2.2. Code Files

C and C++ code files follow a similar structure to the header files. These files should contain the following information in the given order.

1. Copyright statement comment
2. Module abstract comment
3. Preprocessor directives, `#include` and `#define`
4. Revision-string variable
5. Other module-specific variable definitions
6. Local function interface prototypes
7. Class/function definitions

Unlike in the header file, the implementation file revision string should be stored as a program variable rather than in a comment. This way **ident** will be able to identify the source version from the compiled object file. For C files use:

```
static const char rcs_id[] __attribute__ ((unused)) =
"$Id$";
```

The `__attribute__` modifier is a GNU C feature that keeps the compiler from complaining about the unused variable. This may be omitted for non-GNU projects. For C++ files, use the following form for the revision string:

```
namespace { const char rcs_id[] = "$Id$"; }
```

Precede each function or class method implementation with a form-feed character (Ctrl-L) so that when printed the function starts at the start of a new page.

```
    int get_x() const { return x_; }
    void set_x(const int new_x) { x_ = new_x; }
    ...
    void display() {
        ...
    }
}
```

4. Choosing Meaningful Names

4.1. Variable Names

The name formatting conventions described here are essentially the GNU coding standards. These are available online using `info`.

Use lower case for all variable names. For multi-word names, use an underscore as the separator. Use all capitals for the names of constants (i.e. variables declared `const` and enumerated types). Use an underscore as a word separator.

Choose variable names carefully. While studies show that the choice of variable names has a strong influence on the time required to debug code, there are unfortunately no clear and fixed rules for how to choose good names. Review Chapter 9 of *Code Complete* periodically. In the mean time, here are some general guidelines to follow:

- Be consistent! The most important thing is to establish a clear, easily recognizable pattern to your code so that others will be able to understand your implementation and intent as quickly and reliably as possible.
- Use similar names for similar data types, dissimilar names for dissimilar types.

Example 7. Capitalization of user-defined types

```
/* Straight C */

struct complex {
    int r;    /* real */
    int i;    /* imaginary */
};
typedef struct complex Complex;

// C++ interface example

class Canvas {
public:
    enum Pen_style {
        NONE = 0,
        PENCIL,
        BRUSH,
        BUCKET
    };

    Canvas();
    ~Canvas();

    void set_pen_style(Pen_style p);
    ...
private:
    int cached_x_;    // to avoid recomputing coordinates
    int cached_y_;
};

// C++ usage example

Canvas sketch_pad;

sketch_pad.set_pen_style(Canvas::BRUSH);
```

for each function longer than a few lines. Avoid using `break` and `continue` to escape loop and branch code. Consider instead adding or changing the exit conditions of the the control statement. Do not use `goto`.

Prefer using `if/else/else/...` over the `switch/case/case/...` with non-trivial branch conditions. For both constructs use default conditions only to detect legitimate defaults, or to generate an error condition when there is no default behavior. Using a `switch/case` block with overlapping conditions only when the cases have identical code so that fall-through is obvious.

Prefer using `while() { ... }` instead of `do { ... } while();`. It is easier for humans to parse the control structure if they know the exit condition upon entering the block of code. The `do { ... } while();` form buries the exit criterion at the end of the loop.

Avoid overly long control structures. If you find loop or branch constructs spanning several printed pages or screens, consider rewriting the structure or creating a new function. At the very least place a comment at the end of the structure to indicate the exit conditions.

Avoid deeply nested code. Humans have a hard time keeping track of more than three or four things at a time. Try to avoid code structure that requires more than three or four levels of indentation as a general rule. Again, consider creating a new function if you have too many embedded levels of logic in your code.

Avoid the use of global variables. They make your code hard to support in a multi-threaded environment. If you do use global variables, understand how they affect the ability of your module to be reentrant.

6.3. Functions and Error Checking

Do not use preprocessor function macros. There are too many possible problems associated with them and modern computer speeds and compiler optimizations obviate any benefit they once may have had. Define a function instead.

Write function declarations/prototypes for all functions and put them either in the