C Coding standard

Raphael 'kena' Poss

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 Systems programming @ VU Amsterdam

For the 3rd year *Systems Programming* course at the Vrije Universiteit, academic year 2014-2015, **adherence to this standard is mandatory**.

The grading process will go as follows:

1. does the submission follow the rules in File system layout below?

   no: reject from grading. Yes: proceed to step 2.

2. does the C code follow the standard Code style below?

   no: reject from grading. Yes: proceed to step 3.

3. does the code compile without errors or warnings as per Compiler warnings and Build rules below?

   no: reject from grading. Yes: proceed to step 4.

4. grade the assignment using the assignment's criteria, constrained by the section Run-time behavior below.

No exceptions!

3 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/`).
- 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.).

4 Code style

4.1 Comments and preprocessor

- all `.h` files must use a preprocessor fence (`#ifndef-#define-#endif`) to prevent double inclusion.
- the macro name used as fence must be the name of the `.h` file, capitalized. (for example, the fence for `foo.h` must be `FOO_H`).
- comments must be written in correct English. (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.)
- disabled code must be delimited by "`#if 0 ... #endif`"; ie. NOT by comment delimiters, which do not nest.

4.2 Namespaces and translation units

4.2.1 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.
- Preprocessor macros names and `enum` values must be fully capitalized; preprocessor macro parameters must be simply capitalized; all other identifiers must be in small letters. Words must be separated by underscores.

```
 // Good:
#define XFREE(Var)    \
      do              \
      {               \
        if (Var)      \
          free(Var); \
      }               \
      while (0)
#define MAX_LINE_SIZE 10
enum t
{
   FIRST = 0,
   SECOND = 1
};
struct      my_list;
enum        some_enum;
typedef int custom_t;

void        compute_some();
```

```
 // Bad:
#define xfree(VAR)    \
      do              \
      {               \
        if (VAR)      \
          free(VAR); \
      }               \
      while (0)
#define max_L_sz 10
enum t
{
   First = 0,
   second = 1
};
struct      myList;
enum        SomeENUM;
typedef int Custom_t;

void        computesome();
```

- typedef names must be suffixed with "`_t`".

```c
// Good:
typedef unsigned uint_t;
typedef struct   point
{
   int           x;
   int           y;
}                point_t;
```

```c
// Bad:
typedef unsigned uint;
typedef struct   point
{
   int           x;
   int           y;
}                point;
```

- the name of global variables (when at all necessary, see next section), must start with the prefix "`g_`".

4.2.2 Visibility of identifiers

- there must not be any variable definition in the global scope (either `static` or non-`static`), unless explicitly authorized by an assignment.
- there must be at most 5 non-`static` definitions per `.c` file.
- the prototype of each non-`static` functions must be declared exactly once per project, in some `.h` file. (Different declarations may be split into different `.h` files.)
- the `.h` file where a non-`static` function prototype is declared, must be included by the `.c` file where the function is defined and all files where the function is used.
- all `static` function prototypes must be declared exactly once at the beginning of the `.c` file before they are defined.
- all `static` functions must be used.
- all local variables (including function parameters) must be used.

4.3 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 can contain at most 25 lines of code (opening and closing braces excluded).
- function bodies must not contain any comments; comments that explain a function should be placed before the function definition.
- 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");
 }
}
```

- opening and closing braces for statement blocks must always appear on a line of their own, aligned with one another.

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

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

- the opening brace for a struct, union or enum declaration must appear on a line of its own; the closing brace must be the first token on a line, aligned with the opening brace; if a closing brace is followed by a declarator, then the declarator must begin on the same line.

```
// Good:
struct point
{
   int x;
   int y;
};
typedef enum
{
   FALSE = 0,
   TRUE = 1
} bool_t;
```

```
// Bad
struct point
{
   int x;
   int y; };

typedef enum {
   FALSE = 0,
   TRUE = 1
}
bool_t;
```

- 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);
        }
```

- at most one variable can be declared by line.

```
// Good:
int foo(int x, int y, int z)
{
  int u;
  int v;

  u = x + y;
  v = y - z;
  return u * v;
}
```

```
// Bad:
int foo(int x, int y, int z)
{
  int u, v;

  u = x + y;
  v = y - z;
  return u * v;
}
```

- all variable declarations within a scope must appear at the start of the scope.

```
// Good:
int foo(int x, int y, int z)
{
  int u;
  int v;

  u = x + y;
  v = y - z;
  return u * v;
}

void bar()
{
  int i;

  for (i = 0; i < 10; ++i)
  {
      int j;

      printf("%d\n", i);
      j = i - 1;
      printf("%d\n", j);
  }
}
```

```
// Bad:
int foo(int x, int y, int z)
{
  int u;

  u = x + y;
  int v;

  v = y - z;
  return u * v;
}

void bar()
{
  for (int i = 0; i < 10; ++i)
  {
      printf("%d\n", i);

      int j = i - 1;
      printf("%d\n", j);
  }
}
```

- the declarations must be separated from the first statement with a blank line.

```
// Good:
int foo(int x, int y, int z)
{
  int u;
  int v;

  u = x + y;
  v = y - z;
  return u * v;
}
```

```
// Bad:
int foo(int x, int y, int z)
{
  int u;
  int v;
  u = x + y;
  v = y - z;
  return u * v;
}
```

- all declared identifiers in a group of declarations must be aligned on the same text column within a scope, using spaces between the "declaration specifiers" and the "declarator" parts.

```
// Good:
int          x;
char         *p;
long long    y;
void         (*z)(int, int);
struct       t
{
    char     u;
    char     v;
};
```

```
// Bad:
int x;
char*        p
long         long y;
void (*z)(int, int);
struct t
{
  char u;
  char v;
};
```

- variable initializations must be separated from variable declarations, except when the variable is `static` where there must always be an initializer in the declaration:

```
// Good:
int         x;
static int y = 1;


x = 1;
```

```
// Bad:
int         x = 1;
static int y;
```

- 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.

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

```c
// 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.

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

```c
// 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 ;
```

- the `switch` and `goto` statements must not be used.

5 Compiler warnings

All C code must compile without errors or warnings when the following compiler flags are provided:

`-std=c11 -Werror -pedantic -Wall -Wextra -Wformat=2 -O -Wuninitialized -Winit-self -Wswitch-enum -Wdeclaration-after-statement -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wconversion -Waggregate-return -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -Wnested-externs -Wno-long-long` (for Clang only:) `-Wglobal-constructors -Wshorten-64-to-32`

6 Build rules

- the command:

```
cc -I. -c *.[cS]
```

must always complete successfully (with all flags described in Compiler warnings above). In particular, all source files must be present at the toplevel directory, and no invalid/incomplete source files must remain laying around.

- if a `Makefile` is submitted, then:
  - "`all`" must be the first target, i.e. "`make`" without argument must be equivalent to "`make all`".
  - "`make all`" must complete successfully and produce all targets defined in each assignment.
  - "`make clean`" must complete successfully and remove all generated files. In addition, the tree must be clean to start with, ie. running "`make clean`" just after unpacking the tree must not remove any file.
  - "`make clean && make all`" must complete successfully and result in the same output as "`make all`".
  - "`make clean`" must be idempotent (in particular "`make clean && make clean`" always works).
  - it must be possible to add arbitrary flags to C compilation and linker commands using "`make CFLAGS=...`" or "`make LDFLAGS=...`" on the command line.
  - it must be possible to replace the C compiler or linker using "`make CC=...`" or "`make LD=...`" on the command line.
- if the submission includes a program named "`configure`", then this program must be executable without argument; in addition, it will be executed exactly once before testing for `Makefile` rules as described above.

7 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`).

8 Copyright and licensing