C Coding standard

Author: Raphael ‘kena’ Poss
Date: July 2014

Contents

1 Introduction

This document describes how to format and layout C code when preparing assigments to be graded. This standard exists to streamline the work of teaching assistants and to avoid common mistakes when programming in C.

Note that in the text below, all “bad” examples are actually valid C code: they are *functionally equivalent* to the corresponding “good” example to the left. However, they are called “bad” because their *style* does not follow the standard.

2 File system layout

- source trees must be submitted as a tarball compressed using either gzip or bzip2 (`.tar.gz` or `.tar.bz2`).
- submitted archives must expand to a directory named after the name of the archive without extension (ie. `foo.tar.gz` must expand to `foo/`). But if the provided makefile contains a `tarball` target the tar file that is generated by the command `make tarball` is also acceptable.
- the top-level directory must contain a file named “`AUTHORS`” (capitalized, with no extension), containing the name(s) and student number(s) of the project authors, one per line.

> ***Note***
>
> The student numbers are extracted automatically using a regular expression. Do not enter data that would match the regular expression unintendedly.

- the provided tree must not contain any compiler output (`.o`, `.i`, `.s` etc.) nor temporary files (`*.bak`, `*~`, `#*#`, etc.).

3 Code style

3.1 Comments and preprocessor

- comments must be written in correct English or Dutch (In particular: sentences start with a capital letter, contain at least one verb and end with a period; prefer the active form; and use the present tense to describe what a piece of code does or should do.) Consistently use 1 language for your comments (and edit provided comments if necessary).
- disabled code must be delimited by "`#if 0 ... #endif`"; ie. NOT by comment delimiters, which do not nest.

3.2 Namespaces and translation units

3.2.1 Naming conventions

- all names (within code, but also file or directory names!) must be either fully expressive or use a well-known short mnemonic.
- Abbreviations are tolerated as long as they shorten the code significantly without loss of meaning.

3.3 Code layout

- code may not contain unprintable ASCII characters.
- code must only use ASCII space (code 32) and newline characters (code 10) as white space; in particular ASCII tabs (code 9) must not be used, nor the DOS/Windows carriage return (code 13).
- all code must fit within 80 columns.
- code must not contain trailing whitespaces.
- the body of a function definition should contain at most 30 lines of code (opening and closing braces excluded). This is not a hard rule, but a strongly suggested guideline. Most functions should be broken up into smaller functions if exceeding this limit, but exceptions are possible.
- function bodies should contain minimal comments; comments that explain a function should be placed before the function definition. Short clarifications for non-obvious lines within the function body are encouraged.
- blocks must be indented; the same indentation width must be used consistently throughout a submission, with a minimum of 2 spaces.

```
// Good:
void foo(void) {
    while (1) {
        printf("hello\n");
    }
}
```

```
// Bad:
void foo(void) {
 while (1) {// 1 spaces = too small
       // 5 spaces not consistent
       // with 1 used above
       printf("hello\n");
 }
}
```

- a control structure (`if`, `for`, etc.) must always be followed by a new line.

```
// Good:
if (cp)
   return (cp);
if (cp) {
   return (cp);
}
```

```
// Bad:
if (cp) return (cp);
if (cp) {
          return (cp);
        }
```

- all *keywords* with arguments must be followed by exactly one space before their argument(s).

```
// Good:
if (...)
for (...)
return ...
x = sizeof (...)
```

```
// Bad:
if(...)
for(...)
return(...)
x = sizeof(...)
```

- all binary operators except the comma, and the ternary (“`?:`”) operator, must be separated from their operands with exactly one space (when the operand is on the same line).

```
// Good:
x = 3;
f(x) + g(y)
f(x) && g(y)
for (i = 0; i < 10; ++i)
```

```
// Bad:
x=3;
f(x)+g(y)
f(x)&&g(y)
for (i=0; i<10; ++i)
```

- unary operators must precede or follow their operand without intervening white space.

```
// Good:
x = ++i;
y = *p++;
z = !x;
```

```
// Bad:
x = ++ i;
y = *     p   ++;
z = !
   x;
```

- an opening parenthesis “`(`” or square bracket “`[`” must be followed by the following token without intervening whitespace; conversely, a closing parenthesis “`)`” or square bracket “`]`” must follow the preceding token without intervening whitespace.

```
// Good:
x = f(y);
z = x[y];
```

```
// Bad:
x = f(  y);
z = x[y  ];
```

- a function expression or preprocessor macro name must be followed by the opening parenthesis of the argument list without intervening white space.

```
// Good:
f();
y = (*g)(x);
z = f(x, y);
```

```
// Bad:
f ();
y = (*g)   (x);
z = f   (x, y);
```

- remember that `exit` is a function but `return` and `sizeof` are keywords.

```c
// Good:
if (!cp)
   exit(1);
return cp;
```

```c
// Bad:
if (!cp)
   exit (1);
return(cp);
```

- commas and semicolons must follow the preceding token without whitespace, and must be separated from the next token (if any) with a white space.

```c
// Good:
z = x;
w = f(x, y);
u = g(x);
for (x = 0; x < 10; ++x)
return;
```

```c
// Bad:
z = x ;
w = f(x,y);
u = g( x);
for (x = 0 ;x < 10 ;++x)
return ;
```

4 Run-time behavior

- objects must not be accessed outside of their lifetime or their storage (cf. `valgrind`), in particular:
  - don't let code dereference invalid pointers;
  - don't let code access heap objects after they have been `free`'d;
  - don't let code read variables before their initialization;
  - don't let code access local variables after the function where they were defined has returned;
  - don't let code access local variables after the control flow has left the scope where they were defined;
  - don't let code access arrays out of bounds.
- all allocated memory must be freed (cf. `valgrind --leak-check=full`).

---

5 Copyright and licensing