Coding and Documentation Standards

General Program Comments

Put a documentation box like this at the top of each `.cpp` or `.h` file that you submit:

```
//*************************************************************************
//
//  Your Name
//  #HarperId
//  Your Course and Section Number
//
//  I certify that this is my own work and where appropriate an extension
//  of the starter code provided for the assignment.
//
//*************************************************************************
```

Variable Names

Most variable names in your program should describe the quantity or item that they represent. For example, if a variable is to hold a student's test average, don't call it "sa"; call it "studentAverage". If a variable is to hold a person's name, don't call it "s" or "n"; call it "name" or better yet, "studentName" or "employeeName". If a variable is to hold the number of students in a class, don't name it "n" or even "num" or "count"; do name it "student_count" or "number_of_students".

The exception to this rule is for temporary variables that have no intrinsic meaning, especially index variables in `for` loops, e.g.:

```
for (int i = 0; i < 10; i++)
    ...
```

When declaring variables, group and format them in a neat and logical manner.

```
// Bad:

int sum=0,num_gizmos=0,gizmoID,square_of_gizmos=0,
i, j;
double avgGizmos, overallAvg, gizmoSDT,
this,that, theOtherThing;
```

```
// Good:

int sum = 0;
```

```
int num_gizmos = 0;
int square_of_gizmos = 0;
int gizmoID;
int i;
int j;

double average_gizmos;
double overall_average;
double gizmoSDT;
double thisThing;
double thatThing;
double theOtherThing;
```

In short, declare each variable on its own line. That way there is a place for its documentation and maintenance is easier. You're less likely to type a semicolon when you meant to type a comma (and vice versa). You're less likely to forget a critical asterisk or ampersand when declaring multiple pointers or references. If (read as when) you change the data type on a single variable you won't mess up your other declarations.

Variable Documentation

Well-chosen, descriptive variable names typically should not require any additional documentation. If you feel you need to document a variable declaration, you can do so with a single line comment like so:

```
int sum = 0;    ///< Sum of student test scores.
```

Line and Section Documentation

Line documentation consists of a comment about a single line of code. Often it is on the same line as the code:

```
average = (double) sum / count;    // Calculate the program average.
```

Multi-line comments can be placed at:

```
// Calculate the area of the triangle: semi is semiperimeter;
// s1, s2, and s3 are sides.
area = sqrt(semi * (semi - s1) * (semi - s2) * (semi - s3));
```

A moderate amount of documentation in the main body of your program may be advisable, but if you name your variables and functions with meaningful names, you should not need much. If you name variables well, many programs won't need *any* Line Documentation. The following line does not need documentation:

```
programAverage = (double) sumProgramPoints / numPrograms;
```

You can use either the

```
/* some comment */
```

or the

```
// some comment
```

format.

Note that the general use of // will allow you to place a /* and */ around a section of code to 'comment it out' without any comments within it messing up your attempt to comment it out.

In C and C++, the preprocessor can also be used to comment out a section of code or even allow a choice of two options:

```
void f(double d)
{
    int i = 5;
    int j = 23;

#if 0
    for (int k=0; k<10; ++k)           // this line won't be compiled into the
program
#else
    for (int k=i; k<j; ++i)            // this line will be compiled into the
program
#endif
    {
        ....
    }
    ....
}
```

Placing comments before a section of code within a function/struct/class (or *Section documentation*) tells the reader about a mult-line section of code that follows. It is usually placed before a loop or decision construct or a series of lines that do a single task. Examples:

```
// Loop to accept and process user-supplied glucose measurements.

// Decide if gizmos or widgets are to be used.

// Calculate and store averages and standard deviations of measurements.
```

Your programs should use a moderate amount of Section Documentation for functions (such as `main()`) that are long or have several distinct tasks. You should put a blank line before any Section Documentation and indent it the same amount as the code block that it describes. Obviously, use of Section documentation is partly a matter of personal judgment, but programs should include at least a minimal amount of Section Documentation. (Or you should also consider the need for it as a reason to factor out the code section into its own function.)

Code Indentation

Class definition bodies, function and method definition bodies, loop bodies, decision bodies, etc. should be indented in a consistent fashion. Each nested structure must have its own level of indenting. In my opinion, three or four spaces is optimal. Two spaces is the absolute minimum for readability, while more than four spaces is pointless.

Braces

Class definition bodies, function and method definition bodies, loop bodies, decision bodies, etc. generally need to be delimited by braces. Here are two common styles of indentation, both of which are acceptable for this class:

```
// Allman style:

#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;

int main()
{
    int i;

    for (i = 1; i <= 100; i++)
    {
        cout << setw(4) << i;

        if (i % 10 == 0)
        {
            cout << endl;
        }
    }

    return 0;
}
```

```
// Kernigan & Ritchie style:

#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;

int main() {
    int i;

    for (i = 1; i <= 100; i++) {
        cout << setw(4) << i;

        if (i % 10 == 0) {
            cout << endl;
        }
    }

    return 0;
}
```

I prefer Allman style. However, the most important thing is to **pick one of these two styles and use it consistently throughout your programs.**

Note that a loop or decision body that contains only a single statement does not need to be delimited by braces. That means the braces around the `if` body in the examples above are optional.

(What constitutes a single "statement" in C++ is sometimes confusing for students, since a "statement" is not necessarily a single line of code. For example, an `if` and its entire body is considered a single "statement" by the compiler. if you're in doubt, code the braces. In fact, it doesn't hurt to code the braces even if they're not strictly necessary - after all, you might later insert another statement into the loop or decision body and forget to add the braces. If you get in the habit of always coding them, that mistake won't happen.)

Indentation with Tabs vs. Spaces

Code can be indented using either tabs or spaces. Programmers disagree violently about which is superior; (see S03E06 of the HBO Series “Silicon Valley”. It does a decent job of describing the pros and cons of the two methods:

Once again, I don't particularly care which method you use **but do so consistently**. I tend to use tabs when writing my own code but spaces when writing example

code for the courses I teach, because I want the code to look the same no matter what editor a student is using to view it.

The default tab stop in most Unix editors is set at eight spaces, so if you decide to indent with tabs you should really figure out how to change that setting in your editor. My preferred tab size is 4.

Unfortunately, IDEs like Dev-C++ and Xcode often default to using a mix of tabs and spaces to perform their auto-indentation, which really combines the worst aspects of both methods of indentation. That's something you can (and should) change in your IDE's editor preferences, because otherwise your code will look like a ragged mess unless the person viewing it has set the tab stops in their editor to the same value you used when indenting. Visual Studio Code is a great editor for configuration.

Use of White Space

Use spaces to make your code more readable to other humans. For example:

```
// Bad:

if( num==0){
num=num+5;
}

for(i=0;i<10;i++)
    { cout<<i<<endl;}

total=calculate_total(amount,tax_rate,tip);
```

```
// Good:

if (num == 0)
{
    num = num + 5;
}

for (i = 0; i < 10; i++)
{
    cout << i << endl;
}

total = calculate_total(amount, tax_rate, tip);
```

This is also for your own good. More readable code will make it easier to spot the inevitable syntax errors that you will make. AND it is easier to use the cursor

movement commands in editors like vim that were designed specifically to navigate source code.

Insert blank lines between logical sections of the program (or between functions) to help the reader find the main sections. Often each of these sections should have Section Documentation to explain its role in the program. For example:

```
#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;

int main()
{
    int num1;
    int num2;
    int greatest_common_factor;

    // Obtain user input of two integers.
    cout << "Enter first number: ";
    cin >> num1;
    ...

    // Calculate Greatest Common Factor.
    ... code to do the calculation ...

    // Display the result.
    cout << "... ";
    ...

    return 0;
}
```

Identifier Naming Conventions

Programmers frequently need to come up with identifiers, names that label the identity of either a unique object or a unique class of objects. For example, things you might need to name in a C++ program include:

- Variables and constants
- Programmer-defined data types (like class or structure types)
- Functions (including member functions / methods)
- Namespaces

There are a number of common styles used when writing multi-word identifiers in C++:

- **Snake case:** In this style, elements of a multi-word identifier are separated with one underscore character (_) and no spaces, with each element written in lowercase. For example:

```
total     student_count     forward_list     find_first_of     out_of_range
```

  This style is used extensively in the C and C++ standard libraries for variables, functions, member functions, class and structure types, namespaces, and header file names.

- **Screaming snake case:** A variant of snake case in which the elements are written in uppercase instead of lowercase. For example:

```
NULL     FLT_MIN     DBL_MAX     STUDENT_H
```

  This style is used in the C and C++ standard libraries for macro names and macro constants (items declared with `#define`). It is also commonly used for constants in other languages.

- **Lower camel case:**

  In this style, elements of a multi-word identifier are written such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation. The first element starts with a lowercase letter. For example:

```
total     iPhone     eBay     studentCount     reloadTableData
```

  Many modern programming languages such as Java and Swift use this style to name variables and functions (including member functions / methods). This is the style I personally prefer.

- **Upper camel case:** A variant of lower camel case in which the first element starts with a capital letter. For example:

```
Student     StreamHandler     AbstractButton     MenuItem
OutOfMemoryException
```

  Modern programming languages such as Java and Swift typically use this style to name programmer-defined data types (class names, structure names, interface and protocol names, etc.).

As with many of the other code formatting conventions discussed in this document, there is no single right answer to the question, "Which style should I use?" It's more important to pick one and use it consistently in your programs. You should also try to avoid using a style in a way that no one else would typically use it - for example, ONLY use screaming snake case for macro names (a'la #define), and ONLY upper camel case for struct or class names.

Function and Method Documentation

Each function or member function in your program should have a documentation box in the following format:

```
/**
 * This first line is a brief description.
 *
 * The rest of the lines are a more detailed description of the
 * function that outlines what it does and anything interesting about
 * how it does it.
 *
 * param: x Description of the first parameter.
 * param: y Description of the second parameter.
 * param: z Description of the third parameter.
 *
 * return: This is where you describe the possible return values.
 * If the function is void then there must not be a @return markup
 * element in this doc box! (Don't document something that does not exist.)
 *
 * note: This is how you can add an optional note about the function that
 *    may be of interest to someone using it.
 *
 * warning: This is how you can add an optional warning to a user of the
 *    function suitable for noting things like 'This function is not thread
 *    safe' and so on.
 *
 * bug: This is how you can add an optional description of a known bug in the
 *    function such as: This only works for positive values of z.
 ***/
int f(int x, int y, int z);
```

Often the brief description for a function or method will be sufficient and the more detailed description can be omitted. If a function has no parameters or no return value, those items can be omitted as well. The other tags (`@note`, `@warning`, and `@bug`) will only be needed in rare cases.

When documenting parameters and return values, do not simply list the parameter or return value's data type. That information can easily be found by looking at the first line of the function or method or in its prototype. Your documentation should

instead explain the purpose and meaning of the parameter or return value. It's okay to mention the data type, but it is not sufficient.

You can put your documentation for the methods of a class or struct in the `.h` or `.cpp` files but **NOT BOTH**. *Never replicate anything!* If you do so, your documentation will almost inevitably become inconsistent and contradictory over time.

Example documentation box for a function:

```
/**
 * Searches an array of Student objects for a specified name.
 *
 * @param student_array An array of Student objects to search.
 * @param search_name A name for which to search.
 *
 * @return The index of the array element that contains searchName or
 *    -1 if the name is not found.
 *
 * @note Uses the binary search algorithm, so student_array must be sorted
 *    in ascending order by name.
 ***/
int search_for_name(const Student student_array[], const string& earch_name)
```

Last modified: 2022-02-02 13:15:46 CST
