Kicad C++ Source Code Style Guide

Latest Publishing: October 2010
First Published: September 2010

written by

Wayne Stambaugh <stambaughw@verizon.net>
and
Dick Hollenbeck <dick@softplc.com>

Table of Contents

1. Introduction

The purpose of this document is to provide a reference guide for Kicad developers about how source code should be styled and formatted in Kicad. It is not a comprehensive programming guide because it does not discuss many things such as software engineering strategies, source directories, existing classes, or how to internationalize text. The goal is to make all of the Kicad source conform to this guide.

1.1 Why Coding Style Matters

You may be thinking to yourself that using the style defined in this document will not make you a good programmer and you would be correct. Any given coding style is no substitute for experience. However, any experienced coder will tell that the only thing worse than looking at code that is not in your preferred coding style, is looking at twenty different coding styles that are not your preferred coding style. Consistency makes a) problems easier to spot, and b) looking at code for long periods of time more tolerable.

1.2 Enforcement

The Kicad coding police are not going to break down your door and beat you with your keyboard if you don't follow these guidelines (although there are those who would argue that they should). However, there are some very sound reasons why you should follow them. If you are contributing patches, you are much more likely to be taken seriously by the primary developers if your patches are formatted correctly. Busy developers don't have the time to go back and reformat your code. If you have a desire to become a regular Kicad developer with commit access to the development branch, you're not likely to get a glowing recommendation by the lead developers if you will not follow these guidelines. It is just good programming courtesy to follow this policy because it is respectful of the investment already made by the existing developers. The other Kicad developers will appreciate your effort.

2. Naming Conventions

Before delving into anything as esoteric as indentation and formatting, naming conventions need to be addressed. This section does not attempt to define what names you use for your code. Rather, it defines the style for naming. See the references section for links to some excellent coding references. When defining multiple word names use the following conventions for improved readability:
•Use underscores for all upper and all lower case variables to make multiple word names more readable.
•Use camel case for mixed case variable names.
Avoid mixing camel case and underscores.

Examples

```
CamelCaseName           // if camelcase, then no underscores
all_lower_case_name
ALL_UPPER_CASE_NAME
```

2.1 Class, Type Definitions, Name Space, and Macro Names

Class, typedef, enum, name space, and macro names should be comprised of all capital letters.

Examples

```
class SIMPLE
#define LONG_MACRO_WITH_UNDERSCORES
typedef boost::ptr_vector<PIN>     PIN_LIST;
enum KICAD_T  {…};
```

2.2 Local, Private and Automatic Variables

The first character of automatic, static local, and private variable names should be lower case. This indicates that the variable will not be “visible” outside of the function, file, or class where they are defined, respectively. The limited visibility is being aknowledged with the lowercase starting letter, where lowercase is considered to be less boisterous than uppercase.

Examples

```
int i;
double aPrivateVariable;
static char* static_variable = NULL;
```

2.3 Public and Global Variables

The first character of public and global variable names are to be uppercase. This indicates that the variable is visible outside the class or file in which it was defined. (An exception is the use of prefix g_ which is also sometimes used to indicate a global variable.)

Example

```
char* GlobalVariable;
```

2.4 Local, Private and Static Functions

The first character of local, private, and static functions should be lower case. This indicates that the function is not visible outside the class or file where it is defined.

Example

```
bool isModified();
static int buildList( int* list );
```

2.5 Function Arguments

Function arguments are prefixed with an 'a' to indicate these are arguments to a function. The 'a' stands for “argument”, and it also enables clever and concise Doxygen comments.

Example

```
/**
 * Function SetFoo
 * takes aFoo and copies it into this instance.
 */
void SetFoo( int aFoo );
```

Notice how the reader can say “a Foo” to himself when reading this.

2.6 Pointers

It is not desired to identify a pointer by building a 'p' into the variable name. An experienced C++ programmer can see if a variable is a pointer, almost with his/her eyes closed. The pointer aspect of the variable pertains to type, not purpose. It is better to keep the purpose of the variable separate from its type.

Example

```
MODULE*     module;
```

The purpose of the variable is that it represents a MODULE. Something like p_module would only make that harder to discern.

2.7 Accessing Member Variables and Member Functions

We do not use “this->” to access either member variables or member functions from within the containing class. We let C++ perform this for us.

3. Commenting

Comments in Kicad typically fall into two categories: in line code comments and Doxygen comments. In line comments have no set formatting rules other than they should have the same indent level as the code if they do not follow a statement. In line comments that follow statements should not exceed 99 columns unless absolutely necessary. The prevents word wrapping in an editor when the viewable columns is set to 100. In line comments can use either the C++ or the C commenting style, but C++ comments are preferred for single line comments or comments consisting of only a few lines. C comments are block comments and can be used to comment out multiple lines, but only if the multiple lines do not already have a C comment on them. This is one reason why C++ comments should be used for smaller comments.

3.1 Blank Lines Above Comments.

If a comment is the first thing on a line, then that comment should have one or more blank lines above them. One blank line is preferred.

3.2 Doxygen

Doxygen is a C++ source code documenting tool used by the project. Descriptive *.html files can be generated from the source code by installing Doxygen and building the target named **doxygen-docs**.

```
$ cd <kicad_build_base>
$ make doxygen-docs
```

The *.html files will be placed into <kicad_project_base>/Documentation/doxygen/html/

Doxygen comments are used to build developer documentation from the source code. They should normally be only placed in header files and not in *.cpp files. This eliminates the obligation to keep two comments in agreement with each other. An exception to this rule is if the class, function, or enum, etc. is only defined in a *.cpp source file and not present in any header file, in which case the Doxygen comments should go into the *.cpp source file. Again, avoid duplicating the Doxygen comments in both the header and *.cpp source files.

Kicad uses the JAVADOC comment style defined in the "Documenting the code" section of the Doxygen manual. Don't forget to use the special Doxygen tags: bug, todo, deprecated, etc., so other developers can quickly get useful information about your code. It is good practice to actually generate the Doxygen *.html files by building target doxygen-docs, and then to review the quality of your Doxygen comments with a web browser before submitting a patch.

4. Formatting

This section defines the formatting style used in the Kicad source.

4.1 Indentation

The indentation level for the Kicad source code is defined as four spaces. Please do not use tabs.

4.1.1 Defines

There should be only one space after a #define statement.

4.1.2 Column Alignment

Please try to align multiple consecutive similar lines into consistent columns when possible, such as #define lines which can be thought of as containing 4 columns: #define, symbol, value, and comment. Notice how all 4 columns are aligned in the example below.

Example

```
#define LN_RED          12          // my favorite
#define LN_GREEN        13          // eco friendly
```

Another common case is the declaration of automatic variables. These are preferably shown in columns of type and variable name.

4.2 Blank Lines

4.2.1 Function Declarations

There should be 1 blank line above a function declaration in a class file if that function declaration is presented with a Javadoc comment. This is consist with the statement above about blank lines above comments.

4.2.2 Function Definitions

Function definitions in *.cpp files will not typically be accompanied by any comment, since those are normally only in the header file. It is desirable to set off the function definition within the *.cpp file by leaving two blank lines above the function definition.

4.2.3 If Statements

There should be one blank line above if statements.

4.3 Line Length

The maximum line width is 99 columns. An exception to this is a long quoted string such as the internationalized text required to satisfy MSVC++, described below.

4.4 Strings

Because the Kicad project team supports compiling with Microsoft Visual C++, defining long internationalized strings requires an ugly compromise. Most compilers support breaking long strings in to shorter pieces for code formatting purposes. Unfortunately, Visual C++ chokes on some multi-line internationalized string definitions. Prefixing all subsequent strings with L”” solves the Visual C++ problem but breaks POEdit which is used by translators. Breaking the string with \ solves both problems but is no better than line wrapping in your editor. Until a better solution can be found or MSVC++ is fixed, please use either of the 2 acceptable solutions mentioned below.

Examples

```
// works with C99 compliant compilers, but not with MSVC++ so is not acceptable:
wxChar* foo = _( “this is a long string broken ”
                 “into pieces for readability.” );

// this works with VC++ but breaks POEdit, so is not acceptable:
wxChar* foo = _( “this is a long string broken ”
               L“into pieces for readability” );

// ugly, since it wraps back to column zero, but works in all cases:
wxChar* foo = _( “this is a long string \
broken into pieces for readability” );
```

A second acceptable solution is to simply put the text all on one line, even if it exceeds the 99 character line length limit.

4.5 Trailing Whitespace

Many programming editors conveniently indent your code for you. Some of them do it rather poorly and leave trailing whitespace. Thankfully, most editors come with a remove trailing whitespace macro or at least a setting to make trailing whitespace visible so you can see it and manually remove it. Trailing whitespace is known to break some text parsing tools. It also leads to unnecessary diffs in the version control system. Please remove trailing whitespace.

4.6 Multiple Statements per Line

It is generally preferred that each statement be placed on its own line. This is especially true for statements without keywords.

Example

```
x=1;    y=2;    z=3;   // Bad, should be on separate lines.
```

4.7 Braces

Braces should be placed on the line proceeding the keyword and indented to the same level. It is not necessary to use braces if there is only a single line statement after the keyword. In the case of if..else if..else, indent all to the same level.

Example

```
void function()
{
    if( foo )
    {
        statement1;
        statement2;
    }
    else if( bar )
    {
        statement3;
        statement4;
    }
    else
        statement5;
}
```

4.8 Parenthesis

Parenthesis should be placed immediately after function names and keywords. Spaces should be placed after the opening parenthesis, before the closing parenthesis, and between the comma and the next argument in functions. No space is needed if a function has no arguments.

Example

```
void Function( int aArg1, int aArg2 )
{
    while( busy )
    {
        if( a || b || c )
            doSomething();
        else
            doSomethingElse();
    }
}
```

4.9 Switch Formatting

The case statement is to be indented to the same level as the switch.

Example

```
switch( foo )
{
case 1:
    doOne();
    break;
case 2:
    doTwo();
    // Fall through.
default:
    doDefault();
}
```

5. License Statement

There is a the file copyright.h which you can copy into the top of your new source files and edit the <author> field. Kicad depends on the copyright enforcement capabilities of copyright law, and this means that source files must be copyrighted and not be released into the public domain. Each source file has one or more owners.

6. Header Files

Project *.h source files should:

- contain a license statement
- contain a nested include #ifndef
- be fully self standing and not depend on other headers that are not included within it.

The license statement was described above.

6.1 Nested Include #ifndef

Each header file should include an #ifndef which is commonly used to prevent compiler errors in the case where the header file is seen multiple times in the code stream presented to the compiler. Just after the license statement, at the top of the file there should be lines similar to these (but with a filename specific token other than RICHIO_H_):

```
#ifndef RICHIO_H_
#define RICHIO_H_
```

And at the very bottom of the header file, use a line like this one:

```
#endif    // RICHIO_H_
```

The #ifndef wrapper begins after the license statement, and ends at the very bottom of the file. It is important that it wrap any nested #include statements, so that the compiler can skip them if the #ifndef evaluates to false, which will reduce compilation time.

6.2 Headers Without Unsatisfied Dependencies

Any header file should include other headers that it depends on. (Note: Kicad is not at this point now, but this section is a goal of the project.)

It should be possible to run the compiler on any header file within the project, and with proper include paths being passed to the compiler, the header file should compile without error.

Example

```
$ cd /svn/kicad/testing.checkout/include
$ g++ `wx-config --cxxflags` -I . xnode.h -o /tmp/junk
```

Such structuring of the header files removes the need within a client *.cpp file to include some project header file before some other project header file. (A client *.cpp file is one that intends to **use, not implement,** the public API exposed within the header file.)

Client code should not have to piece together things that a header file wishes to expose. Instead, all the needed dependencies should already be included in the header which is doing the exposing. The exposing header file should be viewed as a fully sufficient **ticket to use** the public API of that header file.

This is not saying anything about how much to expose, only that that which is exposed needs to be fully usable merely by including the header file that exposes it, with no additional includes.

For situations where there is a class header file and an implementation *.cpp file, it is desirable to hide as much of the private implementation as is practical and any header file that is not needed as part of the public API can and should be included only in the implementation *.cpp file. However, the number one concern of this section is that client (using) code can use the public API which is exposed in the header file, merely by including that one header file.

7. I Wrote X Lines of Code Before I Read This Document

It's OK. We all make mistakes. Fortunately, Kicad provides a configuration file for the code beautifier uncrustify. Uncrustify won't fix your naming problems but it does a pretty decent job of formatting your source code. There are a few places where uncrustify makes some less than ideal indentation choices. It struggles with the string declaration macros wxT(“”) and _(“”) and functions used as arguments to other functions. After you uncrusify your source code, please review the indentation for any glaring errors and manually fix them. See the uncrusify website for more information.

8. Show Me an Example

Nothing drives the point home like an example. The source file richio.h below was taken directly from the Kicad source.

```
/*
 * This program source code file is part of KICAD, a free EDA CAD application.
 *
 * Copyright (C) 2007-2010 SoftPLC Corporation, Dick Hollenbeck <dick@softplc.com>
 * Copyright (C) 2007 Kicad Developers, see change_log.txt for contributors.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, you may find one here:
 * http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
 * or you may search the http://www.gnu.org website for the version 2 license,
 * or you may write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA
 */

#ifndef RICHIO_H_
#define RICHIO_H_


// This file defines 3 classes useful for working with DSN text files and is named
// "richio" after its author, Richard Hollenbeck, aka Dick Hollenbeck.


#include <string>
#include <vector>

// I really did not want to be dependent on wxWidgets in richio
// but the errorText needs to be wide char so wxString rules.
#include <wx/wx.h>
#include <cstdio>       // FILE



/**
 * Struct IOError
 * is a class used to hold an error message and may be used to throw exceptions
 * containing meaningful error messages.
 */
struct IOError
{
    wxString    errorText;

    IOError( const wxChar* aMsg ) :
        errorText( aMsg )
    {
    }

    IOError( const wxString& aMsg ) :
```

```
        errorText( aMsg )
    {
    }
};


/**
 * Class LINE_READER
 * reads single lines of text into its buffer and increments a line number counter.
 * It throws an exception if a line is too long.
 */
class LINE_READER
{
protected:

    FILE*               fp;
    int                 lineNum;
    unsigned            maxLineLength;
    unsigned            length;
    char*               line;
    unsigned            capacity;

public:

    /**
     * Constructor LINE_READER
     * takes an open FILE and the size of the desired line buffer.
     * @param aFile An open file in "ascii" mode, not binary mode.
     * @param aMaxLineLength The number of bytes to use in the line buffer.
     */
    LINE_READER( FILE* aFile,  unsigned aMaxLineLength );

    ~LINE_READER()
    {
        delete[] line;
    }

    /*
    int  CharAt( int aNdx )
    {
        if( (unsigned) aNdx < capacity )
            return (char) (unsigned char) line[aNdx];
        return -1;
    }
    */

    /**
     * Function ReadLine
     * reads a line of text into the buffer and increments the line number
     * counter.  If the line is larger than the buffer size, then an exception
     * is thrown.
     * @return int - The number of bytes read, 0 at end of file.
     * @throw IOError only when a line is too long.
     */
    int ReadLine() throw (IOError);

    operator char* ()
    {
        return line;
    }

    int LineNumber()
    {
```

```
        return lineNum;
    }

    unsigned Length()
    {
        return length;
    }
};



/**
 * Class OUTPUTFORMATTER
 * is an interface (abstract class) used to output ASCII text in a convenient
 * way.  The primary interface is printf() like but with support for indentation
 * control.  The destination of the 8 bit wide text is up to the implementer.
 * If you want to output a wxString, then use CONV_TO_UTF8() on it before passing
 * it as an argument to Print().
 * <p>
 * Since this is an abstract interface, only classes derived from this one
 * will be the implementations.
 */
class OUTPUTFORMATTER
{

#if defined(__GNUG__)   // The GNU C++ compiler defines this

// When used on a C++ function, we must account for the "this" pointer,
// so increase the STRING-INDEX and FIRST-TO_CHECK by one.
// See http://docs.freebsd.org/info/gcc/gcc.info.Function_Attributes.html
// Then to get format checking during the compile, compile with -Wall or -Wformat
#define PRINTF_FUNC       __attribute__ ((format (printf, 3, 4)))

#else
#define PRINTF_FUNC       // nothing
#endif

public:

    /**
     * Function Print
     * formats and writes text to the output stream.
     *
     * @param nestLevel The multiple of spaces to preceed the output with.
     * @param fmt A printf() style format string.
     * @param ... a variable list of parameters that will get blended into
     *  the output under control of the format string.
     * @return int - the number of characters output.
     * @throw IOError, if there is a problem outputting, such as a full disk.
     */
    virtual int PRINTF_FUNC Print( int nestLevel, const char* fmt, ... )
throw( IOError ) = 0;

    /**
     * Function GetQuoteChar
     * performs quote character need determination.
     * It returns the quote character as a single character string for a given
     * input wrapee string.  If the wrappee does not need to be quoted,
     * the return value is "" (the null string), such as when there are no
     * delimiters in the input wrapee string.  If you want the quote_char
     * to be assuredly not "", then pass in "(" as the wrappee.
     * <p>
     * Implementations are free to override the default behavior, which is to
```

```
     * call the static function of the same name.

     * @param wrapee A string that might need wrapping on each end.
     * @return const char* - the quote_char as a single character string, or ""
     *   if the wrapee does not need to be wrapped.
     */
    virtual const char* GetQuoteChar( const char* wrapee ) = 0;

    virtual ~OUTPUTFORMATTER() {}

    /**
     * Function GetQuoteChar
     * performs quote character need determination according to the Specctra DSN
     * specification.

     * @param wrapee A string that might need wrapping on each end.
     * @param quote_char A single character C string which provides the current
     *          quote character, should it be needed by the wrapee.
     *
     * @return const char* - the quote_char as a single character string, or ""
     *   if the wrapee does not need to be wrapped.
     */
    static const char* GetQuoteChar( const char* wrapee, const char* quote_char );
};


/**
 * Class STRINGFORMATTER
 * implements OUTPUTFORMATTER to a memory buffer.  After Print()ing the
 * string is available through GetString()
*/
class STRINGFORMATTER : public OUTPUTFORMATTER
{
    std::vector<char>       buffer;
    std::string             mystring;

    int sprint( const char* fmt, ... );
    int vprint( const char* fmt,  va_list ap );

public:

    /**
     * Constructor STRINGFORMATTER
     * reserves space in the buffer
     */
    STRINGFORMATTER( int aReserve = 300 ) :
        buffer( aReserve, '\0' )
    {
    }


    /**
     * Function Clear
     * clears the buffer and empties the internal string.
     */
    void Clear()
    {
        mystring.clear();
    }

    /**
     * Function StripUseless
     * removes whitespace, '(', and ')' from the mystring.
```

```
     */
    void StripUseless();


    std::string GetString()
    {
        return mystring;
    }


    //-----<OUTPUTFORMATTER>--------------------------------------------------
    int PRINTF_FUNC Print( int nestLevel, const char* fmt, ... ) throw( IOError );
    const char* GetQuoteChar( const char* wrapee );
    //-----</OUTPUTFORMATTER>-------------------------------------------------
};


#endif // RICHIO_H_
```

9. Resources

There are plenty of excellent resources on the Internet on C++ coding styles and coding do's and don'ts. Here are a few useful ones. In most cases, the coding styles do not follow the Kicad coding style but there is plenty of other good information here. Besides, most of them have some great humor in them enjoyable to read. Who knows, you might even learn something new.

C++ Coding Standard
Linux Kernel Coding Style
C++ Operator Overloading Guidelines
Wikipedia's Programming Style Page