C / C++ Coding Rules

Luca Abeni
luca.abeni@unitn.it

March 3, 2008

Abstract

This short document collects some simple and stupid coding rules for writing understandable C or C++ code, and has been written to help the students of the SO2 and RTOS courses (following these rules is recommended when submitting a project for the SO2 or RTOS exam).

Please write me an email if you have comments, questions, etc...

1 Introduction

Before starting, a word of warning: note that this is an *informal* document, presenting a lot of "rules of the thumb" that should be respected when writing good C / C++ code, but every rule has its exceptions. So, the rules presented here should be respected *in general*, but there are cases in which violating them can be reasonable. Whenever you think you find one of those situations, you should:

- double think about the code, and see if there really are not better solutions
- ask someone experienced (do not hesitate to send me an email: if you ask reasonable questions in an understandable way, you'll surely get an helpful answer)
- if breaking a rule is really the right thing to do, document this fact, explaining why this violation is needed

From this short introduction, we can derive the first coding rule:

Coding Rule 1 *Whenever you are breaking a coding rule (except this one), put a short comment in the code explaining why the rule's violation cannot be avoided*

Note that *rule's violations must be very infrequent.*

2 Editing the Source Code

The code you write must be easy to read and to understand (I will have to review it for evaluating your project... Writing understandable code is in your interest).

So, first of all you **must** *properly indent* the code; you can use the indentation style that you prefer (you are not forced to follow a specific indentation style), but the indentation **must** be consistent (in general, something like `indent -kr <file>.c` will do a reasonable job). Pay attention to the fact that some brain-damaged text editors end up by mixing tabs and spaces, resulting in code that is wrongly indented when edited with another program.

Finally, note that lines longer than 80 characters make the code more difficult to read, so you should rarely use them.

Coding Rule 2 *Indent the code consistently. Running* `indent` *on your code is a good idea, and avoiding to mix tabs and spaces in the code is strongly suggested*

as an extreme consequence of this rule, some software projects forbid tabs in the source code.

Coding Rule 3 *Do not use lines longer than 80 characters in your code*

As a final suggestion, you can open your source files with the `vim` editor (this is what I will use when I will review your project), and check if the code looks good.

3 Compiling

First of all, always use a proper Makefile to compile your program.

Then, note that modern C and C++ compilers are generally very good in detecting potential problems and bugs in the code, and can generate helpful *warnings*. Such warnings are not fatal for the program compilation, but must be taken **very** seriously:

Coding Rule 4 Always *enable as many warnings as possible, and ensure that* **no warnings** *are produced during the compilation*

If you are developing under Linux, the above rule means:

- **always** use the `-Wall` gcc's switch
- if the gcc's standard output is redirected to `/dev/null`, then the compilation should generate no output

(in other words: the `CFLAGS` variable in your Makefile must contain `-Wall`, and the `make > /dev/null` command must generate no output).

4 Code Modularisation

When developing a complex software project, it is important to split the code in different source files or software modules, to maintain the size and the scope of each module reasonable. In particular, C projects are composed by `.c` files (containing source code) and `.h` files (describing software interfaces):

- each `.c` file implements a software module and is compiled into a `.o` *object file*[1]
- the various modules are then *linked* together with some libraries to generate an *executable file*

Each software module contains data (variables) and code (functions), and not all of such functions and variables can be accessed from other modules: the set of symbols that can be accessed from outside a module (exported symbols) defines its *interface*, which is described in a *header file* (`.h`).

Figure 4 presents a very simple program composed by 2 separate modules (the main module `main.o` and a module `funcs.o` containing some helper functions). This is a very simple example, but it is already possible to note that:

1. global symbols that are not part of the interface (in this case, the `status` variable) **must** be defined as `static`
2. each module generally includes the header defining its interface (`funcs.c` includes `funcs.h`)
3. if a module uses services provided by a different module (in this case, `main.o` uses function `f()`, which is defined in `funcs.o`), it must include the header file defining the used interface (`main.c` includes `funcs.h`)
4. the declaration of symbols that are part of the interface **must** be in the header file. The declaration of a function is its prototype (example: `int f(int input);`), and the declaration of a variable is equal to the definition + the `extern` keyword (example: `extern int debug;`)

Note that in real life things are more complex because of "inline" functions and static local variables, but such constructs are considered advanced topics (**use them only when you know what you are doing!**), and will not be considered in this document. Anyway, the previous considerations result in the following rules:

Coding Rule 5 Never *write function prototypes or* `extern` *declarations in* `.c` *files (they must be in* `.h` *files)*

[1] Note that the decomposition of the source code in modules is not arbitrary, but must be performed to group similar functionalities in a single file. However, this discussion is out of the scope of this document

Coding Rule 6 Always *include all the needed header files. This means that you should never see anything like* `warning: implicit declaration of function '...'`

Coding Rule 7 *Mark as many symbols as possible* `static`*. This means that if a global variable or a function can be marked* `static`*, then it* **must** *be* `static`*.*

This decreases the namespace pollution, and helps finding errors and cleaning up the code (if a static symbol is not used in a module, the compiler will generate a warning).

5 Final Remarks

The C and C++ standards provide various constructs that, although valid from a syntactical point of view, should be used with extreme caution. The (in)famous `goto` keyword is a good example: it is a perfectly legal construct, but can lead to *spaghetti programming*. Hence, the following coding rule[2]:

Coding Rule 8 *Never use* `goto` *statements (unless you* **really** *know what you are doing)*

Another example is the possibility to mix code and variable declarations, initially introduced in the C++ and then in the C99 standard: the usefulness of mixing code and variable declarations is debatable, and separating variable declarations from code improves the source readability. So, students are requested to respect the following rule:

Coding Rule 9 *Variable declarations* **must** *be at the beginning of a block (between the '{' and the first instruction)*

This means that the code shown in Figure 2 is ok, whereas the code in Figure 1 is not.

As a suggestion, an empty line can be introduced after variable declarations to separate them from the real code.

Also note that the C and C++ languages allow statements like `for (int i = 0; i < 10; i++)`, but some compilers get this feature wrong. So, do not use it (otherwise you might obtain code that is not compatible with some very well known commercial products).

[2] `goto` statements are often useful in handling error paths, but their usage is only suggested to experienced programmers (who do not need these notes anyway)

```
/*********************************************/
/* funcs.h:  interface definition for funcs.c */
/*********************************************/
extern int debug;
int f(int input);
/* ... */

/*********************************************/
/* funcs.c:      some helper functions       */
/*********************************************/
#include "funcs.h"

static int state;
int debug;

int f(int input)
{
    state = state + input / 2;

    return state + 5;
}
/* ... */

/*********************************************/
/* main.c:         main program file         */
/*********************************************/
#include <stdio.h>
#include "funcs.h"

int main(int argc, char *argv[])
{
    int in, out;

    /* ... */
    out = f(in);
    /* ... */
}
/* ... */
```

```
int main(int argc, char *argv[])
{
  printf("Hello!\n");
  int done = 0;
  while (!done) {
    printf("Looping...\n");
    FILE *f;
    f = fopen(...);
      /*...*/
  }
}
```

Figure 1: Example of mixed code and variable declarations.

```
int main(int argc, char *argv[])
{
  int done = 0;

  printf("Hello!\n");
  while (!done) {
    FILE *f;

    printf("Looping...\n");
    f = fopen(...);
      /*...*/
  }
}
```

Figure 2: Example of correct variable declarations.