OSE - Tools User Guide

Graham Dumpleton
Dumpleton Software Consulting Pty Limited
PO BOX 3150
Parramatta, 2124
N.S.W, Australia
email: grahamd@nms.otc.com.au

Table of Contents

Classinfo Tools

1 Available Programs

The documentation tools provided with the OSE build environment, allow comments in your code files, to be extracted and formatted into end user documentation. To enable the tools to do this in a predictable manner, the comments must be formatted according to a small set of rules. At present, the tools can extract comments from C++ class header files and produce either UNIX style manual pages, of Frame mml files. The tools provided are described below.

Program	Purpose
class2info	Parses a C++ header file and produces a file which contains infor mation about the class in the header file, in a form which is more easily parsed by other tools.
info2man	Takes the output file from class2info and produces a UNIX style manual page for each class described in the input file.
class2man	Combines the programs class2info and info2man, to pro duce a UNIX style manual page for each class declaration in a header file.
info2mml	Takes the output file from class2info and produces a Frame mml file for each class described in the inout file.
class2mml	Combines the programs class2info and info2mml, to pro

```
            duce a Frame mml file for each class declaration in a header file.
--------------------------------------------------------------------------------
```

Using the same front end parser as the documentation extraction tools, are the following tools:

Program	Purpose
info2src	Takes the output file from class2info and produces on stan dard output, empty stub functions for each of the member func tions listed in the class declarations described in the file.
class2src	Combines the programs class2info and info2src, to pro duce on standard output, empty stub functions for each of the member functions listed in the class declarations described in a header file.

These use information contained in a C++ class header file, to produce code stubs suitable for the corresponding implementation file for that class.

The format which C++ class header files must adhere to is the subject of the remainder of this document.

2 Global Sections

To include a section in the manual page of each class in a header file, the comment containing the body text of the section, must appear at global scope within the header file. The title of the section must appear on a separate line, immediately prior to the body text of the section. To distinguish the title of the section from the body text, you must precede it with a tag, consisting of the character '='. There must be a single space only, before and after the tag. For example:

```
// = SECTION TITLE

//     Body text.
```

Any text following the title of the section, up till the end of the comment, the start of another section, or a line commencing with '==', is interpreted to be the body text for that section. Multiple sections may be documented in a single comment block. Any leading, or trailing blank lines will be discarded in the output. An example of a comment block containing multiple sections is given below.

```
// =====================================================

//

// = SECTION TITLE1

//     Body text.

//

// = SECTION TITLE2

//     Body text.

//
```

```
// ========================================================
```

Certain sections, appearing at global scope, are given special treatment in the generated manual pages. These are:

Section	Description
LIBRARY	The name of the library, of which the file is a part. This will appear in the header of each manual page.
FILENAME	The name of the file. This will be used in conjunction with the library name, to produce an include line in the synopsis.
AUTHOR(S)	The names of the authors. If the authors are placed on sepa rate lines, commas should not be used to terminate each line. If more than one author is placed on a single line, a comma should be used to separate each.
COPYRIGHT	The copyright applying to the contents of the file.
VERSION	The version number of the file. If using RCS, this would be set to '$Revision$'.
DATE RELEASED	The date that the current version of the file was released. If using RCS, this would be set to '$Date$'.
RCSID	The raw ID produced by RCS. Normally set to 'Id'.
SCCSID	The raw ID produced by SCCS. Normally set to '%W%'.

Using these tags, you can construct a comment block at the start of each file, which describes the file, authors and version of the file. For example:

```
/*

// ===========================================================

//

// = LIBRARY

//     OTC

//

// = FILENAME

//     collctn/list.hh

//

// = RCSID

//     $Id$

//

// = AUTHOR(S)

//     Graham Dumpleton

//

// = COPYRIGHT
```

```
//     Copyright 1991 1992 1993 OTC LIMITED
//
// ======================================================
*/
```

3 Block Formatting

When the text of your comments appear in the manual page, they will be reformatted. If you need to include an excerpt of code in the manual page, for example:

```
// = EXAMPLE
//     void dump(Class* theClass)
//     {
//       cout << theClass->name() << endl;
//     }
```

it will appear as:

```
void dump(Class* theClass) { cout << *theClass << endl; }
```

If the example is complicated, the result will be unreadable.

To let the documentation tools know that the text you have included is special and should not be reformatted, you can mark it as being a code example. For example:

```
// = EXAMPLE
// = BEGIN<CODE>
//     void dump(Class* theClass)
//     {
//       cout << *theClass << endl;
//     }
// = END<CODE>
```

The commands in this example are, ‘BEGIN<CODE>’and ‘END<CODE>’. In all cases, the format of commands used to change the formatting of a block of text is:

```
// = BEGIN<COMMAND>
// ....
// = END<COMMAND>
```

The ‘BEGIN’ for a command, must always have a matching ‘END’. The commands which are currently understood by the documentation tools are:

Command	Description
INDENT	Text will be indented with respect to the surrounding text.
NOFILL	Text will be output with indentation and formatting as it appears.
CODE	Text is output in ‘NOFILL’ mode using a fixed width font.
COMMENT	Text will not be output. This command would be used to sur round comments about the implementation, which should not appear in the user documentation.

Commands may be nested, with the exception that no commands should be nested within a ‘CODE’ block.

4 Inline Formatting

As well as changing the formatting of a block of text, you can change the font style used. This is done by placing special sequences of characters in the body of the text. Inline formatting commands come in the following three forms:

Command	Description
<text>	Text between ‘<‘ and ‘>’, is displayed in a fixed with font.
<{text}>	Text between ‘<{‘ and ‘}>’, is displayed in an italic font.
<[text]>	Text between ‘<[‘ and ‘]>’, is displayed in a bold font.

‘<text>’ should be used for all C++ keywords, function names, variable names, class names, expressions etc. The others can be used to highlight text. An example of how inline formatting commands may be used, follows:

```
// = ASSERTIONS

//     Assertion checks can be placed into code using

//     the <OTCLIB_ASSERT()> macro. Note that you should

//     <[NOT]> use an expression in the condition check,

//     which has a side effect, the result of which is

//     necessary for the correct operation of the

//     program. The reason for this, is that assertions

//     can be compiled out of your code by defining the

//     <NDEBUG> symbol.
```

Since ‘<‘, and ‘>’ are special characters, you must prefix them with a ‘\’ if the actual character needs to printed. For example:

```
<operator\<()>
```

If ‘{‘, ‘}’, ‘[‘ or ‘]’ appear adjacent to either ‘<‘ or ‘>’ in the forms described above, they will also need to be prefixed with a ‘\’. For example:

```
<\[A-Za-z_]*>
```

5 Class Sections

Sections pertaining to a class, are placed in the class declaration before the initial opening brace. For example:

```
template<class T>

class OTC_List : public OTC_Collection<T>

    // = SECTION NAME

    //     Body text.

{

  ...

};
```

These sections will only appear in the manual page for that class. Sections appearing with a class, which are given special treatment in the manual page are:

Section	Description
TITLE	A one line description of the class. This will be used in the ‘NAME’ section of the manual page.
CLASS TYPE	The type of class. For example: Abstract or Concrete.
AUDIENCE	Who may use the class. For example, the class may only be of interest to the class librarian.
DESCRIPTION	A description of the class.
NOTES	Any special features, unimplemented but desirable features, side effects, or known bugs.
SEE ALSO	References to any associated documents, systems or classes. If listed separately on each line, no comma should be used at the end of the line. If more than one appears on a single line, a comma should be used to separate them.
EXAMPLE	If realistic, a brief example of how the class may be used. Copious examples should not be included here, but should be described in a separate user guide.

6 Member Documentation

Documentation for member variables and member functions do not have to be tagged, as has been the case so far. It does however have to be put in a certain place with respect to the member variable or function declaration. The locations, where comments related to a member function or variable can be placed, are on the same line as the declaration following the semi-colon,

```
void func1(); // Comment

void func2(); // Comment
```

or on the line immediately following the declaration.

```
void func1();

  // Comment


void func2();

  // Comment
```

The comment may extend over more than one line. A comment for a particular member variable or function will be terminated by any line which doesn't contain only a comment. A comment will only be associated with the declaration which immediately preceeds it. For example:

```
void func1();

void func2();

  // Comment
```

the comment is associated with func2() only and not func1().

If a member function is defined inline, the comment must be placed after the code. For example:

```
void func1() { ... }

  // Comment


void func2()

  { ... }

  // Comment
```

A limitation of the extraction tools, currently requires that you do not place comments, or blank lines, within the body of the inline code.

7 Contract Section

Contracts allow grouping of related member functions and variables. This

makes it easier to find a function which serves the purpose you require.

A contract is a grouping of operations which fulfill a responsibility of the class. Contracts could include:

- Initialisation of the class.
- Destruction of the class.
- The ability to insert items into, or modify a particular aspect of a class.

- The ability to make queries about a class or iterate over data it contains.

A contract section is started by placing a comment containing the section title, prior to the functions in the class that are being grouped together. The list of functions in a contract is terminated by the commencement of another contract, a new public, protected, or private section, or the end of the class.

If contract sections are used, only functions which appear under a contract section will appear in the manual page. Functions which are not in a contract section are not displayed. If no contract sections are used, two default contract sections, with labels ‘PROTECTED MEMBERS’ and ‘PUBLIC MEMBERS’, will be created.

An example of a class using contracts is:

```
class Class
    // ...
{
  public:

    ~Class();
      // This will not appear in the
      // manual page.

    // = INITIALISATION

    Class(char const* theString);
      // This will appear in the manual
      // page in section INITIALISATION.
      //
      // Inline formatting, for example:
      // <theString>, may be used, as may
      // block formatting commands.

    // = QUERY
    //     As with all other sections, a contract
    //     section can have body text. This will appear
    //     in the manual page after the section name and
    //     before the functions are listed.
```

```
    char const* string();

    // Extra information for the current contract may
    // also be included between members. This is
    // generally used where it is necessary to make a
    // comment about a specific set of members in a
    // contract.

    OTC_Boolean isValid() const;
};
```