C++ Style Guide for CS161A/B and CS162

Introduction

This simplified style guide is intended to help beginner C++ programmers adhere to basic coding conventions. Properly styled computer code is more easily read and understood by humans. You may revisit the code you write later, or you may work on code with other people in the future. Your code must be easily understood by yourself and others.

Your code will be graded using the guidelines in this code/style guide in the following areas:

- Coding Guidelines
- Consistency
- Code Layout
- Whitespace in Expressions and Statements
- Comments
- Naming Conventions
- User Experience (UX) / User Interface (UI)

Compiler Versions

The compiler used for this class is:

GNU GCC C++ compiler using Modern C++ on MCECS linux systems
g++ -std=c++17 program.cpp

However, not all features of C++ 17 can be used in these courses! Features like Smart Pointers are not allowed in your code. If you have any questions about any features, please contact your instructor first.

Coding Guidelines

The program must follow the below coding guidelines. Apart from Style guidelines there are some syntactical guidelines to be followed for all our CS classes. They are listed below and not adhering to these guidelines will affect your grade.

1. No global variables - you may use global constants.
2. Do not use "goto" statements
3. Do not use "auto" unless you are creating an STL iterator - which we don't do in any of our CS classes.
4. You may not use while true loops - use the right condition in the appropriate loop.
5. You may not use any breaks, continue, or return statements inside any loops - you are allowed to use breaks inside a switch statement.
6. It is important to note that a single return path and not using "break" means that you need to think your control path through in a careful and holistic manner. Just adding extra boolean variables and extra if statements is usually not the right answer. Remember, "for" loops are for when you know how many times you are iterating and

"while" loops are for when you are iterating while a condition is true (or false). Use the right loop construct for the situation and you almost never need a break.

7. Each method or function should have a single return path. You may not have multiple return statements inside a function.
8. Use only concepts you have learned in each class.
9. Do not use any <vectors> in any of your programs unless you are instructed to do so.
10. Your program must have function prototypes. Place the prototypes for your functions globally, after your #includes. All functions must be implemented after main().
11. You may not use the String class after CS161B. You are required to use char arrays in CS161B, CS162 and CS260.

Consistency

Code is read much more often than it is written. The guidelines provided here are intended to improve code readability and make it consistent across the wide spectrum of C++ code.

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Code Layout

Style Guidelines	Good	Bad

Each statement usually appears on its own line.	`x = 25;`
	`y = x + 1;`
		`x = 25;  y = x + 1;  // No`
		`if (x == 5) { y = 14; }  // No`

A blank line can separate conceptually distinct groups of statements, but related statements usually have no blank lines between them.	`x = 25;`
	`y = x + 1;`
		`x = 25;`
		`// No`
		`y = x + 1;`

Most items are separated by one space (and not less or more). No space precedes an ending semicolon.	`celsius = 25;`
	`fahrenheit = ((9 * celsius) / 5) + 32;`
	`fahrenheit = fahrenheit / 2;`
		`celsius=25;  // No`
		`fahrenheit = ((9*celsius)/5) + 32; // No`
		`fahrenheit=fahrenheit/2 ;`
		`// No`

Sub-statements are indented 3	`if (a < b) {`	`if (a < b) {`

spaces from parent statements. Tabs are not used as tabs may behave inconsistently if code is copied to different editors.
Or you may use the indentation offered by IDEs like Visual Studio, or XCode or Repl.it.
	x = 25;
	y = x + 1;
	}
		x = 25; // No
		y = x + 1; // No
		}
		if (a < b) {
		x = 25; // No
		}

Braces		

For branches, loops, functions, or classes, an opening brace appears at the end of the item's line. The closing brace appears under the item's start.	if (a < b) {
	// Called K&R style
	}
	while (x < y) {
	// K&R style
	}
		if (a < b)
		{
		// Also popular - so this is OK.
		}

For if-else, the else appears on its own line	if (a < b) {
	...
	}
	else {
	// Called Stroustrup style
	// (modified K&R)
	}
		if (a < b) {
		...
		} else {
		// No
		}

Braces always used even if only one sub-statement	if (a < b) {
	x = 25;
	}
		if (a < b)
		x = 25; // No, can lead to error later

Maximum Line Length

Limit all lines to a maximum of 79 characters.

Many devices are still limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters.

The preferred way of wrapping long lines is by using C++'s implied line continuation inside parentheses, brackets, and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better.

Blank Lines

Use blank lines in functions sparingly to indicate logical sections. Never use more than one blank line together.

Function Definitions

Every C++ program has a special function named main. The execution of all C++ programs begins with the main function.

Whitespace in Expressions and Statements

Whitespace should be used sparingly. Spaces between operators, variables, and other items make your code easier to read by other humans. The compiler ignores whitespace unless it is made meaningful such as in a string:

```
string userString = " leading space, and trailing space included ";
```

Yes:

```
varThree = varOne + varTwo;     // good whitespace usage
```

No:

```
varThree=varOne+varTwo;    // bad coding practice
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. Suppose a comment is a phrase or sentence, in that case, its first word should be capitalized unless it is an identifier that begins with a lowercase letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

Header Comments

Header comments appear at the top of a file. For class assignments, headers will include your name, assignment number, date, description of the program, input/output list, and a sources list.

```
/*******************************************************************************
# Author:           (programmer's name)
# Assignment:       (fill in with Assignment 1, etc.)
# Date:             (fill in)
# Description:      Example: This program prompts a user for their name and
#                   prints a message to the screen.
# Input:            (What the program asks for, and data type, e.g., string)
# Output:           (Summary of messages displayed by the program)
# Sources:          Lab 1 specifications
#                   and any other substantial aids, like web pages or students
#                   who helped you.
#******************************************************************************
/
// Neither comments nor code should be wider than 79 characters.
// The lines of asterisks above are 79 characters long for easy reference.
// You don't need to include these three lines in your program!
```

Function Comments

Function comments (including the main() function) are like the description part of a header comment but contain information specific to a function. These comments should also include a description of the purpose and expected input arguments, and the expected output values.

```
//Name:   promptName()
//Desc:   This function reads a string from the user and returns it
//input:  None
//output: prompt
//return: string name
string promptName()
{
   string name;
   cout << "Enter name: ";
   cin >> name;
   return name
}

//Name:   findSum(int num1, int num2)
//Desc:   This function adds 2 numbers and returns the sum
//input:  2 integers
//output: None
//return: int (sum of 2 numbers)
int findSum(int num1, int num2)
{
   return num1 + num2;
}
```

Block Comments

Block comments generally apply to some (or all) code that follows them and are indented to the same level as that code. Each line of a block comment starts with a //. Unobvious things and calculations must be explained.

```
int findSum(int num1, int num2)
{
   // this is an example of a block comment
   // notice it lines up with the same indentation
   // as the block it is commenting on
   // return the sum of the two arguments
   return num1 + num2;
}
```

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. At least two spaces should separate inline comments from the statement. They should start with a // and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 // Increment x
```

But sometimes, this is useful:

```
x = x + 1                 // Compensate for border
```

Naming Conventions

There are various naming conventions used in C++ and other programming languages. The two common ones we will see are lowercase_with_underscore (aka snake_case) and mixedCase (aka lowerCamelCase).

Variables

- Variables should always be declared and initialized to a default value at the top of your functions. Not within your code.
- This is good:

```
double sum_of_squares = 0.0;
string student_name = "";
int total_apples = 0;
```

- This is bad - variables are not initialized when they are declared.

```
int i;
char userKey;
userKey = 'c';
int j;
```

- Variables should always be declared within a function - global variables are not allowed, but global constants are permitted. See the section below on Constants.
- For more info see zyBooks Chapter User Defined Functions Part 2, section Scope of variable/function definitions.

Constants

- Constants are declared outside of the main() function.
- Constants are variables whose values don't change once they've been initialized. Use constants instead of "magic numbers" (a number in a program with no meaning), for example, instead of:

```
tax = cost * .0778;
```

- Declare a variable to hold the tax rate and use the constant in calculation. Constants are all UPPER_CASE:

```
double cost = 100;
```

```
const double TAX_RATE = .0778; //TAX_RATE cannot be changed in the program
double tax = cost * TAX_RATE;
```

Names to Avoid

- Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single-character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero.
- In general, try to avoid using single character names. There are some times when using x, y, z for axis names, or a, b, c for the sides of a triangle will make sense. For most other cases, create names that have meaning for their function.

User Experience (UX) / User Interface (UI)

Introduction Message

Always include a welcome message explaining what the program will do. Keep the output line length at an 80 character maximum.

```
This program will prompt you for an item’s cost
and discount percentage and will display the total cost.
```

Prompts (Input)

- Include example input in the prompts if not obvious:

  ```
  Enter the percentage as a whole number (10, not .10 for 10%):

  Do you want to continue (y/n):
  ```

- Always add a space at the end of the prompt; the user should not type next to the last character.

  Example (user input is in **blue**):

  Yes:

  ```
  Enter integer #1: 4
  ```

  No:

  ```
  Enter integer #1:4
  ```

- Use natural language, for example, when prompting the user to enter 5 integers, start at integer #1, not #0:

  Yes:

  ```
  Enter integer #1: 4
  Enter integer #2: 2
  Enter integer #3: 23
  Enter integer #4: -1
  Enter integer #5: 7
  ```

  No:

  ```
  Enter integer #0: 4
  Enter integer #1: 7
  Enter integer #2: 2
  Enter integer #3: 23
  Enter integer #4: -1
  ```

Output

- Your output should look professional. Use uppercase for the first letter of a sentence and one blank line between input and output.
- Check for spelling errors and too many contiguous spaces in a sentence.

  Yes:

  ```
  Enter your name: Sally
  Hello Sally!

  Enter integer #1: 2
  Enter integer #2: 3

  The sum of 2 and 3 is 5.
  ```

  No:

  ```
  enter your name:Sally
  hello Sally !
  Enter: 2
  Enter:3
  The sum of     2 and3is 5
  ```

- Always use labels in your output, avoid naked numbers:

  Yes:

  ```
  Enter the rectangle height in inches: 5.5
  Enter the rectangle width in inches: 3

  Rectangle area: 16.5 inches square.
  ```

  No:

  ```
  Enter the rectangle height in inches: 5.5
  Enter the rectangle width in inches: 3
  16.5
  ```