C Coding Standards

Ken Clowes (kclowes@ee.ryerson.ca)

November 11, 2000

Contents

1 Introduction

The source code for programming projects should always be organized and written with the future tasks of testing, debugging and maintenance (possibly by others) in mind. These tasks will be easier if the project is well organized and the source code is written in a clear and consistent fashion. In addition, the future possibility of porting the program to different environments (portability) should be addressed at the outset.

This document describes some basic rules for C coding and project organization. Some general aspects of the portability problem are also addressed.

Many of the standards discussed here are dealt with from a more general (language-independent) way in *General Coding Standards*[Clo].

2 Recommended Organizational and Coding Standards

Organization: Each project has a separate directory. For example,

Each project directory must have `README` and `Makefile` files. The README file should give a general overview of the project and the files that implement it.

Documentation-I (public): The "public" documentation of your source code should inform a reader of who wrote the code and describe *what* it does and how to *use* the interfaces described. The public comments should provide sufficient information for a reader to use the functions without having to read the actual C code that implements the functions. It is strongly recommended that you write these public comments *before* you write your code. (Writing a function header comment focuses your mind on what you really want the function to do.)

All source code files (*.c and *.h) should conform to the following general commenting standards:

- Identify your work with, for example, a Copyright notice including your name and userid. For example:

```
/* Copyright (C) 1999 Jane Smith (jsmith@ryerson.ca) */
```

- The next comment—the *header*—briefly describes the purpose of the file. For example, a file called `towers.c` might have the following header comment:

```
/**
* The functions in this file solve the classic
* towers of Hanoi problem.
*/
```

- Each function (in a .c file) is preceded by a *function header* comment that briefly describes what the function does as well as indicating the nature of any passed parameters or return value. For example:

```
/**
```

```
 * ``main'' manages the command line interface to solving
 * the towers of Hanoi problem.  The command line arg
 * (which must be string representations of numbers)
 * indicate the number of disks to be moved and the source
 * and destination tower numbers. (The towers are
 * identified with the numbers 1, 2 and 3.)
 *
 * @param  argc  the number of command line arguments
 * @param  argv  a pointer to an array of strings where:
 *      There must be exactly 3 arguments where:
 *             -- the fist arg is the number of disks to move
 *             -- the second is the ID-num of the source
 *             -- the third is the ID-num of the destination
 * @return   always returns an exit code of 0.
 */
int main(int argc, char * argv[])
 {}
```

Notes

The conventions used for the public comments, specifically the `/**` (with the extra `*`) and the tags `@param` and `@return`, correspond to Java commenting standards. In particular, a Java tool called *javadoc* can parse these specially formatted comments and the following declaration to produce nicely formatted HTML documentation automatically. While there is no version of *javadoc* for C code at this time, it does no harm to use the clear conventions of Java in your C code[1].

Documentation-II (private): While the public documentation should be written so that it does not require the reader to understand or even look at the implementation, private documentation is meant to help the reader understand the actual C code implementing a function. The comments should be written under the assumption that the reader is a competent C programmer. For example:

```
i++;  /* Increment i by one */
```

[1]The `doc++` package seems to do this, but I have not yet tested or installed it

is a useless comment since it is entirely obvious to a C programmer.

Often, no private comments are required at all in well written programs. The use of descriptive variable and function names is also a great help. Indeed, Rob Pike states:

> Basically, avoid comments. If your code needs a comment to be understood, it would be better to rewrite it so it's easier to understand.
>
> Rob Pike[Pik]

Using descriptive names often eliminates the need for comments, Consider:

```
foo = foo->bar;  /* move "foo" to next item */
```

The comment would be unnecessary with the more intelligent variable and field names:

```
item = item->next;
```

Avoid "magic numbers": Numbers should rarely be placed directly in the source code. (Common exceptions are the numbers 0 (zero), 1 or −1.) Instead use an `enum` data type or the `#define` preprocessor directive. (It is usually preferable to use an `enum` for a small number of integers instead of a `#define`.)

For example, do *not* write code like:

```
double x = 3.14159265358979323846*2.6*2.6;
```

or

```
for(i = 32; i < 212; i += 2)
```

or

```
if ((j = foo()) == 2)
```

instead, use:

```
#include <math.h>  /* This defines the value of PI */
#define RADIUS 2.6
double x = M_PI*RADIUS*RADIUS;
```

or

```
/* Note following temperatures assume Farenheit scale */
#define FREEZING 32
#define BOILING 212
#define TEMP_INCREMENT 2
for(i = FREEZING; i < BOILING; i += TEMP_INCREMENT)
```

or

```
typedef enum {FooGood = 0,
              FooWarn = 1,
              FooBad = 2} FooReturn_t;
if ((j = foo()) == FooBad)
```

Compile with all warnings turned on: You should compile C source code will all warnings turned on. Your C code should produce **no** warnings.

Portability Try to use only POSIX/ANSI compatible library functions.

No gotos You can use all of the ANSI C language *except* for the `goto` statement. (The single exception to this rule involves the study of "tail-recursion elimination".)

Header files: Header files should only contain declarations (such as `typedef`s or function prototypes) and preprocessor directives (such as constants and macros). Executable C code (such as a function body) should *never* be placed in a header file.

.h protection All .h files must be protected so that they are never included more than once and that the order of their inclusion is less critical. For example, the header file `foo.h` should be structured as:

```
#ifndef FOO_H
#define FOO_H

/* Body of foo.h with (possibly) other #includes...  */

#endif /* FOO_H */
```

Line length and avoiding tabs: No source code line should be longer than 80 characters. Use spaces, not tabs, for indentation.

3 Other C programming conventions

3.1 The *eprintf* library

You may note functions such as `eprintf` or `emalloc` sprinkled through the C code. These functions come from Kernighan and Pike's[KP99, p. 109–111] utility library which we have called `eprintf.o`.

Their use is summarized in Table 1.

K&P name	"Almost" like	Comments
`eprintf(...)`	`fprintf(stderr, ...)`	Exits
`weprintf(...)`	`fprintf(stderr, ...)`	Prints warning
`emalloc(...)`	`malloc( ...)`	Exits on failure
`erealloc(...)`	`realloc( ...)`	Exits on failure

Table 1: Summary of *eprintf* functions

3.2 Using *asserts*

My view is that *asserts* should not be used as a lazy programmer's way of informing end-users of predictable error conditions in the operation of a program. Rather, they should be used mainly during the development stage to help the programmer figure out where things are going wrong.

Despite this, the source code often uses asserts in this "lazy" way.

3.3 Incorrect conventions

I use one coding "standard" that is incorrect and may lead to portability problems. In particular, there are occasions where the following assumptions are made:

1. A generic pointer `void *` is the same size as an integer (`int`) and data of one type can be cast to the other.

2. A generic pointer `void *` and a pointer to a function `void *()(...)` are the same size and either can be cast to the other.

Both of these assumptions *violate* the formal specifications in the ANSI C standard. They are, however, very commonly encountered.

Using these conventions makes some of the code easier to write and more readable. Note that there is NO assumption about the size of these things. Normally, however, they are all either 16 or 32 bits.

Some of the problems explore ways to avoid these assumptions.

3.4 Miscellaneous conventions

Some of the source code follows some other arbitrary conventions that are a matter of personal choice. These include:

Addresses of functions: If `funcP` is a function pointer data type and `foo()` is a function, I use `funcP = &foo` to set `funcP` to be a pointer to the function `foo()`. Other programmers use the shorter and equivalent form: `funcP = foo`.

Private header files: If a header file is used *only* by the programmer implementing a module and does not need to be viewed or included by programmers using the module, I append the letter 'P' to the name of the header. For example, `foo.h` would be a public header file, while `fooP.h` would be a private (non-distributed) header file.

private and public: In implementing modules that have "static globals" (i.e. declarations made at the highest level in the source code file but qualified as "static" to prevent their names being exported to the linker), I often use the pre-processor statements:

```
#define private static
#define public
```

This allows me to declare things at the global level as "private" or "public" which I find closer to the semantics I have in mind.

Underscores for private names: When a name (such as a `typedef` or a private variable is used in a module, I usually prepend an underscore character ('_') to its "logical" name.

References

[Clo] Ken Clowes. *General Coding Standards.* file: `CodingStdGen.ps`.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley, Reading, Massachusetts, 1999. 267 pages.

[Pik] Rob Pike. *Notes on Programming in C.* 5 pages.