CSCI at CU: C++ Style Guide

Version 4.2 - January 23, 2010

In order to allow easy reading and understanding of everybody's programs, and to avoid risky programming practices, all software organizations have guidelines that programmers must follow. This page contains the guidelines we require you to follow this semester.

The Ten Essentials

1. A comment at the top of the program lists the file name, your name, your e-mail address, and a short description of the purpose of the program.
2. Constants are declared with ALL CAPITAL LETTERS. Variables, parameters and function names are declared with all lower-case letters.
3. No global variables. Instead, every variable is declared AT THE TOP of the function it is used in (right after the open curly bracket). Each variable has a short, but meaningful name and a short comment indicating its purpose.
4. Each curly bracket is on a line by itself (or with a comment).
5. Function prototypes appear before the main program in alphabetical order with a comment that indicates how to use each function. The comment must mention the purpose of every parameter.
6. Function definitions (the "implementations") must appear after the main function in alphabetical order. Every function must have a a blank line and a line of dashes immediately before its defintion; immediately after the definition is another line of dashes and then another blank line.
7. No line exceeds 80 characters.
8. Each level of indentation (such as the body of a function or the body of a loop) is indented 4 extra spaces.
9. Blank lines are used in a meaningful way. For example, if there are six lines of code that perform some job, then put a comment before the six lines, and one blank line afterward.
10. Formulas are nicely spaced with a space around every operator (except *, / and %), and a space after every comma in an argument list.

The complete style guide is listed below, but you might also look at a real-world programming stylesheets.

At the same time, this page defines ***abbreviations*** that instructors can use in providing feedback on your programs. For example, if we mark II on a program you turn in, this sheet tells you that we think you have used incorrect indenting.

So, here's a list of bad things we'll be looking for in your programs, and the abbreviations we'll use to notify you if we find them.

FCI File comment incomplete.

> Each .cxx or .h file must have a comment at the top indicating the file name, the name and email of the programmer (that's you), the date it was written, and a

concise but complete description of the purpose.

LTL Line too long.

Please keep all lines under 80 characters so that when the program is printed, the resutl is still readable.

II Incorrect indenting.

There are many ways people indent their C++ code, but the *one way* that we'll accept is described here. This way has some advantages in readability, but more than that (1) if you are all consistent it helps us read your code, and (2) in real life you'll have to get used to following specific guidelines in your code, so why not get some practice now.

The rules are:

- (a) opening curly braces are always placed on a line by themselves immediately below the first character of the keyword that controls the code in the brace, such as for, if, while or else, or the header of a function. This line may also contain a comment starting with //.
- (b) if you prefer, the entire body of a function may be indented an extra four spaces beyond the line that contains the function name and parameters.
- (c) lines inside braces are indented four spaces to the right of the brace
- (d) closing curly braces always occur on lines by themselves immediately below the corresponding opening brace

  (e) With an if, else, for, while or similar construct that controls a single statement, without curly braces, indent the statement four spaces.

For example...

```
void foo(int x, int y)
{
    int i;
    for (i = 0; i < N; i++)
    {
        cout << i << endl;
        cout << i*i << endl;
        if (i > M)
        {
            cout << "M exceeded" << endl;
        }
    }
}
```

WSS White space and slashes.

Programs are more readable if the code has meaningful white space separating parts. In particular:

- Put two blank lines and one long line of slashes before every function defintion
- Put one long line of slasshes after every function definition
- Put one blank line after the local variable declarations at the top of a function
- Put one blank line before each "//" comment that describes the purpose of a group of lines

One space should occur before and after all binary operations in an expression (except for multiplication, division and % operator). This includes the input (>>), output (<<) and assignment (=) operators. Also one space after keywords `for`, `while`, `if`, and one space after each semi-colon in the control of a for-loop.

BN Bad name.

Choose names for functions and variables that convey what they are used for. Variables that are just used for indexing or counting in a loop should have short names such as `i` or `n` or `t`.

CN Capitalization and naming style:

You may use one of two naming conventions, so long as you are consistent. Most programmers will probably prefer the underscore style, but much Win32 programming uses the Hungarian notation developed by Charles Simonyi.

Underscore Style:

Variable names, function names and new class names are all lower-case letters. Names of constants are all upper-case letters. If a name consists of several words, then these words are separated by an underscore (such as `plant_age` or `MANY_COLUMNS`). Functions that do not return a value (void function) should have a verb as the first part of the name (such as `print_table`). Functions that return a bool value should have the word "is" as the first part of the name (such as `is_inside`). Other functions that return a value should have a noun or noun phrase as the name (such as `feet_per_second`).

Hungarian Notation:

Variable names, function names and new class names all have their first letter of each word capitalized. Constants are all capitals with no underscores. Any variable can have a prefix to indicate its type or purpose. Prefixes that I have seen:

b	Boolean	bool bInside;
by	BYTE (an 8-bit value)	BYTE byOffset;
c	Character	char cLetterGrade;
cb	Count in Bytes	UINT cbBuffer;
d	Double Floating-Point Number	double dAltitude;
dw	DWORD (a 32-bit value in WIN32 programming)	DWORD dFlags;
f	Floating-Point Number	float fHeight;
fn	Function	void repeat(void fnWhatToRepeat( )...
h	Handle	HANDLE hBrush;
hdc	Handle to a Device Context	HDC hdcDrawingArea;
hwnd	Handle to a Window	HWND hwndScoreDisplay;
i	Integer (32-bits in WIN32 programming)	int iPixelsPerInch;
if	Input File Stream	ifstream ifCorpus;
is	Input Stream	istream isKeyboard;
msg	A Windows Message	MSG msg;
n	An Integer That Counts the Number of Instances of Some Data	int nStudents;
of	Output File Stream	ofstream ofResults;
os	Output Stream	ostream ofScreen;
si	Short Integer	short siCents;
str	C++ String Object	string strSurname;
sz	A Null-Terminated C String	char szSurname[MAXLENGTH];
w	WORD (which is still 16-bits in WIN32 programming)	WORD wFlags;
x,y	Short used as Cartesian coordinates	short xLocation, yLocation;
struct or class name (or abbreviation)		point pointWhere; statistician statGrades;

Certain modifiers can appear before the Hungarian prefixes. Sometimes you'll see m_ as a prefix for a member variable, for example, though Iy dislike that one. Some others:

ar or rg	Array (rg means "range")	double rgdScores[MANYSCORES];
l	Long	long double dAltitude;
p or lp	Pointer (always long 32-bits in WIN32) to a single item	char* lpszWindowName;

s	Static	static int siCurrentValue;
u	Unsigned	unsigned int uiPopulation;

GV Global variable.

Global variables are forbidden. Global constants are allowed.

DC Define constant.

If the same value occurs in more than one place in the code, define a constant for that value. If a number comes from some physical value with an easily described name, define a constant for that number. Also, if 5,6, and 7 (say) occur in the program, with 5 arising from 6-1 and 7 from 6+1, define a constant for 6 (say C) and express the others as C-1 and C+1. This is crucial in making easily changed programs.

NC Needs comment.

Each function needs a comment explaining the use of the function. The comment must contain enough information that any programmer could use the function. One form for these comments is to provide a precondition and a postcondition. Make sure that the purpose of each parameter is clearly explained. This comment should appear both with the function's definition and in any header files that provide the function.

If there's something difficult about the code in a function definition, explain it in a comment near the difficult code. Long stretches of code should be broken into smaller pieces (around 6-10 lines per piece) with a short comment at the top of each section to explain its purpose. A programmer should be able to read these comments to get a general idea of a program's flow.

The implementation file for any class requires a comment at the top to tell a programmer exactly how all the member variables are used.

UC Unnecessary comment.

With carefully choosen names, you'll need few comments, and a programmer can read these few comments to get a general idea of the program flow (or to understand difficult parts). Comments that explain the obvious are bad; leave them out.

CTC Code too complex.

Find a simpler approach.

FTL Function too long.

A function should have no more than four dozen lines after the declarations. We'll allow an exception if the function is little more than

one large switch statement, and none of the cases get too long.

FP Function format problem.

Use value parameters or const reference parameters to get info into functions. Use `return` to produce values, if you can, and use non-const reference parameters only when you have to (when the function produces several pieces of information). Avoid functions that return information in two different ways (through the `return` statement and also through its parameters).

BUI Bad user interface.

Is output clearly labelled, so the user can tell what the program has done? If user input is requested, is there a reasonable prompt? Is the user asked to provide information that the program should be able to figure out for itself?

DNW Does not work.

This part of the code won't do what it needs to.

WNC

Will not compile.

WARN

Program must compile under the -Wall option with no compiler warnings.

PDR Poor data representation.

An example would be using separate variables to hold information that could be more easily handled in an array.

MD Misplaced declaration or definition.

All variable declarations must be together at the top of a function's definition. This makes it easier for a programmer to find the declarations. For a complete program, function prototypes should precede main() and function definitions should follow main().

NMR Needs to be more robust.

If the user makes an error this part of the program could blow up or behave unpredictably.

NGE Not general enough.

This marks a place where the program is unnecessarily limited in what it can handle. For example, if a program can handle only input names with up to ten letters, then the program is unneccessarily restrictive.

Surprisingly, more general programs are often shorter and simpler, not longer and more complex, than ones which handle special cases.

CF Combine functions.

The code uses two or more different functions to do very similar jobs. This mark says that one single function with parameters would be a better approach.

PORT Portability.

Avoid using non-portable code (such as the conio.h header file). The only exception that we'll allow is the use of winbgim.h for graphics.