out] to document the direction. For inline documentation this is also possible by starting with the direction attribute, e.g.

```
void foo(int v /**< [in] docs for input parameter v. */);
```

Note that these blocks have the same structure and meaning as the special comment blocks in the previous section only the < indicates that the member is located in front of the block instead of after the block.

Here is an example of the use of these comment blocks:

```
/*! A test class */

class Test
{
  public:
    /** An enum type.
     *  The documentation block cannot be put after the enum!
     */
    enum EnumType
    {
      int EVal1,     /**< enum value 1 */
      int EVal2      /**< enum value 2 */
    };
    void member();   //!< a member function.

  protected:
    int value;       /*!< an integer value */
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

Warning

These blocks can only be used to document *members* and *parameters*. They cannot be used to document files, classes, unions, structs, groups, namespaces and enums themselves. Furthermore, the structural commands mentioned in the next section (like \class) are not allowed inside these comment blocks.

Examples

Here is an example of a documented piece of C++ code using the Qt style:

```
//!  A test class.
/*!
  A more elaborate class description.
*/

class Test
{
  public:

    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
              TVal1, /*!< Enum value TVal1. */
              TVal2, /*!< Enum value TVal2. */
              TVal3  /*!< Enum value TVal3. */
             }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    //! A constructor.
    /*!
      A more elaborate description of the constructor.
    */
    Test();

    //! A destructor.
    /*!
      A more elaborate description of the destructor.
    */
    ~Test();
```

```
    //! A normal member taking two arguments and returning an integer value.
    /*!
      \param a an integer argument.
      \param s a constant character pointer.
      \return The test results
      \sa Test(), ~Test(), testMeToo() and publicVar()
    */
    int testMe(int a,const char *s);

    //! A pure virtual member.
    /*!
      \sa testMe()
      \param c1 the first argument.
      \param c2 the second argument.
    */
    virtual void testMeToo(char c1,char c2) = 0;

    //! A public variable.
    /*!
      Details.
    */
    int publicVar;

    //! A function variable.
    /*!
      Details.
    */
    int (*handler)(int a,int b);
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

The brief descriptions are included in the member overview of a class, namespace or file and are printed using a small italic font (this description can be hidden by setting **BRIEF_MEMBER_DESC** to NO in the config file). By default the brief descriptions become the first sentence of the detailed descriptions (but this can be changed by setting the **REPEAT_BRIEF** tag to NO). Both the brief and the detailed descriptions are optional for the Qt style.

By default a JavaDoc style documentation block behaves the same way as a Qt style documentation block. This is not according the JavaDoc specification however, where the first sentence of the documentation block is automatically treated as a brief description. To enable this behavior you should set **JAVADOC_AUTOBRIEF** to YES in the configuration file. If you enable this option and want to put a dot in the middle of a sentence without ending it, you should put a backslash and a space after it. Here is an example:

```
/** Brief description (e.g.\ using only a few words). Details follow. */
```

Here is the same piece of code as shown above, this time documented using the JavaDoc style and **JAVADOC_AUTOBRIEF** set to YES:

```
/**
 *  A test class. A more elaborate class description.
 */

class Test
{
  public:

    /**
     * An enum.
     * More detailed enum description.
     */

    enum TEnum {
          TVal1, /**< enum value TVal1. */
          TVal2, /**< enum value TVal2. */
          TVal3  /**< enum value TVal3. */
          }
       *enumPtr, /**< enum pointer. Details. */
       enumVar;  /**< enum variable. Details. */

      /**
       * A constructor.
       * A more elaborate description of the constructor.
       */
      Test();

      /**
       * A destructor.
       * A more elaborate description of the destructor.
       */
     ~Test();

      /**
       * a normal member taking two arguments and returning an integer value.
       * @param a an integer argument.
       * @param s a constant character pointer.
       * @see Test()
       * @see ~Test()
       * @see testMeToo()
       * @see publicVar()
       * @return The test results
       */
       int testMe(int a,const char *s);

      /**
       * A pure virtual member.
       * @see testMe()
       * @param c1 the first argument.
       * @param c2 the second argument.
       */
       virtual void testMeToo(char c1,char c2) = 0;

      /**
       * a public variable.
       * Details.
       */
```

```
        int publicVar;

    /**
     * a function variable.
     * Details.
     */
    int (*handler)(int a,int b);
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

Similarly, if one wishes the first sentence of a Qt style documentation block to automatically be treated as a brief description, one may set **QT_AUTOBRIEF** to YES in the configuration file.

Documentation at other places

In the examples in the previous section the comment blocks were always located *in front* of the declaration or definition of a file, class or namespace or *in front* or *after* one of its members. Although this is often comfortable, there may sometimes be reasons to put the documentation somewhere else. For documenting a file this is even required since there is no such thing as "in front of a file".

Doxygen allows you to put your documentation blocks practically anywhere (the exception is inside the body of a function or inside a normal C style comment block).

The price you pay for not putting the documentation block directly before (or after) an item is the need to put a structural command inside the documentation block, which leads to some duplication of information. So in practice you should *avoid* the use of structural commands *unless* other requirements force you to do so.

Structural commands (like **all other commands**) start with a backslash (`\`), or an at-sign (`@`) if you prefer JavaDoc style, followed by a command name and one or more parameters. For instance, if you want to document the class `Test` in the example above, you could have also put the following documentation block somewhere in the input that is read by doxygen:

```
/*! \class Test
    \brief A test class.

    A more detailed class description.
*/
```

Here the special command `\class` is used to indicate that the comment block contains documentation for the class `Test`. Other structural commands are:

- `\struct` to document a C-struct.
- `\union` to document a union.
- `\enum` to document an enumeration type.
- `\fn` to document a function.
- `\var` to document a variable or typedef or enum value.
- `\def` to document a #define.
- `\typedef` to document a type definition.
- `\file` to document a file.
- `\namespace` to document a namespace.
- `\package` to document a Java package.
- `\interface` to document an IDL interface.

See section **Special Commands** for detailed information about these and many other commands.

To document a member of a C++ class, you must also document the class itself. The same holds for namespaces. To document a global C function, typedef, enum or preprocessor definition you must first document the file that contains it (usually this will be a header file, because that file contains the information that is exported to other source files).

Attention

Let's repeat that, because it is often overlooked: to document global objects (functions, typedefs, enum, macros, etc), you *must* document the file in which they are defined. In other words, there *must* at least be a

```
/*! \file */
```

or a

```
/** @file */
```

line in this file.

Here is an example of a C header named `structcmd.h` that is documented using structural commands:

```
/*! \file structcmd.h
    \brief A Documented file.

    Details.
*/

/*! \def MAX(a,b)
```