CS 161

C/C++ Programming Style Guideline

Comments

The comments should describe what is happening, what parameters and variables mean, any restrictions or bugs, etc. Avoid comments that are clear from the code. Don't write comments that disagree with the code. Short comments should be *what* comments, such as "compute mean value", rather than *how* comments such as "sum of values divided by n". C/C++ is not assembly; putting a comment at the top of a 3-10 line section telling what it does overall is often more useful than a comment on each line describing micrologic.

Programs should include a program header at the very top of the file, and EVERY function should be preceded by headers blocks, which describe the purpose, usage, and type of any parameters, pre-conditions, post-conditions, and return values. For those unclear on these concepts, a pre-condition is a condition that must hold prior to beginning the function, and post-conditions are conditions that must hold upon exiting the function. As code is updated, these may change, and must be updated accordingly.

```
/*********************************************************************
 ** Program Filename:
 ** Author:
 ** Date:
 ** Description:
 ** Input:
 ** Output:
 *********************************************************************/

/*********************************************************************
 ** Function:
 ** Description:
 ** Parameters:
 ** Pre-Conditions:
 ** Post-Conditions:
 ** Return:
 *********************************************************************/
```

Comments that describe algorithms, data structures, etc. should be in block comment form.

```
/*
 * Here is a block comment.
 * The comment text should be tabbed or spaced over uniformly.
 * The opening slash-star and closing star-slash are alone on a line.
 */

/*
** Alternate format for block comments
*/
```

Block comments inside a function are appropriate, and they should be spaced over to the same indentation setting as the code that they describe. One-line comments alone on a line should be indented to the same space setting of the code that follows.

```
int main() {

   float radius;

   /* Read radius value from user. */
   printf(“Enter the circle’s radius value: ”);
   scanf(“%f”, &radius);

   return 0;
}
```

Very short comments may appear on the same line as the code they describe, and should be spaced over to separate them from the statements.

```
if (a == EXCEPTION) {
        b = TRUE;               /* special case */
}
else {
        b = isprime(a);         /* works only for odd a */
}
```

Whitespace

Use vertical and horizontal whitespace generously. Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 2 blank lines between the end of one function and the comments for the next. For indentation, you can use tabs or spaces, but BE CONSISTENT. If you choose to use spaces, then I suggest you use at least 3 spaces for indenting because it is hard to see 1 or 2 spaces.

It is a good idea to have spaces after commas in argument/variable lists and after semicolons in *for* loops to help separate the arguments/variables and statements visually.

```
int length, width;

int calculate_rectangle_area( int length, int width) {

for(i = 0; i < MAX_SIZE; i++) {
```

Operators should be surrounded by a space. For example, use

```
z = x + y
```

rather than

```
z=x+y
```

This greatly enhances readability, and makes it significantly easier to spot operators within an expression. Prefix and postfix increment and decrement are not considered operators in this context.

Variable Declarations

Related declarations of the same type can be on the same line, but you should put unrelated declarations of the same type on separate lines. A comment describing the role of the object being declared should be included, with the exception that a list of #defined constants does not need comments if the constant names are sufficient documentation. The names, values, and comments are usually indented so that they line up underneath each other. Use spaces rather than tabs for alignment.

The "pointer" qualifier, '*', should be with the variable name rather than with the type.

```
char    *s, *t, *u;
```
instead of
```
char*   s, t, u;
```

which is wrong, since 't' and 'u' do not get declared as pointers.

Compound Statements

A compound statement is a list of statements enclosed by braces. There are many common ways of formatting the braces. Be consistent with your local standard, if you have one, or pick one and use it consistently. When editing someone else's code, always use the style used in that code.

```
control {
          statement;
          statement;
}
```

When a block of code has several labels (unless there are a lot of them), the labels are placed on separate lines. The fall-through feature of the C/C++ switch statement, (that is, when there is no break between a code segment and the next case statement) must be commented for future maintenance.

```
switch (expr) {
  case ABC:
  case DEF:
        statement;
        break;
  case UVW:
        statement; /*FALLTHROUGH*/
  case XYZ:
        statement; break;
}
```

Here, the last break is unnecessary, but is required because it prevents a fall-through error if another case is added later after the last one. The default case, if used, should be last and does not require a break if it is last.

Naming Conventions

Individual projects will no doubt have their own naming conventions. There are some general rules however.

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Function, typedef, and variable names, as well as struct, union, (C++) class, and enum tag names should be in lower case, with words separated by an underscore.
- Avoid names that differ only in case, like foo and FOO. Similarly, avoid foobar and foo_bar. The potential for confusion is considerable. Similarly, avoid names that look like each other. On many terminals and printers, 'l', '1' and 'I' look quite similar. A variable named 'l' is particularly bad because it looks so much like the constant '1'.
- #define constants should be in all CAPS.
- Enum constants are in all CAPS.

Constants

Numerical constants should not be coded directly. The #define feature of the C/C++ preprocessor should be used to give constants meaningful names. Symbolic constants make the code easier to read. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available. Some specific constraints:

- Constants should be defined consistently with their use; e.g. use 540.0 for a float instead of 540 with an implicit float cast.
- There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a for loop indexes through an array, then

  ```
  for (i = 0; i < ARYBOUND; i++)
  ```

  is reasonable.

- Always compare pointers to NULL, rather than 0.
- Even simple boolean values like 1 or 0 are often better expressed using defines like TRUE and FALSE (sometimes YES and NO read better).

Line Length

Lines should be a decent length. Try not to have your code wrap on a terminal because it lowers readability.

A long string of conditional operators and output statements should be split onto separate lines. When separating a statement or conditional operators, make sure the separation is logical and readable, i.e. look at the example below.

```
if (foo->next == NULL && totalcount < needed
&& needed <= MAX_ALLOT &&
server_active(current_input)) {
        ...
}
```

might be better as

```
if (foo->next == NULL
    && totalcount < needed
    && needed <= MAX_ALLOT
    && server_active(current_input)
    ){
        ...
}
```