Programming Style and Organization Guide

As you begin programming, we want to provide you with a guide covering some of the best practices when it comes to coding style and organization. Writing in a clear, consistent and readable style should not an afterthought to composing your programs because your stylistic habits directly affect the usefulness and maintainability of your code. The computer, of course, doesn't care how you format your code. However, we don't just write code for the compiler, but for other programmers to read and tinker with. When you are reading thousands of lines of code day after day, having that code adhere to a set, predictable standard reduces your cognitive load, allowing you to process information more effectively. When other programmers go to modify code you have written, keeping up a good style helps them understand your code and avoid making errors.

Generally speaking, there are many opinions as to what constitutes the "best" coding style. What matters more in this course is not the convention you choose, but that your code is consistent within a single assignment. That means you are free to experiment with different styles from one homework to the next, but not *within* a single homework. In general, you should strive to write in whatever style other programmers are using as the convention for the language you are working with. If you are curious, you may explore some style guides written for C++, but this is not required.

The purpose of this guide is not to be the final, definite word on every stylistic issue, but to offer a broad introduction to some of the most common and basic issues students face when writing code for the first time. Please look over these items carefully and follow them as you write your code; we will give warnings and may take off points for poor style.

Item 1: Brace Position

- There are, generally speaking, two predominant styles with positioning your braces: either they go on the same line, or they are staggered. Choose whichever you prefer and use it consistently within an assignment.

```
// Allman style
int main()
{
    return 0;
}

// K & R style
int main() {
	return 0;
}
```

Item 2: Indentation

- Every line of code that is placed within the same **scope** (i.e. inside a pair of matching curly braces) *must* have the same indentation.

```
// BAD INDENTATION EXAMPLE
int main()
{
	if (x < 0) {
		a += 5;
```

```
                b = a;                          // wrong indentation
        }

        for (int i = 0; i < maxNum; i++) {
        if (arr[i] == null) {                   // wrong indentation
                return null                     // wrong indentation
                }                               // wrong indentation
                arr[i] = 5;
        }

        return 0;
}



// GOOD INDENTATION EXAMPLE
int main()
{
        if (x < 0) {
                a += 5;
                b = a;
        }

        for (int i = 0; i < maxNum; i++) {
                if (arr[i] == null) {
                        return null
                }
                arr[i] = 5;
        }

    return 0;
}
```

Item 3: Spaces around operators

- You must always include spaces around all arithmetic operators and most Boolean operators.

```
// BAD EXAMPLE
x = c+5;

result=(x*x)/2;

apples==oranges;

// GOOD EXAMPLE
a += b - c * (4 + d);

arg1 <= 10;

return (!arr.contains(num));
```

Item 4: Variables names should be written in either camelCase or snake_case

- All variables names should use a consistent naming convention: either camel-case or snake-case

```
// BAD EXAMPLE
std::string mybestfriendsname = "";
```

```
long bad_snakecaseExample = 132598317183  // mixing styles

// GOOD EXAMPLE
int lengthOfStandardRulerInInches = 12;

bool isFoundInArray = false;

double return_rate_of_annual_compunded_interest = 0.08;
```

Item 5: All constants must be written in UPPER_SNAKE_CASE

- All constant variables must be declared in upper snake-case.

```
// BAD EXAMPLE
const int maxNumberOfTickets = 100;

// GOOD EXAMPLE
const int DEFAULT_ARRAY_LENGTH = 10;
```

Item 6: Use descriptive variables names

- Always give your variables descriptive names that immediately make clear what the variable is representing.
- Don't worry so much about character length when naming variables.
- Compare the two examples below. Both represent code that does the same exact thing, but which one is easier to read and understand on a first pass?

```
// BAD EXAMPLE
double mInMl = 1609;
double d = 92.96;
double c = 300000000;
double secs = (d * mInMl) / c;

// GOOD EXAMPLE
double metersInMile = 1609;
double distanceToTheSunInMillionMiles = 92.96;
double speedOfLightInMetersPerSecond = 300000000;
double timeForLightToReachEarthInSeconds = (distanceToTheSunInMillionMiles *
                                            metersInMile) / speedOfLightInMetersPerSecond;
```

Item 7: Use comments judiciously

- In general, it is considered good practice to comment your code. Many students do not comment at all.
- However, it is also possible to over-comment your code. In general, you should strive to make your code as "self-commenting" as possible (see **Item 6**).
- Comments should be used only when you are trying to clarify something for the reader that is not immediately apparent from the code itself; be judicious about when you should do this.
- Typically functions are commented to describe what they do and what parameters they accept. Same applies to classes.

```
// BAD EXAMPLE
// if r is true      <-- this is obvious from the code
```

```
if (r == true) {
	// do something
}

// adding integer variables a and b
int result = a + b;


// GOOD EXAMPLE
// range for valid longitude coordinates is [-180..180]  <-- domain specific knowledge
if (longitudeGpsCoordinate < -180.00 || longitudeGpsCoordinate > 180.00) {
	return -1;
}
```

Item 8: Never use magic numbers

- Magic numbers are literals that are used directly in your code. I.e. they are raw numbers found in the middle of your code.
- Any literal values used in your code should be stored as variables or constants.

```
// BAD EXAMPLE
Process[] procArray = Process[6];
doSomething(procArray, 6);

// GOOD EXAMPLE
const int MAX_NUMBER_PROCESSES = 6;

Process[] processArray = Process[MAX_NUMBER_PROCESSES];
doSomething(procArray, MAX_NUMBER_PROCESSES);
```

Item 9: Declare variables as close as possible to where they are first used

- Do **not** simply declare all the variable at the top of the program!
- While this is fine for very short programs, it does not scale to code spanning hundreds or thousands of lines. Get into the habit of declaring variables close to their initial use now.

```
// BAD EXAMPLE
int a = 10, b = 20, c, temp;

temp = a * b;

// more code...
// more code...
// more code...

c = temp + 5;  // by this point, you may have forgotten what temp or c are


// GOOD EXAMPLE
int a = 10;
int b = 20;
int temp = a * b;

// more code...
// more code...
// more code...
```

```
int c = temp + 5;
```

Item 10: Always use braces around single-statement blocks (e.g. if, while, for, etc.)

- Leaving braces out of single-statement blocks is error-prone
- If you ever want to add more logic to the block (e.g. for debugging), it is very easy to forget that the braces are missing. You will add statements that the compiler will not interpret as being part of the block, but you will probably not get any warnings because the compiler will just execute it as the next line of code in your program.

```
// BAD EXAMPLE
int grade = 85;
if (grade >= 80 && grade <= 89)
	std::cout << "You got a 'B+'!\n";

Node* currentNode = head;
while (currentNode != null)
	currentNode = currentNode->next;

// GOOD EXAMPLE
for (int i = 0; i < totalLength; i++) {
	doOneThingCorrectlyInBraces();
}

if (name == "Robert") {
	std::cout << "Bob" << std::endl;
}
```

Item 11: Avoid excessive line length

- Try to avoid having lines that run on for too-long. The human eye is better at tracking shorter lines.
- Use well-placed line-breaks to make your code more readable. Your IDE will typically auto-format your line break when you hit "Enter."
- Use common-sense to decide when a line is too long. As a loose rule-of-thumb a single line of code should not exceed the width of your screen.

```
// BAD EXAMPLE
std::cout << "Error: This is a really long error message that does exceeds the maximum
permitted length.\n";

// GOOD EXAMPLE
std::cout << "Error: This is a really long error message "
	"that does not exceed the maximum permitted length.\n";

std::vector<std::string> results((std::istream_iterator<std::string>(iss)),
	std::istream_iterator<std::string>());
```

Item 12: Break larger functions up into smaller functions

- You should strive to make your functions as simple and straightforward as possible. Practically, this means that each function should handle no more than one task. If you need multiple tasks

performed within a single function, you should delegate any additional subtasks to other **helper functions** that you can call wherever necessary.

- This improves code readability (because your functions are smaller and easier to comprehend) and reusability (because you modularize your code into discrete, independent blocks).
- As a rule of thumb your function should not exceed the height of your screen. If you find yourself writing an enormous function, start thinking of ways to break it up. A good typical size to aim for is somewhere between 10 – 20 lines of code.

```
// BAD EXAMPLE
void adjust(Time unit, int amount, int temperature) {

	int timeToAdjustInSeconds;

	switch (unit) {
		case Seconds:
		timeToAdjustInSeconds = amount;
		break;
	default:
		timeToAdjustInSeconds = amount * SECONDS_IN_A_MINUTE;
		break;
	}

	double previousRemainingSecondsUntilDone = remainingSecondsUntilDone->get();
	percentageDone->set(percentageUndone->get() - (percentageDone->get() *
			percentageDone)));

	double calculatedNewTime = calculateTimeFromTemperature(temperature);
	double calculatedNewRemainingTime = calculatedNewTime * percentageUndone->get();
	remainingSecondsUntilDone.set(calculatedNewRemainingTime));
}

// GOOD EXAMPLE
void adjust(Time unit, int amount, int temperature) {
	int timeToAdjustInSeconds = getTimeToAdjustInSeconds(unit, amount);

	double previousRemainingSecondsUntilDone = remainingSecondsUntilDone->get();
	percentageDone->set(percentageUndone->get() - (percentageDone->get() *
			percentageDone)));

	double calculatedNewTime = calculateTimeFromTemperature(temperature);
	double calculatedNewRemainingTime = calculatedNewTime * percentageUndone->get();
	remainingSecondsUntilDone.set(calculatedNewRemainingTime));
}

private int getTimeToAdjustInSeconds(Time unit, int amount) {
	int timeToAdjustInSeconds;

	switch (unit) {
		case Seconds:
		timeToAdjustInSeconds = amount;
		break;
	default:
		timeToAdjustInSeconds = amount * SECONDS_IN_A_MINUTE;
		break;
	}
}
```

Item 13: Avoid explicit Boolean values when implicit equivalents can be used.

```
// BAD EXAMPLE
if(isEqual(alpha1, alpha2, alphaLength) == true) {
	return true;
}
else if (isEqual(alpha1, alpha2, alphaLength) == false) {
	return false;
}

// GOOD EXAMPLE
Return isEqual(alpha1, alpha2, alphaLength);
```