# CS Department Style Guidelines for Programming in C++

One of the goals of modern software engineering is to craft programs so that they are well designed, readable, and maintainable. Programming style guidelines go along way toward insuring that is the case. Most large programming shops have such guidelines and they will require you to follow them. The programming style guidelines for the C++ language are to be used in this course:

## Identifiers

Use names that have semantic meaning. Avoid single character, very short, or very long names.

***Examples:***

**Semantic Names**  **Baffling Names Six Months From Now**

amount a

isFinished xl

Follow the following C++ language naming conventions.

### **Constants**

###### All upper case with words separated by an underscore

***Examples:*** SIZE

CONVERSION\_FACTOR

### **Classes and Interfaces**

Title case (capitalization of the first letter in each word)

***Examples:*** SimpleCalc

ActionListener

### **Methods and Variables**

Lower case for the first word and title case for every word thereafter.

***Examples:*** name

makeDeposit( )

### **GUI Variables**

Use the same naming conventions as regular variables and then append a three-character GUI component suffix. The trailing extension helps distinguish between the non-visual representation of the data (model) and the associated GUI component (view). It also makes it easier to cluster related GUI components without having name clashes.

##### ***GUI Component Suffix Example***

Button Btn calculateBtn

ComboBox Cbo examScoresCbo

Menu Mnu fileMnu

MenuItem Mni saveAsMni

TextField Txf loanAmountTxf

##### Memo Mmo displayMmp

### **Reserved words**

#### Reserved words will be all lower case

#### ***Examples:*** true

class

## Block Delimiting and Indentation

### **Braces**

Even though the C++ language does not always require braces for some statements it is good programming practice to provide them. Use braces liberally to visually delimit the beginning and end of code blocks. Including braces now avoids the possibility of errors creeping into your code when you add additional statements at the last minute.

Place the opening (left) brace **{** so that it lines up with the left side of class header, method header, conditional statement, or repetitive statement. Place the closing (right) brace **}** in the same column as the opening brace. Always enter braces in opening/closing pairs to avoid forgetting to add one or the other or both. For braces that span more than three to five lines, comment the ending brace to indicate its nature (e.g., // end if ).

***Example:***

class GuideLines

{

public void reviewCode( )

{

if ( meetsGuidelines )

{

cout << “Proceed to the next assignment”;

}

else

{

cout << “Rework your documentation”;

} // end if/else

} // end reviewCode( )

} // end class GuideLines

### **Indentation**

As you move from block to block, indent a minimum of two spaces. A good text editor will assist in the necessary tabbing.

### **Spacing**

Although the C++ compiler ignores most spacing, placing white space around code elements makes reading code easier.

### **Binary Operators**

One space before and after the binary operator.

***Examples:***  2 + 2

945 >= 234

### **Parameter Lists**

One space after the comma that separates each parameter.

***Example:*** drawString(drawingSurface, 25, 50);

Checkbox( “large”, sizeGrp, true);

### **Parentheses**

One space before or after the adjoining parenthesis when you have series of nested parenthesis, otherwise no space is necessary.

## Comments

### **In-Line Comments**

Use in-line comments ( // ) for most of your commenting. Semantic identifiers go a long way toward eliminating the need for most comments. In-line comments above your code are easier to maintain. However, in-line comments on the same line are best for documenting a brace.

### **Multi-Line**

Use the multi-line comment ( /\* \*/ ) to “comment out” code for debugging purposes.

### **Source Code File Prologue**

All source code files should include a prologue of the following form:

// Author: *studentName*

// Assignment: p*Number*

// File: *fileName.cpp*

// Instructor: *InstructorName*

// Class: CSIS *XXXX* Section: *sectionNumber*

// Date Written: *the Date*

// Description: *the description*

### **Function Prologue**

All functions will contain a function prologue of the following form:

// Function Name: *swap*

// Purpose: swaps two integer values

// Parameters: *two integers, passed by reference*

// Returns: *none*

// Pre-conditions: none

// Post-conditions:none

## Magic Numbers

A magic number is any numeric literal other than 1, 0, or –1 used in your program. However if 1, 0 and –1 are used to represent something other than the integers 1, 0, or –1 they will be considered magic numbers. Unfortunately, most code you will see in Java books or programming books in general will include magic numbers because it’s easier to code in the short run. In the long rung, six months from today, you will be clueless as to what the number means. Therefore, DON’T USE MAGIC NUMBERS in your assignments.

So how do you avoid magic numbers? Define constants in you class or methods using semantic identifiers for each magic number.

***Example:***

const int SIZE = 10;

TextField nameTxf = new TextField( SIZE );

## User Prompts and Output Labeling

It is expected that all output will be appropriately labeled. Don’t just output a number without telling the user what it is. All prompts issued to the user will be clear, courteous, and words will be spelled correctly.