C++ Language Coding Requirements

Note: These coding guidelines are derived from the book *Big C++, Second Edition* by Cay S. Horstmann and Timothy A. Budd, published by John Wiley & Sons and the Google C++ Style Guide (http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml). They are copyright © 2009, by John Wiley & Sons. All Rights Reserved. Revised by Helmick and Talaga.

Introduction

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses. It lays down rules that you must follow for your programming assignments.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate the consistency if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it *reduces gratuitous choice*. If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines a number of constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are of marginal utility and can be expressed just as well or even better with other language constructs.

If you have already programmed in C or C++, you may be initially uncomfortable about giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat long and dull. They also mention features that you may not yet have seen in the class. Here are the most important highlights:

1. Tabs are set every two spaces.
2. Variables are lowercase, with _ between words.
3. Function names are camelcase.
4. Class names start with an uppercase letter.
5. Constant names are all uppercase.
6. Ending braces must line up with the statement that started the block.
7. No magic numbers may be used.
8. Every function must have a comment.

9. At most 30 lines of code may be used per function.
10. No line can be longer than 80 characters.
11. All flow and conditional statements must use {..}
12. At most two global variables may be used per file.
    1. Zero global variables allowed in class definition files

Source Files

Each program is a collection of one or more files or modules. The executable program is obtained by compiling and linking these files. Organize the material in each file as follows:

- Header comments
- `#include` statements
- Constants
- Classes
- Global variables
- Functions

Note some files may not contain all of the above components. See Multi-File Programs below.

It is common to start each file with a comment block. Here is a typical format:

```
/*    @file invoice.cpp
      @author Jenny Koo
      @date 2010-01-24
*/
```

You may also want to include a copyright notice, such as

```
 /* Copyright 2010 Jenny Koo */
```

A valid copyright notice consists of

- the copyright symbol © or the word "Copyright" or the abbreviation "Copr."
- the year of first publication of the work
- the name of the owner of the copyright

Next, list all included header files.

```
#include <iostream>
#include "ccc_empl.h"
```

Do not embed absolute path names, such as

```
 #include "c:\me\my_homework\widgets.h"  // Don't !!!
```

After the header files, list constants that are needed throughout the program file.

```
const int GRID_SIZE = 20;
const double CLOCK_RADIUS = 5;
```

Then supply the definitions of all classes.

```
class Product   {
     ...
};
```

Order the class definitions so that a class is defined before it is used in another class. Very occasionally, you may have mutually dependent classes. To break cycles, you can declare a class, then use it in another class, then define it:

```
class Link; // Class declaration
class List   {      ...
  Link* first;
};
class Link // Class definition
{      ...
};
```

Continue with the definitions of global variables.

```
ofstream out; // The stream for the program output
```

Every global variable must have a comment explaining its purpose. Avoid global variables whenever possible. You may use at most two global variables in any one file.

Finally, list all functions, including member functions of classes and nonmember functions. Order the nonmember functions so that a function is defined before it is called. As a consequence, the `main` function will be the last function in your file.

Multi-File Programs

As our programs grow in size we'll need to split the source files into smaller components to keep the programs manageable. Each class should consist of two files, a definition file in the form of a .h file (header), and an implementation file .cpp. Both files should have the same prefix filename, which matches the classname.

For example the LList class should be implemented in LList.cpp and defined in LList.h.

ONLY class definitions and should exist in a header file and ONLY class implementations should exist in the .cpp file. It is allowable for a class method to be implemented in an .h file only if it can fit on the same line as the definition.

Both files need to have header comments. Definition files (.h files) need to have detailed method descriptions.

Functions and Methods

Supply a comment of the following form for every function.

```
/**   Explanation.
```

```
      @param argument_1 explanation
      @param argument_2 explanation
      ...
      @return explanation   */
```

The introductory explanation is required for all functions except `main`. It should start with an uppercase letter and end with a period. Some documentation tools extract the first sentence of the explanation into a summary table. Thus, if you provide an explanation that consists of multiple sentences, formulate the explanation such that the first sentence is a concise explanation of the function's purpose.

Omit the `@param` comment if the function takes no parameters. Omit the `@return` comment for `void` functions. Here is a typical example.

```
/**   Converts calendar date into Julian day. This algorithm is
      from Press et al., Numerical Recipes in C, 2nd ed.,
      Cambridge University Press, 1992.
      @param year  the year of the date to be converted
      @param month the month of the date to be converted
      @param day the day of the date to be converted
      @return the Julian day number that begins at noon of
      the given calendar date   */
long dat2jul(int year, int month, int day){
 ...
}
```

Parameter names must be explicit, especially if they are integers or Boolean.

```
Employee remove(int d, double s); // Huh?
Employee remove(int department, double severance_pay); // OK
```

Of course, for very generic functions, short names may be very appropriate.

Do not write `void` functions that return exactly one answer through a reference. Instead, make the result into a return value.

```
void find(vector<Employee> c, bool& found); // Don't!  bool
find(vector<Employee> c); // OK
```

Of course, if the function computes more than one value, some or all results can be returned through reference parameters.

Functions must have at most 30 lines of code. (Comments, blank lines, and lines containing only braces are not included in this count.) Functions that consist of one long `if/else/else` statement sequence may be longer, provided each branch is 10 lines or less. This rule forces you to break up complex computations into separate functions.

Local Variables

Do not define all local variables at the beginning of a block. Define each variable just before it is used for the first time.

Every variable must be either explicitly initialized when defined or set in the immediately following statement (for example, through a `>>` instruction).

```
 int pennies = 0;
```

or

```
 int pennies;
 cin >> pennies;
```

Move variables to the innermost block in which they are needed.

```
 while (...){
      double xnew = (xold + a / xold) / 2;
      ...
}
```

Do not define two variables in one statement:

```
 int dimes = 0, nickels = 0; // Don't
```

When defining a pointer variable, place the `*` with the type, not the variable:

```
 Link* p; // OK
```

not

```
 Link *p; // Don't
```

Constants

In C++, do not use `#define` to define constants:

```
 #define CLOCK_RADIUS 5 // Don't
```

Use `const` instead:

```
 const double CLOCK_RADIUS = 5; // The radius of the clock face
```

You may not use magic numbers in your code. (A magic number is an integer constant embedded in code without a constant definition.) Any number except 0, 1, or 2 is considered magic:

```
 if (p.get_x() < 10) // Don't
```

Use a `const` variable instead:

```
const double WINDOW_XMAX = 10;
if (p.get_x() < WINDOW_XMAX) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice.

Make a constant

```
 const int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365's, 364's, 366's, 367's, and so on in your code.

Classes

Lay out the items of a class as follows:

```
class ClassName{
public:
    constructors
    mutators
    accessors
private:
    data
};
```

All data fields of classes must be private.

Control Flow

The `if` Statement

Always use { …} when using an if statement.

```
if (...)
  stuff          // BAD
  morestuff      // BAD
if (...){
  stuff          // GOOD
}
morestuff
if (...)stuff   // BAD
morestuff
```

Avoid the "`if...if...else`" trap. The code

```
if (...)
  if (...) ...;
else{
  ...;
  ...;
}
```

will not do what the indentation level suggests, and it can take hours to find such a bug. Always use `{...}` when dealing with if statements:

The `for` Statement

Use `for` loops only when a variable runs from somewhere to somewhere else with some constant increment/decrement.

```
for (int i = 0; i < a.size(); i++) print(a[i]);        // BAD
for (int i = 0; i < a.size(); i++){
  print(a[i]);                                          // GOOD
}
```

{} Must be used for all loops.

Do not use the `for` loop for weird constructs such as

```
for (xnew = a / 2; count < ITERATIONS; cout << xnew){ // Don't
```

```
  xold = xnew;
  xnew = xold + a / xold;
  count++;
}
```

Make such a loop into a `while` loop, so the sequence of instructions is much clearer.

```
xnew = a / 2;
while (count < ITERATIONS){ // OK
  xold = xnew;
  xnew = xold + a / xold;
  count++;
  cout << xnew;
}
```

A `for` loop traversing a linked list can be neat and intuitive:

```
for (p = a.begin(); p != a.end(); p++){  cout << *p << "\n";}
```

Lexical Issues

Naming Conventions

The following rules specify when to use upper- and lowercase letters in identifier names.

1. All variable and all data fields of classes are in lowercase, sometimes with an underscore in the middle. For example,`first_player`.
2. Function names are in camelcase. For example `myFunction()`
3. All constants are in uppercase, with an occasional underscore. For example, `CLOCK_RADIUS`.
4. All class names start with uppercase and are followed by lowercase letters, with an occasional uppercase letter in the middle. For example,`BankTeller`.
5. Template type parameters are in uppercase, usually a single letter.

Names must be reasonably long and descriptive. Use `first_player` instead of `fp`. No drppng f vwls. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for five counter variables in your function. Surely each of these variables has a specific purpose and can be named to remind the reader of it (for example,`ccurrent, cnext, cprevious, cnew, cresult`). However, it is customary to use single-letter names such as `T` for template type parameters.

Indentation and White Space

Use tab stops every two columns. Save your file so that it contains no tabs at all. That means you will need to change the tab stop setting in your editor! In the editor, make sure to select "2 spaces per tab stop" and "save all tabs as spaces".

Every programming editor has these settings. If yours doesn't, don't use tabs at all but type the correct number of spaces to achieve indentation.

Use blank lines freely to separate logically distinct parts of a function.

Use a blank space around every binary operator:

```
x1 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a); // Good
x1=(-b-sqrt(b*b-4*a*c))/(2*a); // Bad
```

Leave a blank space after (and not before) each comma, semicolon, and keyword, but not after a function name.

```
 if (x == 0) ...    f(a, b[i]);
```

Every line must fit on 100 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = ..............................................
  + ................;
```

If a line break happens in an `if` or `while` condition, be sure to brace the body in, *even if it consists of a single statement:*

```
 if (..................................................
     &&   ..................         || ..........){
   ...
 }
```

If it weren't for the braces, it would be hard to distinguish the continuation of the condition visually from the statement to be executed.

Braces

Opening and closing braces must line up, either horizontally or vertically.

```
while (i < n) { print(a[i]); i++; } // OK
while (i < n){
  print(a[i]);
  i++;
} // OK
```

Some programmers put the starting `{` *on the next line* of the `while`:

```
while (i < n)
{
  print(a[i]);
  i++;
}
```

This style is allowable.

Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```
 class Employee{
  ...
  private:
    string  name;
```

```
      int  age;
   double  hourly_wage;
     Time  start_time;
};
```

This is undeniably neat, and we recommend it if your editor does it for you, but *don't* do it manually. The layout is not *stable* under change. A data type that is longer than the preallotted number of columns requires that you move *all* entries around.

Some programmers like to format multiline comments so that every line starts with `**`:

```
 /* This is a comment
** that extends over
** three source lines   */
```

Again, this is neat if your editor has a command to add and remove the asterisks, and if you know that all programmers who will maintain your code also have such an editor. Otherwise, it can be a powerful method of *discouraging* programmers from editing the comment. If you have to choose between pretty comments and comments that reflect the current facts of the program, facts win over beauty.