SİVAS UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ENGINEERING AND NATURAL SCIENCES

ALGORITHM AND PROGRAMMING - I

Experiments Manual

Supervisor: Asst. Prof. Nurhan GÜNEŞ

Laboratory Instructor: Res. Asst. Semih OKTAY

SİVAS

TABLE OF CONTENT

Experiment 0: Getting Started with C++

1. Objective of the Experiment

- To introduce the laboratory computing environment.
- To introduce the editing, compiling, and execution of a program.
- To introduce simple file manipulation commands.

2. Theoretical Background

Introduction

The main purpose of this lab is to introduce you to the computing environment of your laboratory. You will use the ideas in this lab again and again throughout this course, so you should make every effort to understand not only *what* but *why* you are doing what you are doing at each step.

Before we can begin our session, your instructor will inform you how to begin a session with the computer at your particular institution. The way this is done differs from school to school, according to the kind of computer being used, whether they are networked or stand-alone, whether a security system is in place, and so on. You should ask and get answers to these questions:

1. What **operating system** am I using?
2. Should the computer be turned on at the beginning of the exercise and off at the end of the exercise, or does it remain on all of the time? Turning them off is sometimes optional, but in many computer labs, you should *never* turn the computer off.
3. Do I need a **personal account** to log in first before I can use the computer? If so, then I have some further questions:
    a. How do I find out my **username**?
    b. How do I find out my **password**?
    c. Is it necessary to change my password? If it is, how?
4. If I'm using a windowing environment (X Window System, OpenWindows, MacOS, Microsoft Windows, etc.):
    a. How do I start that environment?
    b. How do I use that environment?
    c. How do I exit that environment?
5. How do I print? What command should I use? Which printer should I sent my output to? Will I be charged?
6. What must I do to quit a session using the computer (especially if I'm using a personal account)?

About this Manual

Throughout this lab manual, instructions will be printed in this default font (the one you are reading). To help you distinguish the instructions from what appears on your screen, text that you should see displayed on your screen will be shown in `this typewriter font`.

There's a subtle distinction between **typing** and **entering** information:

- To *type* the letter y, simply press the keyboard key marked `y`.
- To *enter* the letter y, press the keyboard key marked `y`, followed by the key marked `Return` or `Enter`.

Operating System

First, you have to become familiar with Microsoft Windows, the operating system you're using (if you're not already).

You may find it challenging enough getting used to your operating system, so don't be afraid to play around to get used to it. *Keep the appropriate instructions handy for* ***every*** *lab since you'll need to create folders and copy files for* ***every*** *lab.*

Programming Environment

There are many different compilers you can use to compile C++ programs. Visual Studio from Microsoft is already picked for you.

As with the computing environment, you'll continually need these instructions for **every** lab. *Bookmark the appropriate instructions so that you can use them for* ***every*** *lab.*

Work on a Program

This is the program that you need for the environment instructions in the previous section.

```
/* bases.cpp demonstrates basic I/O in C++.
 *
 * Written by: Jane Doe, Feb 29, 1999.
 * Written for: CS I, at City University.
 *
 * Specification:
 *   Input(keyboard): aNumber, an integer;
 *   Output(screen): the base 10, 8 and 16 representations of
aNumber.

***************************************************************
*********/

#include <iostream>
using namespace std;

int main()
{
   // declare an integer container to hold the input number
   int aNumber;

   // 0. print a message explaining the purpose of the
program.
   cout << "\nThis program inputs a base-10 integer"
        << "\n\tand displays its value in bases 8 and 16\n";
   // 1. ask the user to enter an integer.
   cout << "\nPlease enter an integer: ";

   // 2. input an integer, storing it in variable aNumber.
   cin >> aNumber;

   // 3. output the base-8 and base-16 representations of
aNumber.
   cout << "\n\nThe base-8 representation of " << aNumber << "
is "
        << oct << aNumber
        << ",\n\tand the base-16 representation is "
        << hex << aNumber
        << "\n\n";
}
```

Applying the Scientific Method

After you've successfully copied, compiled, and executed the program above, continue on with this exercise.

An important part of any science, including the science of computing, is to be able to observe behavior, form hypotheses, and then design and carry out experiments to test your hypotheses. The next part of this exercise involves applying the scientific method to infer (from the statements within `bases.cpp`) how the certain aspects of C++ output work. Since two heads are (sometimes) better than one, feel free to work through this section with the person sitting next to you.

Observe

Run the `bases` program one or more times and study its behavior; then study the text of `bases.cpp`. Here's the question we want to address: What in the world do the symbols `\n` and `\t` do for the program?

Hypothesis

Question #0.1: Construct a hypothesis (i.e., a statement) that states what you *think* the symbols `\n` and `\t` do in the program.

Hint: they do slightly different things, but they both have an effect on the output of the program.

You do not have to be *right* yet; you're simply making a prediction. If your experimentation proves the hypothesis, then your conclusion is easy; if your experimentation proves the hypothesis is wrong, then you change it for your conclusion. It's really the conclusion that matters.

Experiment

Design an experiment using `bases.cpp` that tests your hypothesis. Modify `bases.cpp` as necessary to perform your experiment, compile the program, and run it.

Question #0.2: What results did you get?
Question #0.3: Do these results confirm or contradict your hypothesis?

If the experiment confirms the hypothesis, construct another experiment, a *tougher* experiment to test the hypothesis further. If the experiment contradicts the hypothesis, come up with a new hypothesis, and try it all again.

For each iteration through this process, write down your hypothesis, your experiment, and your results.

Conclusion

Question #0.4: What conclusion have you reached about the `\n` and `\t` symbols?
Question #0.5: Why do you suppose that these symbols are known as whitespace characters?

You may find it very useful to compare your hypotheses and experiments with the hypotheses and experiments of your fellow classmates; remember: the more experiments you have, the stronger your hypothesis.

Submit

Turn the answers to the questions in the lab exercise. Turn in a copy of the program that you copied over and compiled. Also, turn in a sample execution of your program.

Terminology

entering data, operating system, password, personal account, typing data, username, whitespace character

3. Project 0

Your instructor will assign you one of the problems below. To solve your problem, write a program that outputs the necessary information.

Project #0.1: Write `flies.cpp` -- a program that produces the following output:

```
                  like
           flies        an
      Time                    arrow ...

      Fruit       like     banana.
           flies        a
```

Project #0.2: Write `recipe.cpp` -- a program that produces the following output:

```
Pop 1 cup of popcorn
Melt 1 stick of butter
Combine popcorn and butter
Salt to taste
```

Project #0.3: Write `direct.cpp` -- a program that produces detailed directions to your house, such as:

```
Take I-96 to U.S. 131
Take 131 3 miles south to Hall St.
Take Hall 1 mile east to Madison Ave.
Take Madison 3 blocks north to Sherman St.
Take Sherman 2 blocks east to 1000 Sherman St.
```

Project #0.4: Write `me.cpp` -- a program that produces output describing yourself, such as:

```
Name:    John Whorfin
Gender:  Male
Year:    Freshman
Phone:   555-9876
Hobbies: Swimming, volleyball and eighth-dimensional physics
Quote:   "Laugh while you can, monkey-boy!"
```

Turn In

Turn the following things:

1. Your source program.
2. The output from an execution of your program.

Experiment 1: Software Engineering

1. Objective of the Experiment

- To practice using the computing environment.
- To gain experience designing programs to solve problems.
- To gain experience building programs from a design.
- To gain experience with compiler errors, deliberately generating them.

2. Theoretical Background

Introduction

Our exercise is to learn how to design and write a program. Since doing so is creating software, this endeavor is sometimes called **software engineering**. The particular design technique we will be using in this manual is called **object-centered design** (**OCD**), a methodology explicitly designed to help novice programmers get started writing software.

The process of developing a program to solve a problem involves several stages:

1. **Design**: Carefully plan your program using OCD.
    1. **Behavior**: Describe as precisely as possible what the program must do.
    2. **Objects**: Using the behavioral description, identify the objects needed to solve the problem.
    3. **Operations**: Using the behavioral description, identify the *operations* needed to solve the problem.
    4. **Algorithm**: Organize the objects and operations into an algorithm---a sequence of steps that solves the problem.
2. **Coding**: Translate your algorithm into a programming language like C++.
3. **Testing**: Run your program with sample data to check for errors, and *debug* your errors until there aren't any more.
4. **Maintenance**: Perform any modifications needed to improve the program.

This lab will use an example problem to elaborate a bit on these steps and to see an example C++ program. The main work you'll have to do for this lab will be modifying the program and observing the results.

Do not worry or panic if a topic is confusing and it's not explained in great detail. We will go through each of these topics in subsequent labs.

Design

Yogi Berra once supposedly said,

If you don't know where you're going, you'll end up somewhere else!

This idea is especially important when writing software: you need to know where you are going before you sit down to write a C++ program. This is done by spending some time *designing* your program.

Helpful hint: *A good design makes a program easy to write.*

If you are not a novice programmer, spending the time to design a program may seem like a waste, especially at the beginning when problems are relatively easy. However, the problems will soon become more difficult. If you get in the habit of carefully designing your software now (while doing so is easy), then designing elegant solutions for more difficult problems will also be relatively easy. But if you take a short cut now and skip the design stage, you will not master object-centered design, and when the problems get more difficult, you will find that your peers are writing better programs and writing them faster than you.

So be disciplined and get into the habit of carefully designing your software. The remainder of this section will teach you how.

Object-centered design consists of four sub-stages: describing how the program must behave to solve the problem, identifying its objects, identifying its operations, and organizing those objects and operations into an algorithm.

Behavior

We can describe our program's behavior as follows:

Our program should first display on the screen a greeting, after which it should display a prompt for the lengths of the legs of a right triangle. It should then read these two values from the keyboard. Once it has the two leg lengths, it should compute the hypotenuse length. It should then display the hypotenuse length (and appropriate labels) on the screen.

This **behavioral description** gives us all of the information we need to get started designing our program. In particular, it provides us with the objects and the operations our program requires to solve the problem. It does not magically fall from the sky, but it's a natural explanation of what we want the program to do. What are the steps *you* would do to solve the problem? We then throw in steps to read in data and to display the results of the solution.

Once we have our behavior description, we can pick out our objects and operations. True to its name, OCD says we pick out our objects first...

Objects

Picking out our objects is as simple as picking out the *nouns* in our behavioral description (ignoring nouns like "program" and "user"). The result is a list of the objects our program needs to define:

Description	Type	Kind	Name
the screen	`ostream`	varying	`cout`
a greeting	`string`	constant	--
a prompt for the legs	`string`	constant	--
the length of the first leg	`double`	varying	*`leg1`*
the length of the second leg	`double`	varying	*`leg2`*

the keyboard	`istream`	varying	`cin`
the hypotenuse value	`double`	varying	*`hypotenuse`*
labels for the hypotenuse	`string`	constant	--

Once we know the objects in our problem, it is useful to specify the basic flow of information in our program in terms of the objects going in and data going out. For example, we can write:

Specification:

input (keyboard): *`leg1`* and *`leg2`*, two `double` values.
output (screen): *`hypotenuse`*, a `double` value.

Such a **specification** succinctly states what the program does to solve the problem in terms of its inputs and outputs. (Output items like a greeting, prompts and labels are assumed.) It is good programming style to provide such a specification as part of the program's opening comment, for documentation purposes.

Operations

Just as our nouns are objects, *verbs* are our operations for our algorithm:

Description	Predefined?	Name	Library
display the greeting	yes	`<<`	`iostream`
display a prompt	yes	`<<`	`iostream`
read the length of the first leg	yes	`>>`	`iostream`
read the length of the second leg	yes	`>>`	`iostream`
compute the hypotenuse length	??	??	??
display the hypotenuse	yes	`<<`	`iostream`
display the label	yes	`<<`	`iostream`

In addition to listing the basic operations, we also list whether or not the operation is predefined; if so, how it is specified, and where the operation is defined.

This table provides us with the basic operations needed to solve our problem; however, there is no predefined operation to compute the length of the hypotenuse of a right triangle. To do so, we must refine this operation by breaking it down into smaller ones. In particular, the Pythagorean Theorem tells us that we need these operations to compute the hypotenuse length:

Description	Predefined?	Name	Library
compute the square of a leg	yes	`pow()`	`cmath`
add two squared legs	yes	`+`	built-in
compute the square root	yes	`sqrt()`	`cmath`
store a value in a variable	yes	`=`	built-in

Algorithm

Once we have identified all of the objects and operations needed to solve our problem, we can organize them into a sequence of steps that solves our problem---an algorithm. An algorithm should specify what a program does in fair detail, but it need not worry about the syntax details of a particular language like C++.

We've already noted above some C++ objects that we'll use (specifically `cin` and `cout`) and some C++ operations (e.g., >> and <<). We'll cheat a bit in our algorithm by using these C++ names, but if we wanted to switch languages, we could easily switch these names.

1. Display via `cout` a greeting for the user.
2. Display via `cout` prompts for the two leg lengths.
3. Read values from `cin` for *`leg1`* and *`leg2`*.
4. Compute the hypotenuse length using the Pythagorean Theorem:

   ```
   hypotenuse = sqrt(pow(leg1, 2.0) + pow(leg2, 2.0))
   ```

5. Display via `cout` the value of *`hypotenuse`* along with an appropriate label.

This algorithm provides the blueprint for our C++ program. Once we have it, we are done designing our program, and are ready to begin **implementing** our design.

Coding

Before working through the rest of this lab, make sure you have work through the operating-system and compiler specific portions of Lab #0 to become familiar with your environment.

In the `lab1` directory, open up the `hypot.cpp` file. If you obtained this from a zipped file, you should find the file in the `lab1` directory; otherwise, you'll have to create the directory and download the file.

We are going to take a big jump here and simply look at the code for this program; in most labs, you'll write a lot of this code. Take a look at the `hypot.cpp` file that's provided for you.

You'll play around with the syntax of this program below, but for now compare how the program statements match up with the algorithm steps. For each step of the algorithm, there's usually one and occasionally two C++ statements for the algorithm step. This indicates you have a good algorithm; if you find yourself needing more statements for an algorithm step, then you probably have to revise your algorithm.

After the program is written, we compile the program. Go ahead and do this. (You will not know how to do this unless you've read the appropriate OS and compiler directions from Lab #0.)

In practice, it is best to compile the program on a regular basis, as you write it, so that compiler errors don't overwhelm you at the end of the coding. It will make finding the errors easier. But since this program was given to you, you can move abnormally fast through this step.

Testing

The third stage of program development is a thorough testing. The basic idea is to execute the program using sample data values that test the program, to see if it contains any logic errors.

You compiled the program in the previous section, so execute the program with the following values, to see if you get the correct results.

		predicted	**observed**
`leg1`	***`leg2`***	***`hypotenuse`***	**`hypotenuse`**
1.0	1.0	1.414214	
3.0	4.0	5.0	
5.0	12.0	13.0	

Question #1.1: Write down the observed values that you get testing these inputs on the program you compiled.

Maintenance

Unlike programs that are written by students, real world programs may be used for many years. It is often the case that such programs must be modified several times over the course of their lifetimes, a task which is called **program maintenance**. Maintenance is thus the final (and usually the longest) stage of program development. Some studies have shown that the cost of maintaining a program can account for as much as *80%* of its total cost! One of the goals of object-oriented programming is to try to reduce this maintenance cost, by writing code that is reusable.

Question #1.2: What changes might you like to make to this program? (You *won't* have to implement them, so let the sky be your limit!)

Playing with the Program

Load `hypot.cpp` into your editor.

First of all, you will be making changes to the program, so you should add a modification comment in the comment at the top of the file. Add something like this:

```
* Modification history:
*    by John VanDoe in September 2002 for CPSC 185 at Calvin
College
*       Modified to run the experiments for Lab #1.
```

This should go right after the author information. Recompile and execute the program. It shouldn't execute any differently. That's because that text at the beginning of the document are all comments. But you might wonder what makes a comment.

Comments

A comment is text that's useful for the programmer. The compiler will ignore it completely. So if you type junk into a comment, the compiler will ignore it. If you type junk outside of a comment, the compiler will take it as program code and will get confused.

For each of the following questions, make the change suggested and recompile your program. If it compiles okay, we'll assume it runs okay; your answer for the question can be "compiles fine". But if a change generates a compiler error, your answer for the question should be the *first* error message that the compiler gives you. *Always undo your change before going on to the next question.*

Question #1.3: What happens when you remove the asterisks (i.e., `*`) before the lines you just added?

Undo your change.

Question #1.4: What happens when you remove the `/*` at the very beginning of the file?

Undo your change.

Question #1.5: What happens when you remove just the `/` at the very beginning of the file?

Undo your change.

Question #1.6: What happens when you remove just the first `*` (right after the `/` at the beginning of the file?

Undo your change.

You should have a good idea where this comment starts.

Question #1.7: Where do you suppose this opening comment ends?

Another way to introduce a comment is with `//`, two slashes. This type of comment is used to indicate the algorithm steps in the program.

Question #1.8: Remove one of the `//`s in the program. What happens when you compile the code?

And, yet again, make sure your program is restored back to a compilable state.

Includes

You'll find two lines that use the `#include` directive right after the opening comment. These lines tell the compiler that it needs to access some library files. They're necessary for some of the operations that the program does.

Question #1.9: What happens when you delete one of the `#include` lines?

Question #1.10: What happens when you add some spaces before a `#include`?

Question #1.11: What happens when you move one of the includes to the end of the file?

Remember to restore your file after each question.

The Program: int main()

The main program is designated by `main()`. The main algorithm must be encoded in this function.

Question #1.12: What happens when you drop the parentheses: `main()` becomes just `main`?

Question #1.13: What happens when you drop the `int` before `main()`?

Question #1.14: What happens when you replace the curly braces (i.e., `{` and `}`) with parentheses (i.e., `(` and `)`)?

Question #1.15: What happens when you add extra spaces before the `int`?

Again, restore your program after each question so that it compiles and executes correctly.

Input and Output

The input and output statements for Steps 1, 2, 3, and 5 all begin with either `cin` or `cout` followed by objects that should be displayed and used to read in values. The objects are separated with the operators << and >>.

Question #1.16: What happens when you replace a `cout` with `cin`?

Question #1.17: What happens when you remove a >>?

Question #1.18: What happens when you remove a <<?

Question #1.19: What happens when you replace a >> with <<?

Question #1.20: What happens when you replace a << with >>?

And yet again, restore your program after each question so that it compiles and executes correctly.

Declarations

Before we can use a variable in a C++ program, we must first declare it. The `leg1` and `leg2` variables are declared with this line:

```
double leg1, leg2;
```

Question #1.21: What happens when you delete this line?

Question #1.22: What happens when you move it after Step 4?

Question #1.23: What happens when you move it before `main()`?

By this point, you've probably run across a change or two that didn't matter. You made the change, the program still compiled, and the program still executed correctly. However, *not all of these changes are good.* You'll see again and again that there's often many ways to write your program, many of them bad ways.

Helpful hint: *Pay careful attention to the way code is written in the examples that you see in this lab manual. Try to mimic this as closely as you can* ***even if something else works just as well****.*

Be sure to return your file to this original state.

Semicolons

Semicolons end almost every C++ statement.

Question #1.24: What happens when you remove a semicolon from one of the statements?

Question #1.25: What happens when you add an extra semicolon?

Question #1.26: What happens when you add some extra spaces before a semicolon?

You might be a bit worried about the `#include` directive; it doesn't have a semicolon!

Question #1.27: What happens when you add a semicolon after an `#include` directive on the same line?

Question #1.28: How do statements and directives differ with respect to semicolons?

Submit

Turn in your answers to the questions in this lab exercise.

Terminology

algorithm, behavior, behavioral description, coding, design, implement, maintenance, object, object-centered design, OCD, operation, program maintenance, software engineering, specification, testing

3. Project 1

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #1.1: Modify the program from the lab exercise to also print out the area of the triangle:

```
double area = 0.5 * leg1 * leg2;
cout << "The area is " << area << endl;
```

Note: In the following formulas, π is the symbol for "pi" on this browser.

Project 1.2. Write a program to input the radius of a circle and then compute and output its circumference and area. The formulas for these quantities are as follows:

circumference =$2\pi r$
area =πr^2 where r is the radius of the circle.

Project 1.3. Write a program to input the radius of the circular base and the height of a right circular cylinder, and then compute and output its surface area and volume. The formulas for these quantities are as follows:

surface area = $2\pi rh$
volume = $\pi r^2 h$
where r is the radius of the circular bases of the cylinder and h is its height.

Project 1.4. Write a program to input the lengths of the two axes of an ellipse and compute and output its circumference and area. The formulas for these quantities are as follows:

circumference = 2π x the square root of $(((a/2)^2 + (b/2)^2) / 2)$
area = πab
where a and b are the lengths of the major and minor axes of the ellipse.

Project 1.5. Write a program to input the radius of a sphere and then compute and output its the surface area and volume. The formulas for these quantities are as follows:

surface area = $4\pi r^2$
volume =$(4/3)\pi r^3$
where r is the radius of the sphere.

Turn In

Turn the following things:

1. You OCD design.
2. Your source program.
3. The output from an execution of your program.

Experiment 2: Types

1. Objective of The Experiment

- To work with the basic data types of C++.
- To work with variables and their declarations.
- To work with constants and their declarations.
- To gain experience with compiler errors related to declarations and data types.

2. Theoretical Background

Introduction

In this lab, you will perform several experiments with different types of data using a pre-written program.

Files

Starting with this lab exercise, each lab exercise will have a section named "Files", and it will contain information like this:

Directory: `lab2`

- `experiment.cpp` is our playground for the experiments in this lab.

You should create the specified directory and download the files listed here. Every lab will have just the one directory, but any labs will have multiple files which should all be downloaded and saved in the directory that you create.

You will also be asked to personalize the documentation at the beginning of the files. For this lab, add some lines to the comment at the top of the file similar to this:

```
* Modification history:
*     by John VanDoe in September 2002 for CPSC 185 at Calvin
College
*       Modified to run the experiments for Lab #2.
```

You should make similar additions to every code and documentation file you see during *any* lab.

C++ Output

All of the things we look at in this lab are really internal to the workings of a program. In order to get some feedback, the program needs to **display**, or **output**, some information.

Output in C++ is done with the `cout` object. This is the name of the screen in C++. To actually send something to the screen, C++ gives us the << operator. In general, an output statement looks like this:

cout << *Value*$_1$ << *Value*$_2$ << ... << *Value*$_N$;

where each `ValueI` is replaced with objects. The << operators separate each of the values. For now, you can send any object to the screen this way.

Compiler Errors

Most (if not all) of the experiments for this lab ask you to compile program that will deliberately generate **compiler errors**. This is to get you familiar with the error messages from your compiler.

Helpful hint: *The first error message from a compiler is the only one that matters.*

This is a bit overstated, but not by much. Usually one mistake can generate several error messages. The compiler may generate several messages for the same mistake; it can also get so confused by the problem that it starts to complain about *good* code!

Consequently, it is fastest to fix the *first* error message you get and then recompile. If you try to fix all of the error messages, you'll waste too much time trying to figure out if your fix for the first error already fixed the other errors. When you recompile, you'll often get a whole new batch of errors, but as long as you keep working on the first error message each time, you should eventually arrive at a program that compiles.

In these experiments, you'll be asked to write down the first error message when you're deliberately told to compile a program with an error in it. Write down the error message *verbatim* as you read it off the screen. Write down *only* the first error message since it's really the only one that matters.

Most error messages are quite awkward, and it'll help you to recognize what they mean if you write them down carefully. Try to pick out key words (e.g., "initialization", "undeclared", etc.) in the error message that indicate what's going wrong. Learn to recognize these words.

Compilers will also issue **compiler warnings** for code that's officially correct according to the C++ standard, but probably it causes some problem. If a program generates only warnings from the compiler, but no errors, you often get a executable program (depending on the compiler), but most likely there's a bug in the program that will be much harder to find at run time. Heed your compiler's warnings!

The Experiments

This lab consists of four experiments. Your instructor may ask you to do only some of them; be sure to do those first; you will find working on the others helpful, if you have the time.

Submit

Submit your answer to the questions for the experiments that you completed. Your instructor may also ask you to submit a copy or multiple copies of your program.

Terminology

compiler error, compiler warning, display, output

3. Sub-Experiments

a) Sub-Experiment 1

The Basic Program

Our program is already set up to handle two integer values. It declares two integer variables and then prints their values.

Variables can be declared in basically two ways, with or without initialization:

```
type variable name;
type variable name = initializer expression;
```

You can list several variables (with or without initialization) after the data type, separating them with commas.

Our program has opted to declare and initialize two integer variables:

```
int i = 3, j = 5;
```

So `int` is the *type*, `i` and `j` are both *variable names*, and `3` and `5` are both *initializer expressions*.

After the declaration is a statement to print the values. You *must* use explicit output statements to see the values in your variables.

Question #2.1.1: What do you think this program will display on the screen when it executes?

Note: there is *no* wrong answer to this question, or any question that asks you to speculate what your program will do. The intent of these questions is to build your skill at reading programs and encouraging you to speculate about how a program will execute. You'll "fix" wrong answers in later questions.

Compile and execute the program.

Question #2.1.2: Compare your answer to the previous question and the actual output. How do your answers differ? Explain why they different.

Initial Values in a Declaration

The original program initializes `i` and `j` to specific values. Those same values should be printed out when you execute the program. But maybe we just got lucky.

Change the initial values of `i` and `j` to be `66` and `-5`, respectively.

Question #2.1.3: What does the declaration line look like now?

Question #2.1.4: What will this new version print when it executes?

Compile and execute this new version.

Question #2.1.5: What did the program display? Evaluate how good your prediction was.

No Initial Values

As noted above, initializing a variable is not required. Drop the initializations so that the declaration looks like this:

```
int i, j;
```

Question #2.1.6: Predict what this version will print when it executes.

Compile and execute this version.

Question #2.1.7: What did the program display? Evaluate how good your prediction was.

Most likely, `i` and `j` were initialized to 0 by default. *Do not count on this.* Some compilers will initialize integer variables to 0, but not all of them. Depending on the compiler and even the version of the compiler, `i` and `j` may have been initialized to seemingly random values.

Compilers change, and you may switch compilers, so it's risky relying on automatic initialization. Since initialization is so easy, it's much better for you to do it explicitly.

Change your program back so that `i` and `j` are initialized to `66` and `-5` again.

No Declaration at All

You've been told that variables must be declared before you can use them. What happens if you don't? Use a `//` comment to comment out the declaration of `i` and `j` (i.e., place the two slashes before `int`, on the same line as the declaration).

Try compiling it. Ooops!

Question #2.1.8: What is the first compiler error that you get?

Variable declarations are a very common source of programming error. You may forget to declare a variable, you may misspell it, or you may put it in the wrong place. Learn to recognize the error message that you just got; when you see it, you most likely have a problem with a variable declaration.

Restore the declaration (i.e., uncomment it).

Declaring Twice

What happens if you declare the same variable more than once? Copy the declaration line in your program two times. Compile your program.

Question #2.1.9: What is the first error message that your compiler gives you?

Better not do that. Remove one of the declarations.

Identifiers for Variables

Some words in C++ have a special meaning, and you cannot use them in other contexts. Such a **keyword**, or **reserved word**, can be used only for its particular meaning, not for a variable that you create.

For example, `int` is a reserved word. You cannot use it as the name of a variable. Try it: replace the variable `i` with `int` in the declaration. Try compiling your program.

Question #2.1.10: What is the first compiler error that you get?

Restore `i` to your program before you forget where it should go.

So, we must avoid keywords. Any C++ book will list the C++ keywords; check the list occasionally so that you become familiar with the keywords. It's probably not worthwhile deliberately memorizing them; you'll end up learning them through experience.

So then what constitutes a valid **identifier**? An identifier must begin with a letter and can be followed by letters (`a` through `z` and `A` through `Z`), digits (`0` through `9`), and underscores (_).

Replace the variable `i` in your program with `i3`. Replace *every* use of the variable `i`.

Question #2.1.11: Predict: what will your program display?

Compile and execute your program. Your program *will* compile; make sure you've changed every *i* variable to *i3*.

Question #2.1.12: What did your program actually display? Compare the output to your prediction.

Replace the variable `i3` in your program with `integerVariable`. (Using your editor's search-and-replace makes this faster than doing it by hand!)

Question #2.1.13: Predict: what will your program display?

Compile and execute your program.

Question #2.1.14: What did your program actually display? Compare the output to your prediction.

Replace the identifier `integerVariable` with `3i` in your program. (Hey, if `i3` is fine, why not `3i`?) Compile your program.

Question #2.1.15: What is the first error message your compiler gave you? What's wrong with the identifier?

We could play this game all day, but you get the idea. Restore the variable `i` in your program.

Now, just because an identifier is valid does not mean that it's a good identifier. There are several things to consider in picking names for identifiers. The first is readability. A good identifier should indicate what object it holds. What value would you expect to find in a variable named `b`? It could be a book, a Boolean, a bird, or just about anything---open up your dictionary to the 'b's. `book` would be a much better name (if, in fact, it stored a book object).

Some names have implied meanings. If your value is an integer used to count, the identifiers `i`, `j`, and `k` are very popular. The identifiers `x`, `y`, and `z` are often used for real numbers. If you use these in other contexts, you may confuse other programmers. In general, it's best to use *full* words or even *phrases* for your identifiers to better describe the objects they hold.

You must also consider the *form* of the identifier. Conventions vary, but they're usually similar to the one we'll use for variables:

- The identifier consists primarily of lowercase letters and (possibly) a few digits.
- Every word in the identifier, except the first word, begins with an upper case letter. So we use `integerVariable` instead of `IntegerVariable` or `integervariable`.
- Do not use underscores. So use `integerVariable` instead of `integer_variable`.

Not Just Integers

These declaration experiments are not particular to integer variables or integer values. *All* variables must be declared regardless of their type, and they should all be initialized also regardless of their type. The types and initializer expressions change, but never go away.

Terminology

identifier, keyword, reserved word

b) Sub-Experiment 2

The Program

For this experiment, we will play around with the integer declarations in the original program.

The integer data type in C++ is denoted with the `int` keyword. This is the type used in the declaration of integer variables.

Integer Initialization

From the first experiment, you know that you can change the initial values of `i` and `j` by changing the initialization expression. What happens though, if the initialization expression is not an integer?

Change the initialization expression of `i` to be a literal string instead of an integer. A string begins and ends with double quotes:

```
"this is a string"
```

Consider: if someone told you they were going to give you a hug, but then kicked you, you'd complain, right? Well, the compiler probably doesn't have the same emotional connection to an `int` that you might have to a hug, but it's still going to complain.

Compile the program.

Question #2.2.1: What is the first error message that the compiler gives you?

Seems like the compiler will save you from making an initialization mistake. The problem here is that a string of characters cannot be turned into an integer very easily. Ask yourself: what *is* the best integer to represent "this is a string"?

But what if the data types were very similar and *could* be converted easily?

Change the initialization expression of `i` to be a literal character. A character is a single character surrounded by single quotes: `'f'`.

Compile the program.

Question #2.2.2: What is the first error message that the compiler gives you?

You might be very confused at this point because your compiler *didn't* give you an error message. Some don't. (Answer the previous question appropriately: "I didn't get an error message.") The `int` and `char` data types in C++ are very closely related, and you probably won't get an error for initializing an `int` with a `char`. This is because a single `char` is represented in the computer's memory as an integer. So it's very easy for the compiler to switch between the two.

Some compilers will issue a warning for this initialization since often this is a programming mistake; either `i` should be declared as a `char`, or the initialization is wrong. Either way, it's a warning you should watch out for, just like any compiler error.

What about a real number, something with a decimal point in it? Change the initialization of `i` to be `3.14159` (the most overused real-number literal in programming examples), and compile the program.

Question #2.2.3: What is the first error message or warning that the compiler gives you?

Again, you probably don't get a error message, but you might have gotten a warning.

Let's try something really daring: `cin`. Initialize `i` to be equal to `cin`. It's a completely ridiculous thing to do, but let's see what the compiler does. Compile the program.

Question #2.2.4: What is the first error message or warning that the compiler gives you?

This time you should get an actual error message.

Finish

Restore your program so that `i` is initialized to a proper integer.

c) Sub-Experiment 3

The Program

For this experiment, you will write code for real-number variables.

The keyword for the real-number data type in C++ is `double`. (There's also `float`, but it's half as precise.)

The difference between a real number and an integer is a decimal point: real numbers have a decimal point, integers do not.

Real Number Declarations

Consider the initialization of our integer variables in the original program:

```
int i = 3, j = 5;
```

Write declarations for `double` variables `x` and `y`, initializing them to `3.1` and `5.667`, respectively. *Do not get rid of any code.* Compile and execute your code.

Question #2.3.1: What is the declaration you successfully added to your program?

You won't see any change in the execution since the declaration and initialization is purely internal. Once you can compile and execute the code without problems, add a statement to print out `x` and `y`. The original program had this statement to print `i` and `j`:

```
cout << "i is " << i << "\n"
     << "j is " << j << endl;
```

Your new statement for `x` and `y` should look quite similar, replacing the integer variables with the real-number variables and changing the labels appropriately.

Now compile and execute the program. Make sure it prints the proper labels and values for the variables.

Question #2.3.2: What is the output statement you just added to your program?

Let's play around with these declarations.

Real-Number Initialization

In the integer experiment, we tried initializing an integer variable with various types. Let's try this again for the `double` variables you just added to your program.

For the following questions, "I didn't get an error message or warning" might be perfectly acceptible.

Initialize `x` to an integer. Compile your program.

Question #2.3.3: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `x` to a character. Compile your program.

Question #2.3.4: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `x` to a string. Compile your program.

Question #2.3.5: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `x` to be equal to `cin`. Compile your program.

Question #2.3.6: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Observations

You probably didn't get any warning or error for initializing `x` to be an integer. Keep in mind that a real number, a `double`, *may* have a decimal point in it. It's not required to. Any number without a decimal point can easily have one added: `3` becomes `3.0`. C++ compilers are happy to add this decimal point for you.

In the previous experiment, though, the compiler complained when you tried initializing an `int` variable with a floating-point number. This is because an `int` *cannot* have a decimal point it in. The compatibility between these two types is one-way.

Real-Number Literals

Integers are simple to write: just a bunch of digits, possibly with a negative sign on the front.

Simple real numbers, known as **fixed-point** real literals, are also similar to write: write some digits and put in (at most) one decimal point. However, if you want to represent *really* big or *really* small numbers, we'd need a simpler notation. For example, chemists like to measure amounts of an element in *moles*. One mole is equivalent to about 602 million trillion atoms---that's 602 with *21* zeros after it. Yikes!

A **floating-point** real literal uses **scientific notation** (also known as **exponential notation** or **floating-point notation**) to represent these extreme numbers. Scientific notation uses 10 to a power to shift the decimal point in a number. So, for example, the number 602 million trillion can be written 6.02×10^{23} in scientific notation. It's just a math equation: take 6.02 and multiply it by 10 to the 23th power. This effectively shifts the decimal point over 23 positions to the right, adding zeros as needed.

However, we don't have the ability to write superscripts, and programmers are notoriously lazy. So C++ uses a short hand for floating-point notation: just the letter `e` for "exponent". So 602 million trillion can be written as `6.02e23` in C++.

Change your program so that `x` is initialized to `6.02e23` and `y` is initialized to `60.2e22`.

Question #2.3.7: Predict: what will your program display when you execute it?

Compile and execute your program to test your prediction.

Question #2.3.8: What did your program actually display? Compare the actual output to your prediction.

It's possible to have negative real numbers and negative exponents. Negative exponents move the decimal point to the left, making the number a small fraction, that is, close to zero. A negative number just makes it negative.

Terminology

exponential notation, fixed-point, floating-point, floating-point notation, scientific notation

d) Sub-Experiment 4

The Program

For this experiment, you will write code for character variables. The keyword for the character data type in C++ is `char`. It represents exactly one character. A character literal begins and ends with single quotes, with the character between them (e.g., `'a'` and `'A'`).

Character Declarations

Write declarations for `char` variables `ch1` and `ch2`, initializing them to `'a'` and `'Z'`, respectively. *Do not get rid of any code.* Compile and execute your program.

Question #2.4.1: What is the declaration that you added to your program?

You won't see any change in the execution since the declaration and initialization is purely internal. Once you can compile and execute the code without problems, add a statement to print out `ch1` and `ch2`. The original program had this statement to print `i` and `j`:

```
cout << "i is " << i << "\n"
     << "j is " << j << endl;
```

Your new statement for `ch1` and `ch2` should look quite similar, replacing the integer variables with the character variables and changing the labels appropriately.

Now compile and execute the program. Make sure it prints the proper labels and values for the variables.

Question #2.4.2: What is the ouput statement you just added to your program?

Let's play around with these declarations.

Character Initialization

In the integer experiment, we tried initializing an integer variable with various different types. Let's try this again for the `char` variables you just added to your program. Again, "I didn't get any compiler errors or warnings" is an acceptible answer for these questions.

Initialize `ch1` to an integer (try something between 64 and 100). Compile your program.

Question #2.4.3: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `ch1` to a double. Compile your program.

Question #2.4.4: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `ch1` to a string. Compile your program.

Question #2.4.5: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Initialize `ch1` to be equal to `cin`. Compile your program.

Question #2.4.6: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

Change your program back so that `ch1` is initialized to a character.

Character Literals

As mentioned above, a character literal consists of two single quotes with a character between them.

What happens if there's more than one character between them? Change the character literal for `ch1` to be `'abc'`. Compile your program.

Question #2.4.7: What is the first error or warning message that the compiler gives you? If it does compile (with or without warnings), what does it print for the new initialization?

There are some characters that perform an action. For example, if you want the displayed output to start on a new line you must use the newline character. But you're using the newline character in your program to control how *it* looks, so it's not entirely clear how you would enter a newline into a string so that it appears in the program's output.

C++ solves this mild dilemma with **escape characters**. An escape character begins with the backslash character, \, followed by another character that *represents* the character we want. For example, the newline character literal is `'\n'`. You, the programmer, must type in two separate characters, but to C++ this *represents* a *single* character, the newline character.

Some other escape characters include `'\t'` for the tab character, `'\''` for a single quote, `'\"'` for a double quote (more useful in strings), and `'\\'` for a backslash character.

Initialize `ch1` to be the newline character and `ch2` to be a single quote.

Question #2.4.8: Predict: what will your program display when you execute it?

Compile and execute your program to test your prediction.

Question #2.4.9: What did your program actually display? Compare the actual output to your prediction.

As you'll see in the next experiment, these escape characters are also used in strings.

Terminology

escape character

4. Project 2

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

The projects for this lab ask for a relatively simple computation that is *not* given to you. You should be able to find the computation in your textbook, from your own knowledge, or with a simple search on the Internet. Be sure to indicate in your program *where* you got the information.

Project #2.1: Write a program that reads a number of feet, a real (i.e., `double`) value, and prints the equivalent number of inches.

Project #2.2: Write a program that reads in a Fahrenheit temperature and prints the equivalent temperature in Kelvin.

Project #2.3: Write a program that reads in a number of days (as a `double`) and prints the equivalent number of hours.

Project #2.4: Write a program that reads in an age as a `double` and prints the equivalent number of "dog years".

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 3: Operations and Expressions

1. Objective of The Experiment

- To explore expressions involving the C++ fundamental types.
- To explore the C++ assignment and related expressions.
- To explore the C++ input and output expressions.

2. Theoretical Background

Introduction

The exercise for this lab involves a series of experiments, each investigating a different aspect of C++ expressions.

Definition: An **expression** is a sequence of one or more **operands**, and zero or more **operators**, that when combined, produce a **value**.

The operands are objects; the operators are actions. Let's consider a few simple examples:

```
12
```

is an expression. It's has one operand (the object 12), there are no operators, and it produces a value (the object 12).

Here's a more familiar example:

```
2 + 3
```

This too fits the definition of an expression, since it consists of two operands (the objects 2 and 3) and one operator (+) that combine to produce a value (the object 5).

Operands need not be constants:

```
2.5 * x - 1.0
```

This also fits the definition of an expression, since it consists of three operands (the objects 2.5, `x`, and 1.0) and two operators (*, -) that combine to produce a value (1 less than the product of 2.5 and `x`).

These examples have been **arithmetic expressions**, expressions whose operators are familiar arithmetic operations applied to numbers. C++ provides a rich set of arithmetic operators, but since C++ has other types of data (like characters and strings), we can also write expressions for other types of data.

Ultimately, all computation in any programming language boils down to expressions. There are other language features to abstract and control which and when expressions are evaluated, but in the end, expressions do all the real work. So, to build our knowledge of writing programs, we start at the bottom with expressions.

Expression Versus Statement

It is important to distinguish between an expression and a **statement**. Expressions are really an incomplete thought, like a sentence fragment. They are the building blocks for us to write our statements. A statement *is* complete thought; the compiler should have everything it needs to carry out an action. A statement always ends in a semicolon. (Actually, this is a bit of a lie as we'll see later.)

For example, consider the expression `2+3` again. We try to turn this into a statement by adding a semicolon:

```
2+3;
```

But what does this mean? Translated into English, we might say "add 2 and 3", but what are supposed to do with the result? Print it? Add it to a database? Do further computations with it? We could come up with hundreds of legitimate things we could do with the expression. The compiler will not guess; we *must* be specific.

If we want to print it

```
cout << 2 + 3;
```

or do further computations with it

```
int result = 2 + 3;
```

we have to specify these actions explicitly. Consider the English translations of these statements: "Send the addition of 2 and 3 to the screen (i.e., `cout`)." and "Save the addition of 2 and 3 in `result` so that I can use it later." Those are complete thoughts with concrete results.

Some expressions are complete in themselves. In fact, the output statement above is actually an output *expression* that contains an arithmetic expression. Output expressions are complete in themselves, so we can just tack on a semicolon to make it a statement. (The same applies to input expressions.)

Files

Directory: `lab3`

- `experiment.cpp` is our playground for the experiments in this lab exercise.

Create the directory, and save the file there.

Add some lines to the comment at the top of the file similar to this:

```
* Modification history:
*    by John VanDoe in September 2002 for CPSC 185 at Calvin
College
*       Modified to run the experiments for Lab #3.
```

The Experiments

You will find the variable declarations that you wrote previously to be useful for some of the experiments in this lab.

Submit

Submit your answer to the questions for the experiments that you completed. Your instructor may also ask you to submit a copy or multiple copies of your program.

Terminology

arithmetic expression, expression, operand, operator, statement, value (of an expression)

3. Sub-Experiments

a) Sub Experiment 1

Output Expressions

We took a quick look at output in the previous lab; we'll expound on that just a bit.

Output in C++ is done with the `cout` object. This is the name of the output screen or window. To actually send something to the screen, C++ gives us the << operator. In general, an output statement looks like this:

```
cout << Value1 <<  Value2 << ... <<  ValueN;
```

where each *`Value`*$_I$ is replaced with objects. Note that << operators separate each of the values.

As mentioned in the introduction to the lab, output *statements* are actually output *expressions*. However, we need the semicolon to make it a statement which is what the compiler demands.

The `endl` Object

You may have noticed a strange object in the output statements of our programs: `endl`. In your output, you should notice that `endl` stops the current line of output and starts the next output on the next line. This suggests that `endl` is interchangeable with `"\n"`. But just how interchangeable?

Add this line of code in your program:

```
cout << "Line #1." << "\n" << "Line #2" << "\n";
```

Don't both removing anything from your program since you'll need it later. Recompile and run your program.

Question #3.1.1: What does this statement print? Be *very* precise with the line breaks.

Replace each `"\n"` with an `endl`. Recompile and run your program.

Question #3.1.2: How has the output changed?

`endl` is a variable for an object, apparently some type of `string` object with a newline character in it. (It might have other characters we can't see.) As we know, variables do not magically appear in C++. Like `cin` and `cout`, the object `endl` comes from the `iostream` library.

But that's a bit strange, right? Why not use `"\n"` all of the time? We could make the output expression from above much simpler:

```
cout << "Line #1.\nLine #2\n";
```

So why bother with `endl`? Actually, `endl` is more than just the newline character; it also indicates that you want the output to appear on the screen *right now*. Usually, for these labs, you won't find this to be a big deal. It may never matter to you (depending on your program, compiler, and operating system). Play it safe and use `endl` at the end of a prompt, but otherwise it doesn't really matter if you use `endl` or `"\n"`.

Output Anything

We can nearly output anything. Our program currently prints strings, integer objects, and even the result of an expression. We can also print complex objects:

```
cout << cin << endl;
```

Add this line to your program, recompile, and execute the program.

Question #3.1.3: What does this line print?

Printing the value of `cin` isn't really useful; the point is that we *can* print it. For some objects we'll see later on, this can be very useful, especially in debugging our code.

b) Sub-Experiment 2

Input Expressions

To make some of the other experiments easier, we'll take a look at input expressions.

Variables Only

Consider this statement:

```
cin >> 3;
```

What do you suppose this would mean? Maybe after this executes, every `3` the program encounters should be replaced with the value the user types in from the keyboard? That seems awfully silly and quite dangerous. So, a better question: does C++ even allow this? Try it. Add the input statement to your program and recompile.

Question #3.2.1: What is the first compiler error that you get?

Fortunately, C++ doesn't let us do something so silly.

Instead, all of the objects in an input expression must be variables. Let's try this:

```
cin >> i >> j;
```

Add this statement *after* the declaration of `i` and `j`, but before the output statement. When you run your program, enter these values:

```
123  456
```

Question #3.2.2: What values are printed for `i` and `j`? Did these values come from the declaration or from your keyboard input?

An input statement *replaces* the value in a variable used in the input statement. The variable does *not* remember its old values; the old value is gone.

Replacing Values

Try this variation:

Question #3.2.3: What happens if the input line is moved *before* the declaration?

That's actually a review of the previous lab. Remember that you can't use a variable unless you declare it first. The order of the statements matter.

Here's a variation that *will* compile and execute:

Question #3.2.4: What happens if the input line is moved *after* the output statement?

Question #3.2.5: Use the results of this experiment to justify the claim that an input statement replaces the values in its variables.

Question #3.2.6: Is it necessary to initialize `i` and `j` in their declaration if we read in values for them in the very next statement?

Wrapping Up

Move the input statement back between the declaration and output statements.

c) Sub-Experiment 3

Arithmetic Expressions

The original version of our program prints the sum of `i` and `j`. C++ provides us with several arithmetic operators:

`+` **addition**, computes the sum of two (integer or real) operands
`-` **subtraction**, computes the difference of two (integer or real) operands
`*` **multiplication**, computes the product of two (integer or real) operands
`/` **division**, computes the quotient of the division two (integer or real) operands
`%` **modulus**, computes the remainder of the division two integer operands

Suppose we wanted to compute the product (i.e., the multiplication) of `i` and `j` instead of their sum.

Question #3.3.1: What change would we have to make to the program?

Test your change to make sure.

These arithmetic operators should be familiar to you, except for the last one. Let's spend some time looking at it.

Integer and Real Division

There is an important difference between the division of two integers and two real numbers. Let's make some changes to our program so that we can explore these differences.

The first change is to use an input statement to get values for `i` and `j`. Experiment #2 had you add the input statement. With this input statement you can compile the program *once* but execute it *many* times for different values for `i` and `j`.

The second change is to compute two values. The output statement already computes one value; make it compute `i/j`. Then add to the output expression so that it *also* prints out `i%j`. *Be sure to change the string labels so that you can read your output easily.*

Once you have these changes made, compile and execute the program to make sure it's correct. Then, using this code as a basis, write similar lines of code for `double` variables `x` and `y`. For these real-number variables, you shouldn't compute `x%y` since C++ won't let you.

Question #3.3.2: Use your program to fill in this chart:

`i`	**`j`**	**`i / j`**	**`i % j`**	**`x`**	**`y`**	**`x / y`**
4	1			4.0	1.0	
4	2			4.0	2.0	
4	3			4.0	3.0	
4	4			4.0	4.0	

4	5			4.0	5.0	
4	6			4.0	6.0	
4	7			4.0	7.0	
4	8			4.0	8.0	
4	9			4.0	9.0	

Recall that an integer is a whole number *without any* fractional part. So when you divide the integer 4 by the integer 5 (i.e, `4/5`), you can't do this evenly, not even once. So the integer division is 0.

But as a real number, you can compute fractional amounts. So `4.0/5.0` is `0.8` as a real number. Note that this is the decimal equivalent of the fraction `4.0/5.0`.

But what about that modulus computation? "**Modulus**" is (for the most part) just another name for "**remainder**". When you first learned about division, you probably learned to talk about your result in terms of quotient and remainder:

"28 divided by 3 is 9 with a remainder of 1."

Splitting Apart Integers

Division and modulus by 10 allow us to split up integers into their decimal digits.

Question #3.3.3: Use your program to fill in this chart:

`i`	`j`	`i / j`	`i % j`
1234	1		
1234	10		
1234	100		
1234	1000		

Can you see a pattern? How significant is the number of 0s in `j`?

Question #3.3.4: Use your observation to fill in this chart:

`i`	`j`	`i / j`	`i % j`
5678	1		
5678	10		
5678	100		
5678	1000		

Use your program if you get stuck and to check your answers.

Multiples

The multiples of 2 are 0, 2, 4, 6, 8, 10, 12, ... The multiples of 3 are 0, 3, 6, 9, 12, ...

What do the multiples of 2 have in common? Two evenly divides each multiple of two; that is, two divides each multiple *without any remainder*. Ah! So if `m % 2` is 0, then `m` must be a multiple of two.

Use this same thinking to come up with an arithmetic expression to use as a test for these multiples:

Question #3.3.5: What do the multiples of 3 have in common?

Question #3.3.6: What do the multiples of 4 have in common?

The multiples of 2 are also known as the **even numbers**.

Question #3.3.7: What do the even numbers have in common?

The **odd numbers** are all the whole numbers that aren't even.

Question #3.3.8: What do the odd numbers have in common?

Terminology

modulus, remainder

d) Sub-Experiment 4

Logical Expressions

We wish to find out if the value of `i` is within the range 1 to 100. Mathematically, you can write

```
1 <= i <= 100
```

However, while C++ is very mathematically based, mathematics can often be too ambiguous for C++. It's unclear which operator should be done first. Mathematically, we do *both* of the at the same time, but C++ can't do this. Unfortunately, we *don't* get an error for compiling this expression; C++ finds a different and (for us) very undesirable meaning for the expression.

Modify your program to evaluate and print the result of this expression. Compile and execute your program.

Question #3.5.1: Fill in this chart using your program:

`i`	`1 <= i <= 100`	**should be**
-5		
50		
5000		

The "should be" column is the mathematical relationship. The `1<=i<=100` column is the result given by your C++ program. As hinted above, there should be some disagreement between the two columns.

In order to make this expression palatable for C++, we have to break it into *two* comparisons with two subexpressions. (Hmmmm... two operators, two subexpressions. Coincidence?) In English, we might ask if "1 is less than or equal to *`i`*, and *`i`* is less than or equal to 100". Verbally, we've broken it into two separate expressions!

Let's turn the English into an expression:

1	is less than or equal to	*`i`*	and	*`i`*	is less than or equal to	100
`1`	`<=`	`i`	????	`i`	`<=`	`100`

We're close; we're just need the ability to express the "and" relationship. C++ gives us the `&&` operator:

1	is less than or equal to	*`i`*	and	*`i`*	is less than or equal to	100
`1`	`<=`	`i`	&&	`i`	`<=`	`100`

To make it easier to read, you'll find many C++ programmers add some parentheses:

```
(1 <= i) && (i <= 100)
```

Try this instead in your program. Compile and execute it.

Question #3.5.2: Fill in this chart using your program:

`i`	**`(1 <= i) && (i <= 100)`**	**should be**
-5		
50		
5000		

The "should be" column should be the same as before.

C++ gives three logical operators:

`&&`	**and**, produces true if and only if both of its operands are true.
`||`	**or**, produces true if and only if either of its operands are true.
`!`	**not**, produces true if and only its operand is false.

The AND and OR operators each require *two* operands. The NOT operator requires only *one* operand.

Boolean Expressions

Relational expressions and logical expressions both produce `bool` values as their result. For this reason, such expressions are collectively referred to as **boolean expressions**.

Boolean expressions have a variety of uses. One of these is the `assert()` mechanism, that allows you to halt the program if something bad happens, like a variable gets out of range. For example, suppose we have a problem that requires us to input a number `i`. Further suppose that this number must be non-zero, or the program will be unable to solve the problem correctly.

The requirement that the number be non-zero is known as a **precondition**; a precondition is an expression---a condition---that must be true in order for the code to succeed.. We can check a precondition using the `assert()` mechanism like so:

```
assert(i != 0);
```

This statement should be put after the statement that computes or reads in a value for `i`.

The parentheses in this statement are required. You can put any boolean expression between those parentheses. If the boolean expression is true, execution will continue as usual; but if the boolean expression is false, the program will terminate and a **diagnostic message** will be displayed, listing the precondition that failed.

Add this assert after the input statement that reads in a value for `i`.

Question #3.5.3: What value do you need to enter to make the assert fail?

Question #3.5.4: Write down the *exact* message that your program gives you when the assert fails.

Terminology

boolean expression, diagnostic message, precondition

4. Project 3

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

For any of these projects, declare `PI` as a constant:

```
const double PI = 3.14159;
```

Project #3.1: Write a program to find the circumference and area of any circle. The formulas for these quantities are as follows:

$circumference = 2 * PI * uservar$.

$area = PI * radius^2$.

Project #3.2: Write a program to find the side surface area and volume of any regular cylinder. The formulas for these quantities are as follows:

$sideSurfaceArea = 2 * PI * radius * height$.

$volume = PI * radius^2 * height$.

Project #3.3: Write a program to find the circumference and area of any regular ellipse. The formulas for these quantities are as follows:

$circumference = 2 * PI *$ the square root of $(((height/2)^2 + (width/2)^2)/2)$.

$area = PI * height/2 * width/2$.

Project #3.4: Write a program to find the surface area and volume of any sphere. The formulas for these quantities are as follows:

$surface\ area = 4 * PI * radius^2$.

$volume = 4/3 * PI * radius^3$.

Turn In

Turn the following things:

1. Your OCD design.
2. Your source program.
3. The output from three different executions of your program.

Experiment 4: Functions and Libraries

1. Objective of the Experiment

- To learn how to declare simple functions.
- To learn how to define simple functions.
- To learn how to create and use a library of functions.

2. Theoretical Background

Introduction

As you likely found out the last time you purchased your last textbook, books can be quite expensive. One way you could save yourself some money would be to share the book with someone else. In fact, the more people you can share your book with, the more you can save.

A **library** (a.k.a. a **library module** or simply a **module**) is a generalization on this idea of sharing. When a community of people pool their resources, then they can buy and share a centralized collection of books. By cooperating this way, everyone has access to a greater set of books than they could afford individually.

In the digital world, sharing libraries is even easier (much to the consternation of the mass media companies). While a physical book can be actively used by only one person at a time, a digital file can be shared by many people all at the same time. More particular to our situation, we can write code once and share it among many other programmers.

In fact, we've already been sharing code. More accurately, we've be taking advantage of the code others have written. For example, in all of the programs we've seen so far, we've used the `iostream` library. This library defined `cout` and `cin` as well as the input and output operators. There's actually a lot in that library, and we certainly don't want to write all of that code for every program we write. By reusing the library, all C++ programmers saves themselves enormous amounts of time.

As we'll see in this lab, there's nothing magical about a library, You'll see all the basic tools for creating your own library. The main difference between the libraries you write and the standard libraries of C++ is size, but that's mostly because these standard libraries have had so many people and so much time spent on them.

We'll write our own library in this lab.

Review

Let us briefly review what we know about function libraries. We have seen that C++ provides a variety of function libraries, including `iostream` that provides input and ouput, `cmath` that provides various mathematical functions, and `cctype` that provides character-processing functions.

To use a library function, a program must use the `#include` directive to include the library's header file:

```
#include <cmath>
```
This includes basic definitions of the library into your program.

To **call** a function, you must specify its name and a list of **arguments** in parentheses. For example, the `cmath` library contains a function named `pow` which computes the exponential power of a number. If we have variables `x` and `y` (which should be `double`s), we can call the function like so:

```
pow(x, y)
```

Question #4.1: In the function call above, what is the name of the function being called? What are the arguments in the function call?

The code associated with the function is executed, and the result is returned, `x` to the `y`th power. Since this is an expression, you will undoubtedly put it into a context:

```
double result = pow(x, y);
```
or
```
cout << "The answer is " << pow(x,y) << ".\n";
```

Planning a Library

The library we create will provide us with a set of functions to convert English-system measurements into their metric-system counterparts. We will call our library `metric`.

The first thing that we must decide is what measurement conversions we wish our library to provide. Here are just *some* of the conversion we could do:

English Unit	**Metric Unit**	**Conversion Equation**
Inches	Centimeters	1 inch = 2.54 cm
Feet	Centimeters	1 foot = 30.48 cm
Feet	Meters	1 foot = 0.3048 m
Yards	Meters	1 yard = 0.9144 m
Miles	Kilometers	1 mile = 1.609344 km
Ounces	Grams	1 ounce = 28.349523 g
Pounds	Kilograms	1 pound = 0.453592 kg
Tons	Kilograms	1 ton = 907.18474 kg
Pints	Liters	1 pint = 0.473163 l
Quarts	Liters	1 quart = 0.946326 l
Gallons	Liters	1 gallon = 3.785306 l

Our exercise is to write two **functions** to convert feet into meters and meters into feet. By storing these function in our library `metric`, it can be shared by any and all programs that need to convert between feet and meters, allowing all of us to avoid "redefining the wheel."

You may notice that you've already been provided with the feet-to-meters conversion. We'll walk through this example, and you'll be in charge of writing the meters-to-feet conversion.

Library Structure

A library consists three separate files:

- A **header file** in which you declare each name that is to be accessible outside the library.
- An **implementation file** in which you define each name declared in the header file.
- A **documentation file** in which you can write comments about how to use the library.

The documentation file is not strictly necessary, any more than are comments within a program; however it is good style to somehow provide such a file, and store within it all of the information needed to use the library. Our custom is to place function specifications in this file.

The header file is necessary, and its role is to provide an **interface** to the library, by providing just enough information for a program to use the functions in the library, without specifying their details. In contrast, the role of the implementation file is to provide complete definitions of the library's functions. You'll notice some redundancy in these two files, but this is unavoidable.

Helpful hint: *Remember to check **both** your header file and implementation file when something goes wrong with something in your library.*

Why all of these files? The reason has to do with program maintenance. If we are writing a library, then we expect that many programs will use of it. It is often the case that even a well-designed library may need to be updated if a better way is discovered to perform one of its functions or even if a function needs to be fixed. If we have designed our functions carefully, then updating a library function should simply involve altering its definition which is in the implementation file. We may be able to leave the header file alone which would save many people from having to recompile their programs.

Files

Directory: `lab4`

- `metric.h`, `metric.cpp`, and `metric.doc` implement a metric conversion library.
- `driver.cpp` implements a driver for the metric conversion library.
- `Makefile` is a makefile.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

Header File Structure

Edit `metric.h`, the header file of our library. For convenience, we base the name of this file after our library. Common convention tacks a `.h` to the end of the filename to designate this as the header file.

Personalize the opening comment to the modification section of the library.

A simplified general pattern for a library header file is as follows:

```
OpeningDocumentation
PrototypeList
```

The opening documentation is a comment describing the purpose of the library, who wrote it, and when it was last updated. The prototype list is a sequence of prototypes; we have to learn about designing functions to learn about prototypes.

Function Design

Nothing beats a good design. So we'll use OCD to design our functions.

Behavior

First, we describe the behavior of our feet-to-meters function:

Our function should receive from its caller the number of feet to be converted, and should check that this value is positive. It should convert that quantity to meters by multiplying it by 0.3048, and then return the resulting value to the caller.

Note that where a *program* typically inputs values from the *keyboard*, a function typically *receives* values from whoever called the function. Similarly, where a *program* typically outputs values to the *screen*, a function typically *returns* a value to whoever called the function.

This distinction is *very* important. Overwhelmingly, most functions do *not* use `cin` or `cout`. We won't need them in our functions.

We also want to try and anticipate what could go wrong. The function will only produce a correct result if the value it receives is not negative; this is a **precondition** since it's a condition that must be true before the function can be executed.

Now let's consider our other function, converting meters to feet. This function won't be much different than the feet-to-meters function. So, the basic structure for the behavior should be the same. The values we receive and return will be different, and the computation is different (division instead of multiplication), but still pretty much the same.

Question #4.2: Write the behavior paragraph for the meters-to-feet function.

Objects

We list the objects of a function in the same way that we do for a program, except we also want to describe their movement. For each object, does it move *into* the function from outside, does it move from the function *out* to the outside, or is it purely *local* to the function?

Description	Type	Kind	Movement	Name
the number of feet	`double`	varying	in	*`feet`*
the conversion factor	`double`	constant	local	0.3048
the corresponding number of meters	`double`	varying	out	*`meters`*

The movement column of this chart summarizes what data is coming in and going out very nicely. This allows us to write our specification which also includes our precondition:

Specification:
receive: *`feet`*, the number of feet to be converted.
precondition: *`feet`* is not negative.
return: *`meters`*, the equivalent of feet in meters.

Keep in mind that none of this information here is magical. The behavior paragraph is used to create the object chart. The nouns in the behavior are our objects, and the behavior explains the type, kind, and movement of the objects. The behavior even suggests the names of our objects.

Once we have the object chart, we can come up with the specification, looking for the objects moving in and out of the function. The behavior paragraph establishes our preconditions.

Question #4.3: Using your behavior description, come up with an object chart for the meters-to-feet function.

Question #4.4: Using your behavior description and object chart, come up with a specification for the meters-to-feet function.

The specification of a function provides us all we need to write a **prototype**. A prototype is the way we write the specification for our library. We'll need a prototype for *every* function we write.

The simplified pattern for a function prototype is

`ReturnType` *`FunctionName`* `(`*`ParameterList`*`) ;`
where

- *`ReturnType`* is the type of value the function returns. It's any data type like `int` or `double`.
- *`FunctionName`* is the name of the function. You get to choose this.
- *`ParameterList`* is a list of declarations of parameters.

Each value that our function *receives* must be declared as a **parameter**. A parameter is really a variable just like the ones we've already declared and used. The differences are subtle: a parameter is declared as part of the function (not inside it) and each parameter represents a value that the function receives.

With our specification and some imagination, we can come up with the components of our prototype:

- First, the specification tells us that our function must return the number of meters, a real value. So our return type must be `double`.
- Second, we use our imagination to come up with a function name. Since the function converts a number of feet to meters, let's try `feetToMeters()`.
- Third, the specification also tells us that our function must receive the number of feet, also a real value. So we'll have one parameter of type `double` that we'll call *`feet`*.

Now we can assemble these components into our prototype:

```
double feetToMeters(double feet);
```

You will see this prototype in both the header file (`metric.h`) and the documentation file (`metric.doc`). C++ requires the prototype in the header file so that our program (and others) can see it. We document it in the documentation file so that others know how to use our function; that's why you'll see the specification just above it.

Question #4.5: Write a prototype for the meters-to-feet function.

Add your prototype to both the header file and the documentation file. Also, add a comment containing the specification for the prototype in the documentation file.

Since parameters are like variables, their names begin with a lowercase letter, with every word after the first capitalized. The same convention is used for naming function. These are common conventions, although not universal; they *are* the convention we'll use.

As this illustrates, the stages of software design are fluid, not firm. We can build a function's prototype (a part of *coding*) during our design stage, as soon as we have enough information.

Operations

Continuing with our design, our list of needed operations is quite short:

Description	Predefined?	Name	Library
get a value from the caller	yes	`feet`	built-in
check that the value is not negative	yes	`>= 0`	built-in
halt the program if the value is negative	yes	`assert()`	`cassert`
multiply two real values	yes	`*`	built-in
return a value to the caller	yes	`return`	built-in

We've already partially seen the parameter mechanism that allows us to get the `feet` value. We'll see more when we write our function. We'll also see the return mechanism in action. The key now is that all of these operations are built into C++, and we'll definitely make use of them.

The relational operators and the `assert()` function will all help us in handling our preconditions.

Question #4.6: Write down the operation chart for the meters-to-feet function.

Algorithm

We can then organize these operations and objects into the following algorithm:

1. Receive *`feet`* from the caller.
2. Halt the program if *`feet`* is negative.
3. Compute *`meters`* = *`feet`* * 0.3048.
4. Return *`meters`*.

Our behavior description and the operations lead directly to this algorithm.

Question #4.7: Write an algorithm for the meters-to-feet function.

Implementation File Structure

We store **declarations** in the header file. That's all a function prototype is: a declaration. It declares the function's name, it's parameters, and return type. But we haven't written any code for the function to execute!

That's what the implementation file is for. In an implementation file, we put the **definitions** of our library functions. This will include another declaration of the function *plus* the code for the function. Our implementation file are often named the same as our header file, excepting ending in `.cpp`.

The general pattern for an implementation file is as follows:

```
OpeningDocumentation
Includes
DefinitionList
```

Take a moment to personalize the opening documentation in `metric.cpp` if you haven't already.

While it's not needed for every library, it's a good habit to include our library's header file in the implementation file. However, we have to use a variation of the `#include` directive. We've normally used this form:

```
#include <iosteam>
```

for accessing system libraries. A personal library that we have in our working directory uses different punctuation:

```
#include "metric.h"
```

Go ahead and add this line to the implementation file.

Defining A Function

The general pattern for a function definition is

`ReturnType` *`FunctionName`* (*`ParameterList`*) { *`StatementList`* }
where

- *`ReturnType`*, *`FunctionName`*, and *`ParameterList`* are *exactly* the same as in a function prototype.
- *`StatementList`* is a sequence of valid C++ statements.

The first bullet point here is very important. Once you've written your prototype, you can copy this line over to the implementation file for the beginning of the function definition. Just be sure to remove the semicolon at the end. A prototype requires that punctuation, but it will cause an error in a function definition.

A **function stub** is a minimal function definition. For example, this is the function stub for `feetToMeters()`:

```
double feetToMeters (double feet)
{
}
```

Why bother with a function stub? We can compile the program with a function stub. It gives us a great place to stop to check our program to make sure that at least the compiler likes what we've done.

Of course, we eventually fill the stub with code so that the function does something. That's already been done for `feetToMeters()`. Nothing's been done yet for the meters-to-feet function.

Start by writing a function stub for the meters-to-feet function, and add it to the implementation file. Compile the program. You should be able to compile the program without errors. You may get a warning that your function does not return anything, but we'll let that slide. Also, you only have to compile the `metric.cpp` file itself; you do *not* have to link it with the driver yet. You can, but you may get errors that you should ignore.

Now we can actually write code for our function using our algorithm.

1. Receive *feet* from the caller.

This step is taken care of for us by the C++ function-call mechanism. We've already done it! The mere act of declaring a parameter tells C++ that we want to receive this value. That's the whole purpose for parameters.

It's good to put this step in your algorithm to remind you where the data comes from, but once you have the parameters declared, the function has already received the values from whatever called it.

2. Halt the program if *feet* is negative.

This is our precondition. We decided that we'd do this with the `assert()` statement. That's exactly what's been done in `feetToMeters()`.

3. Compute *meters* = *feet* * 0.3048.

This step has been implemented in one declaration. Remember that we have to declare all of our variables. The *`feet`* variable has already been declared as a parameter---that counts!

However, this is the first time we've used `meters`, so we declare it and initialize it with our computation.

4. Return *meters*.

The third operation can be performed using the C++ **return statement**. Its general pattern is:

```
return Expression;
```
where *Expression* is any valid C++ expression. The result of the expression is sent back to whatever code called this function.

Now it's your turn. Use your algorithm for the meters-to-feet function plus the `feetToMeters()` example to implement your algorithm in the stub that you wrote earlier. Compile the code as you go along (probably after you write each line of code); be sure it compiles without errors or warnings when you finish.

Writing a Program to Test the Library

Next, open `driver.cpp` and personalize its documentation.

There's a program in this file to test the functions we've written. Such programs are called **driver programs** because all they do is test drive the functions in a library.

Designing The Driver Program

Most driver programs use the same algorithm:

1. Display a prompt for whatever values the function requires as arguments.
2. Read those values from `cin`.
3. Call the function that you're testing using the input values as arguments.
4. Display via `cout` the result of calling the function plus a descriptive label.

Since every driver program uses this same basic algorithm, we're going to jump to the coding stage of our development.

Coding the Driver Program

You've been given the code for the driver for this lab; well, most of the code. You'll have to write the drivers for future labs, so make sure you read over this code carefully and that you understand what it does.

Step 1 of the algorithm is merely an output statement. You've written a few of these already.

Step 2 is a matter of reading in data. This take a little more thought: you need to know what objects you need to read in and their data types. Declare variables for these objects, and then read in the data.

Step 3 will usually consist of a variable declaration with an initialization that calls the function we're testing.

Step 4 is just an output statement that displays a label describing the result and displays the result itself (otherwise, what's the point!).

Now, before we can use the library, we must include the header file in the driver file. Notice the saving we're already getting. We write the header file *once*, but we're using it *twice*. And this is just in one program! Be sure to use the `#include "..."` form to include a header file from your personal library.

Calling a Function

Let's look more closely at Step 3 of our driver algorithm.

At the beginning of this lab, we quickly looked at the `pow()` function. If we had two variables `x` and `y`, we can call the function like so: `pow(x,y)`. Put in a context, we can then use the value that this returns.

But we got the `pow()` function from a system library. What do we do with a function that we write ourselves? *Exactly the same thing.*

Consider `feetToMeters()`. It receives a number of feet. If we want to convert 1.0 foot to meters, we could write this:

```
feetToMeters(1.0)
```
(For now, don't worry about the context, what we do with the result). If we want to convert 2.0 feet to meters, we could write this:
```
feetToMeters(2.0)
```
If we want to convert `x` raised to the `y`th power feet to meters, we could write:
```
feetToMeters(pow(x,y))
```
In the case of our driver, though, the driver reads in a value, places it in `feet`, and we want to convert that value:
```
feetToMeters(feet)
```
The driver uses this as the expression to initialize the `meters` variable.

Our approach to the function has changed. When we declared the function, we declared a parameter---a variable---to receive the number of feet. But now that we're calling the function, we can pass in a literal, a variable, or any `double` expression. A value that we *pass into* a function is known as an **argument**.

Helpful hint: *A parameter is always a variable, it is part of a function **definition**, and we get to choose the name. An argument can be any expression, and it is part of a function **call**.*

There are three important rules to remember when calling a function:

1. The number of arguments in a function call must equal the number of parameters in the function's *prototype*, or a *compilation* error will occur.
2. The number of arguments in a function call must equal the number of parameters in the function's *definition*, or a *linking* error will occur.

3. The types of the first argument and first parameter, the second argument and second parameter, the third argument and third parameter, etc. must be the same, or a compilation error will occur.

If you get any complaints about your function, make sure you check the function call, the function prototype, *and* the function definition.

Compile and link all of your code, and test out your program.

As it stands, though, the program only tests `feetToMeters()`. It should also test `metersToFeet()`. There are several ways to do this:

- Write a brand new driver for each function that you write. This will lead to *lots* of programs, more than you really need.
- Duplicate the code to test `feetToMeters()` to test the meters-to-feet function. This leads to a lot of extra code, but not nearly as much as the first option.
- Duplicate only the code you *have* to in order to test both functions. This is the most general solution.

Let's try the third option here. So instead of reading in a number of feet, let's read in a generic number. Let's call it `measurement`. We can then use this one value as an argument to *both* functions. Its meaning in these functions will be different, a number of feet in the one and a number of meters in the other, but that's fine. Here's what the code would look like to test `feetToMeters()`:

```
// Step 2.
double measurement;
cin >> measurement;
// Step 3.
double meters = feetToMeters (measurement);
// Step 4.
cout << measurement << " feet == " << meters << " meters\n";
```

Now all you have to do is use this code in the driver, and then duplicate Steps 3 and 4 for the meters-to-feet function.

Testing

Once the program is successfully translated, we are ready to test what we have written, to see if we can discover any logic errors. Execute the program, and use the following sample data to test `feetToMeters()` correctness:

Feet	Meters
1.0	0.3048
3.3	1.00584
-5.9	ERROR

Since `feetToMeters()` is a linear function, two test values are sufficient to verify its correctness. Be sure to note and fix any discrepancies.

The third test value tests our precondition. This is also a good idea to make sure that the function *fails* when it's supposed to.

Question #4.8: What error message does the program give when you enter the invalid value?

Since you've been given `feetToMeters()`, it should run correctly. Your meters-to-feet function may not. The sample data above can also be used to test your meters-to-feet function.

Question #4.9: What sample data are you going to use and expect for your meters-to-feet function.

Debug your program until you get the correct results from both functions.

Maintenance

Better Error Messages

After you release your library, you might get some complaints about the error message the functions give when they fail. The error message is far too cryptic. An `assert()` statement is not always the best way to react to problems; alternatively, we can code our own precondition handling.

Let's consider `feetToMeters()`. We came up with its precondition almost from the beginning, and we have an expression for it: `feet >= 0`. We chose to use the `assert()` statement to test the precondition; this is not the only choice we have.

Think about what we want to happen: if the precondition is false (i.e., `feet` is *negative*), then we want to display a helpful error message and stop the program.

This new behavior, introduces one new object: `cerr`. Similar to `cout`, `cerr` is of type `ostream`, and will display its output to the screen. However, `cout` should be used for normal **out**put with the user while `cerr` should be use for **err**or messages. *You use `cerr` just the same way you use `cout`.*

We have four operations here:

Description	Predefined?	Name	Library
`feet` is negative	yes	<	built-in
if...then...	yes	`if` statement	built-in
display error message	yes	<<	`iostream`
stop the program	yes	`exit()`	`cstdlib`

The `exit()` function, when executed, will stop the program immediately. The function takes one argument, an `int`. The actual value of the argument is really up to you; convention is that any non-zero value means the program stopped because of a problem. (Yes, a zero value does mean that the program stopped normally.) You could be very sophisticated with your exit codes, but we'll keep it simple. So, use `1` or `-1`.

Using all of this information, we can replace Step 2 of the algorithm for `feetToMeters()`:

2. If *`feet`* is negative, then...
    a. Display a helpful error message.
    b. Exit the program.

What about that "if" operation? This is a language feature in C++ that we haven't seen yet, but it is built in to C++. The form for a simple **`if` statement** looks like this:

```
if (BooleanExpression)
{
   Statements
}
```

So, in the code, we can *replace* the call to `assert()` with this code:

```
if (feet < 0)
{
  cerr << "The number of feet is negative in feetToMeters()."
<< endl;
  exit (-1);
}
```

That's a much more informative message; the people who use the library will be much happier.

Make this change to `feetToMeters()`. Compile and test your program, especially the failure test.

Once you have this working for `feetToMeters()`, make similar changes to your meters-to-feet function. The new behavior, objects, operations, and algorithm should parallel the new design for `FeetToMeteres()` as should the new code.

Repeated Tests

You might find it a bit annoying to have to run the driver three times for the three test values we have above. It might be easier if the program ran through three tests. When we want something done more than once, we turn to **loops**. And when we do any counting, we turn to a **counting `for` loop**.

The general form of a counting `for` loop looks like this:

```
for (Type loopVar = firstValue; loopVar <= lastValue;
loopVar++)
{
  Statements
}
```

There's a fair amount of information that we need to provide for a counting `for` loop:

- *`Type`* will nearly always be `int`; it's the data type of *`loopVar`*.
- *`loopVar`* is the counting variable. It is initialized to *`firstValue`*, is incremented each time through the loop (with *`loopVar`*`++`), and stops when *`loopVar`* is greater than *`lastValue`*.
- *`Statements`* are C++ statements which will be executed once for *each* value of *`loopVar`*, *`firstValue`* through *`lastvalue`*.

This sounds like it could work for our driver. Here's a new algorithm:

1. Repeat three times:
    a. Display a prompt for whatever values the function requires as arguments.
    b. Read those values from `cin`.
    c. Call the functions that you're testing using the input values as arguments.
    d. Display via `cout` the result of calling the function plus a descriptive label.

The "repeat" action can be implemented with a loop:

```
for (int i = 1; i <= 3; i++)
{
  ...
}
```

The question is, what goes in the body of the function, replacing the `...`? According to the new algorithm, it's the same code we had before!

Add this counting `for` loop to your program, making sure the old steps of the algorithm are all put between the curly braces of the loop. Also make sure the counting `for` loop is inside the curly braces of `main()`.

Now that our code is getting a little more complex, we should address the issue of indentation:

Helpful hint: *You will find it much easier to debug your program if you indent your code properly.*

You may also find yourself losing a lot of points on your assignments if you don't indent properly.

Convention suggests that every time you have an opening curly brace, you should indent a little to the right (two to four spaces). We've already done this with our functions; you should also do this with your loops and selection statements.

Compile and execute your program. Make sure it works correctly.

Finally, here's something to think about: how could you get this loop to execute five times? Fifty times? Even better: how could you let the *user* choose how many times to execute the loop? You have the tools to do this, but we've already done quite a bit in this lab.

Submit

Turn in a copy of all of your code, the answers to the question, and a sample run of your program proving that the functions work properly.

Terminology

argument, argument, counting `for` loo, declaration, definition, documentation file, driver program, function, function call, function stub, header file, `if` statment, implementation file, interface, library, library module, loop, module, parameter, precondition, prototype, return statement

3. Project 4

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #4.1: Work in a group with other students to complete the `metric` library started in the exercise. Assign a different person to work on each of the following groups of functions:

Lengths	**Weights**
Inches to centimeters (1 inch = 2.54 cm) Yards to meters (1 yard = 0.9144 m) Miles to kilometers (1 mile = 1.609344 km)	Ounces to grams (1 ounce = 28.349523 g) Pounds to kilograms (1 pound = 0.453592 kg) Tons to kilograms (1 ton = 907.18474 kg)
Volumes	**Areas**
Pints to liters (1 pint = 0.473163 l) Quarts to liters (1 quart = 0.946326 l) Gallons to liters (1 gallon = 3.785306 l)	Square inches to square millimeters (1 sq. in. = 645.16 sq.mm) Square feet to square meters (1 sq. foot = 0.09290304 sq.m) Acres to square meters (1 Acre = 4.04686x10^3 sq.m).

Each person is responsible for writing the necessary function prototypes, definitions, and documentation specifications. Each person should write up an OCD design for at least one of their functions.

Each *group*, though, should submit *exactly one* library, bringing together all of the prototypes, definitions, and documentations (in three separate files, of course).

Then one of the people in the group (under the close scrutiny of the other group members) should extend the driver from the lab exercise to test all of the functions with one input value.

The documentation file should indicate which group member was responsible for which functions. Only that group member will be penalized for any errors in that set of functions or its documentation.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 5: Using Classes

1. Objective of the Experiment

- To learn the use of classes and methods.
- To experience working with text-processing problems.
- To see language-translation problems.

2. Theoretical Background

Introduction

In this lab's exercise, we examine two language translation problems. The first problem is the problem of making plural nouns, and the second problem is converting a word from English into Pig Latin. While they're rather different problems, both involve string manipulation. We will have to explore the `string` library of C++.

Making Plural Nouns

Making a noun plural usually consists of adding an 's', but sometimes there are special cases we must watch out for. Consider these examples:

Singular noun	**Plural noun**
exercise	exercises
noun	nouns
word	words
abscess	abscesses
summons	summonses
box	boxes
hobby	hobbies
party	parties

Translating English into Pig Latin

As for Pig Latin, there are two rules:

1. If the English word begins with a consonant, move the initial consonants to the end of the word and tack on "ay".
2. If the English word begins with a vowel, tack on "yay" to the end of the word.

English	**Pig Latin**	**English**	**Pig Latin**
alphabet	alphabetyay	nerd	erdnay
billygoat	illygoatbay	orthodox	orthodoxyay
crazy	azycray	prickly	icklypray
dripping	ippingdray	quasimodo	asimodoquay

eligible	eligibleyay	rhythm	ythmrhay
farm	armfay	spry	yspray
ghost	ostghay	three	eethray
happy	appyhay	ugly	uglyyay
illegal	illegalyay	vigilant	igilantvay
jury	uryjay	wretched	etchedwray
killjoy	illjoykay	xerxes	erxesxay
limit	imitlay	yellow	ellowyay
messy	essymay	zippy	ippyzay

Files

Directory: `lab5`

- `piglatin.h`, `piglatin.cpp`, and `piglatin.doc` implement a library for translating into Pig Latin.
- `translate.cpp` implements a driver for translating keyboard input from English to Pig Latin.
- `Makefile` is a makefile.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

Strings of Characters

To solve both problems, we first look at the `string` library.

The `string` data type in C++ is implemented with a **class**. A class is a way for us to **encapsulate** both data *and* actions together into one object. In a future lab, we'll look at how we can write our own classes, but for now we'll use the classes that C++ provides for us.

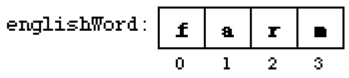
A `string` object is used to hold a *string (or sequence) of characters*. A single character (i.e., a `char`) is rarely that useful by itself, so you'll find yourself using `string`s any time you need to process text.

`string` Objects

We can easily declare and initialize `string` objects:

```
string englishWord = "farm";
```

After the declaration, `englishWord` is a `string` object. A `string` is an **indexed type**, so that we can access the individual characters in the string:



The numbers 0, 1, 2 and 3 are the **index values** of the characters within `englishWord`, and can be used to access the individual characters within the `string` object.

The most important thing about indexing the characters of a `string` is where the indexing starts:

Helpful hint: *The indexing of a* `string` ***always*** *starts at index 0.*

So the first character of the `string` has index 0, the second characters has index 1, the third character has index 2, and so on up to the last character whose index is *one less than the size of the* `string`. Keep all of this in mind as you use indices to access the characters and substrings of a `string`.

`string` Operations

As mentioned above, a class encapsulates both data and actions into one object. As a class, the `string` class has many operations we can use on `string` objects.

An operation in a class is usually implemented as a **method** (also known as an **instance method** or a **function member**). A method is merely a function defined in a class. We'll see how to make these definitions ourselves in a later lab; for now all we need is the ability to call them. This requires a slightly different syntax for our function call:

```
obj.method(args);
```

In the past, we didn't have an object before the function name when we called the function; but with a method, the method must be applied to an object. Many object-oriented programmers think of methods as **messages** that they send to the objects. For example,

```
int size = stringObject.size();
```

We'd think of `stringObject` receiving a message `size()` which asks the object to figure out its size. In other words, "Hey, `stringObject`, what's your size?" This metaphor becomes more important when we implement classes, but it's also helpful in mastering this new syntax for calling methods.

Let's look at some of the provided operations of the `string` class. Suppose that `str` is a `string`:

Description	Syntax	Explanation
Index operator	`str[index]`	Returns the character at `index` from `str`. If `index` is out of range for `str`, it will return junk or crash your program.
Size of `string`	`str.size()`	Returns the size of the string.

Concatenation of two strings	`str1 + str2`	Returns a `string` equal to the first `string` followed by the second.
Equality of two strings	`str1 == str2`	Returns `true` if the two `string`s are equal, `false` otherwise.
Substrings	`str.substr(start,size);`	Returns the substring starting at index `start` of length `size`.
Finding characters in a string.	`str.find_first_of(pattern,index)`	Returns the index of the first character in `pattern` that it can find in `str`, starting at `index`. If it cannot find a matching character, then it returns `string::npos`.

The first three operations are pretty straightforward. Let's look at some examples of the `substr()` method:

```
string str = "hello";
string sub1 = str.substr(0,2);
assert (sub1 == "he");
string sub2 = str.substr(3,2);
assert (sub2 == "lo");
```

This code compiles and executes without any problems. The last example (involving `sub2`) is important because the second argument is a *size*, not an index (a common mistake). Often we have the *index* of the last character we want; if we also have the index of the first character of the substring, computing the size is simple:

```
size = last - first + 1
```

This is a *very* important computation, and you should keep this *very* handy.

The `find_first_of()` is a little more involved. Consider this code:

```
string sample = "Hello.  Nice to meet you!";
int firstIndex = sample.find_first_of(".!?", 0);
```
This searches `sample` to find the first occurrence of a character from the pattern `".!?"`. It does *not* need to find the entire pattern, just *one* of the characters from the pattern. So, in the code above, `firstIndex` should be set to 5, the index of the first period in the string.

The starting index (`0` in our example) allows us to start a search in the middle of a `string`. Often you'll use `0` as your starting index just like above, but we could continue our example:

```
int secondIndex = sample.find_first_of(".!?", firstIndex+1);
```

This starts the search right after the previous search and finds the *second* occurrence of one of the characters.

Question #5.1: What will be the value of `secondIndex`?

Question #5.2: What would be the value of `secondIndex` if we started the search at `firstIndex` instead of `firstIndex+1`?

Remember that it is *not* important to memorize a library description like this. It *is* important to generally know what's available in the library, where you can find the library, and (most importantly) where you can find documentation for the library (Appendix D of *C++: An Introduction to Computing* by Adams and Nyhoff is one place). The discussion in this section will suffice for this lab---don't memorize this, but be sure to come back to this section when working with these methods.

`if` Statements

In the previous lab, we used a simple `if` statement. Our functions in this lab require a more powerful `if` statement. A full `if` statement looks like this:

```
if (BooleanExpression)
{
   Statements1
}
else
{
   Statements2
}
```

If the `BooleanExpression` is true, then only `Statements`$_1$ will be executed; otherwise (that is, if `BooleanExpression` is false), then only `Statements`$_2$ will be executed. We can chain two `if` statements into what is known as a multi-branch `if`:

```
if (BooleanExpression1)
{
  Statements1
}
else if (BooleanExpression2)
{
  Statements2
}
else
{
  Statements3
}
```

The pattern generalizes (as we'll see in the next lab). The boolean expressions are evaluated *in order* until the *first* one that evaluates to true, and then its corresponding statements are executed. If none are true, the last block of statements (which an `if` test) is executed. Exactly *one* block of statements is executed each time through the statement.

Plural Nouns

Analysis

Examine the chart of singular and plural nouns at the beginning of this exercise. Can you pick out the rules we need?

We'll use some simple rules:

1. In general, add "s" to the end of the word.
2. If the word ends in "s" or "x", add "es" to the end of the word.
3. If the word ends in "y", replace the "y" with "ies".

These rules are not comprehensive, but deriving a set of comprehensive rules for making plural nouns in English is rather difficult. For example, consider the word "radius" whose plural is "radii", not "radiuses". However, this only applies to English words with Latin roots: the plural of "bonus" is "bonuses"; "-us" words are very tricky. Making the rules comprehensive would take a very long time, so we'll stick with just these simple rules.

Question #5.3: For each words in the chart above, match it up with the rule that made it plural.

Seems simple enough. It also seems reasonable that a "pluralizing function" could be generally useful. So it certainly deserves a function and even a library.

OCD

Here's our behavior:

Our function should receive a singular noun. If the noun ends in "s" or "x", return the noun with "es" tacked on the end. If the noun ends in "y", return the noun with the "y" replaced with "ies". Otherwise, return the noun with "s" tacked on the end.

So far so good. It's just a transliteration of the rules. Note that the general case comes last.

Here are the objects we need:

Description	Type	Kind	Movement	Name
the singular noun	`string`	varying	in	`singularNoun`
the index of the last character of the noun	`int`	varying	local	`lastCharIndex`
the last character of the noun	`char`	varying	local	`lastChar`
the plural noun	`string`	varying	in	--

We have a specification for our function:

Specification:

receive: a singular noun, a `string`
precondition: the noun should be singular
return: the plural version of the noun, a `string`

In the previous lab, we used an `assert()` statement to enforce our precondition. We could *try* to enforce the precondition for this function, but it would take a *lot* of work. In fact, we'd have to implement an entire dictionary because you *cannot* tell just by looking at the letters in a word if it is already plural or not.

This is okay. Not every precondition can or should be enforced by your code. It certainly makes the code better if you can enforce your preconditions, but it's not absolutely necessary. It *is* absolutely necessary, however, to clearly indicate that this is a precondition for the function. If a programmer uses your function and passes in a plural word (like "nouns") and gets strange results (like "nounses") it shouldn't be a surprise to the programmer---we warned him!

Using the `string` operations listed above and other operations we've seen before, here are the operations we need for this function:

Description	Predefined?	Name	Library
receive a value	yes	parameter	built-in
if...then...	yes	`if` statement	built-in
get last character of string	yes	`[]` and `size()`	`string`
compare characters	yes	`==`	built-in
concatenate two strings	yes	`+`	`string`
extract a substring	yes	`substr()`	`string`
return a string	yes	`return`	built-in

So, finally, we have our algorithm:

```
1. Receive singularNoun.
2. Let lastCharIndex be the size of singularNoun minus 1.
3. Let lastChar be the character of singularNoun at index lastCharIndex.
4. If lastChar is 's' or 'x',
     Return singularNoun + "es".
   Otherwise if lastChar is 'y',
       a. Let base be singularNoun without the trailing "y".
       b. Return base + "ies".

       Otherwise,
         Return singularNoun + "s".
```

Pig Latin Translator

Let's turn to your coding of the Pig Latin translator.

Analysis

Like the pluralizer, our Pig Latin translator will have certain rules to transform a word based on the contents of the word. In the case of the pluralizer, the rule we applied was determined by the last letter of the word. For your Pig Latin translator, it's the location of the first vowel.

Consider some examples:

- The Pig Latin equivalent of "dog" is "ogday"; "style" becomes "ylestay"; "stripe" becomes "ipestray". The initial consonants are moved to the end of the word and "ay" is tacked on the end.
- If the word begins with a vowel: "apple" and "orange" become "appleyay" and "orangeyay". There aren't any consonants to move, but we tack "yay" (not just "ay") to the end of the word.

The focal point in both rules is the first vowel. We'll define 'a', 'e', 'i', 'o', 'u', and 'y' as vowels. While 'y' is only sometimes a vowel, the rules work better if we consider it a vowel.

OCD

Let's try this behavior:

Receive an English word. Find the position of the first vowel in that word, checking that a vowel was actually present. If a vowel begins the word, then return the concatenation of the English word and "yay". Otherwise, the Pig Latin word consists of three parts (in order): (1) the portion of the English word from the first vowel to its end, (2) the initial consonants of the English word, and (3) "ay". Return the Pig Latin word.

Here are the objects that you need:

Description	Type	Kind	Movement	Name
the English word	`string`	varying	in	`englishWord`
index of the first vowel	`int`	varying	local	`vowelPosition`
the Pig Latin word	code string	varying	out	`piglatinWord`
portion of the English word from its first vowel until its end	`string`	varying	local	`lastPart`
consonants at the beginning of the English word	`string`	varying	local	`firstPart`
"yay"	`string`	constant	local	--
"ay"	`string`	constant	local	--

This gives us the following specification for our function:

Specification:
receive: `englishWord`, a `string`.
precondition: `englishWord` should have at least one vowel in it.
return: `piglatinWord`, a `string`.

Since this method could be useful in many places (okay, just a bit of a stretch there), we'll use a library. Use the specification to create a prototype for a function named `EnglishToPigLatin()` in `piglatin.h`; copy the prototype into the documentation file; and then add a stub for the same function in `piglatin.cpp`.

The rest of the design is up to you:

Question #5.4: Write out a chart outlining the operations you need for the Pig Latin translator.

Question #5.5: Write an algorithm for the Pig Latin translator.

Plural Nouns (Again)

Coding

We can code up our algorithm for the "pluralizing" algorithm like so:

```
string Pluralize (string singularNoun)
{
  int lastCharIndex = singularNoun.size() - 1;
  char lastChar = singularNoun[lastCharIndex];
  if ((lastChar == 's') || (lastChar == 'x'))
    return singularNoun + "es";
  else if (lastChar == 'y')
    {
      string base = singularNoun.substr (0, lastCharIndex);
      return base + "ies";
    }
  else
    return singularNoun + "s";
}
```

Spend some time comparing the algorithm to the code.

Question #5.6: How much time did you spend comparing the algorithm and the code?

It wasn't enough time. Spend some more time.

Now let's break this down:

Step 1 is done automatically through the parameter passing mechanism.

Step 2 uses the `size()` method to get the index of the last character. Step 3 uses this index to get the last character of `singularNoun`.

Step 4 is a multi-branch `if` statement. The conditions of the `if`s compare two `char`s. When we figure out what rule we need to apply, each rule results in a string concatenation. Two of those are quite simple, but the second one deserves a closer examination.

Recall that the `substr()` method needs a beginning index and a size of the substring. The beginning index is easy: since we want everything but the last character (which is a 'y' we're discarding), we need to start at index 0. It doesn't matter what's in `singularNoun` or how long it is, a `string` always begins at index 0.

But how long is this substring? `lastCharIndex` is the index of the last character. That's the character we need to avoid, so for our substring, the *previous* character is the last character of our substring. That previous character has index `lastCharIndex`-1. Using the formula above, we come up with:

`substrsize` = (`lastCharIndex`-1) - 0 + 1 = `lastCharIndex`
This explains why `lastCharIndex` is doing double duty as a index into `singularNoun` *and* as a size for the substring.

Go back one more time to compare the algorithm and the code. There's supposed to be little thought going from the algorithm to the code. You'll struggle with the syntax, but you don't have to get too creative. The double use of `lastCharIndex` is a bit tricky, but otherwise the two match up very well.

Pig Latin Translator (Again)

Coding

The trickiest operation you'll have to worry about is finding the first vowel in a word. Unlike the noun pluralizer, you cannot test a fixed position with the index operator. You have to search for it. However, read the list of operations above carefully because one of those methods (probably the one we *didn't* use in the pluralizer---hint! hint! hint!) will mostly solve the problem for you.

Then, extract substrings from the English word to build up the Pig Latin word. Remember to use the formula above to compute the size of a substring.

Code up your algorithm, compile, and run your program.

Testing

Test your Pig Latin translator on all of the words listed above plus others that you come up with.

Maintenance

As you test your function, you'll discover that your function *doesn't* work on all words. (This should teach you to read the entire lab before working on it!) In particular, your function has problems with words that start with 'y' (like "yellow") or that contain a 'q' in the initial consonants (like "quiet" or "squire").

Words beginning with 'y' pose a problem because we consider 'y' to be a vowel. If we didn't, words like "style" or "spry" wouldn't translate properly. It appears that we need a special case for words beginning with 'y' because then it usually works as a consonant. But this new rule isn't perfect: "Ypsilanti" begins with a 'y' vowel.

Words with a 'q' in the initial consonants also poses a problem because the 'u' after it should also be moved to the end of the word: "squire" should translate to "iresquay", not "uiresqay". This requires a special case. We can complicate the special case further if we permit English words that do *not* have a 'u' after a 'q'. Strictly speaking, this isn't permitted in English, but many foreign words absorbed into English have a 'q' without a 'u' right after it.

These are both cases that should be considered for a full-fledged Pig Latin translator, although we won't do them here. See Project #5.1.

Submit

Turn in a copy of your code, your answers to the questions, and sample runs of your program demonstrating that it works correctly.

Terminology

class, encapsulate, function member, index, indexed type, instance method, message, method

3. Project 5

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #5.1: As written, our `Piglatin()` function fails to correctly translate words such as *yes* (*esyay*) and *yellow* (*ellowyay*) in which the first vowel 'y' is actually being used as a consonant. It also fails to correct translate words such as *quiet* (*ietquay*) and *quack* (*ackquay*), in which the first vowel is a 'u' following a 'q'. Redesign the algorithm used by `Piglatin()` so that it will correctly translate words like *yes*, *yellow*, *quiet*, *quack*, and *squire* without losing the ability to correctly translate words like *style*, *rhythm*, *luck*, and *blue*.

Project #5.2: A character string is said to be a palindrome if it reads the same when the order of its characters is reversed. For example, the following are all palindromes:

```
madam
smada bob adams
able was I ere I saw elba
```

Write a function that, given a character string, returns true if that string is a palindrome, and returns false otherwise.

Project #5.3: A (very) simple encryption method is to reverse the order of the characters in a word, so that if the characters in the message

```
OEMOR OEMOR EROFEREHW TRA UOHT
```

are reversed word-by-word, then the decoded message reads:

```
ROMEO ROMEO WHEREFORE ART THOU
```

Write a function that, given a character string, returns the reverse of that character string.

Project #5.4: Internet mailers and news-readers often contain a "rot13" function that can be used to "encrypt" words by rotating their characters 13 positions (i.e., A becomes N, B becomes O, C becomes P, ... Z becomes M). This is convenient, because the same function can be used to both "encrypt" and "decrypt" a word:

```
rot13("ROMEO") == "EBZRB"
```

and

```
rot13("EBZRB") == "ROMEO"
```

Write a `Rot13()` function that, given a string, returns a string whose characters are those of the first string rotated 13 positions. (Hint: You may find the `%` operator to be useful.)

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 6: Selection

1. Objective of the Experiment

- To review selective execution.
- To learn the switch statement.
- To practice writing functions that use selection statements.

2. Theoretical Background

Introduction

One of the easiest ways to make a program user-friendly is to make it **menu driven**. That is, rather than prompting the user in some vague sort of way, we present them with a **menu** of the choices available to them. Then all the user has to do is look at the menu and choose one of the choices. Since the menu always tells the user what their options are, the user needs no special knowledge, making such a program easy to use.

For example, a simple 4-function calculator program might prompt the user by providing the following menu:

```
Please enter:
  + to add two numbers.
  - to subtract two numbers.
  * to multiple two numbers.
  / to divide two numbers.
```

Thanks to the menu, a user knows exactly what to enter.

This lab's exercise is to complete such a program, and at the same time learn about some of the C++ **control structures** for **selection**.

Old Code

Let's first take a look at some old code. In the previous lab, we looked at this function:

```
string pluralize (string singularNoun)
{
  int lastCharIndex = singularNoun.size() - 1;
  char lastChar = singularNoun[lastCharIndex];
  if ((lastChar == 's') || (lastChar == 'x'))
    return singularNoun + "es";
  else if (lastChar == 'y')
    {
      string base = singularNoun.substr (0, lastCharIndex);
      return base + "ies";
    }
  else
    return singularNoun + "s";
}
```

This function used a multi-branch `if` to decide which rule to apply in making the noun plural.

General `if` Statement

The general pattern for an `if` statement looks like this:

```
if ( Condition )
  Statement1
[else
  Statement2 ]
```

Either *Statement*$_1$ *or* *Statement*$_2$, but not both, are executed based on the value of *Condition*. If *Condition* is `true`, then *Statement*$_1$ is executed; otherwise (i.e., *Condition* is `false`), *Statement*$_2$ is executed.

The square brackets `[...]` in the pattern indicates that the `else` clause is *optional*. Sometimes a simple `if`, without an `else`, is all we need.

Multi-branch `if` Statement

We can also chain several `if`s together into one statement. Now generally it's not advisable to put an `if` statement in as *Statement*$_1$ of another `if`. These can get confusing. However, using an `if` statement for *Statement*$_2$ is perfectly acceptable---encouraged, even. In general, a multi-branch `if` looks like this:

```
if (Condition1)
  Statement1
else if (Condition2)
  Statement2
...
else if (ConditionN)
  StatementN
else
  StatementN+1
```

Consider the pluralizer above. There are four possibilities for the singular noun: it ends in an 's', it ends in an 'x', it ends in a 'y', or it ends in something else. This chain of rules/conditions leads to a chain of `if`s.

Exactly *one* of the *Statement*$_I$ will be executed. It will be the *first* *Statement*$_I$ where *Condition*$_I$ is `true`. All of the previous conditions must be false. If all conditions fail, then the fail-safe *Statement*$_{N+1}$ is executed. This is *exactly* what we want and need for the pluralizer; you'll also need it for the code in this lab.

Helpful hint: *Include a fail-safe* `else` *in a multibranch* `if`*, even if it's just a debugging statement.*

The debugging statement can be as simple as a statement that prints an error message like "This statement should not print". This will make things much easier for you to debug your program when (not if) something goes wrong.

Also, there is *no condition after the last* `else`.

Helpful hint: *The last* `else` *in any* `if` *statement should* **not** *have a condition test.*

When any `else` clause is triggered, we *know* that the previous test failed, so there's no need to make a further test. For example, we *could* write:

```
if ((lastChar == 's') || (lastChar == 'x'))
  return singularNoun + "es";
else if ((lastChar != 's') && (lastChar != 'x') && (lastChar
== 'y'))
{
  string base = singularNoun.substr (0, lastCharIndex);
  return base + "ies";
}
else if ((lastChar != 's') && (lastChar != 'x') && (lastChar
!= 'y'))
  return singularNoun + "s";
```

But the extra tests in this version are *completely* unnecessary. When *any* of the `!=` tests are evaluated in this new multi-branch `if`, they will *always* be true because they only made *after* the corresponding `==` tests were discovered false. This is why the original version of `pluralize()` does *not* have these unnecessary `!=` tests.

Compound Statements

The patterns above for the `if` and multi-branch `if` statements were carefully crafted. In particular, we used the *singular* "*Statement*", not "*Statements*". We can put only *one* statement in those places in the patterns.

What if we need more than one statement? Well, that's exactly what we needed with the second rule for pluralizing a word which involves *two* statements. C++ allows us to wrap several statements in curly braces and treat them as one:

```
{
  Statement1
  Statement2
  ...
  StatementN
}
```

So that's why there are those curly braces around the two statements for the second rule. If we dropped them, then the compiler would get confused over the last `else` since there isn't a corresponding `if` close enough for it.

Files

Directory: `lab6`

- `calculate.cpp` implements the driver.
- `mathops.h`, `mathops.cpp`, and `mathops.doc` implement a math library.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

Looking at the Code

Take a few moments to study `calculate.cpp`, particularly the code in the `main()` routine. Make sure you understand the purpose of each statement in `calculate.cpp` before going any further in this lab.

Ignoring its error-checking code, `main()` should behave like so:

Our program should display on the screen a greeting, followed by a menu of permissible operations. It should then read a user-specified operation from the keyboard. It should then prompt the user for the two operands for that operation, and then read those operands from the keyboard. **It should then compute the result of applying the user-specified operation to the two operands.** It should conclude by displaying that result on the screen, with appropriate labeling.

All of this behavior is coded up in `main()` expect for the sentence written in boldface. Since there is no predefined C++ capability to directly perform that operation, we will write a function `apply()` to do that action. This function is (arguably) useful beyond just this program, so we'll put it off in a library.

Function Design

As usual, we use object-centered design to develop this function.

Behavior. Our function must apply *`operation`* to *`op1`* and *`op2`* and return the result. We can describe the needed behavior as follows:

Our function should receive from its caller an operation, and two operands. If the operation is `'+'`, our function should return the sum of the two operands. If the operation is `'-'`, our function should return their difference. If the operation is `'*'`, our function should return their product of the two operands. If the operation is `'/'`, our function should return their quotient.

Objects. From this behavioral description, we can identify the following objects:

Description	Type	Kind	Movement	Name
The operation	`char`	varying	received	*`operation`*
One of the operands	`double`	varying	received	*`op1`*
The other operand	`double`	varying	received	*`op2`*
The sum of the operands	`double`	varying	out	*`op1 + op2`*
The difference of the operands	`double`	varying	out	*`op1 - op2`*
The product of the operands	`double`	varying	out	*`op1 * op2`*
The quotient of the operands	`double`	varying	out	*`op1 / op2`*

From this list, we can specify the behavior of our function as follows:

Specification:
receive: `operation`, a `char`; `op1` and `op2`, two `double` values.
return: the `double` result of applying `operation` to `op1` and `op2`.

Please, please, please save yourself much pain and watch the name of the first parameter:

Helpful hint: *Do not name the first parameter `operator`. It's a keyword, and the compiler may not give you the best error messages if you do use it as a variable.*

Using the specification (and heeding the hint), add a prototype for a function named `apply()` in `mathops.h` and a stub in `mathops.cpp`.

Function Operations From our behavioral description, we have these operations:

Description	Predefined?	Name	Library
Receive *`operation`* (a `char`)	yes	function call mechanism	built-in
Receive *`op1`* (a `double`)	yes	function call mechanism	built-in
Receive *`op2`* (a `double`)	yes	function call mechanism	built-in
Return the...	yes	`return`	built-in
...sum of *`op1`* and *`op2`*	yes	`+`	built-in
...difference of *`op1`* and *`op2`*	yes	`-`	built-in
...product of *`op1`* and *`op2`*	yes	`*`	built-in
...quotient of *`op1`* and *`op2`*	yes	`/`	built-in
Do exactly **one** of math operations	yes	**`if` statement**	built-in

C++ provides facilities for performing each of these operations as noted in this chart. The last operation is different from the others, in that it requires **selective behavior**. We can use the `if` statement.

Function Algorithm. We can organize these operations into the following algorithm:

```
1. Receive operation, op1 and op2.
2. If operation is '+':
      Return op1 + op2.
   Otherwise, if operation is '-':
      Return op1 - op2.
   Otherwise, if operation is '*':
      Return op1 * op2.
   Otherwise, if operation is '/':
      Return op1 / op2.
   Otherwise,
      Print an error message and return 0.
```

From this algorithm it is clear that a multi-branch `if` will do the trick. The last case is technically unnecessary since our program checks the operation, but as noted above you shouldn't rely on that. Print out an error message here so that it's *very* clear that something went wrong. If you want, you can even quit the program (with `exit(1);`) instead of returning 0, which is merely an arbitrary value in this case.

Add a multi-branch `if` to your `apply()` stub to encode this step.

Testing and Debugging.

When you are done, compile everything and test your program. Fix it up until it works correctly. Be sure to test each operator at least twice.

Coding 2: The `switch` Statement

The multi-branch `if` suffers from one drawback:

- To perform the addition operation, one condition is evaluated: `(operation == '+')`.
- To perform subtraction, two conditions are evaluated: `(operation == '+')` and `(operation == '-')`.
- To perform multiplication, three conditions must be evaluated: `(operation == '+')`, `(operation == '-')`, and `(operation == '*')`.
- And so on...

In general, selecting *`Statementi`* using a multi-branch `if` statement requires the evaluation of *i* conditions. This can be good because we can assume that the failed test did, in fact, fail. When testing for ranges of values, this is incredibly useful (and even necessary).

On the other hand, the evaluation of each condition consumes time, statements that occur later in the multi-branch `if` statement take longer to execute than do statements that occur earlier. It would be great if we could avoid this if at all possible. One possibility is using a `switch` statement.

A `switch statement` looks like this:

```
switch ( ConstantExpression )
{
CaseList1
  StatementList1
CaseList2
  StatementList2
...
CaseListN
  StatementListN
default:
  StatementListN+1
}
```

`ConstantExpression` is any C++ expression that evaluates to a **integer-compatible constant**. Each *`StatementListi`* is a sequence of valid C++ statements. Each *`CaseListi`* is one or more **cases** of the form:

```
case Constant :
```

`Constant` is an integer-compatible constant.

That's a fair bit of code. Watch the careful use of terminology here, particularly "integer-compatible constant". Let's first figure out what a `switch` statement does:

1. The *`ConstantExpression`* is evaluated.
2. If the value of the expression is present in *`CaseListI`*, then execution *begins* in *`StatementListI`* and proceeds, until a `break` statement, a `return` statement, or the end of the `switch` statement is encountered.
3. If the value of *`ConstantExpression`* is not present in any *`CaseListI`*, then the (optional) default *`StatementListN+1`* is executed.

A given *`Constant`* can appear in only one *`CaseListi`*.

Helpful hint: *Just as a multi-branch `if` should always have a fail-safe `else` clause, you should always have a `default` case in a `switch` statement, even if it just prints an error message.*

Perhaps the trickiest thing about a `switch` statement is the purpose of the cases. The cases of the switch statement are merely *labels* for the compiler to tell it where to *start* executing code. Consider this code:

```
switch (i)
{
case 0:
  cout << "Hi ";
case 1:
  cout << "there, ";
default:
  cout << "people!";
}
```

Then, if `i` is 0, this prints

```
Hi there, people!
```

The cases tell the compiler where to *start* executing the code, *not* when to stop. While this is useful in some situations, it won't be for us. When I make a noun plural, I want to apply only *one* of the rules. When I add two numbers together, I don't also want to multiply them. To stop executing the code at the end of a statement list, we can the statement list with a `break` statement, a `return` statement, or a call to `exit()`.

We've used `returns` for returning values from a function. We've also use the `exit()` function for stopping the program in case of an error. The `break` statement is even simpler:

```
break;
```

Helpful hint: *The statement list of every case list in a* `switch` *statement should end with a* `break`*, a* `return`*, or an* `exit()`*.*

So when would we use a `switch` statement? Whenever we can. Unfortunately, there are a few restrictions:

- Each `case` *must* be a constant.
- The *`ConstantExpression`* and the case values must all be *integer-compatible*.
- Finding a matching `case` is done *only* through equality tests.

The integer-compatible data types in C++ include `int` (of course), `char`, and `bool`. *It does not include* `double` *or* `string`. That's very important, so maybe you should read it again, this time out loud: *"integer-compatible" does not include* `double` *or* `string`*.*

Consider the chart below. If we write the algorithm on the left, we can use the `switch` in the middle which is equivalent to the multi-branch `if` on the right:

Set *Variable* to *ConstantExpression*.
If (*Variable* is equal to *Value*$_1$) then
  StatementList$_1$
Else if (*Variable* is equal to *Value*$_2$) then
  StatementList$_2$
...
	switch (*ConstantExpression*)
	{
	case *Value*$_1$:
	  StatementList$_1$
	  break;
	case *Value*$_2$:
	  StatementList$_2$
	  break;
	...
	case *Value*$_N$:
	  StatementList$_N$
	  break;
		Variable = *ConstantExpression*;
		if (*Variable* == *Value*$_1$)
		{
		  StatementList$_1$
		}
		else if (*Variable* == *Value*$_2$)
		{
		  StatementList$_2$
		}
		...

Else if (*Variable* is equal to *Value*$_N$) then *StatementList*$_N$ Else *StatementList*$_{N+1}$ End if.	default: *StatementList*$_{N+1}$ }	else if (*Variable* == *Value*$_N$) { *StatementList*$_N$ } else { *StatementList*$_{N+1}$ }

A `switch` statement is more efficient than the multi-branch `if` statement because a `switch` statement can select *StatementList*$_I$ in the time it takes to evaluate one condition. The `switch` statement thus eliminates the non-uniform execution time of statements controlled by a multi-branch `if` statement. But since the `switch` statement has so many restrictions on it, we use it only occasionally.

We have a slight wrinkle with `pluralize()` because it's first test actually tests *two* things. However, *both* tests are equality tests with constants and one variable. We can use *two* cases for *one* statement list in a `switch` statement.

So we could rewrite `pluralize()` like so:

```
string pluralize (string singularNoun)
{
  int lastCharIndex = singularNoun.size() - 1;
  char lastChar = singularNoun[lastCharIndex];
  switch (lastChar)
  {
  case 's':
  case 'x':
    return singularNoun + "es";
  case 'y':
    string base = singularNoun.substr (0, lastCharIndex);
    return base + "ies";
  default:
    return singularNoun + "s";
  }
}
```

We're careful in this code to end each statement list with a `return`. Each `case` has an integer-compatible (i.e., `char`) constant.

Try it out for yourself,

1. Copy the multi-branch `if` version of `apply()` so that you have two copies of it in the same file.
2. Rename one of them to be `apply2()`.
3. Now replace the multi-branch `if` in `apply()` with an equivalent `switch` statement. (Leave `apply2()` alone.)

Translate and test what you have written.

When everything is working right, save your work and print a hard copy of your code.

Submit

Turn in your code and sample runs that proves your calculator works for all operations.

Terminology

case, control structure, integer-compatible constant, menu, menu driven, selection, selective behavior

3. Project 6

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #6.1: There are three possible sources of user error in `calculate.cpp`:

1. The user could enter an invalid *operation* (something other than `+`, `-`, `*` or `/`).
2. The user could enter a non-numeric value for *op1* or *op2*.
3. The user could enter `/` for *operation* and `0` for *op2* (i.e., a divide-by-zero error).

Our program uses `assert()` to guard against these errors, but the diagnostic message displayed by `assert()` is not particularly informative or user-friendly. Replace each of the calls to `assert()` in `calculate.cpp` with a selective-behavior statement that displays a more user-friendly diagnostic message and terminates the program (using `exit()` from `cstdlib`) if the user enters erroneous information.

Project #6.2: Write a menu-driven "police sketch artist" program. The program should use four different menus for:

- hairstyle (e.g., bald, crew-cut, curley, wearing a hat)
- eyes (e.g., beady, bug-eyed, glasses, closed)
- nose (e.g., pug, small, medium, large)
- mouth (e.g., puzzled, smiling, bearded, frowning)

Each menu must provide at least four different choices. Your program should display "sketches" of the person being described, along the lines of those below (hopefully yours will be even better!):

```
  -----
 |     |        .......     \|||||||/
---------       .     .      |     |
(| O O |)      (|-0-0-|)    (| . . |)
 |  _\ |        |  ^  |      |  >  |
 |\___/|        | --- |      |||-|||
  -----          -----        |||||
                                |||
```

Organize your program in such a way that it contains no redundant code. For each of the user's choices, write a separate function to process that choice.

Project #6.3: A year is a leap year if it is evenly divisible by 4, unless it is divisible by 100, in which case it must also be divisible by 400. That is, 1996 was a leap year because it is divisible by 4 and not 100, 1997 was not a leap year because it is not divisible by 4, 2000 was a leap year because it is divisible by 400, but 2100 will not be a leap year because it is divisible by 100 but not 400.

Write a `LeapYear()` function that, given a year, returns true if that year is a leap year, and returns false otherwise. Then write a driver program that tests your function.

Project #6.4: Using the `metric` library you created for the project of Lab #4, write a menu-driven program that permits the user to select one of the metric conversions in the library.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 7: Repetition

1. Objective of the Experiment

- To practice using boolean expressions.
- To learn more about repetitive behavior.
- To examine all of the repetition statements of C++.

2. Theoretical Background

Introduction

We have seen that the C++ `if` statement uses a **condition** to **selectively execute** statements. In addition to facilitating selection, most modern programming languages also use conditions to permit statements to be executed **repeatedly**.

Let's consider an example. Suppose we have 2500 words that we want to translate into Pig Latin---about as many words as this lab exercise. The program we wrote in Lab #5 could do this, although the `englishToPigLatin()` function only worked on one word. The key was that the main driver repeatedly called `englishToPigLatin()` for each word that the user typed in. We can use that program to translate all 2500 words, and that's *so* much easier than running the program 2500 times or writing 2500 calls to `englishToPigLatin()`.

C++ technically provides three different loops, although we'll have four different uses for them. These are called the `while` loop, the `do` loop, the `for` loop, and what we call the *forever* loop.

This lab's exercise is to make a calculator program that has more functionality and is more user-friendly than the one we wrote in the last exercise.

Files

Directory: `lab7`

- `calculate.cpp` is the driver.
- `mathops.h`, `mathops.cpp`, and `mathops.doc` implement a math library.
- `menus.h`, `menus.cpp`, and `menus.doc` implement a library for menus.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

The Exponentiation Operation

We will implement x^n by writing a function `power()`, such that a call:

`power(x, n)`
will compute and return `x` raised to the power `n`. To simplify our task, we will assume that `n` is a nonnegative integer.

You may recall that exponentiation is available in C++ via the function `pow()` in the `cmath` library. Just for this lab, we will not use `pow()` so that we can try to implement it with loops.

Function Design

As usual, we begin by using object-centered design to carefully design an exponentiation function. However, before we can describe its behavior, some simple analysis may shed light on what our function must do.

Function Analysis. When faced with a new problem, it is often helpful to solve it by hand first. For example, to calculate `power(2,0)`, `power(2,1)`, `power(2,3)` and `power(2,4)` by hand, we might write out expressions for these computations:

power(2,0) = 1
power(2,1) = 2
power(2,2) = 2 * 2
power(2,3) = 2 * 2 * 2
power(2,4) = 2 * 2 * 2 * 2

The first case defines our starting point. The other cases make explicit what we already know: *n* tells us how many times we need to use *x* as factor. How do you know we have enough factors? You count them! Now if we could only count in our programs...

Function Behavior. We can describe what we want to happen as follows:

Our function should receive a base value and an exponent value from the caller of the function. It should initialize *`result`* to one, and then repeatedly multiple *`result`* by the base value, with the number of repetitions being the exponent value. Our function should then return *`result`*.

Function Objects. From our behavioral description, we can identify the following objects:

Description	Type	Kind	Movement	Name
The base value	`double`	varying	received	*`base`*
The exponent value	`int`	varying	received	*`exponent`*
The result value	`double`	varying	returned	*`result`*

From this, we can specify the task of our function as follows:

Specification:

receive: `base`, a `double`; `exponent`, an `int`.
precondition: `exponent` is non-negative
return: `result`, a `double`

While we could worry about negative exponents, we choose not to since it would require a bit more work. We also won't test this precondition since our specification makes it quite clear that we aren't coding for this; we've given other programmers fair notice of our assumption. Our function will simply return 1, as if the exponent were 0.

Using this specification, go to `mathops.h` and `mathops.cpp` and replace the appropriate lines with a prototype and a stub for `power()`. Then, uncomment the call to `power()` within `apply()`. Finally, compile the code and fix your errors.

The call to `power()` in `apply()` uses `int()` to convert the `double` argument *op2* to an integer. Without this conversion, our attempt to pass a `double` argument into an `int` parameter would result in a compilation error.

Function Operations. From our behavioral description, we have the following operations:

Description	Predefined?	Name	Library
Receive `base` and `exponent`	yes	parameter	built-in
Initialize `result` to 1.0	yes	declaration	built-in
Set `result` to `result`*`base`	yes	`*=`	built-in
Repeat `*=` `exponent` times	yes	`for` loop	built-in
Return `result`	yes	`return`	built-in

Function Algorithm. We can organize these operations into the following algorithm:

```
1. Receive base and exponent from caller.
2. Initialize result to 1.0.
3. For each count from 1 to exponent:
      result *= base.
4. Return result.
```

Coding

The C++ `for` loop is designed to facilitate the counting behavior required by step 2. The pattern for the **counting `for` loop** is:

```
for (Type Var = Start; Var <= Stop; Var++)
   Statement
```

where *Type* is a C++ numeric type, *Var* is the loop control variable used to do the counting, *Start* is the first value, and *Stop* is the last value.

For example, if we wanted to print `x` 20 times, here's the algorithm and the code:

1. For each `i` from 1 to 20:
a. Display `x`.
	`for (int i = 1; i <= 20; i++)`
	`  cout << x;`

Compare the algorithm and code, then using the pattern and the example, complete the `power()` function. *Read the algorithm carefully.* There are several variables that you have to create and use in writing your loop, so *read the algorithm carefully*. Remember that you aren't supposed to invent new things with your code; just translate the algorithm above.

When you've written the code, compile and test your program. Test exponentiation on *several* values. Make sure you test 0 as an exponent. Try values larger than 2 and 3 as both bases and exponents.

Characterizing Loops

The pattern for a C++ `for` loop is actually more general:

```
for (InitializationExpr; Condition; StepExpr)
  Statement
```

where *`InitializationExpr`* is any initialization expression, *`Condition`* is any boolean expression, and *`StepExpr`* is an arbitrary expression.

If for some reason we wanted to count *downwards* and output the multiples of 12 from 1200 to 12, then we'd have this algorithm and code:

1. For each *i* from 100 down to 1:
a. Display 12 * `i`.
	`for (int i = 100; i >= 1; i--)`
	`  cout << 12 * i << endl;`

The *`Condition`* controls the loop. As long as it is true when it is tested, then the loop *`Statement`* is executed. C++ `for` loops are controlled by conditions, just as `if` statements are controlled by conditions. As we shall see, each of the other C++ loop statements are also controlled by conditions.

A loop is categorized by when it evaluates its condition:

1. A **pretest loop** evaluates its condition *before* its statements.
2. A **posttest loop** evaluates its condition *after* its statements.
3. An **unrestricted loop** evaluates its condition whenever you like.

The `for` loop is a pretest loop, because it evaluates its condition before the loop's statement is executed. You can prove it to yourself: `power()` should return 1 if the exponent is 0 (which you know since you tested this above). The *only* way this happens is if the loop in `power()` does not execute its statement; if the statement *did* execute, then the value returned wouldn't be 1.

The `for` loop is designed primarily for problems that involve counting through ranges of numbers, or problems in which the number of repetitions can be determined in advance. Many problems need other types of repetition.

The other three C++ loops differ from the `for` loop in that they are **general-purpose loops**, designed for problems where the number of repetitions is not known in advance.

1. The C++ `while` loop provides a general pretest loop.
2. The C++ `do` loop provides a general posttest loop.
3. The C++ forever loop provides a general unrestricted loop.

Let's try out these loops to handle our menu input.

Getting a Valid Menu Choice

`getMenuChoice()` is defined in `menus.cpp`.

```
char getMenuChoice(const string MENU)
{
  cout << MENU;
  char choice;
  cin >> choice;
  return choice;
}
```

What happens if the user types in a bad value? Try it out.

We can be more user-friendly by handling the input errors in this function.

One way to handle such errors is to repeatedly display the menu and input the user's choice, so long as they continue to enter invalid menu choices. This isn't something we can predict since we have *no* idea how many times the user will enter in bad data.

The general-purpose loops give us a way to handle user errors, but we must decide which one to use. We can begin by writing a partial algorithm for this problem using a **generic `Loop`** statement, in which we don't worry (for the moment) how control will leave the loop:

```
1. Loop:
      a. Display MENU.
      b. Read choice.

      End loop.

2. Return choice.
```

We've looked at how conditions are used in `if` and `for` statements. The other loops work quite similar to the `for` loop. There are two types of conditions we have to work with.

- A **continuation condition** defines the circumstances under which we want repetition to continue. "When do I keep on going?"
- A **termination condition** defines the circumstances under which repetition should terminate. "When do I stop?"

The loop we use will determine what type of condition we need. When we switch from one loop to another, we might have to switch the type of condition. Rethinking the logic isn't too tricky.

For our problem, we want the repetition

- to continue so long as `choice` is an invalid menu choice, and
- to terminate when `choice` is a valid menu choice.

Since `choice` is not known until *after* Step (b), it seems logical to see if we're done *after* that step, which we can describe using our termination condition as follows:

```
1. Loop:
      a. Display MENU.
      b. Read choice.
      c. If choice is a valid menu choice, exit the loop.

      End loop.

2. Return choice.
```

In this algorithm, it is apparent that the controlling condition is evaluated at the bottom of the loop, which implies that a post-test loop is the appropriate loop to choose.

In C++, the post-test loop is called the **`do` loop**, and its pattern is as follows:

```
do
{
  Statements
}
while ( Condition );
```

When execution reaches such a loop, the following actions occur:

1. `Statements` execute.
2. `Condition` is evaluated.
3. If `Condition` is true, control returns to step 1; otherwise, control proceeds to the next statement after the loop.

Since repetition continues so long as the condition is true, a `do` loop uses a continuation condition.

Since its `Condition` is not evaluated until after the first execution of `Statements`, the `do` loop guarantees that `Statements` will be executed one or more times. For this reason, the `do` loop is said to exhibit **one-trip behavior**: we must make at least one trip through *`Statements`*.

The notion of a "valid" menu choice is a bit tricky. One way to handle it is to require that the valid menu choices be consecutive letters of the alphabet (e.g., `'a'` through `'e'`). If we then pass the first and last valid choices to `getMenuChoice()` as arguments:

```
char operation = getMenuChoice(MENU, 'a', 'e');
```

and add parameters to the function prototype and definition to store these arguments:

```
char getMenuChoice(const string MENU, char firstChoice, char
lastChoice);
```

Add the new arguments to the function call in the driver. Add the new parameters to the prototype and the function itself. You can recompile the code to make sure the compiler is happy with your changes, but the program won't run any differently yet.

Since `firstChoice` and `lastChoice` will define the range of valid choices, we can express our loop's continuation condition in terms of those values: `choice` is invalid if and only if

```
choice < firstChoice || choice > lastChoice
```

We are then ready to add a `do` loop to the function to implement our algorithm.

Code it up. Here are the pieces we need for the `do` loop:

- The *`Statements`* are the output of `MENU` and the input of `choice`.
- The *`Condition`* is in the previous paragraph.
- We'll have to declare `choice` outside the loop so that it can be returned outside the loop.

Compile and test your code.

Getting Valid Numeric Input

Another potential source of error occurs when our program reads values for `op1` and `op2` from the keyboard -- the user might enter some non-numeric value. Our current version checks for this using the `assert()` mechanism and the `good()` method of `cin`.

But suppose we wished to give the user another chance to enter valid values, instead of just terminating the program (which seems terribly drastic)? At first glance, it looks like we could use the same approach as before, by replacing the `assert()` with a posttest loop that repeats the steps so long as `good()` returns false:

```
do
{
// prompt for op1 and op2
// input op1 and op2
}
while ( !cin.good() );
```

However, this approach is inappropriate because of two subtleties about `cin` input (actually, `istream` input in general). When the >> operator is expecting a real value, but gets a non-real value, two things happen, each of which causes a difficulty:

1. The *internal* status of `cin` is set so that its method `good()` will return false. *No input operations can be performed with* `cin` *so long as* `cin.good()` *returns false.*
2. After failed input, the bad input value is left (unread) in the input stream.

The first problem can be fixed with `cin.clear()`, which resets the status of `cin` so that `cin.good()` returns true again.

The second problem takes some more work. We don't want the loop to continue reading the same bad value over and over and over again. We need some way to skip over the bad input. This can be accomplished using the `ignore()` method:

```
cin.ignore( NumChars, UntilChar );
```
Characters in `cin` will be skipped until *NumChars* have been skipped or until the character *UntilChar* is encountered, whichever comes first. Since lines are usually not more than 120 characters in length, and the end of a line of input is marked by the newline character (`'\n'`), we can use the following call to solve this difficulty:
```
cin.ignore(120, '\n');
```

All of this complicates our choice of which loop to use, because we now have four steps to perform:

a. Display a prompt for two real values. [One or more times.]
b. Input *op1* and *op2*. [One or more times.]
c. `cin.clear();` [Only if necessary, zero or more times.]
d. `cin,ignore(120, '\n');` [Only if necessary, zero or more times.]

It looks like the test should be placed in the middle there. A `do` loop isn't going to work.

One traditional solution is modify our algorithm to use a pre-test loop in the following way:

```
a. Display a prompt for two real values.
b. Input op1 and op2.
c. Loop so long as cin.good() is false:
     i.   cin.clear();
     ii.  cin.ignore(120, '\n');
     iii. Display a prompt for two real values.
     iv.  Input op1 and op2.

   End loop.
```

The drawback to this approach is its redundancy: the prompt and input steps are written twice. This is not too much of an inefficiency, so long as one does not mind the extra typing. In the final part of this exercise, we will see a way to avoid this redundancy.

The C++ pretest loop is called the `while` loop:

```
while ( Condition )
   Statement
```

As usual, *`Condition`* is any C++ Boolean expression, and *`Statement`* can be either a single or compound C++ statement. *`Statement`* is often referred to as the **body** of the loop.

This works just like a `do` loop except that *`Condition`* is evaluated before the *`Statement`* is evaluated. If *`Condition`* is false right away, *`Statement`* is never executed. For this reason, a `while` loop is said to exhibit **zero-trip behavior**: we might make zero trips through *`Statement`*.

Add a `while` loop to encode our modified algorithm. The condition `!cin.good()` can be used to control the `while` loop. Since the `while` loop must repeat multiple statements, its *`Statement`* must be a compound statement or bad things will result.

Compile and thoroughly test what you have written. Continue when what you have written makes the entry of numeric values fool-proof.

One Execution, Multiple Calculations

The algorithm that our program is using is basically as follows:

```
1. Display a greeting.
2. Display a menu of operations and read operation, guaranteed to be a valid menu
   operation.
3. Display a prompt for two real values.
4. Read op1 and op2.
5. Loop (pretest), so long as a real value was not entered:
      a. Clear the status of cin.
      b. Skip the invalid input.
      c. Display a prompt for two real values.
      d. Input Op1 and Op2.

         End loop.

6. Compute result by applying operation to op1 and op2.
7. Display result.
```

Using this algorithm, we have to re-run our program for every calculation, which is inconvenient for the user. A more convenient calculator would wrap some of the statements in a loop, so that the user could perform multiple calculations without having to re-run the program.

To add the loop, we modify our algorithm:

```
1. Display a greeting.
2. Loop:
      a. Display a menu of operations and read operation, guaranteed to be a valid
         menu operation.
      b. Display a prompt for two real values.
```

```
        c. Read op1 and op2.
        d. Loop so long as a real value was not entered:
             i. Clear the status of cin.
            ii. Skip the invalid input.
           iii. Display a prompt for two real values.
            iv. Input op1 and op2.

                End loop.

        e. Compute result by applying operation to op1 and op2.
        f. Display result.

           End loop.
```

In order to determine which kind of loop to use, we must determine where to evaluate the loop's termination condition, which we might express as "the user wants to quit".

One way to have the user indicate that they want to quit is to view quitting as an operation, and provide an additional menu choice (i.e., `'f'`) by which the user can indicate that they want to quit. The condition `(operation == 'f')` evaluates to true if the user wishes to quit.

So when do we check this condition? The ideal place is to evaluate it *as soon as it can be known*; in this case this would be immediately following the input of `operation`. We thus have a situation where the loop's condition should not be evaluated at the loop's beginning nor at its end. It must be evaluated in the middle of the loop.

```
1. Display a greeting.
2. Loop:
        a. Display a menu of operations and read operation, guaranteed to be a valid
           menu operation.
        b. If operation is quit, exit the loop.
        c. Display a prompt for two real values.
        d. Read op1 and op2.
        e. Loop so long as a real value was not entered:
             i. Clear the status of cin.
            ii. Skip the invalid input.
           iii. Display a prompt for two real values.
            iv. Input op1 and op2.

                End loop.

        f. Compute result by applying operation to op1 and op2.
        g. Display result.

           End loop.
```

In C++, the unrestricted loop is a simplification of the `for` loop that we call the **forever loop**, that has the following pattern:

```
for (;;)
{
  StatementList1
  if ( Condition ) break;
  StatementList2
}
```
By leaving out the three expressions that normally control a `for` loop, it becomes an **infinite loop**. Of course, we do not want an infinite number of repetitions, but instead we want execution to leave the loop when a termination condition becomes true. As shown in the pattern above, this can be accomplished by placing an `if` statement that uses a termination condition to control a `break` statement within the body of the loop.

When a `break` statement is executed, execution is immediately transferred out of the forever loop to the first statement following the loop. By only selecting the `break` statement when *`Condition`* is true, a forever loop's repetition can be controlled in a manner similar to the other general-purpose loops.

Modify your source program to incorporate this approach, and then translate and test the newly added statements.

This loop can also be used for the "valid numeric input" loop that we wrote before. Give it a try.

Submit

Turn in your code and sample runs of your program to demonstrate that it does everything we added to it this week. We added a *lot* of things, including some error handling, so make sure you test it fully.

Terminology

condition, continuation condition, counting `for` loop, `do` loop, forever loop, general-purpose loop, generic loop, infinite loop, loop body, one-trip behavior, posttest loop, pretestloop, repeated, selective execution, termination condition, unrestricted loop, zero-trip behavior

3. Project 7

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #7.1: Write a program that will read a sequence of numbers from the keyboard, and display the minimum, maximum, average, and range of the entered values. Make the input step "fool-proof".

Project #7.2: Extend `calculate.cpp` into a six-function calculator, as follows:

1. Add a `factorial()` operation that, given an integer n, computes n! = 1 * 2 * ... * (n-1) * n.
2. Redesign `Power()` so that it handles negative exponents.

Project #7.3: Write (at least) two ASCII graphics functions:

1. `PrintStripe(`*`n`*`, `*`ch`*`);` that displays *n* consecutive *ch* characters.
   (e.g., `PrintStripe(5, 'X');` should display `XXXXX`.)
2. `PrintAlternating(`*`n`*`, `*`ch1`*`, `*`ch2`*`);` that displays *n* consecutive pairs of the characters *ch1ch2*. (e.g., `PrintAlternating(3, 'X', 'Y');` should display `XYXYXY`.)

Use these functions to write a program that draws a picture, such as a flag. For example, you might use these functions to draw a crude facsimile of the U.S. flag that looks like this:

```
* * * * * * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * * * * *
* * * * * * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * * * * *
* * * * * * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * * * * *
* * * * * * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * * * * *
* * * * * * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Try and draw your picture efficiently (i.e., using loops to minimize the number of statements). Feel free to create additional "graphics" functions.

Project #7.4: Build a "police sketch artist" program as described in Project #6.2, but write a dynamic program that lets the user experiment with different combinations of facial parts in a single execution. Control the program using a loop. Start with a 'blank' face. Allow the user to modify this face using a two-level hierarchical menu, with the first level allowing the user to select which facial part they want to modify (i.e., a menu of menus), and the second level allowing the user to select from among the choices for that particular facial part.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 8: Parameter-Passing and Scope

1. Objective of the Experiment

- To gain experience using reference parameters.
- To practice writing functions.
- To understand the rules of scope.

2. Theoretical Background

Introduction

This lab's exercise involves a series of experiments that investigate some of the features and restrictions of C++ functions that we have glossed over until now. The experiments are divided into two categories:

1. **Parameters.** In the first part, we will begin by exploring the nature of function parameters, and what rules govern the relationship between parameters and their arguments. We will also examine the different kinds of parameters available in C++.
2. **Scope.** In the second part, we will examine the relationship between definitions that appear in different functions, and some of the **rules of scope** in C++.

There is a different page for each experiment which requires you to write out answers for your work. Write, or better yet type, your answers up and hand them in to your instructor. Your instructor may also want you to hand in your program after each experiment.

Experiments

Each experiment for this lab starts out with an issue and a hypothesis about the issue. Your work will be to prove or disprove the hypothesis.

Did you catch that, "prove *or disprove*"? *The hypothesis might be **wrong***.

Files

Directory: `lab8`

- `params.cpp` is our playground for the experiments in this lab exercise.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

a) Part I: Parameter-Passing Mechanisms

The general form of a C++ function heading is:

ReturnType *Name* (*ParameterDeclarationList*)
where *ParameterDeclarationList* is an optional sequence of one or more *ParameterDeclarations* separated by commas, each of which has the form:
Type *ParameterName*
where *Type* is a valid type, and *ParameterName* is a valid identifier.

We've used this in both the prototype and definition of our functions.

Terminology

1. A **value parameter** is a parameter whose *Type* *is not* followed by an ampersand (`&`). A value parameter is a variable that is local to the function; when the function is called, it receives a *copy* of the value of the corresponding argument.
2. A **reference parameter** is parameter whose *Type* *is* followed by an ampersand (`&`). (Technically, the ampersand is *part* of the type.) A reference parameter is an *alias* (e.g., another name) for its corresponding argument.

Question #8.1: We have used parameters in several of our programs. In every case, which kind did we use?

Using your editor, take a moment to look over the program in `params.cpp`. Its behavior consists of 3 steps:

1. Initialize a set of variables to some initial value (-1 in this case).
2. Call function `change()` that *tries* to modify the values of those variables.
3. Output the values of those variables to view the effects of function `change()`.

The function `change()` is the "laboratory" in which we will experiment with parameters. The key word above is "tries". `change()` *tries* to alter the variables in `main()` through its parameters, but it's not always successful.

b) Part 2: Scope in C++

A declaration can be thought of as giving the compiler a meaning for the name being declared.

Think about this for a moment. In Experiment #3, we saw that the name `arg1` can be declared twice: once as an integer variable in the main program, and a second time as an integer value parameter in `change()`. We also saw that when `arg1` is a value parameter, altering it in `change()` leaves the value of `arg1` in the main program unchanged. We might conclude from this that `arg1` in `main` and `arg1` in `change()` are two different variables.

Put differently, *the same name can have different meanings in different places in a program.*

There is nothing that prevents us from declaring any variable as an integer in `main` and then redeclaring that variable with the same name as a real number in `change()`. The same name will have one meaning when execution is in `main`, and an different meaning when execution is in `change()`.

Keep in mind that a compiler *hates* ambiguity. It cannot tolerant confusion. So there *must* be some underlying rules that keep the two declarations separate. These rules are known as the "rules of scope".

Definition: The set of all places in a program where a name has a particular meaning is called the **scope** of that name.

The (Simplified) C++ Rules of Scope

The basic rules governing the scope of C++ names can be summarized as follows:

1. **Local block scope.** The scope of any name declared within a block starts at its declaration and ends at the close-brace (i.e., `}`) of the block. The variable is defined in any of the code that *appears in the program text* between its declaration and the closing brace (including nested compound statements). These variables are called "local" because they automatically come into existence whenever execution enters the surrounding braces.
2. **Local parameter scope.** The scope of a parameter starts at its declaration and ends at the function's close brace. Parameters are also local because their scope covers only a local area of the code.
3. **`for`-loop scope.** A variable declared in the initialization expression of a `for` loop is local to just the `for` loop.
4. **Non-local scope.** There are many other types of scope that we'll just label as "non-local". In particular, the scope of a variable declared outside any curly-brace blocks lasts from the variable's declaration to the end of the file.

C++ classes have their own rules of scope, which we will examine in a later exercise.

A variable's scope determines when it's valid. So if we try to access a variable outside its scope, the access is invalid, and the compiler will complain.

3. Sub-Experiments

a) Sub-Experiment 1

Variable Parameters and Identical Argument Names

The Issue: *If a value parameter has the same name as its corresponding argument, will altering the parameter alter the corresponding argument?*

Hypothesis: You try this one:

Question #8.3.1: What hypothesis do you want to try to prove? Will the same names make a difference or not?

The Experiment: Modify the definition of `change()`, so that its parameter-names match the argument names:

```
void change(int arg1, int arg2, int arg3)
{
    arg1 = 1;
    arg2 = 2;
    arg3 = 3;
}
```

This function is already called in the main program with the statement:

```
change(arg1, arg2, arg3);
```

If giving parameters the same names as their arguments is a problem, compiling `params.cpp` will produce errors. If the names matter, then we might see a change in the values in the main program.

Observation: Compile and execute your program.

Question #8.3.2: Were you able to compile your program?
Question #8.3.3: If not, what error message did you get? If you could compile, what did the program print when you ran it?

Conclusions:

Question #8.3.4: Is your hypothesis correct? How do you know? If it's not correct, what should it be?

Undo your modifications and then continue.

b) Sub-Experiment 2

Using a Twice-Declared Identifier in Outer Block

Issue: *Suppose an identifier has two different declarations, one in an outer block and one in a nested inner block. If the name is accessed within the outer block, but after the inner block, which declaration is used?*

Hypothesis: The closest definition will be used.

Proximity seemed to work out fine in the last experiment, so why not here?

The Experiment: If we move the statement that displays `arg1`, we can get an answer to our question:

```
int main()
{
  int arg1;
  arg1 = -1;
  ...
  {
    char arg1 = 'A';
  }
  cout << arg1 << endl;
}
```

Question #8.3.5: If `-1` is printed, what does this mean? If `A` is printed, what does this mean?

Observation:

Question #8.3.6: What does the program display?

Conclusions:

Question #8.3.7: Is our hypothesis correct? How do you know? If it's not correct, what should it be?

This calls into question the "proximity" rule of thumb. It's *not* a matter of which declaration is seen last. Apparently that one close curly brace makes the difference.

Question #8.3.8: If proximity does not matter for resolving duplication declarations, what *does*?

Hint: look at the nesting of the curly braces. Consult the scoping rules for more help.

Submit

Turn in the answers to the questions of the experiments that you run. Your instructor may also want you to turn in a copy of your program after each experiment.

Terminology

`for`-loop scope, local block scope, local parameter scope, non-local scope, parameter, reference parameter, rules of scope, scope, scope, value parameter

3. Project 8

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #8.1: White water rapids are classified by their *gradient* (the number of feet per mile they descend) and their *class* (a difficulty rating in the range 1-6, with 1 being *calm* and 6 being *unnavigable*.) Popular white water rafting trips in Pennsylvania and West Virginia include the lower Youghiogheny (gradient 15, class 3), the New (gradient 25, class 4), the Cheat (gradient 35, class 4), and the Upper Youghiogheny (gradient 125, class 5).

Write a menu-drive program that displays a menu of the trips, and displays the gradient and class of whatever trip the user selects. Your program should include a function that, given the trip selected by the user, passes back the gradient and class of that trip.

Project #8.2: Write a function `Sort3()` that, given three integer arguments *`int1`*, *`int2`*, and *`int3`*, changes the values of those arguments so that their values are in ascending order (i.e., *`int1`* <= *`int2`* <= *`int3`*---looks like a postcondition!). Write a driver program that demonstrates the correctness of `Sort3()`.

Project #8.3: Your local painting supplies store would like a paint-mixing "expert system." Write a function that, given a non-primary color (i.e., orange, green, purple) will pass back the two primary colors that must be mixed to produce that color (i.e., red and yellow must be mixed to produce orange, yellow and blue to produce green, and blue and red to produce purple). Use a `string` object to store a color. Then write a menu-driven program that allows its user to enter, process and display the solution of as many paint-mixing problems as the user wishes.

Project #8.4: A quadratic equation is an equation of the form
$ax^2 + bx + c = 0$
Write a function that, given the `a`, `b` and `c` values defining a quadratic, passes back the two roots of that quadratic. To compute the roots, use the quadratic formula:

```
         -b ± sqrt(b² - 4ac)
roots = -------------------------
                   2a
```

Your function should check for common errors, such as `a == 0` and $b^2 - 4ac$ being negative.

Create a user-friendly driver program that allows its user to compute the roots of an arbitrary number of quadratic equations.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 9: Files and Streams

1. Objective of the Experiment

- To learn more about iostream objects.
- To learn the fundamentals of file input and output.
- To learn about simple encryption techniques.

2. Theoretical Background

Introduction

Throughout this lab manual, we have made extensive use of **files**---containers on a hard or floppy disk that can be used to store information for long periods of time. Each source program that we have written has been stored in a file, and each binary executable program has also been stored in a file.

Files differ from programs in that a program is a sequence of instructions, and a file is a container in which data (like program code) can be stored. For example, the word processing documents that you work on are saved as files. Those files look quite different from the files that store your programs and from the files that store your executables.

But this begs the question, why can't we read and write data from and to a file like a word processor? This would be particularly useful for problems where the amount of data to be processed is so large that interactively entering the data each time the program is executed becomes inconvenient. If we could only store our data in a file, then we could test our program many times over without having to retype the data each time.

Let's consider a simple problem.

When many of us were younger, we enjoyed writing secret messages, in which messages were encoded in such a way as to prevent others from reading them, unless they were in possession of a secret that enabled them to decode the message. Coded messages of this sort have a long history. For example, the **Caesar cipher** (invented, it is said, by Julius Caesar himself) is a simple way to encoding messages.

For example, consider this message and its encoding:

Message	Encoded
`One if by land, two if by sea.`	`Rqh li eb odqg, wzr li eb vhd.`

What is the relationship between the letters in the original sentence and those in the encoded sentence? Hint: compare the "difference" between the corresponding characters in the two sentences.

This lab's exercise is to use the Caesar cipher to encode and decode messages stored in files.

Files

Directory: `lab9`

- `caesar.h`, `caesar.cpp`, and `caesar.doc` implement a Caesar encryption function.
- `encode.cpp` and `decode.cpp` are the two drivers needed for this lab exercise.
- `message.text` and `alice.code` are two sample input files.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

An Encoding Program

The first part of this exercise is to write a program that can be used to encode a message that is stored in a file. The encoded message will then be saved to a second file.

Design

As usual, we will apply object-centered design to solve this problem.

Behavior

Our program should display a greeting and then prompt for and read the name of the input file. It should then connect an input stream to that file so that we can read from it, and check that the stream opened correctly. It should then prompt for and read the name of the output file. It should then connect an output stream to that file so that we can write to it, and check that the stream opened correctly. For each character in the input file, our program should read the character, encode it using the Caesar cipher, and output the encoded character to the output file. Our program should conclude by disconnecting the streams from the files.

This behavior is a bit verbose since we're just learning about files, but it's perhaps better to err on the verbose side of things rather than on the forgetting side of things. (Programs tend to crash when you forget to do things.)

Objects. Using this behavioral description, we can identify the following objects:

Description	Type	Kind	Name
a greeting	`string`	constant	---
The name of the input file	`string`	varying	*`inFile`*
An input stream	`ifstream`	varying	*`inStream`*
The name of the output file	`string`	varying	*`outFile`*
An output stream	`ofstream`	varying	*`outStream`*
a character from the input file	`char`	varying	*`inChar`*

an encoded character	`char`	varying	*`outChar`*

Using this list of objects, here's our specification:

Specification:
input (`inFile`): a sequence of unencoded characters.
output (`outFile`): a sequence of encoded characters.

This list of objects raises an important question that you should think about:

Question #9.1: What is the difference between a *file name* and a *file stream*?

One immediate hint: consider the data types. The data type determines the operations you can perform on an object. You should answer to this question *after* you've read through this section. It's an important distinction that, if you understand the difference, you'll save yourself much heartache when writing and debugging your programs.

Operations. From our behavioral description, we have these operations:

Description	Predefined?	Name	Library
Display a `string`	yes	<<	`iostream`
Read a `string`	yes	>>	`iostream`
Connect an input stream to a file	yes	`ifstream` declaration	`fstream`
Connect an output stream to a file	yes	`ofstream` declaration	`fstream`
Check...	yes	`assert()`	`cassert`
...that a stream opened properly	yes	`is_open()`	`fstream`
Read a `char` from an input stream	yes	`get()`	`fstream`
Encode a `char` using the Caesar cipher	yes	`caesarEncode()`	caesar
Write a `char` to an output stream	yes	<<	`fstream`
Repeat input, encoding, and output operations	yes	input loop	built-in
Determine when all `char`s have been read	yes	`eof()`	`fstream`
Disconnect a stream from a file	yes	`close()`	`fstream`

Algorithm. We can organize these operations into the following algorithm:

1. Display a greeting.
2. Prompt for and read *`inFile`*, the name of the input file.
3. Create an `ifstream` named *`inStream`* connecting our program to *`inFile`*.
4. Check that *`inStream`* opened correctly.
5. Prompt for and read *`outFile`*, the name of the output file.
6. Create an *`ofstream`* named *`outStream`* connecting our program to *`outFile`*.
7. Check that *`outStream`* opened correctly.

8. Loop through the following steps:
    a. Read a character from the input file.
    b. If the end-of-file was reached, then terminate repetition.
    c. Encode the character.
    d. Write the encoded character to the output file.

    End loop.

9. Close the input and output connections.
10. Display a "successful completion" message.

Coding

This algorithm should be encoded in `main()` in `encode.cpp`.

- Steps 1, 2, and 5 are very familiar. Print a prompt, read in a value. The values are only `string`s, so nothing new yet.
- The loop is tested in the middle, so we'll use a forever loop (i.e., `for(;;)...`) with an `if-break` to stop the loop.
- Encoding can be done with `caesarEncode()` from the `caesar` library.

Write the code for these steps. The `if-break` statement can wait, but the other steps listed here are straightforward.

We only have to figure out the file I/O steps.

Opening a Connection to a File. When we want to get input from a file, we have tell the compiler that's what we want. It's a fairly expensive operation (since data moves *much* slower to and from a disk than to and from a computer's main memory). We also have to be precise about what file we want. We certainly don't want *all* files on the machine.

So we need to open a connection between the program and a file. A connection is a thing, and all things in C++ are represented as objects. File connections are known as **streams**, and there are two types of streams: `ifstream` for input file streams and `ofstream` for output file streams.

Like any other object, a stream must be declared before it can be used. If *inputFileName* is a `string` object containing the name of an input file, then the declaration

```
ifstream inFileStream(inputFileName.data());
```

constructs a stream object named *inFileStream* as a connection to the file.
The `string` method `data()` extracts the actual characters from a `string`. If your compiler is not fully ANSI compliant, you may have to use the `c_str()` method instead. The stream classes are a bit particular about the strings that they'll accept.

If the file does not exist, bad things happen. More on this later.

Using this information, implement the step of our algorithm that creates and opens a connection `inStream` to the input file named `inFile`.

An output stream is similar:

```
ofstream outFileStream(outputFileName.data());
```
This declaration constructs an object named `outFileStream` as a connection to the file named `outputFileName`.

If the file does not exist, then a file by that name is created in the working directory. If the file does exist, then its contents are *erased*. An `ofstream` thus provides a connection to a file so that we write data to the file.

Using this information, implement the appropriate step of our algorithm by declaring an `ofstream` named `outStream` that serves as a connection between our program and the file whose name is in `outFile`.

Libraries. Try compiling your code. Ooops. You should get complaints about `ifstream` and `ofstream`. The answer is in the object chart above: you haven't included the proper library.

Include the proper library for these identifiers, and then compile your code. *Don't run your code yet* because your loop doesn't have a termination test.

Checking that a Connection Opened Correctly. Opening files is an operation that is highly susceptible to user errors. Suppose the user has accidentally deleted the input file and our program tries to open a connection to it? What if it never existed in the first place! If an `fstream` opens as expected, the operation is said to **succeed**, but if it does not open as expected, the operation is said to **fail**.

To detect the success of an open operation, fstream objects contain an `is_open()` method:

```
fileStream.is_open()
```
which returns true if *fileStream* is open, and it returns false otherwise.

In an `assert()`, the `is_open()` method provides a readable way to perform the checking steps of our algorithm.

Write the code for these steps. Compile (but again *don't* execute) your program.

Input from an `ifstream`. The most important thing about input (and output) is that you already know how to do it:

Helpful hint: *File I/O is done the same way a screen and keyboard I/O.*

Just as we have used the >> operator is used to read data from the `istream` named `cin`, the >> operator can be used to read data from an `ifstream` opened for input. Since the `ifstream` connects a file to a program, applying >> to it transfers data from the file to the program. For this reason, this operation is described as **reading** from a file, even though we are actually operating on the `ifstream`. An expression of the form:

```
inputFileStream >> VariableName
```

thus serves to read values from an `ifstream` named *`inputFileStream`* into the variable *`VariableName`*. The type of the value being read must match the type of *`VariableName`*, or the operation will fail.

However, while the input operator is the appropriate operator to solve many problems involving file input, it is *not* the appropriate operator for our problem. The reason is that the >> operator *skips leading whitespace characters*. That is, if our input were

```
  One if by land.
  Two if by sea.
```

and we were to use the >> operator (in a loop) to read each of these characters:

```
inStream >> inChar;
```

then all whitespace characters (blanks, tabs and newlines) would be skipped, so that only non-whitespace characters would be processed, as if the file contained

```
Oneifbyland.Twoifbysea.
```

To avoid this problem, `ifstream` objects contain a `get()` method:

```
inputFileStream.get( CharacterVariable );
```

When execution reaches this statement, the next character, *including whitespace characters*, is read from *`inputFileStream`* and stored in *`CharacterVariable`*.

Use the `get()` method of the `inStream` object to perform the `char` input in the loop. Then compile your program, and continue when your program compiles without error. *Don't run it yet!*

Controlling a File-Input Loop. Files are created by a computer's operating system. When the operating system creates a file, it marks the end of the file with a special end-of-file mark. Input operations are then implemented in such a way as to prevent them from reading beyond the end-of-file mark, since doing so could allow a programmer unauthorized access to the files of another programmer. The input operations will just keep you at the end-of-file forever until you realize where you are.

An `ifstream` object has a method named `eof()` that can be used to control an input loop:

```
inputFileStream.eof()
```

This expression returns true if the last read from *`inputFileStream`* tried to read the end-of-file mark, and it returns false otherwise. We have to read *first*, *then* test for end-of-file.

In a forever loop like the one in the source program, the `eof()` method can be used as our termination test. By placing an if-break combination:

```
if ( /* end-of-file has been reached */ ) break;
```

following the input step, repetition will be terminated when all of the data in the file has been processed.

In your source program, place an `if-break` in the appropriate place in our algorithm, using the `eof()` method of `inStream` as the condition in the if statement. Then compile your source program, to check the syntax of what you have written. When it is syntactically correct, continue to the next part of the exercise. You probably could run the program now, but it won't do anything interesting because it's not generating any output.

File Output. Just as we have used the << operator to write data to the `ostream` named `cout`, the << operator can be used to write data to an `ofstream` opened for output. Since the `ofstream` connects a program to a file, applying << to it transfers data from the program to the file. This operation is thus described as **writing** to the file, even though it is an `ofstream` operation.

The pattern for output should look pretty familiar:

```
outputFileStream << Value ;
```

outputFileStream is an `ofstream`, and *Value* is value that should be written in the file.

Use this example as a basis for a statement to finish up the loop, writing the encoded character (*not* the original!) to the output file. Compile your program to test the syntax of what you have written, and fix all of your compilation errors.

Closing Files. Once we are done using an stream to read from or write to a file, we should **close** it, to break the connection between our program and the file. This is accomplished using the method `close()`. Both the `ifstream` and `ofstream` classes have this method:

```
fileStream.close();
```

When execution reaches this statement, the program severs its connection to *fileStream*.

In the appropriate place in the source program, place calls to `close()` on the input stream and on the output stream. Then compile your program, and ensure that it is free of syntax errors.

Testing and Debugging

When your program's syntax is correct, test it using the provided file named `message.text`. If what you have written is correct, your program should create an output file, containing the output:

```
Rqh Li Eb Odqg
Wzr Li Eb Vhd
```

If this file is not produced, then your program contains a logical error. Retrace your steps, comparing the statements in your source program to those described in the preceding parts of the exercise, until you find your error. Or, pretend you're the computer and walk through your program. Correct your program, recompile it, and retest your program until it performs correctly.

Applying What We Have Learned

The last part of this exercise is for you to apply what you have learned to the problem of **decoding** a file encoded using the Caesar cipher. Complete the skeleton program `decode.cpp`, that can be used to decode a message encoded using the Caesar cipher.

Do all that is necessary to get this program operational, so that messages encoded with `encode.cpp` can be decoded with `decode.cpp`. Put differently, the two programs should complement one another.

The difficult part has been done for you. The `caesar` library contains a `caesarDecode()` function which does all the work of decoding. Your job is to build the driver to handle file I/O.

To test your program, you can use the output file created by `encode.cpp`, or `alice.code`, a selection from Lewis Carroll's *Alice In WonderLand*.

Beware! Watch the names of your output files. Every application you've used has probably warned you when you're about to overwrite an existing file. *This is behavior that the program had to implement.* You *haven't* implemented it in this program, so you won't get this warning. So if you encode `message.text` to be `message.code`, and then you decode `message.code` to be `message.text`, then say goodbye to the old `message.text`! The old version will disappear, and you'll have to copy it over again. It's better to use `message.decode`, perhaps, when you decode `message.code`.

Submit

Turn in your code as well as the output from your programs.

Terminology

Caesar cipher, decode, file, reading (from a file), stream, writing (to a file)

3. Project 9

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #9.1: Write a program that reads from a file of real numbers, and displays the *minimum*, *maximum*, *average*, and *range* of the numbers in the file. The user should be able to enter the name of the input file from the keyboard.

Project #9.2: Write a program that reads the contents of a file and creates an exact copy of the file, except that each line is numbered. For example, suppose the input file contains the following text:

```
'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.
                        - Lewis Carroll
```

Then the output file should appear something like this:

```
1: 'Twas brillig, and the slithy toves
2:    Did gyre and gimble in the wabe;
3: All mimsy were the borogoves,
4:    And the mome raths outgrabe.
5:                         - Lewis Carroll
```

The user should be able to enter the names of the input and output files from the keyboard.

Project #9.3: Using the `PigLatin()` function we wrote in Exercise #4, write a program that reads a file of words and translates each word in that file into Pig Latin. The user should be able enter the names of the input and output files from the keyboard.

Project #9.4: Write a text-analysis program that reads an essay or composition stored in a text file, and determines the number of words, the number of sentences, the average number of words per sentence, the average number of letters per word, and a *complexity* rating, using:

```
complexity = 0.5 * averageSentenceLength + 0.5 *
averageWordLength
```

Based on these calculations, your program should assess the writing level of the essay as:

- Grammar School if complexity < 6.
- Junior High if 6 <= complexity < 7.
- High School if 7 <= Complexity < 8.
- College if 8 <= complexity < 9.
- Graduate if complexity >= 9.

The user should be able to enter the name of the input file from the keyboard.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 10: Vectors

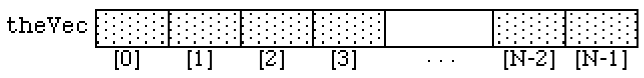
1. Objective of the Experiment

- To learn to use and process sequences of objects.
- To learn how to define and use vectors.
- To implement some common vector operations.

2. Theoretical Background

Introduction

In past exercises, we have dealt with **sequences** of values by processing the values one at a time. Now we want to examine a new kind of object called a **vector** that can store not just one value, but an entire sequence of values. Like a `string`, a vector is a **subscripted object**, meaning the values within it can be accessed via a subscript or index:



The components or "spaces" within a vector in which values can be stored are called the vector's **elements**. Unlike `string`s which only store `char`s, `vector`s can be used to store any type of data.

By storing a sequence of values in a vector, we can design operations for the sequence, and then implement such operations in a function that receives the entire sequence through a vector parameter.

Vectors are implemented for us in C++ as the `vector` class in the `vector` library which is part of the C++ **Standard Template Library** (**STL**). The STL contains many containers and algorithms that are useful in many different situations. We'll only explore a very small portion of the STL in this lab.

The Problem

This lab's exercise is to write a program that processes the names and scores of students in a course, and assigns letter grades (A, B, C, D or F) based on those scores, curving the grades. The format of the input file is a series of lines, each having the form:

`name` `score`

As usual, we will use object-centered design to design our solution.

Files

Directory: `lab10`

- `DoubleVectorOps.h`, `DoubleVectorOps.cpp`, and `DoubleVectorOps.doc` implement various operations for vectors of `double`s.
- `grades.cpp` implements a skeleton driver.
- `scores1.data`, `scores2.data`, and `scores3.data` are sample input files.
- `Makefile` is a makefile.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

Begin by editing the file `grades.cpp`, and take a few moments to look it over. As indicated by its documentation, `grades.cpp` is a skeleton program that (partially) encodes an algorithm to solve the problem. This lab's exercise will involve completing this program.

Design

Up until now, our programs have been able to fully process their input as soon as it was read in. This is actually quite rare. Often, as in this lab's problem, we need to keep our information around for a long time.

Let's start at the end product: to assign a letter grade, we need to know the student's score. That's fine: read in the score, and print the corresponding letter grade. However, how can we assign the letter grade? We first need to know the average and standard deviation of all of the scores; the only way to know this is to sum *all* of the scores, and then we can compute the average and standard deviation.

Consequently, the scores have to be processed *twice*: once to figure out the average and standard deviation; a second time to assign letter grades based on the average and standard deviation.

We could use a file to store the sequence, and then reread the file each time we need to process the values. The problem with this approach is that reading from a file is very, *very*, ***very*** slow compared to reading from main memory---a thousand times as slow!

A *much* better solution is to use an internal container, in this case, a `vector`. In addition to being *much* faster than input from a file, a `vector` gives us immediate access to any element stored in the `vector`.

Behavior

Our program should display a greeting, and then prompt for and read the name of an input file. It should then read a sequence of names and a sequence of scores from the input file. It should then compute the average and standard deviation of the sequence of scores, and display these values. Using the average and standard deviation, it should compute the sequence of letter grades that correspond to the sequence of scores. It should then display each student's name, score and letter grade.

Objects

When we examine our behavioral description for objects, we find these objects:

Description	Type	Kind	Name
a greeting	`string`	literal	--
The name of the input file	`string`	varying	`inFileName`
a sequence of names	`vector<string>`	varying	`nameVec`
a sequence of scores	`vector<double>`	varying	`scoreVec`
the average score	`double`	varying	`average`
the standard deviation of the scores	`double`	varying	`stdDeviation`
a sequence of letter grades	`vector<char>`	varying	`gradeVec`

We can thus specify the behavior of our program this way:

Specification:
input (keyboard): the name of an input file.
input (input file): a sequence of names and scores.
output (screen): the average and standard deviation of the scores, plus the sequences of names and scores, and the sequence of corresponding letter grades.

Operations

From our behavioral description, we have these operations:

Description	Predefined?	Name	Library
Display a `string`	yes	<<	`iostream`
Read a `string`	yes	>>	`iostream`
Read a sequence of names and scores from a file	no	--	--
Compute the average of a sequence scores	no	--	--
Compute the standard deviation of a sequence scores	no	--	--
Compute the sequence of letter grades corresponding to a sequence scores	no	--	--
Display sequences of names, scores and letter grades	no	--	--

There are several undefined operations, so it appears we'll be writing quite a few functions for this lab.

Algorithm

We can organize the preceding objects and operations into the following algorithm:

1. Display a greeting.
2. Prompt for and read the name of the input file.

3. Fill nameVec and scoreVec with values from the input file.
4. Output the mean and standard deviation of the values in scoreVec.
5. Compute gradeVec, a vector of the letter grades corresponding to the scores in scoreVec.
6. Display the elements in nameVec, scoreVec and gradeVec.

We will thus use three
different vector objects: nameVec storing string values, scoreVec storing double v alues, and gradeVec storing char values.

Defining vector Objects

Vectors are declared like so in C++:

```
vector<Type> vecObj;
```

This declaration defines vecObj as a vector containing values of type Type.

A programmer using a vector must specify the type of value they want to store in a given vector. While a vector is *generally* useful for storing *any* type of data, we always know *exactly* what type of data we want to put in given a particular problem. While we could try to keep track of the particular type ourselves, it's much, much better to let the compiler worry about it.

The vector class is known as a **template class**; that is, it's be templated to allow us to specify the Type of data that is stored in each vector.. A complete discussion of templates and template classes is beyond us at this point, but for now it suffices to know why and how to use them.

With the vector class template, the compiler will check everything done with vecObj. Only Type objects can be put into vecObj; only Type objects will come out. No one (neither the programmer nor the compiler) is ever surprised by the contents of a vector.

In grade.cpp, use this information to define the three vectors needed in the main program: nameVec, scoreVec, and gradeVec. The first two vectors are declared early in the function; gradeVec is declared later. (The comments tell you where.) Each of these vectors stores a different type of data so write the declarations with this in mind. Also make sure you include the vector library.

Filling vector Objects from a File

Now let's start tackling the undefined operations in our main algorithm. The first undefined operation is to read in names and scores from a file. The name of the file is in inFileName; the names and scores should be placed into the vectors that you just declared in the main program.

We'll call this function fillVectors().

Design

Behavior

We can describe how this function should behave as follows:

Our function should receive the name of an input file, an empty `vector` of strings and an empty `vector` of doubles from its caller. The function should open an `ifstream` to the input file. It should then read each name and score from the `ifstream`, appending them to the `vector` of strings and the `vector` of doubles, respectively. When no values remain to be read, our function should close the `ifstream`, and pass back the filled `vector` objects.

Behavior

Using the behavioral description, our function needs the following objects:

Description	Type	Kind	Movement	Name
The name of the input file	`const string &`	constant	in	`inFileName`
An empty `vector` of strings	`vector<string> &`	varying	in & out	`nameVec`
An empty `vector` of doubles	`vector<double> &`	varying	in & out	`scoreVec`
a stream to `inFileName`	`ifstream`	varying	local	`inStream`
a student's name	`string`	varying	local	`name`
a student's score	`double`	varying	local	`score`

Given these objects, we can specify the behavior of our function as follows:

Specification:
receive: `inFileName`, a `string`; `nameVec`, a `vector<string>`; and `scoreVec`, a `vector<double>`.
passback: `nameVec` and `scoreVec`, filled with the values from `inFileName`.

Since this function seems unlikely to be generally reuseable, we will define it within the same file as our main function. Using the above information, place a prototype for `fillVectors()` before the main function, and a function stub after the main function.

Since `inFileName` is a class object that is received but not passed back, it should be defined as a constant reference parameter, not as a value parameter. In contrast, `nameVec` and `scoreVec` are passed back, and so should be defined as reference parameters.

Operations

In our behavioral description, we have the following operations:

Description	Predefined ?	Name	Library

Receive `inFileName`, `nameVec` and `scoreVec`	yes	function call mechanism	built-in
Open an `ifstream` to `inFileName`	yes	`ifstream` declaration	`fstream`
Read a `string` from an `ifstream`	yes	`>>`	`fstream`
Read a `double` from an `ifstream`	yes	`>>`	`fstream`
Append a `string` to a `vector<string>`	yes	`push_back()`	`vector`
Append a `double` to a `vector<double>`	yes	`push_back()`	`vector`
Repeat reading and appending for whole file	yes	eof-controlled input loop	built-in
Close an `ifstream`	yes	`close()`	`fstream`
Pass back `vector` objects	yes	reference parameter	built-in

Algorithm

We can organize these objects and operations into the following algorithm:

```
1. Receive inFileName, nameVec and scoreVec.
2. Open inStream, a stream to inFileName, and check that it opened successfully.
3. Loop
      a. Read name and score from inStream.
      b. If end-of-file was reached, terminate repetition.
      c. Append name to nameVec.
      d. Append score to scoreVec.

      End loop.

4. Close inStream.
```

Coding

To append values to a `vector`, use the `vector` method named `push_back()`:

```
vectorObject.push_back(newValue);
```

Such a statement "pushes" `newValue` onto the back of the `vector` named `vectorObject`.

Using this information, complete the stub of `fillVectors()`. Then compile what you have written and make sure that it is free of syntax errors before proceeding.

Average

The next undefined operation should average the scores. An important observation here is that *it doesn't matter that the values are scores.* They could be temperatures, heights, weights, times, or *any* `double` *values*. So the `average()` function that we'll write computes the average of some generic `double` values, regardless of what they actually mean.

It's important to note this now because it will have an effect on the way we name our variables. Yes, we'll continue to think of the scores as scores in the main program (and other functions), but in `average()` we'll think of them as just a sequence of real numbers.

Design

Behavior

We can describe how this function should behave as follows:

Our function should receive a `vector` of `double` values. It should add up the values in the `vector`, and if there is at least one value in the `vector`, our function should return the average of the values (i.e., the sum divided by the number of values). Otherwise, our function should display an error message and return a default value (e.g., 0.0).

Objects

Using the behavioral description, our function needs the following objects:

Description	Type	Kind	Movement	Name
A `vector` of doubles	`const vector<double> &`	varying	in	`numVec`
the sum of the values in the `vector`	`double`	varying	local	`sum`
the number of values in the `vector`	`int`	varying	local	`numValues`
an error message	`string`	literal	local	--
a default return value	`double`	literal	local	0.0

Given these objects, we can specify the behavior of our function as follows:

Specification:
receive: `numVec`, a `vector<double>`.
precondition: `numVec` is not empty.
return: the average of the values in `numVec`.

Since we've designed this function to be generally reusable, we will define it within a library named `DoubleVectorOps`.

Place a prototype
for `average()` in `DoubleVectorOps.h` and `DoubleVectorOps.doc`, and a function stub in `DoubleVectorOps.cpp`.

Since `numVec` is a class object that is received but not passed back, it should be defined as a constant reference parameter.

Operations

In our behavioral description, we have the following operations:

Description	Predefined?	Name	Library
Receive `numVec`	yes	function call mechanism	built-in
Sum the values in a `vector<double>`	yes	`accumulate()`	`numeric`
Determine the number of values in a `vector<double>`	yes	`size()`	`vector`
Divide two `double` values	yes	`/`	built-in
Return a `double` value	yes	`return`	built-in
Display an error message	yes	`<<`	`iostream`
Select between normal case and error case	yes	if statement	built-in

Algorithm

We can organize these objects and operations into the following algorithm:

```
1. Receive numVec.
2. Compute sum, the sum of the values in numVec.
3. Compute numValues, the number of values in numVec.
4. If numValues > 0:
      Return sum / numValues.
   Otherwise
      a. Display an error message via cerr.
      b. Return 0.0 as a default value.

      End if.
```

Coding

As noted above, the STL contains not only containers (like `vector`), but it also contains algorithms. The function `accumulate()` is one such algorithm, and it comes from the library named `numeric`. This function adds up the values in a `vector`:

```
accumulate(vectorObject.begin(), vectorObject.end(), 0.0)
```

This expression returns the sum of the values in `vectorObject`. To use `accumulate()`, you must include the `numeric` library in your program.

Another operation we need is to determine the number of values in a `vector`. This is considerably easier:

```
vectorObject.size()
```

This expression evaluates to the number of values in `vectorObject`. The `size` method is part of the `vector` class, and so you must include the `vector` library (which must be done anyone so that you can declare `vectorObject` in the first place).

Using this information, complete the stub for `average()`. Then uncomment the call to `average()` in the main function, translate and test your program. Don't forget to include the `numeric` library! Verify that your program is correctly computing the average of the values in the input file before you proceed.

`vector` and the Standard Template Library

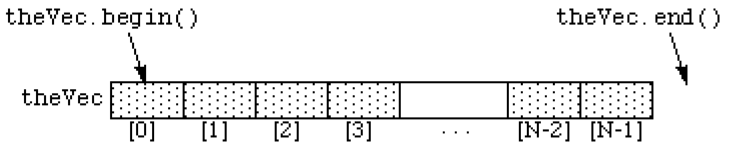
In writing the `average()` function, we used of one of the `vector` methods, `size()`; and we used one of the C++ standard library algorithms, `accumulate()`. Being knowledgeable about what methods and algorithms can be applied to an object is important because it allows you to avoid reinventing the wheel. We *could* have written our own functions to find the number of values in a `vector`, or add up the values in a `vector`, but why go to all that extra effort when we can use the provided one? Learning about the functions and algorithms that can be applied to an object takes time, but it is time well-spent, since it provides ready-made solutions for many of the problems you will encounter.

As mentioned earlier, `vector` is our first look at a template class. The `vector` template is just one of many different kinds of containers provided by the C++ Standard Template Library. Other containers include the `list`, the `set`, the `map`, the `stack`, the `queue` and a variety of others. Each of these containers has its own unique properties that distinquish it from the others. These containers are typically the subject of later computer science courses, and so we will not discuss them further here.

Some of the `vector` methods and each of the STL algorithms make use of a new kind of object called an **iterator**. An iterator is an object that can move through a sequence of values and access each of the values in turn. Each of the containers in STL provide iterators for processing their elements, and each of the STL algorithms take a container's iterators as arguments, which they use to access the elements in that container.

It is through iterators that the STL algorithms can be written easily to handle *any* of the STL containers. Otherwise, the authors of the STL would have to write an `accumulate()` function for *every* container implemented in the STL. In fact, we could write our own, brand-new container (perhaps never seen before in the history of computing); as long as we wrote iterators for our new container, we could use all of the STL algorithms without having to change the code for those algorithms.

A complete discussion of iterators is beyond us at this point; however, it is useful to understand that each of the containers (including `vector`) contains a method `begin()` that returns an iterator to its first element, plus a method `end()` that returns an iterator pointing beyond its final element.



These two iterators thus mark the beginning and end of a container. An STL algorithm (like `accumulate()`) can then use these two end points of the container to iterate through the whole container and do its work (e.g., add the elements together).

Standard Deviation

Returning to our problem at hand, we next need to write function to compute the standard deviation of our scores. `standardDev()` will be given a `vector` of `double` values, and it will return the standard deviation of those values.

Since a proper discussion of standard deviation would take us into statistics, we'll black box the computation. That is, the specification and algorithm are given below without the rest of the OCD design. Generally you do want a good understanding of the problem you're working on and of the solution you come up with for the problem. But in some cases, one person may design a solution, and some one else may be asked to implement it.

Design

The specification for this function is as follows:

Specification:
receive: `numVec`, a `vector<double>`.
precondition: `numVec` is not empty.
return: The standard deviation of the values in `numVec`.

Since this function is generally reusable (just like `average()`), it should also be defined in our DoubleVectorOps library. Using this specification, place a prototype for `standardDev()` in `DoubleVectorOps.h` and `DoubleVectorOps.doc`, and a function stub in `DoubleVectorOps.cpp`.

As before, `numVec` should be defined as a constant reference parameter.

An algorithm to compute the standard deviation is as follows:

```
1. Receive numVec.
2. Compute numValues, the number of values in numVec.
3. If numValues > 0:
      a. Define avg, a double initialized to the average of the values in numVec.
```

```
b. Define sumSqrTerms, a double initialized to zero.
c. For each index i of numVec:
      i.  Define term, a double initialized to $numVec_i$ - avg.
      ii. Add $term^2$ to sumSqrTerms.

    End loop.

d. Return sqrt(sumSqrTerms / numValues).

Otherwise

e. Display an error message via cerr.
f. Return 0.0 as a default value.

End if.
```

We use the notation $numVec_i$ to refer to the value in `numVec` whose index is `i`.

Coding

Vector size. To determine the number of values in a `vector`, we can use the `size()` method, as before.

Average. To compute the average of the values in a `vector`, we can use the `average()` function that we just defined.

The loop. The loop can be implemented as a counting `for` loop that counts from 0 to `numValues` minus one:

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

This is perhaps one of the most common ways to process a `vector`. Become very familiar with this loop since you will see it in any program that uses a `vector`. It's a little different from the counting loops we've used in the past; the primary reason for this is because `vector` indices start at index 0.

Element access. To access a value from a `vector`, the subscript operation can be applied to the `vector`, in the same manner as a `string`. The pattern is:

```
vectorObject [ index ]
```

This expression evaluates to the value that is stored within `vectorObject` at index `index`.

Do it. Using this information, complete the stub for `standardDev()`. Then uncomment the call to `standardDev()` in the main function, and translate and test what you have written.

You should get the following standard deviations (*approximately*):

File	Standard Deviation
`scores1.data`	11.1803
`scores2.data`	2.69186
`scores3.data`	0.927025

Make sure that your program is correctly computing the standard deviation of the values in the input file before you proceed.

Computing Letter Grades

Print a hard copy of each of the three scores files. Take a moment and and look over the distribution of scores in each file.

Question #10.1: If you were assigning grades, what letter grades would you assign for the scores in `scores1.data`, `scores2.data` and `scores3.data`? Assume the scores are out of a possible 100 points. Write your letter grades on the printouts.

Our current task is to write a function that computes the appropriate letter grades for the input scores, curving the grades as appropriate. This function needs a sequence of scores to process, and it must return a corresponding sequence of letter grades; we can specify what the function must do as follows:

Specification:
receive: `scoreVec`, a `vector` of `double` values.
return: `gradeVec`, a `vector` of `char` values.

Since this function seems pretty tightly tied to this particular grading problem, we will define it locally, within `grades.cpp`.

Using this information, prototype `computeLetterGrades()` before the main function and define a stub following the main function. Since `scoreVec` is a class object that is received but not passed back, define it as a constant reference parameter.

There are various ways to curve the grades based on a collection of scores. We'll base it on the average and standard deviation of the scores:

- Scores more than 1.5 standard deviations (s-ds) below the mean receive the 'F' (failing) grade.
- Scores that are between 0.5 and 1.5 s-ds below the mean receive the 'D' grade.
- Scores that are between 0.5 below and 0.5 s-ds above the mean receive the 'C' grade.
- Scores that are between 0.5 and 1.5 s-ds above the mean receive the 'B' grade.
- Scores that are more than 1.5 s-ds above the mean receive the 'A' grade.

Here is the algorithm:

1. Receive `scoreVec`, a `vector` of scores.
2. Define `numValues`, the number of values in the `vector`.
3. Define `gradeVec`, a `vector` of characters the same length as `scoreVec`.

```
4. If numValues > 0:
      a. Define avg, the average of the values.
      b. Define standardDev, the standard deviation of the values.
      c. Define F_CUT_OFF as avg - 1.5 * standardDev.
      d. Define D_CUT_OFF as avg - 0.5 * standardDev.
      e. Define C_CUT_OFF as avg + 0.5 * standardDev.
      f. Define B_CUT_OFF as avg + 1.5 * standardDev.
      g. For each index i of scoreVec:
            i.  If scoreVec_i < F_CUT_OFF:
                   Set gradeVec_i to 'F'.
                Else if scoreVec_i < D_CUT_OFF:
                   Set gradeVec_i to 'D'.
                Else if scoreVec_i < C_CUT_OFF:
                   Set gradeVec_i to 'C'.
                Else if scoreVec_i < B_CUT_OFF:
                   Set gradeVec_i to 'B'.
                Else
                   Set gradeVec_i to 'A'.
                End if.

         End loop.

5. End if.
6. Return gradeVec.
```

Using the information above, complete the definition of `computeLetterGrades()`. Then compile what you have written to check for syntax errors.

Display the Names, Scores and Grades

The final operation is to display the information we have computed.

Design and write a function that, given an `ostream`, a `vector` of names, a `vector` of scores, and a `vector` of letter grades, displays these three `vector` objects in tabular form. For example, the output produced when `scores1.data` is processed should appear something like the following:

```
Enter the name of the scores file: scores1.data

Mean score: 75
Std. Dev: 11.1803

joan    55      F
joe     60      D
jane    60      D
jim     65      D
janet   70      C
john    70      C
johanna 75      C
```

```
jack    75      C
joeline 75      C
jacques 75      C
josh    80      C
janna   80      C
jason   85      B
jadzia  90      B
jon     90      B
jackie  95      A
```

If you find that you have logic errors in what you have written, use the debugger to find them.

Question #10.2: When `scores2.data` or `scores3.data` are processed using our curve, how do the curved grades compare with the grades you assigned?

Question #10.3: What have you learned about grading on the curve?

Submit

Turn in your answers to the questions in this exercise, a copy of your code, and a sample run of the program on at least the three provided data files.

Terminology

element, iterator, sequence (of values), Standard Template Library, STL, subscript, template class, vector

3. Project 10

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #10.1: The **median** value of a sequence of numbers is a value *v* such that one half of the numbers are greater than *v*, and one half of the numbers are less than *v*. A simple algorithm to compute the median of a sequence is

```
1. Let seq be the sequence.
2. Let n be the number of values in the sequence.
3. Sort seq.
4. If n is an odd number:
      Return the middle value of the sorted sequence.
   Otherwise (n is even):
      Return the average of the two middle values in the sorted sequence.
```

Write a function that, given a vector of numbers, computes the median value of those numbers. Your function should use the STL `sort()` algorithm for the third step. Write a program that, given the name of a file containing an arbitrary sequence of numbers, computes and displays the median value of that sequence.

Project #10.2: A certain on-line testing program records student exam results in a text file, each line of which has the form:
`name examScore`
Write a program that analyzes student performance on an exam using the information from such a file (although you should store the data in `vector`s). The program should input the name of the text file, and then display the worst score, the best score, the average score, the standard deviation, and a *histogram*---a bar graph indicating the frequency with which a given score occurred. For example, if three people scored 74, five people scored 75, six people scored 76, no one scored 77 and two people scored 78, then that portion of the histogram should appear as:

```
74: ***
75: *****
76: ******
77:
78: **
```

Entries below the worst or above the best should not be displayed.

Project #10.3: Each year, the well-known meteorologist Dr. H. Tu Oh creates a file containing the year's 12 monthly precipitation totals. Write a program that, given the names of two of these files, will create a file containing an easy-to-read analysis comparing the two sets of readings, including which of the two years was the wettest (and by how much), the average monthly precipitation for each year, and the wettest and driest months in each year.

Project #10.4: Write a program that, given the name of a text file, reads that file and counts the number of occurrences of each alphabetic letter in the file. Your program should use a `vector` of length 26 to count the occurrences of the 26 alphabetic letters, and treat upper and lower-case instances of a letter as the same letter. (Hint #1: look up `tolower()` ---testing the case isn't necessary. Hint #2: if `ch` is a `char`, then `ch-'a'` is a valid C++ expression and will evaluate to 0 when `ch` is equal to 'a'---try 'b', 'c', and 'd'!)

Run your program on several large text files.

Study the results looking for patterns; then write a paragraph addressing these two questions:

1. How would these numbers be useful to someone decoding a message encoded using the Caesar cipher?
2. How would these numbers be useful to someone competing on the *Wheel of Fortune* game show?

Submit this paragraph as part of your documentation.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 11: Building Classes

1. Objective of the Experiment

- To learn how to write classes.
- To learn about instance variables and methods.
- To learn how to overload C++ operators.
- To learn about inline functions.

2. Theoretical Background

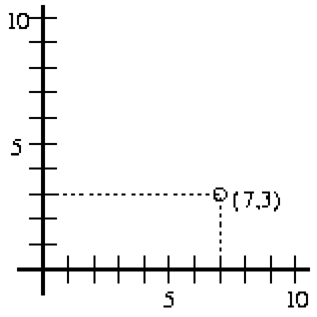
Introduction

Most of the problems we have examined have had relatively simple solutions, because the data objects in the problem could be represented using the predefined C++ types. We can represent a menu with a `string`, a choice from that menu with a `char`, the radius of a circle with a `double`, and so on.

The problem is that real-world problems often involve data objects that cannot be directly represented using just a single, predefined C++ type.

Let's consider two different problems.

Problem #1: Cartesian points. In the algebra courses you've taken, you have probably graphed some functions on a two-dimensional graph. These graphs typically use a *Cartesian coordinate system,* with *x*- and *y-coordinates,* to plot the points.



This graph plots the point (7,3), which is 7 units to the right along the x-axis and 3 units up along the y-axis. In this lab, we'll only worry about representing a single point. This work, though, could easily be used in a program to generate some actual graphs in a Cartesian coordinate system.

Problem #2: Fractions. The other problem is doing fractional arithmetic. Suppose that we know a certain gourmet chef named Pièrre whose recipes are written to make 12 servings. There a few difficulties:

1. Pièrre frequently must prepare a dish for more or fewer than 12 customers, requiring the scaling of his recipes (e.g., 1 customer results in 1/12 of a recipe, 2 customers results in 1/6 of a recipe, 15 customers results in 15/12 = 5/4 of a recipe, etc.).

2. Pièrre's recipes are written using fractions (e.g., 1/2 tsp., 3/4 cup, etc.) so that he must multiply fractions when scaling a recipe.
3. Pièrre is so poor at multiplying fractions, that he has hired us to write a program that will enable him to conveniently multiply two fractions.

We will work through two programs to manipulate fractions.

Coding. Keep in mind that *you* will write the code for the `Fraction` class. The `Coordinate` class is presented here as an example to help you write code for the `Fraction` class.

Files

Directory: `lab11`

- `Fraction.h`, `Fraction.cpp`, and `Fraction.doc` implement the `Fraction` class.
- `gcd.h` and `gcd.cpp` implement Euclid's greatest-common-divisor algorithm (used by the `Fraction` class).
- `pierre1.cpp` and `pierre2.cpp` are drivers for the `Fraction` class.
- `Makefile` is a makefile.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

Looking at the Code

Take a moment to compare the programs in `pierre1.cpp` and `pierre2.cpp`. Both programs implement the same basic algorithm:

1. Get `oldMeasure`, the fractional measure to be converted.
2. Get `scaleFactor`, the fractional conversion factor.
3. Compute `newMeasure = oldMeasure * scaleFactor`.
4. Output `newMeasure`.

A solution to Pièrre's problem is quite simple, *if* we have the ability to define, input, multiply and output fraction objects. That's the work we'll be doing in this lab.

The two programs differ only in how they read and write fraction objects. The second version has a more familiar look to it, but will require more work from us. So the first version will be useful for getting something workable, but it's missing some input and output features.

Creating Classes

The difficulty is that there is no predefined C++ type `Fraction`. In such *very* common situations, C++ provides a mechanism by which a programmer can create a new type and its operations. This mechanism is called a **class**.

In C++, a new type can be created by

1. Defining the data objects, known as **instance variables**, that make up the **attributes** of an object of the new type.
2. Surrounding those definitions with a class structure.

The class structure mentioned in this second step looks like this:

```
class TypeName
{
  public:

  private:

};
```
where `TypeName` is a name describing the new type. A class is (almost) always defined in a header file.

A class has two sections, a **public section** and a **private section**. The public section is where class operations are declared, and the private section is where class attributes are declared.

Let's first consider a point in Cartesian coordinates. A point consists of an x-coordinate and a y-coordinate. We can easily write two variables to hold these two values:

```
double myX, myY;
```

Of course, the context for this declaration is all important. We declare these variables in an appropriately-named class structure, making them *instance variables*:

```
class Coordinate
{
  public:

  private:
    double myX, myY;
};
```

We use the prefix "`my`" at the beginning of a name for an instance variable name to encourage an **internal perspective**. We imagine ourselves as the object we're trying to describe. For example, when figuring out these instance variables, I could say, "I am a Cartesian coordinate, and I have an x-coordinate and a y-coordinate." Since they are *my* attributes, I label them as such. This way I shouldn't get them confused with other objects that I work with.

The result is a new type, named `Coordinate`, which can be used as a type for new objects:

```
Coordinate point1, point2, point3, point4;
```

Each `pointI` has two `double` components, one named `myX` and the other named `myY`. This is why `myX` and `myY` are known as instance variables: every instance of a `Coordinate` has *its own copies* of these variables. The neat thing is that we only had to declare `myX` and `myY` once in the class definition.

We can now expand our general pattern for a class definition:

```
class TypeName
{
  public:

  private:
    Type1 InstVarList1;
    Type2 InstVarList2;
    ...
    TypeN InstVarListN;
};
```

where each `TypeI` is any defined type and each `InstVarListI` is a list of instance variables of type `TypeI`.

Now let's apply this same thinking to the fraction problem. We first need to identify the attributes of a `Fraction` object. Here are a few fractions:

```
1/2
4/3
4/16
16/4
```

Each fraction has the form:

`number1/number2`

where `number1` is called the **numerator** and `number2` is called the **denominator**. A fraction needs both a numerator and a denominator. However, the `/` symbol is common to all fractions, and so it is not recorded as an attribute of a fraction. Consequently, a fraction has just two attributes, a numerator and a denominator, both of which are integers.

Edit the file `Fraction.h`:

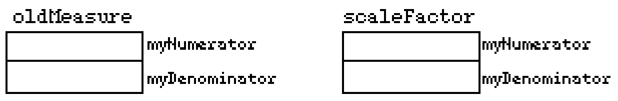
- Write a class definitions for a class named `Fraction`.
- In the appropriate place, define two integer instance variables named `myNumerator` and `myDenominator`.

Look at the patterns and example above for help with these two steps.

Given this declaration of `Fraction`, we can make the following declarations:

```
Fraction oldMeasure;
...
Fraction scaleFactor;
```

These declarations define two objects with the following forms:



Again, note that each instance of type `Fraction` has its own copy of each of the attributes we defined; that's why they're called *instance* variables. It's also why we prefix the variable name with "`my`", to encourage an internal perspective.

In the source program `pierre1.cpp`, uncomment the definition of `oldMeasure`. (Do not uncomment anything else, yet.) Compile your source program to test the syntax of what you have written. When it is correct, continue to the next part of the exercise.

Methods

Besides having instance variables, a class can also have methods. A **method** is a function declared *inside* a class to provide an operation for objects of the class.

We've used several methods already. A `vector` has a `size()` method. A `string` has a `substr()` method. But until now we've only *used* methods. Now we get to define our own methods in our own classes.

Methods are prototyped within the class structure itself (in the header file), and they are normally defined in the class implementation file. However, very simple methods (i.e., ones that fit on a single line) are defined "**inline**": in the header file, following the class declaration, prefixed by the keyword `inline`. The reasons for this are a bit technical and beyond the scope of what we're really trying to do. Just remember to define simple methods in the header file with an `inline` in front of them.

Class Structure and Information Hiding

One of the characteristics of a class is that its instance variables are kept **private**, meaning that a program using the class is not permitted to directly access them. Mostly this is an issue of trust. In the case of a `Coordinate` object, it probably wouldn't be too bad if other programmers had direct access to the `myX` and `myY` instance variables. However, in the `Fraction` class, we must make sure that `myDenominator` does not become zero since division by 0 is undefined. If we hide away `myDenominator` from the casual programmer, we can write the methods of the `Fraction` class so that `myDenominator` never becomes 0.

In practice, it is not worth the time and effort to try to figure out which instance variables are safe to let everyone access directly. Common practice is to make *all* instance variables private.

On the other hand, we *do* want users of the class to be able to access the operations of a class. As a result, the operations should be declared in the **public** section of the class. Anything defined in the public section can be accessed through an instance of the class from any part of the program. So, wherever we have a `Fraction` object, we will be able to access its public operations.

By convention, the public section of a class comes first, so that a user of the class can quickly see what operations it provides. It is good style to have one public section and one private section in a class; however, C++ allows the keywords `public:` and `private:` to appear an arbitrary number of times in a class declaration.

Methods As Messages

We have seen that methods are called differently from normal functions: if two `string` objects named `greeting` and `closing` are defined as follows:

```
string greeting = "hello",
       closing = "goodbye";
```
then the expression
```
cout << greeting.size() << ' ' << closing.size() << endl;
```
prints `5 7`, the sizes of the two `string`s.

Object-oriented programmers like to think of a call to a method as a **message** to which the object responds. When we send the `size()` message to `greeting`, `greeting` responds with the number of characters it contains; and if we send the same `size()` message to `closing`, `closing` responds with the number of characters it contains. This is part of the internal perspective for designing and writing a class. When we use the `string` class, we're not responsible for its operations. We just simple ask the `string` objects to carry out various actions.

As you might expect, *defining* a method is also a bit different from defining a normal function.

a) Part I: `pierre1.cpp`

In this first part of this exercise, we will focus on adding the methods to class `Fraction` needed to get `pierre1.cpp` operational.

An Output Method

To facilitate debugging a class, it is often a good idea to begin with a method that can be used to display class objects---an output method.

From the perspective of our `Coordinate` class, we can specify the task of such a method, which we'll call `print()`, as follows:

Specification:
receive: `out`, an `ostream` to which I write my values.
output: `myX` and `myY`, as an ordered pair.
passback: `out`, containing the output values.
Note the use of internal perspective (e.g., "...*I* write *my* value.").

Also note that this method receives and passes back an `ostream`. The rule is very simple:

Rule: *All streams are passed by reference---received and passed back.*

The reasons for this are fairly technical and are beyond what we really need to know right now, but this is hard-and-fast rule.

Methods must be prototyped within the class declaration, so we would write:

```
class Coordinate
{
  public:
    void print(ostream & out) const;
  private:
    double myX, myY;
};
```

Whoa! Where did that `const` come from? Internal perspective is an easy perspective for explaining this: "I am a `Coordinate` object, and when I am printed using `print()`, I should not change." It's the `const` that adds the "I should not change". Think about it: a `Coordinate` object that changes every time we print it would be quite useless. We can get the compiler to enforce this observation by adding `const` after the parameter list for the method; now, the compiler will not let us change `myX` or `myY` in the method definition.

Also, we've put `print()` in the public section of the class. This is quite typical. It's very common for programmers to print out coordinates in their programs, so it's an operation that we want to make available to everyone. Hence, it goes in the public section.

Whenever you declare a method, take a bit of time to think if the method should change the instance variables of the class. If you even *suspect* they shouldn't change, add the `const`. You can always take it away later if you discover you *do* need to change the contents of an instance variable. (We'll see a method that does very soon.)

This is a fairly simple method, so we would define it within the header file `Coordinate.h`, following the declaration of class `Coordinate`. As an inline method, we would precede its definition with the keyword `inline`.

To define `print()`, we must inform the compiler that this is a method of a class. This is done by preceding the name of the method with the name of the class of which the method is a member and the **scope operator** (::). That is, we would define `print()` as a method of our `Coordinate` class as follows:

```
inline void Coordinate::print(ostream & out) const
{
  out << '(' << myX << ',' << myY << ')';
}
```

- `inline` indicates that this is a simple method that should be inlined.
- `void` tells the compiler that this method returns nothing.
- `Coordinate::` tells the compiler that this method is a member of class `Coordinate`.
- `print` is the name of the method.
- `(ostream & out)` is the parameter list of the method.

- `const` tells the compiler that this method should not modify any of the instance variables of class `Coordinate`.

Note that `const` must be present in *both* the prototype *and* the definition of a method that does not modify its instance variables. In contrast, `inline` is used *only* with the definition, not the prototype.

Let's consider what happens when this method is invoked. As with anything class related, this makes the most sense in an internal perspective: "I am a `Coordinate` object, and when someone sends me a message telling me to print myself out, I send a left parenthesis, my x-coordinate, a comma, my y-coordinate, and a right parenthesis to the provided `ostream`." Note how the "`my`" prefix makes the code similar to this statement; also note the message metaphor.

Let's see if you have the hang of this:

Question #11.1: If `point` is a `Coordinate` object whose x-coordinate is 3 and whose y-coordinate is 4, then what does the statement `point.print(cout);` display?

Question #11.2: If `origin` is a `Coordinate` object whose x-coordinate is 0 and whose y-coordinate is 0, then what does the statement `origin.print(cerr);` display?

As seen in the method definition above, a method can directly access the private instance variables of the object. Otherwise, the private instance variables would remain hidden to all code everywhere, making them quite useless. Someone should be able to access them, and why not myself? If I'm a `Coordinate` object and someone asks me to print myself, I *should* be allowed to access my private instance variables.

Using all of this, prototype and define a similar `print()` method for your `Fraction` class. Write the method so that when `oldMeasure` is a `Fraction` whose numerator is 3 and whose denominator is 4, then a message:

```
oldMeasure.print(cout);
```
will display

```
3/4
```

Prototype this method in the public section of class `Fraction`, and define it as an `inline` method following the declaration of class `Fraction` in `Fraction.h`. Check the syntax of your method, and continue when it is correct.

Constructors

An output operation for a class is of little use unless we are able to define and initialize objects of that class. The action of defining and initializing an object is called **constructing** that object. To allow the designer of a class to control the construction of class objects, C++ allows us to define a special function of a class called a **constructor**. A constructor specifies what actions are to be taken when a class object is constructed. When an instance is declared, the compiler calls this constructor function to initialize the object's instance variables.

For example, what should happen when we declare a new `Coordinate` object like so:

```
Coordinate point;
```

Presently, there would be junk in this object's instance variables. It seems reasonable, though, to initialize the x- and y-coordinates to 0.

We might specify this as a **postcondition** (as part of a specification):

Specification:
postcondition: `myX` == 0.0 and `myY` == 0.0.

A constructor does *not* return anything to its caller; it initializes the instance variables of an object when that object is defined. We specify this behavior as a boolean expression which is true when the constructor terminates. Such an expression is called a postcondition, since it is a condition that holds true *after* (i.e., "post") the constructor finishes.

A postcondition is *not* code you should write. It's a test that should be true *if* you tested it at the end of the constructor. You *can* test a postcondition if you want (using an `assert()`), and this may be a good idea (especially if there's a lot of code and there's a bug you can't find). But it's not necessary. Instead, a postcondition indicates what *other* code you should write. In this case, what code can I execute so that `myX` is 0.0? The answer is below.

In order for a constructor to be a member of a class, its prototype must appear within the class. Unlike other functions, C++ determines the name of this function:

Rule: *The name of a constructor is always the name of the class.*

So we prototype this constructor in the public section of class `Coordinate`, as follows:

```
class Coordinate
{
  public:
    Coordinate();
    void print(ostream & out) const;
  private:
    double myX, myY;
};
```

Unlike our `print()` method, a constructor initializes (i.e., modifies) the instance variables of a class. As a result, it should *not* (and cannot) be prototyped or defined as `const`. So the "missing" `const` is understandable.

However, *where's the return type?!! Every* function has had a return type. C++ insists on it. Well, in fact, C++ will equally insist that a constructor does *not* have a return type. As observed above, a constructor never returns anything to its caller; it initializes its instance variables. The object itself has *already* been created, so that doesn't have to be returned. It just needs to be initialized. So constructors do not need a return type.

As with the `print()` method, we want everyone to be able to construct `Coordinate` objects, so the prototype should be placed in the public section of the class.

Also, as before, simple definitions should be placed in the class header file, designated as inline functions. To define an inline `Coordinate` constructor, we thus write this funny-looking definition in the header file:

```
inline Coordinate::Coordinate()
{
  myX = 0.0;
  myY = 0.0;
}
```

The first `Coordinate` is the name of the class, telling the compiler that this is some member of class `Coordinate`. The second `Coordinate` is the name of the constructor.

Given this definition, when a `Coordinate` object is defined, the compiler will call this constructor to initialize the new object, setting the object's `myX` and `myY` instance variables to zero.

The pattern for a constructor is thus:

```
ClassName::ClassName(ParameterList)
{
  StatementList
}
```

where the first `ClassName` refers to the name of the class, the second `ClassName` names the constructor, and `StatementList` is a sequence of statements that initialize the instance variables of the class.

Constructors can take parameters, which are defined as they would be for any other function, and any valid C++ statement can appear in the body of such a constructor.

Using this information, prototype and define a constructor for your `Fraction` class, that satisfies the following specification:

Specification:
postcondition: `myNumerator` == 0 and `myDenominator` == 1.
That is, the definition:
`Fraction oldMeasure;`
should initialize the instance variables of `oldMeasure` appropriately to represent the fraction 0/1. (Remember that division by 0 is a Bad Thing, so initializing it to 0/0 is not advisable.)

Store the prototype in the public section of class `Fraction`, and define it as `inline`, following the declaration of class `Fraction`, in `Fraction.h`. Test the syntax of what you have written, and continue when it is correct.

A Second Constructor

A class can have multiple constructors, so long as each definition is distinct in either the number or the type of its parameters. Defining the same function multiple times is called **overloading** the function. Overloading works for normal functions, methods, and constructors.

Suppose that we would like to be able to explicitly initialize the x- and y-coordinates of a `Coordinate` object to two values specified by the programmer creating the object. We can specify this as follows:

Specification:
receive: `x` and `y`, two `double` values.
postcondition: `myX` == `x` and `myY` == `y`.

We can overload the `Coordinate` constructor with a second definition, one that takes two `double` arguments and uses them to initialize our instance variables:

```
inline Coordinate::Coordinate(double x, double y)
{
  myX = x;
  myY = y;
}
```

Note the benefit of using the "`my`" prefix: we don't have to come up with silly or awkward names for the parameters here. If we called the instance variables `x` and `y`, we'd have to come up with different names for these parameters. This convention also clarifies the internal perspective: `myX` is my x-coordinate while `x` is an x-coordinate that something else has handed to me.

As usual for such a simple function, this constructor is defined inline in the header file, and like all methods of a class, its prototype would be placed in the public section of class `Coordinate`:

```
class Coordinate
{
  public:
    Coordinate();
    Coordinate(double x, double y);
    void print(ostream & out) const;
  private:
    double myX, myY;
};
```

We can declare `Coordinate` objects to invoke this constructor:

```
Coordinate point1,
           point2(1.2, 3.4);
```

Our first constructor initializes `point1` since its declaration has no arguments and the first constructor has no parameters. Our second constructor initializes `point2` since its declaration has two `double` arguments and the second constructor has two `double` parameters. After these declarations, we have these objects:

point1
0.0
0.0
myX
myY
point2
1.2
3.4
myX
myY

Using this information, define and prototype a second `Fraction` constructor that satisfies this specification:

Specification:
receive: `numerator` and `denominator`, two integers.
precondition: `denominator` is *not* 0.
postcondition: `myNumerator` == `numerator` and `myDenominator` == `denominato r`.
Consequently, the definitions

```
Fraction oldMeasure;
...
Fraction scaleFactor(1, 6);
```

should initialize `oldMeasure` to 0/1, and initialize `scaleFactor` to 1/6.

Use a call to `assert()` to ensure the precondition.

Use the compiler to test the syntax of what you have written. When the syntax is correct, use `pierre1.cpp` to test what you have done, by inserting calls to `print()` to display their values:

```
...
oldMeasure.print(cout);
...
scaleFactor.print(cout);
```

Also try initializing `scaleFactor` with a zero denominator. Make sure that the `assert()` is triggered.

When your constructors and methods are working correctly, remove this test code from `pierre1.cpp`.

Accessor Methods

It might be useful to be able to extract the x- and y-coordinates of a `Coordinate` object. It is common to need the values stored in an object, and the methods that return these values are generally known as **accessor methods**.

As with most classes, the accessor methods for the `Coordinate` class have very simple specifications:

Specification:
return: my x-coordinate.
and
Specification:

return: my y-coordinate.

Since these methods do not modify any of the instance variables, we can declare them as `const` methods. Generally, accessor methods begin with the prefix "`get`" followed by the name of the attribute (*without* "`my`"). So, two new prototypes are added:

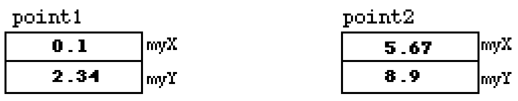
```
class Coordinate
{
  public:
    Coordinate();
    Coordinate(double x, double y);
    double getX() const;
    double getY() const;
    void print(ostream & out) const;
  private:
    double myX, myY;
};
```

We would then define these simple methods in the header file as follows:

```
inline double Coordinate::getX() const
{
  return myX;
}

inline double Coordinate::getY() const
{
  return myY;
}
```

Suppose we declare two `Coordinate` objects `point1` and `point2`:



Question #11.3: What do these expression evaluate to: `point1.getX()` and `point2.getY()`?

This is another reason for the "`my`" prefix. We can use the `my`-free names as the name of a method that accesses the value of that instance variable.

Using this information, add to class `Fraction` an accessor method `getNumerator()` that satisfies this specification:

Specification:
return: `myNumerator`.
Also write an accessor method `getDenominator()` that satisfies this specification:
Specification:

return: `myDenominator`.
Since these are simple methods, define them `inline` following the class declaration in the header file. Test their syntax by compiling the code and continue when they are correct.

Input

Once we are able to define `Fraction` objects, it is useful to be able to input a `Fraction` value. To illustrate, suppose that we wanted to input a `Coordinate` value that looked like this:

```
(3,4)
```
A user would type this in, or perhaps a program would read this in from a file.

We can specify the problem as follows:

Specification:
receive: `in`, an `istream.`
precondition: `in` contains a `Coordinate` of the form `(x,y).`
input: `(x,y)`, from `in`.
passback: `in`, with the input values extracted from it.
postcondition: `myX` == `x` && `myY` == `y`.

We prototype this method in the class (as with all methods), although this one is *not* `const` because it modifies the instance variables:

```
class Coordinate
{
  public:
    Coordinate();
    Coordinate(double x, double y);
    double getX() const;
    double getY() const;
    void read(istream & in);
    void print(ostream & out) const;
  private:
    double myX, myY;
};
```

We can define `read()` as a method that satisfies the specification, as follows:

```
void Coordinate::read(istream & in)
{
  char ch;          // for reading ( , and )
  in >> ch          // read '('
     >> myX         // read x-coordinate
     >> ch          // read ','
     >> myY         // read y-coordinate
     >> ch;         // read ')'
}
```

Testing pre- and postconditions for an input method is tricky at best, so don't worry that we haven't tested them here. They're good to write down anyway so that we're clear on what we expect.

The input specification indicates that a coordinate in the input will have punctuation around it: parentheses and a comma. The `ch` variable is used to read in these characters and throw them away. It's useful to have them in the input to make the input more readable, but they're frivolous in a `Coordinate` object since *every* `Coordinate` object is written with this punctuation. We only need it in the input and output.

Given the length of this method, it's pressing the boundaries of what some compilers define as "simple"; some won't allow us to declare it inline. As a result, we define it in the implementation file (without the keyword `inline`).

With this method, the statements

```
Coordinate point;
point.read(cin);
```
reads a `Coordinate` of the form `(x,y)` from `cin`.

Using this information, define and prototype an input method named `read()` for class `Fraction`. Your method should satisfy this specification:

Specification:
receive: `in`, an `istream`.
precondition: `in` contains a Fraction value of the form `n/d` and `d` is not zero.
input: `n/d`, from `in`.
passback: `in`, with `Fraction` `n/d` extracted from it.
postcondition: `myNumerator` == `n` and `myDenominator` == `d`.

Some differences from the `read()` of `Coordinate`:

- You *can* (and should) test the second half of the precondition (i.e., `d` is not zero). Use a call to `assert()`.
- Watch the punctuation. A coordinate has three punctuation characters, but a fraction has just one. Don't blindly copy the input statement. Change the comments so that it's clear what's being read in by each input operator.

After you've written and successfully compiled this new method, you should be able to uncomment the statements:

```
oldMeasure.read(cin);
...
scaleFactor.read(cin);
```

Test your input method by adding `print()` statements after the `read()` statements in `pierre1.cpp` to test the input routines. Compile and run the program, and continue when `read()` works correctly. Remove the test `print()` statements.

Fractional Multiplication

We have seen that methods like constructors can be overloaded. In addition, C++ allows us to overload operators, such as the arithmetic operators (`+`, `-`, `*`, `/`, and `%`). However to do so, we need to rethink the way expressions work.

Suppose that we want to permit two `Coordinate` objects to be added together. In C++'s object-oriented world, an expression like

`point1 + point2`
is thought of as sending the + message to `point1`, with `point2` as a message argument. We can specify the problem from the perspective of the `Coordinate` receiving this message:
Specification:
receive: `point2`, a Coordinate.
return: `result`, a Coordinate.
postcondition: `result.myX` == `myX` + `point2.myX` and `result.myY` == `myY` + `point2.myY`.

Again, the postcondition here suggests the code that we should write, and it doesn't particularly warrant testing at the end of the method.

According to our specification, this operation does not modify the instance variables of the `Coordinate` that receives it, and so we write the following prototype:

```
class Coordinate
{
  public:
    Coordinate();
    Coordinate(double x, double y);
    double getX() const;
    double getY() const;
    void read(istream & in);
    void print(ostream & out) const;
    Coordinate operator+(const Coordinate & point2) const;
  private:
    double myX, myY;
};
```

Not only is the method itself `const`, but the parameter is `const` as well. We don't change either object. We pass the parameter by reference because it's not a primitive type; recall that passing by reference is a bit faster than by value for non-primitive types.

One way to define this method is as follows:

```
Coordinate Coordinate::operator+(const Coordinate & point2)
const
{
  Coordinate result(myX + point2.getX(), myY + point2.getY());
  return result;
}
```

This definition uses our second constructor to construct and initialize `result` with the appropriate values.

This function illustrates that, for any overloadable operator Δ, we can use the notation `operatorΔ` as the name of a function that overloads Δ with a new definition.

Once such a method has been prototyped and defined as a member of class `Coordinate`, we can write familiar looking expressions, such as

```
point1 + point2
```
to compute the sum of two `Coordinate` objects `point1` and `point2`.

The C++ compiler treats such an expression as an alternative notation for the method call:

```
point1.operator+(point2)
```

While it is useful to overload all of the arithmetic operators for a `Fraction`, the particular operation that we need in order to solve our problem is multiplication (the others, we leave for the exercises). From the preceding discussion, it should be evident that we need to overload `operator*` so that the expression in `pierre1.cpp`:

```
oldMeasure * scaleFactor
```
can be used to multiply the two `Fraction` objects `oldMeasure` and `scaleFactor`.

However, the math here is a bit more involved that in the past examples and tasks. We can get some insight into the problem by working some simple examples:

- 1/2 * 2/3 = 2/6 = 1/3
- 3/4 * 2/3 = 6/12 = 1/2

We multiply the numerators together and the denominators together, and then we simplify.

The specification for such an operation can be written as follows:

Specification:
receive: `rightOperand`, a `Fraction` operand.
return: `result`, a `Fraction`, containing the product of the receiver of this message and `rightOperand`, simplified, if necessary.

We can construct `result` by taking the product of the corresponding instance variables and then simplifying the resulting `Fraction`.

For the moment, ignore the problem of simplifying a `Fraction`. Extend your `Fraction` class with a definition of `operator*` that can be used to multiply two `Fraction` objects. When we add the code to simplify `result`, this method will be reasonably complicated, so define it in the implementation file (and, of course, prototype it in the header file, in the class definition). Test the correctness of what you have written by uncommenting the lines in `pierre1.cpp` that that compute and output `newMeasure`. Continue when your multiplication operation yields correct (if unsimplified) results.

The main deficiency of our implementation of `operator*` is its inability to simplify improper fractions. That is, our multiplication operation would be improved if class `Fraction` had a `simplify()` operation, such that fractions like: 2/6, 6/12, 12/4 could be simplified to 1/3, 1/2, and 3/1, respectively.

Such an operation is useful to keep fractional results as simple and easy to read as possible. To provide this capability, we will implement a `Fraction` method named `simplify()`. In a method like `operator*` which constructs its answer in a `Fraction` named `result`, `simplify()` can be invoked like so:

```
result.simplify();
```
to simplify the `Fraction` in `result`.

It will help to phrase this in message passing terms. This calls says, "Hey, `result`! Simplify yourself!" `result` then goes off and simplifies itself (e.g., changes itself from 12/4 to 3/1). I don't expect anything back; I've told `result` to do all the work and change itself.

That takes care of `result` in the multiplication operation, but what about the definition of `simplify()`? We shift our perspective to the `Fraction` that's been told to simplify itself. There are a number of ways to simplify a fraction. One way is the following algorithm:

1. Find `gcd`, the greatest common divisor of `myNumerator` and `myDenominator`.
2. Replace `myNumerator` by `myNumerator/gcd`.
3. Replace `myDenominator` by `myDenominator/gcd`.

Those "replace" steps are assignment statements: e.g., "Change my numerator to be my numerator (my old one) divided by the GCD." Read this statement carefully, and the code writes itself.

The specification for this method is thus:

Specification:
postcondition: `myNumerator` and `myDenominator` do not share an common factors (i.e., the fraction is simplified).

Remember that `simplify()` doesn't need any extra information, except the `Fraction` object that it's invoked on, so there are no parameters. Also, the object that receives this message changes itself, so it *cannot* be declared `const`. And there's no need to return anything. So we end up with only a postcondition in our specification.

The implementation file `gcd.cpp` contains a function `greatestCommonDivisor()` that implements Euclid's algorithm for finding the greatest common divisor of two integers. Using `greatestCommonDivisor()` and the preceding algorithm, define `simplify()` as a method of class `Fraction`.

Since this is a complicated operation, define it in the implementation file, and only prototype it in the header file.

We have now provided all of the operations needed by `pierre1.cpp`, so the complete program should be operable and can be used to test the operations of our `Fraction` class.

b) Part II. `pierre2.cpp`

In this second part of this exercise, we add the functionality to class `Fraction` in order for `pierre2.cpp` to work properly, so use your text editor to open it for editing.

Output Revisited

While we have provided the capability to output a `Fraction` value via a `print()` method, doing so requires that we write clumsy code like:

```
cout << "\nThe converted measurement is: ";
newMeasure.print(cout);
cout << "\n\n";
```

instead of elegant code like:

```
cout << "\nThe converted measurement is: " << newMeasure <<
"\n\n";
```

Our `print()` method does solve the problem, but its solution doesn't fit in particularly well with the rest of the `iostream` library operations. It would be preferable if we could use the usual insertion operator (<<) to display a `Fraction` value.

Let's revisit our `Coordinate` class. We would like to be able to write this:

```
cout << point << endl;
```

And this should display the `Coordinate` object named `point` on `cout`.

Well, << is an operator just like + or `*`, and overloading those worked well, so we could add a method to class `ostream` overloading `operator<<` with a new definition to display a `Coordinate` value. Then the compiler could treat an expression like

```
cout << point
```

as the call

```
cout.operator<<(point)
```

However, this would require us to modify `ostream`, a predefined, standardized class. This is *never* a good idea, since the resulting class will no longer be standardized. (It's also *very* hard to do.)

We *could* define `operator<<` as a method of `Coordinate`, but C++ would then require us to invoke the method like so:

```
point << cout;
```

Yikes! That looks like `cout` is being sent to `point`---the *complete* opposite of what we want. Technically, we could use the >> operator instead, but we'd still have the output stream on the right, and we're used to our streams on the far left. Everyone keeps their streams on the left.

Instead, we can overload the insertion operator (<<) as a normal function (i.e., *not* as a method) that takes an `ostream` (e.g., `cout`) and a `Coordinate` (e.g., `point`) as its operands. That is, an expression

```
cout << point
```
will be translated by the compiler as a normal function call
```
operator<<(cout, point)
```
That's exactly what we want.

We define the following function in the header file, following the class declaration:

```
inline ostream & operator<<(ostream & out, const Coordinate &
coord)
{
  coord.print(out);
  return out;
}
```

There are several subtle points in this definition that need further explanation:

- Because the function is simple, we define it in the header file as an `inline` function. Since it is not a method, defining it in the header file serves as both prototype and definition for the function. An inline method must still be prototyped in the class so that the compiler fully understands that it is a member of the class.
- When we invoke this function, the left operand is an `ostream` which is altered by the operation---the `Coordinate` is inserted into the output stream. Since it's changed, the `ostream` is a non-`const` reference parameter.
- When we invoke this function, the type of the right operand is the type of the object we want to display, and this object does not change. Since it's not primitive, we pass it by reference; since it does not change, we pass it `const`.
- We want to chain objects in a output statement:

  ```
  cout << Value1 << Value2 << ... << ValueN;
  ```

  The leftmost << is applied first, and the value it returns becomes the left operand to the second <<. Similarly, the value returned by the second << becomes the left operand of the third <<, and so on, down the chain. Consequently, our function must return an `ostream` to its caller to make the chaining work correctly. However, if we simply make the return-type `ostream` (instead of `ostream&`), the C++ function-return mechanism will make and return a *copy* of parameter `out` which (as an alias for `cout`) would return a copy of `cout` for use by the next operator in the chain. As a result, the next value would get inserted into a copy of `cout`, rather than `cout` itself, which would have unpredictable results.

  What we need is a way to tell the compiler to return *the actual object* to which `out` refers, instead of a copy of it. This is accomplished by defining the function with a **reference return-type**, `ostream&`. The compiler then does all the work to return the actual object itself (not a copy), and so the chaining will work as we want it to.

- The actual work of formatting and outputting the `Coordinate` is done by sending the `Coordinate` parameter `coord` the `print()` message we defined in Part I of this exercise, with `out` as its argument. If we hadn't written `print()`, we could still define this function using the accessor methods `getNumerator()` and `getDenominator()`.

For our `Fraction` class, the specification of this operation is thus:

Specification:
receive: `out`, an `ostream`, and `aFraction`, a `Fraction`.
precondition: `aFraction.getNumerator()` == `n` and `aFraction.getDenominator()` == `d`.
output: `aFraction`, in the form `n/d`, via `out`.
passback: `out`, containing `n/d`.
return: `out`, for chaining.

Using this information, overload the output operator for class `Fraction`. Uncomment the appropriate line in `pierre2.cpp`, and test it out. You haven't written the input operator yet, so *don't* uncomment those lines yet. Just use the default values in the `Fraction`s. Continue when `pierre2` compiles and executes correctly (as far as it should for now).

Input Revisited

As a dual to output, all of the things we learned about the output operator also apply to the input operator, just with a slight change in direction (reading information in, instead of sending it out). The syntax is *very* similar.

Suppose we wanted to input a `Coordinate`, entered as

```
(3,4)
```

We could define this operator in the header file:

```
inline istream & operator>>(istream & in, Coordinate & coord)
{
  coord.read(in);
  return in;
}
```

There are a few differences between the input and output (i.e., extractor and insertion) operators:

- The insertion operator is `operator<<`, the extraction operator is `operator>>`.
- The left operand of the extraction operator is an `istream` instead of an `ostream`.
- The right operand is a `Coordinate` reference, rather than a `const Coordinate` reference. Since we *want* to change this `Coordinate`, it cannot be `const`.
- The function returns a reference to an `istream` reference instead of an `ostream` reference. Similar to the output operator, this is for chaining: `cin >> point1 >> point2;`.

This operation is sufficiently simple to define inline within the header file.

Using this information, overload the extraction operator for class `Fraction`, so that a user can enter

```
3/4
```
to input the fraction 3/4.

Uncomment the remaining statements in `pierre2.cpp`, and test the correctness of these operations. Your `Fraction` class should now have sufficient functionality for Chef Pièrre to solve his problem using `pierre2`.

When everything in your `Fraction` class is correct, complete `Fraction.doc`.

c) Part III: Friend Functions

While it is not necessary for this particular problem and the code solution we have, there are certain situations where it is useful for a function that is *not* a member of a class to be able to access the private instance variables. By default, C++ will not allow any other function to access private instance variables.

But suppose we had not written the `read()` method for class `Coordinate`, and wanted to overload `operator>>` in order to input `Coordinate` values. We'd probably write what we *do* have for `read()` in the input operator:

```
istream & operator>>(istream & in, Coordinate & coord)
{
  char ch;
  in >> ch            // consume (
     >> coord.myX     // read x-coordinate
     >> ch            // consume ,
     >> coord.myY     // read y-coordinate
     >> ch;           // consume )
}
```

Syntactically, this is 100% correct. However, semantically, the compiler does not allow this and will generate compilation errors for the accesses to `coord`'s instance variables. As a non-method, this operator does not have the privilege of accessing the private instance variables. However, we might like to be able to grant this permission.

For such situations, C++ provides the **friend mechanism**. If a class names a function as a friend with the keyword `friend`, then that non-member function is permitted to access the private section of the class. A function is declared as a friend by including a prototype of the function preceded by the keyword `friend` in the class definition. Thus, we would have to write this in our `Coordinate` class:

```
class Coordinate
{
  public:
    Coordinate();
    Coordinate(double x, double y);
    double getX() const;
    double getY() const;
    Coordinate operator+(const Coordinate & point2) const;
    friend istream & operator>>(istream & in, Coordinate &
coord);
  private:
    double myX, myY;
};
```

Placing the `friend` keyword before a function prototype in a class thus has two effects:

- It tells the compiler that the function is *not* a member of the class.
- It tells the compiler that a definition of that function is nevertheless permitted to access the private section of the class.

As we have seen in this exercise, an object-oriented programmer can usually find other ways to implement an operation without resorting to the `friend` mechanism. In the object-oriented world, solving a problem through the use of methods is generally preferred to solving it through use of the `friend` mechanism. As a result, `friend` functions tend to be used only when the object-oriented alternatives are too inefficient for a given situation. Nevertheless, they are a part of the C++ language, and you should know the distinction between a method and a `friend` function of a class.

You could make a similar change to your `Fraction` class, even just prototype the input and output operators as `friend` functions, but it's not necessary.

Object-Centered Design Revisited

Now that you have seen how to build a class, we need to expand our design methodology to incorporate classes:

1. Describe the behavior of the program.
2. Identify the objects in the problem. **If an object cannot be directly represented using available types, design and implement a class to represent such objects.**
3. Identify the operations needed to solve the problem. If an operation is not predefined:
    a. Design and implement a function to perform that operation.
    b. **If the operation is to be applied to a class object, design and implement the function as a method (or a friend function).**
4. Organize the objects and operations into an algorithm.

Using this methodology and the C++ class mechanism, we can now create a software model of *any* object! If you can imagine it, you can write a class for it.

The class thus provides the foundation for **object-oriented programming**, and mastering the use of classes is essential for anyone wishing to program in the object-oriented world.

Learning to design and implement classes is an acquired skill, so feel free to practice by creating software models of objects you see in the world around you! You cannot practice this too much.

Submit

Turn in a copy of your code and sample executions of *both* drivers. Answer the questions posed in this lab exercise in comments in the `Fraction.doc` documentation file.

Terminology

accessor method, attribute, class, construct (an object), construct, friend mechanism, inline method, instance variable, internal perspective, message, method, object-oriented programming, overloading, postcondition, private, public, reference return-type, scope operator

3. Project 11

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #11.1: Extend the class `Fraction` by overloading the remaining arithmetic operators (`+`, `-`, and `/`), the six relational operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`). Then construct a menu-driven 4-function calculator that an elementary student can use to check his or her fraction homework assignments.

In your documentation file, include some comments why `%` should *not* (and is not) defined for `Fraction`.

Project #11.2: Create a "drill" program for the `Fraction` class. This program should generate random fractions (using class `RandomInt` from *C++: An Introduction to Computing*) that are displayed and then asks for the user to type in their product. The program checks the answer and tells the user if they were right or wrong. This would be great for grade-school kids learning fractions for the first time. Keep track of the total number of problems the user gets right and gets wrong; report this number when the program finishes.

Project #11.3: A quadratic equation has the form

$ax^2 + bx + c = 0$

where `a`, `b`, and `c` are all real values. Write a class `Quadratic` that can be used to model a quadratic equation, with operations to construct with default values, construct with explicit values, input, output, extract the instance variables of, evaluate (for a given value of `x`), find the roots of, and find the `x` value at which the value of the `Quadratic` is minimized (or maximized).

To test your class, write a menu-driven program that allows a user to enter a quadratic, and repeatedly process it using any of the provided operations.

Project #11.4: A phone number consists of four separate pieces of information: an area code, an exchange, a local number, and a long-distance indicator (true or false). Design and build a `PhoneNumber` class that models a phone number, providing operations to construct, input, output, extract each of the instance variables of a `PhoneNumber` object, and indicate whether or not the number is long-distance. The input operation should read a local or long-distance number and set the long-distance indicator accordingly. The output operation should display a local number differently from the way it displays a long-distance number (e.g., `555-1234` vs. `(616) 555-1234`).

To test your class, write a program that simulates an intelligent computer modem dialer by reading a `PhoneNumber`, and displaying the number to be dialed. If the number is a long distance number, it should be preceded by `1-`; otherwise it should be displayed as a local number.

Project #11.5: A playing card has two attributes, its *rank* (e.g., 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A) and its *suit* (e.g., clubs, diamonds, hearts, spades). Design and build a `PlayingCard` class that models a playing card. Your class should provide operations to construct, input, output, compare, and extract the instance variables of a `PlayingCard` object.

A deck of cards is simply a sequence of cards. Design and create a class `DeckOfCards` that represents such objects by using a `vector` to store a sequence of `PlayingCard` values. Your `DeckOfCards` class should provide operations to construct, shuffle, and take the top card. The class constructor should initialize the `DeckOfCards` as a new deck of 52 cards (i.e., 2-clubs, 3-clubs, ..., A-clubs, 2-diamonds, ..., A-diamonds, 2-hearts, ..., A-hearts, 2-spades, ..., A-spades). The shuffle operation should rearrange the cards in a deck in random order. The final operation should remove the top card from the deck, and return that card.

To test your classes, write a program that plays a simple card game (i.e., blackjack, go fish, etc.) against a human opponent.

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

Experiment 12: Enumerations

1. Objective of the Experiment

- To learn about enumerations.
- To practice writing library functions.
- To learn about automatic code generators.

2. Theoretical Background

Introduction

It is often the case that a programmer needs to represent some real-world object whose possible values are non-numeric. For example:

- **Gender**: *Female*, *Male*.
- **Color**: *Red*, *Orange*, *Yellow*, *Green*, *Blue*, *Indigo*, *Violet*.
- **Day of the Week**: *Monday*, *Tuesday*, *Wednesday*, *Thursday*, *Friday*, *Saturday*, *Sunday*.
- **Months of the Year**: *January*, *February*, *March*, *April*, *May*, *June*, *July*, *August*, *September*, *October*, *November*, *December*.
- **Sports Shoe Manufacturers**: *Adidas*, *Asics*, *Converse*, *Dunlop*, *...*
- ...

One way to represent such values is using the `string` type:

```
cout << "\nEnter your gender: ";
string userGender;
cin >> userGender;
if (userGender == "male")
   // male computations
else if (userGender == "female")
  // female computations
else
  // error
```

For a few applications, this approach is acceptable. However, while it's easy to implement, it's not particular efficient. Comparing two `string` values is a *slow* operation since, in the worst case, *every* character in the two strings must be examined:

```
bool operator==(const string & str1, const string & str2)
{
  if (str1.size() != str2.size())
    return false;
  else
  {
    for (int i = 0; i < str1.size(); i++)
      if (str1[i] != str2[i])
        return false;
    return true;
  }
}
```

The actual definition of this operator in the `string` library might be a bit different, but it's close to this one.

Let's consider the work involved in the test `(userGender == "male")`:

1. The literal `"male"` is converted to a `string` using the `string` constructor function, which makes a copy of the literal using a loop to do the copying.
2. If the sizes are different, another loop is needed to compare all of the characters in the two `string`s.
3. Plus, there are a whole bunch of book-keeping operations, but these are few compared to the two loops.

That's *two* loops, going through the entire `string`s. That's a lot of work for seeing if the gender is male.

In contrast, consider an equality test involving integers, `(integer == 5)`:

1. Call the basic machine instruction comparing `integer` and `5`.

There's no question which one is faster. The integer comparison is a single machine instruction, but the `string` comparison is many, many, many machine instructions depending on the size of the `string`s.

So how can we use integer comparisons for this type of data? That's the question we answer in this lab.

Files

Directory: `lab12`

- `Gender.h`, `Gender.cpp`, and `Gender.doc` implement a gender enumeration.
- `driver.cpp` implements a driver.
- `enumGenerator.cpp` implements an enumeration generator (covered later in the lab).
- `Makefile` is a makefile.

Create the specified directory, and copy the files above into the new directory. Only `gcc` users need a makefile; all others should create a project and add all of the `.cpp` files to it.

Add your name, date, and purpose to the opening documentation of the code and documentation files; if you're modifying and adding to the code written by someone else, add your data as part of the file's modification history.

This lab's exercise has eight parts. The program in `driver.cpp` tests each part. Open `driver.cpp` and take a few moments to study it, noting that most of the program is at present commented out.

a) Part I: Declaring An Enumeration Type

To provide a way to use real-world names in a program without using a `string`, C++ allows a programmer to declare an **enumeration type**. As its name implies, an enumeration is a type in which the programmer **enumerates** (i.e., exhaustively lists) all of the values for that type.

For example, suppose that we want to create a new type named **Season**, with the values *Spring*, *Summer*, *Autumn* and *Winter*. We can do so be creating a `Season` enumeration, as follows:

```
enum Season { SEASON_UNDERFLOW, SPRING, SUMMER,
              AUTUMN, WINTER, SEASON_OVERFLOW   };
```

With this statement, we are declaring the identifier `Season` as the name of a new data type whose valid values are `SEASON_UNDERFLOW`, `SPRING`, `SUMMER`, `AUTUMN`, `WINTER`, and `SEASON_OVERFLOW`.

The "overflow" and "underflow" values provide values for error-handling and other unusual situations. We don't necessarily want to use them in our programs, but they'll be useful for detecting and handling error situations.

Given such a declaration, a programmer can write this:

```
Season aSeason = SPRING;
```

This declares a variable named `aSeason` as type `Season` and initializes it to `SPRING`.

Since an enumeration is not a `string`, *no quotation marks are needed when you use an enumeration value* like `SPRING`, which is called an **enumerator**. Since they are essentially constant values, it is conventional to write enumerators in uppercase.

The general pattern for declaring an enumeration with N values is thus:

```
enum NewType { Value1, Value2, ..., ValueN };
```

This declaration creates *NewType* as the name of a new type whose valid values are *Value*$_1$, *Value*$_2$, ..., *Value*$_N$, which must be valid C++ identifiers, using the same conventions as for constant identifiers.

Open `Gender.h` and replace the appropriate comment with a statement that declares a new type named `Gender`, whose valid values are `GENDER_UNDERFLOW`, `FEMALE`, `MALE`, `UNKNOWN`, and `GENDER_OVERFLOW`. To check the syntax of what you have written, uncomment the declaration statement in `driver.cpp` and compile your code. When it is syntactically correct, continue to the next part.

b) Part II: The Input Operation

In order to input an enumeration, it is important to recognize that the input operator (>>) only works for a type if it has been defined for that type. That is, we cannot simply write

```
cin >> aGender;
```

This operator has not yet been defined. We can, however, provide such a definition.

It is important to remember that an `istream` like `cin` delivers a stream of *characters*. That is, we must read an enumerator as a sequence of characters, i.e., a `string`, and convert that `string` into the corresponding enumerator. The specification of our enumeration-input function is as follows:

Specification:
receive: *`in`*, an `istream`;
`value`, an enumeration variable.
precondition: *`in`* contains the `string` of a valid enumerator.
input: the enumerator-string from *`in`*.
passback: *`in`*, with the enumerator-string extracted from it; *`value`*, containing the corresponding enumerator.
return: *`in`*, for chaining.

Our function reads a `string` corresponding to an enumerator from *`in`*, determines the corresponding enumerator value, and passes back that enumerator value via the *`value`* parameter.

Here is an algorithm to solve this problem for our `Season` enumeration:

```
1. Receive in and value.
2. Read aString from in.
3. Convert the characters in aString to upper-case, if need be.
4. If aString equals "SPRING":
      value = SPRING.
   Else if aString equals "SUMMER":
      value = SUMMER.
   Else if aString equals "FALL" or aString equals "AUTUMN":
      value = AUTUMN.
   Else if aString equals "WINTER":
      value = WINTER.
   Else
      Display error message and terminate.
   End if.
5. Return in.
```

Since we are comparing `string` values, we *cannot* use a `switch` statement, but must instead use a multi-branch `if` to implement this algorithm. The function can be defined as follows:

```
istream & operator>>(istream & in, Season & value)
{
  string aString;
  in >> aString;
  for (int i = 0; i < aString.size(); i++)
    if (islower(aString[i]))
      aString[i] = toupper(aString[i]);

  if (aString == "SPRING")
    value = SPRING;
  else if (aString == "SUMMER")
     value = SUMMER;
  else if (aString == "AUTUMN" || aString == "FALL")
     value = AUTUMN;
  else if (aString == "WINTER")
     value = WINTER;
  else
  {
    cerr << "\n*** Invalid enumerator: " << aString
         << " received by >>\n" << endl;
    exit(1);
  }

  return in;
}
```

Note in particular the difference between a `string` literal and an enumerator. The C++ compiler uses the double-quotes surrounding `"SPRING"` to distinguish it from an enumerator like `SPRING`. If you were to try and assign `"SPRING"` to `value` (or assign `SPRING` to `aString`), the compiler would generate an error, since the types of the objects do not match. Those double quotes make a world of a difference.

You may notice, though, that we're back where we started: comparing `string`s. Didn't we say this was really time consuming? Well, yes, it is. However, input comes to us only as `char`s. Fortunately, `char`s can be turned into a `string` which, in turn, can be turned into an enumerator. The key for all this work is that we do this conversion from `string` to enumerator *once*, at input. From then on, our code will use the more efficient enumerator. The majority of useful programs spend most of their time in non-input functions, and that's where it's most important to have the more efficient enumerator.

Using this function definition as a model, define `operator>>` for your `Gender` enumeration in `Gender.cpp`, so that it implements the following algorithm:

```
1. Receive in and value.
2. Read aString from in.
3. Convert the characters in aString to upper-case, if need be.
4. If aString == "FEMALE":
      value = FEMALE.
   Else if aString == "MALE":
      value = MALE.
   Else if aString == "UNKNOWN":
      value = UNKNOWN.
   Else
```

```
        Display error message and terminate.
    End if.
5.  Return in.
```

Add a prototype for this function in the appropriate places in `Gender.h` and `Gender.doc`. Then uncomment the input step in `driver.cpp` and test your function by translating your driver program. When it is syntactically correct, then continue on.

c) Part III: The Output Operation

As with the input operator, the output operator cannot be applied to an object unless it has been defined for objects of that type. Actually, we *can* write

```
cout << aGender;
```

But this will not print out what we want. We can, however, provide a definition that will print out useful output.

Similar to an `istream`, an `ostream` like `cout` is a stream of *characters*. We must display an enumerator as a sequence of characters, i.e., a `string`. This implies that our function must display the `string` that corresponds to the enumerator it receives. The specification of our function is thus as follows:

Specification:
receive: *`out`*, an `ostream`, *`value`*, an enumeration variable.
precondition: *`value`* contains a valid enumerator.
output: the string corresponding to that enumerator, via *`out`*.
passback: *`out`*, containing the enumerator-string.
return: *`out`*, for chaining.

Our function must determine the `string` that corresponds to the enumerator it receives from the caller, and display that `string`. Here is an algorithm to solve this problem for our `Season` enumeration:

```
1.  Receive out and value.
2.  If value equals SPRING:
        Display "SPRING" via out.
    Else if value equals SUMMER:
        Display "SUMMER" via out.
    Else if value equals AUTUMN:
        Display "AUTUMN" via out.
    Else if value equals WINTER:
        Display "WINTER" via out.
    Else
        Display error message and terminate.
    End if.
3.  Return out.
```

Since we are comparing enumerator values, which *are* integer-compatible, we *can* use a `switch` statement to implement this algorithm. So, here's our function:

```
ostream & operator<<(ostream & out, const Season value)
{
  switch(value)
  {
  case SPRING:
    out << "SPRING";
    break;
  case SUMMER:
    out << "SUMMER";
    break;
  case AUTUMN:
    out << "AUTUMN";
    break;
  case WINTER:
    out << "WINTER";
    break;
  default:
    cerr << "\n*** Invalid enumerator received by <<\n" << endl;
    exit(1);
  }
  return out;
}
```

The `Season` parameter is *not* passed by reference because it is a primitive type. It's perhaps *less* efficient to pass it by reference. This is unlike the class objects we passed into the output operator in the previous lab; then we *did* pass them by reference.

Using this definition as a model, define `operator<<` for your `Gender` enumeration in `Gender.cpp`, so that it implements the following algorithm:

```
1. Receive out and value.
2. If value equals FEMALE:
      Display "FEMALE" via out.
   Else if value equals MALE:
      Display "MALE" via out.
   Else if value equals UNKNOWN:
      Display "UNKNOWN" via out.
   Else
      Display error message and terminate.
   End if.
3. Return out.
```

Add a prototype for this function in the appropriate place in `Gender.h` and `Gender.doc`. Then uncomment the first and last output steps in `driver.cpp`, and test your function by translating your driver program. When it is syntactically correct, continue to the next part of the exercise.

d) Part IV: The Prefix Increment Operator

It is often convenient if we can increment an enumeration variable. For example, we might want to display the four seasons by writing

```
for (Season aSeason = SPRING; aSeason <= WINTER; ++aSeason)
  cout << aSeason << ' ';
```

To write a statement like this, we must provide a definition of the **prefix increment operator** `++`. We can specify its behavior as follows:

Specification:
receive: `value`, an enumeration object.
precondition: `value` contains a valid enumerator.
passback: `value`, containing the next enumerator in the enumeration (or its OVERFLOW value).
return: `value`.

The thing to remember about the prefix increment operator is that its return-value is the *incremented value*. (This is unlike the postfix increment operator that we'll see below.) An algorithm to perform this operation for our `Season` enumeration is as follows:

```
1. Receive value.
2. If value equals SPRING:
      value = SUMMER.
   Else if value equals SUMMER:
      value = AUTUMN.
   Else if value equals AUTUMN:
      value = WINTER.
   Else if value equals WINTER:
      value = SEASON_OVERFLOW.
   Else
      Display an error message and terminate.
   End if.
3. Return value.
```

Since we are comparing enumerators we can implement this algorithm using a `switch` statement:

```
Season operator++(Season & value)
{
  switch (value)
  {
  case SPRING:
    value = SUMMER;
    break;
  case SUMMER:
    value = AUTUMN;
    break;
  case AUTUMN:
    value = WINTER;
```

```
      break;
    case WINTER:
      value = SEASON_OVERFLOW;
      break;
    default:
      cerr << "\n*** Invalid enumerator received by prefix++\n" <<
endl;
      exit(1);
    }
    return value;
}
```

The algorithm for a version of this operation for the `Gender` enumeration is similar:

```
1. Receive value.
2. If value equals FEMALE:
      value = MALE.
   Else if value equals MALE:
      value = GENDER_OVERFLOW.
   Else
      Display an error message and terminate.
   End if.
3. Return value.
```

Implement this algorithm by defining `operator++` in `Gender.cpp`, and add prototypes of it to `Gender.h` and `Gender.doc`. Test what you have written by uncommenting the three lines in `driver.cpp` that refer to `gender1`. Then translate and run your program. Continue when it works correctly.

e) Part V: The Postfix Increment Operator

While overloading the prefix operator provides us with the means of incrementing an enumeration, it is also sometimes desirable to be able to use the **postfix increment operation**. For example, we might want to display the four seasons by writing

```
for (Season aSeason = SPRING; aSeason <= WINTER; aSeason++)
  cout << aSeason << ' ';
```

The syntactic difference here is that the ++ increment operator is *after* (i.e., "post") the `aSeason` variable.

For this to work, we must provide a definition of the postfix increment operator `++`, whose behavior is just slightly different from that of the prefix version:

Specification:
receive: *`value`*, an enumeration object.
precondition: *`value`* contains a valid enumerator.
passback: *`value`*, containing the next enumerator in the enumeration (or its OVERFLOW

value).
return: the original enumerator of *`value`*.

It's identical to the specification for the prefix increment operator expect for one slight change to the return specification: return the *original* value of *`value`*. That's because the postfix increments the value *after* the return value is determined.

An algorithm to perform this operation for our `Season` enumeration is thus slighly different:

```
1. Receive value.
2. Set savedValue to be value.
3. If value equals SPRING:
      value = SUMMER.
   Else if value equals SUMMER:
      value = AUTUMN.
   Else if value equals AUTUMN:
      value = WINTER.
   Else if value equals WINTER:
      value = SEASON_OVERFLOW.
   Else
      Display an error message and terminate.
   End if.
4. Return savedValue.
```

Defining this function poses a syntax problem in C++. We've already used this prototype for the prefix increment operator:

```
Season operator++(Season & value);
```

But the postfix increment operators should have the same parameter list! So, C++ cheats---it has to. As a special case for a postfix increment operator, we add an extra (anonymous) `int` parameter like so:

```
Season operator++(Season & value, int );
```

We can then define this function for our `Season` enumeration as follows:

```
Season operator++(Season & value, int )
{
  Season savedValue = value;
  switch (value)
  {
  case SPRING:
    value = SUMMER;
    break;
  case SUMMER:
    value = AUTUMN;
    break;
  case AUTUMN:
    value = WINTER;
    break;
  case WINTER:
    value = SEASON_OVERFLOW;
    break;
  default:
    cerr << "\n*** Invalid enumerator received by postfix++\n"
<< endl;
    exit(1);
  }
  return savedValue;
}
```

In odd situations like this where a parameter is needed, but not required by the function's definition, C++ allows us to forgo supplying a name for the parameter.

The algorithm for a version of this operation for the `Gender` enumeration is similar:

```
1. Receive value.
2. Set savedValue to value.
3. If value equals FEMALE:
      value = MALE.
   Else if value equals MALE:
      value = GENDER_OVERFLOW.
   Else
      Display an error message and terminate.
   End if.
4. Return savedValue.
```

Implement this algorithm to define a postfix version of `operator++` in `Gender.cpp`. Add prototypes of it to `Gender.h` and `Gender.doc`. Test what you have written by uncommenting the three lines in `driver.cpp` that refer to `gender3`. Then translate and run your program. Continue when it works correctly.

f) Part VI: The Prefix Decrement Operator

Now if we envision incrementing an enumeration, it seems reasonable that we'll also want to decrement it. We might want to display the four seasons in reverse order by writing

```
for (Season aSeason = WINTER; aSeason >= SPRING; --aSeason)
  cout << aSeason << ' ';
```

To write this code, we must provide a definition of the **prefix decrement operator `--`**. The decrement operator is a dual to the increment operator, so the prefix decrement operator should look at lot like the prefix increment operator, just changing increments to decrements and overflows to underflows.

Since we've already gone through the prefix increment operator, we're going to let you try the prefix decrement operator on your own. First, establish a specification for a prefix decrement operator for an enumeration:

Question #12.1: Write down the specification for the prefix decrement operator for an enumeration.

Now the algorithm:

Question #12.2: Write an algorithm for the prefix decrement operator for the `Gender` enumeration.

Implement this algorithm by defining `operator--` in `Gender.cpp`, and add prototypes of it to `Gender.h` and `Gender.doc`. Test what you have written by uncommenting the three lines in `driver.cpp` that refer to `gender2`. Translate and run your program. Continue when it works correctly.

g) Part VII: The Postfix Decrement Operator

Our final operation is the **postfix decrement operation**. For example, we might want to display the four seasons in reverse order by writing

```
for (Season aSeason = WINTER; aSeason >= SPRING; aSeason--)
  cout << aSeason << ' ';
```

Again, we've already seen the postfix increment operator, which is structurally similar (if not identical) to the postfix decrement operator. So the job is all yours:

Question #12.3: Write down the specification for the postfix decrement operator for an enumeration.

Now be a bit more specific for a `Gender` enumeration:

Question #12.4: Write an algorithm for the postfix decrement operator for the `Gender` enumeration.

Implement this algorithm in your code. This postfix decrement operator also takes an unused `int` parameter just like the postfix decrement operator.

h) Part VIII: A Code-Generating Tool for Enumerations

For most enumerations a programmer wants to use, the programmer follows these same steps:

1. Declare the enumeration.
2. Implement the input operation.
3. Implement the output operation.
4. Implement the prefix increment operation.
5. Implement the postfix increment operation.
6. Implement the prefix decrement operation.
7. Implement the postfix decrement operation.

Some enumerations may require additional operations (e.g., a `daysIn()` function would be useful for a `Month` enumeration), but these six operations are a minimal group needed by any enumeration.

The process of defining these operations for a given enumeration is a *mechanical* one. (Hey, implement a few more enumerations if you don't believe us.) Aside from the particular enumerator values, the algorithms for each of these operations follow exactly the same structure. This process is so mechanical, it is straightforward to write a program that, given a sequence of enumerators and the desired name of the enumeration, generates the C++ code to declare the enumeration type and its operations. In fact, some languages automatically define the operations for you by the compiler whenever you create a new enumeration.

We have written a little program in `enumGenerator.cpp` that, given the name of an enumeration and a sequence of enumerators stored (one per line) in a file, generates the header, implementation, and documentation files for that enumeration.

Save a copy of `enumGenerator.cpp` in your directory and translate it.

Use a text editor to create an input file `daysofweek.txt` containing the days of the week:

```
sunday
monday
tuesday
wednesday
thursday
friday
Saturday
```

Run `enumGenerator`; name the enumeration `Day` and use `daysoftheweek.txt` as the input file. `enumGenerator` should then generate the files `Day.h`, `Day.cpp`, and `Day.doc`. Take a few moments to look over these files, and see how much code was just automatically generated for you!
Create a driver to test out the day tester (perhaps transliterate the driver for the `Gender` enumeration).
You may wish to study `enumGenerate.cpp` to see how it does what it does. You will likely run into similar situations in the future where taking the time to write a code-generating program will be a worthwhile investment of your time.
Remember, whenever you find yourself doing the same thing over and over, look for a better way, often one that the computer can do for you. Automated code generators of this sort are one way to solve such problems.
Submit
Submit all of the code for the `Gender` and `Day` enumerations (but not `enumGenerate.cpp`). Also turn in a sample execution of both drivers.
Terminology
enumerate, enumeration type, enumerator, prefix increment operatör

3. Project 12

Your instructor will assign you one of the problems below. To solve your problem, write a program that reads the necessary information to compute and output the indicated values, as efficiently as possible. Following the pattern in the lab exercise, first, *design* using OCD; then *code* your design in C++ using stepwise translation; finally, test your program thoroughly.

Project #12.1: Extend your `Gender` enumeration with these operations:

- `Next(Gender value);` that returns the enumerator that follows `value`, wrapping around from the last valid enumerator to the first valid enumerator---returns `FEMALE` when `value` is `UNKNOWN`, returns `MALE` when `value` is `FEMALE`, and returns `UNKNOWN` when `value` is `MALE`.
- `Previous(Gender value);` that returns the enumerator that precedes `value`, wrapping around from the first valid enumerator to the last valid enumerator---i.e., returns `FEMALE` when `value` is `MALE`, returns `MALE` when `value` is `UNKNOWN`, and returns `UNKNOWN` when `value` is `FEMALE`.

Write a driver program that tests your functions.

Project #12.2: Build a `Month` enumeration. Using it, create a `Date` class that stores the month, day, and year for a date. Test your class by writing a program that reads two dates, and returns the number of days between those two dates (don't forget to consider leap years).

Project #12.3: Extend the classes in `enumGenerator.cpp` to automatically generate the `Next()` and `Previous()` functions described in the previous project for *any* enumeration.

Project #12.4: Build two enumerations:

- **Suit**, containing the values *Clubs*, *Diamonds*, *Hearts* and *Spades*.
- **Rank**, containing the values *Ace*, *Two*, *Three*, *Four*, *Five*, *Six*, *Seven*, *Eight*, *Nine*, *Ten*, *Jack*, *Queen*, *King*.

Use these enumerations to design and build a `PlayingCard` class. Using this class and `vector`, build a `DeckOfCards` class. Provide the following operations:

- A constructor that creates a new (in-order) deck of cards.
- A `Shuffle()` operation that arranges the `PlayingCard`s in a `DeckOfCards` in random order.
- A `TopCard()` operation that returns the top card in the `DeckOfCards()` without removing it.
- A `DealTopCard()` operation that returns the top card in the `DeckOfCards()` and removes it.

Use these classes to write a program that plays a simple card game, such as "Go Fish."

Turn In

Turn the following things:

1. Your OCD.
2. Your source program.
3. The output from an execution of your program.

SOURCES

https://cs.calvin.edu/activities/books/c++/intro/3e/HandsOnC++/