First C++ Programs

http://people.sc.fsu.edu/∼jburkardt/isc/week02/
lecture_03.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 17 May 2011

- **Introduction**
- The HELLO Program
- Add Integers
- Output Redirection
- Conclusion
- In Class Exercise #2

INTRO: A Closer Look at HELLO and ADD_INTS

We have seen that a basic C++ program includes some initial material we don't understand, and a main function, which seems to be where the action is.

We saw how we could put a print statement inside the main function to say "Hello", and in the lab exercise, you typed in a program to add two integers.

Today we'll summarize some of the C++ features we have already been using, and try to understand the rules they follow.

INTRO: Numeric Computations, Variables, Formulas

We will especially want to understand calculations with numbers.

When we want a computer to carry out a numeric computation, the text of the program contains formulas such as:

```
x = y + z;
```

The symbols in the formula refer to variables, that is, names for numbers. We will look at rules for the names of variables, the kinds of numbers they can hold, and the way they can be combined in formulas.

We will also find out the rules for doing arithmetic, and for checking whether certain conditions are true or not.

We will also look at how input commands can be read from a file, and how output data can be written to a file, using the simple idea of redirection.

We saw that the C++ operators **cout** and **cin** can write messages and numbers to the screen, or accept input from the keyboard.

```
cout << "Enter second integer:  ";   <-- (output)
cin >> number2;     <-- (read input from keyboard)
```

In the lab exercise, you were encouraged to save your output with a command like

```
./a.out > results.txt
```

We will look at when it makes sense to do this, and reasons why you might also want to have the program read input from a file, rather than having you type it in.

HELLO: The Source Code

The hello program is stored in the file **hello.cpp**

```
# include <cstdlib>
# include <iostream>

using namespace std;

int main ( )
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

HELLO: # include <iostream>

When you write a program in the C++ language, there are many tools available to help you. The include statements request tools that you expect to use in your program.

Include statements are an example of **preprocessor** statements: they begin with a **#** sign, which means the preprocessor "rewrites" those lines. In this case, it replaces the request to include material by the text of the material.

If your program gets input from the user or prints output, you want to include the tools in iostream, such as:

- **cout**: for "standard" output;
- **cin**: for "standard" input;
- **cerr**: for output that can't be ignored or redirected;
- **endl**: for the end-of-line marker;
  (you can also use **n** for this purpose);

HELLO: # include <cstdlib>

The cstdlib contains so many useful tools that I *usually* include it in my programs, even though in some very simple programs it may not be needed.

DO THIS NOW: Does **hello.cpp** actually need **cstdlib**?

By including the **cstdlib**, we are able to use things like

- **atof()** and **atoi()**, which extract real or integer numbers from a string;
- **exit()**, which terminates a program early;
- **qsort()**, which sorts a list;
- **rand()**, which returns random numbers;

HELLO: using namespace std;

When we use an **include** statement to access C++ tools, the names come with a prefix of **std::** which is meant to indicate where they came from, and to avoid conflict with any names we were already using.

So, strictly speaking, the printing line in our program should read:

```
std::cout << "Hello, world!" << \n;
...or....
std::cout << "Hello, world!" << std::endl;
```

But the statement

```
using namespace std;
```

means that we promise not to use any variable names that are the same as the names in the include files; in exchange for that promise, we can drop the **std::** prefix.

HELLO: The Main Function

A program can include several functions, but must always have one called **main**. That is where the program will start.

The declaration line **int main ()** announces the beginning of a function whose name is **main**. The **int** indicates that the result returned by the function will be an integer. The parentheses **()** hold the input; in this case, there isn't any.

```
int main ( )
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

(The main function is always required to return an integer, and usually that integer is zero. Things get more interesting when we add more functions.)

HELLO: The Main Function

After the declaration line comes the body of the function, which must begin and end with a pair of curly brackets: { and }.

The body of this function contains statements. Each statement ends with a semicolon. When the program is executed, the first statement is carried out, then the next one and so on. The statement **return 0** means that the computation is complete, and that the value that the function should return is the integer 0.

```
int main ( )
{
  cout << "Hello, world!" << endl;
  (room for many more statements here)
  return 0;
}
```

HELLO: The Shortest C++ Program

The **wc** command (which stands for "word count"), will report the number of lines, words, and characters in a file.

Let's measure the size of **hello.cpp**:

```
wc hello.cpp
  10  23  128  hello.cpp
```

So this particular version of the Hello program uses 128 characters.

We already saw that some lines were not needed in the file. What is the smallest possible C++ program you can make, by cutting things out of **hello.cpp**?

The answer will show us the simplest C++ program.

HELLO: The Shortest C++ Program

The simplest program I came up with looks like this:

```
main(){}
```

for a count of 9 characters (because we include the carriage return at the end).

This is a perfect outline of the simplest C++ program: a main program, with **()** for input, and {} to contain statements.

ADD: How Do We Store Data?

When you fill out a tax form, you often are told to enter numbers in box 1, 2 and 3. Then take the sum of boxes 1 and 2, multiply by 5% and put that in box 4, and so on. Each box is a place where we can store the next piece of data or the result of a calculation.

With a form, we only expect to fill in a box once. However, if we wanted to reduce the number of boxes we needed, we could reuse them, as long as the previous result was no longer needed. A program is going to need boxes of some kind to store data.

And what will the data look like? The simplest data would simply be literal values. That is, if we need to multiply two numbers, we "literally" place those values in the program:

```
cout << "  My tip should be " << 0.20 * 17.54 << "\n";
```

ADD: How Do We Store Data?

But in a complicated calculation, where the same number may be used many times, it's handy to replace the literal values by symbolic names. Then we can use the formula for any case we run into. In particular, the values we need to multiply may not be known when we write the program, but rather be calculated by the program when it runs.

```
tip = rate * bill;
```

So now **tip**, **rate** and **bill** are like the boxes in a form; they can hold values. Probably, we need to put values into **rate** and **bill**, but then the value of **tip** can be determined from those values.

Let us look at how C++ enables us to set up "boxes" for data, put numbers into them, combine them, and print them out.

ADD: Variable Names

In C++, quantities used to store values are called **variables**.

The first property of a variable is its name, such as **x** or **rate** or **profit** or **b12**. The name essentially sets aside a box where we can put values, or get them back.

- A variable name must begin with a letter.
- It can contain letters, digits or underscores.
- It should not be longer than 31 characters.

C++ reserves certain keywords which cannot be used as names. These include words such as **and**, **bool**, **false**, **float**, **int**, **new**, **true** and other words that have special meaning.

You might expect that **cout** is a keyword. It's not, because it's not part of the fundamental set of C++ names. It only comes in if we **include <iostream>**, and even then, it has a prefix of **std::** (unless we "promise" not to create a variable called **cout**).

ADD: Variable Types

The second property of a variable is its **type**.

Numeric variables can be integer, real or complex. We will start out working with integers, whose C++ type is called **int** and single precision real numbers, whose type is **float**. When we need more digits of precision, we will look at the **double** type.

If a variable is declared to be an integer, it can only hold whole number values. The storage for an **int** is limited. This means that on the computer, an **int** is limited to values between -2,147,483,648 and +2,147,483,647.

Later, we will see a **bool** type for logical variables, a **char** type for single characters, and a **string** type for strings of text.

ADD: Variable Types

The third property of a variable is its value. A variable is a place for storing values, but it is possible that no value has been assigned to it yet. Such a variable is called **uninitialized**.

A variable is typically given a value by an assignment statement; it can also be given an initial value when its name and type are declared.

If a variable is declared to be a float, then it can hold real number values, such as 3.14159265. We can assign the variable **x** to have this value using the statement

```
x = 3.14159265;
```

We can also use scientific notation for numbers very large or small.

```
y = 1.23E+6; <-- y = 1.23 * 1,000,000 = 1,230,000.
z = 4.7E-5;  <-- z = 4.7  * 0.00001   =         0.000047
```

ADD: Accuracy

Because the storage for a float is limited, a **float** never has more than about 8 digits of accuracy. A double can have about 16 digits.

This means that arithmetic is only approximately accurate, especially when you are adding relatively small **y** to a number **x**. And if **y** is really small, adding it to **x** will make no difference at all!

To see the gory details, let's declare a **float** variable, set it to 1.23456789, and add smaller and smaller values to it.

Normally, **cout** will only print about 6 digits of a number, but we want to see more. We can force **cout** to print more digits by using an **include** statement to add the <**iomanip**> library, and then using the **setprecision(12)** function to ask for 12 digits of output.

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

using namespace std;

int main ( )
{
  float x, y, z;
  x = 1.23456789;
  cin >> y;
  z = x + y;
  cout << setprecision(12) << x
       << " + " << y << " = " << z << "\n";
  return 0;
}
```

ADD: Declarations and Executables

Your numeric C++ program includes two types of statements:

- **declarations** assign type, name, possible initial values;
- **executables**, evaluate formulas using the variables, or print;

```
int main ()
{
  float bill = 17.54;  <-- (3 Declarations)
  float rate = 0.20;
  float tip;

  tip = rate * bill;   <-- (3 Executables)
  cout << "  My tip should be " << tip << "\n";
  return 0;
}
```

First C++ Programs

REDIRECT: The Output Redirection Command

The "standard output" from the C++ operator **cout**, such as the string "Hello, world!", normally appears on your screen.

When you run a program, you have the option of redirecting all the standard output to a file, using the symbol $>$:

```
./a.out > file.txt
```

C++ includes a second output operator, **cerr**, or "standard error", whose output *cannot be redirected*. This is typically used for important error messages, or other information that should always go to the screen. But otherwise, it can be used the same as **cout**:

```
cerr << "Hello, world!\n";
```

Text printed by **cerr** appears on the screen, even when you try to redirect it.

REDIRECT: The Sine Function

We can use the **cout** and **cerr** operators, and the Unix redirection symbol, to create a program that values of the sine function, writes them to a file, and tells the user that it was successful.

A good start for a plot is to make a table. Each "wiggle" of a sine curve has length $2\pi \approx 6.28$, so to make sure we see about 3 wiggles, we'll plot over the range $0 \leq x \leq 20$.

Most line graphs can be plotted with about 500 data points. So we need to generate that many equally spaced x values, compute the corresponding sine value, and print the pair of values.

The sine function is only available to us if we **include** the library <**cmath**>. Then, for any value **x**, we can compute the sine by a statement like:

```
y = sin ( x );
```

REDIRECT: The FOR Statement

To compute 500 equally spaced values between 0 and 20, I'll use the C++ **for** command, which we haven't officially talked about yet (so you're not responsible for it yet!).

The **for** statement will count from 0 to 500 (so I'll actually use 501 points!). If I start **x** at 0.0, and then add 0.04 each time, then 500 steps later I will be at $\mathbf{x} = 20$. Thus, the **for** loop allows me to do something many times, while only writing the statements once.

```
x = 0.0;
for ( i = 0; i <= 500; i = i + 1 )  <-- Repeat statements
{                                       in brackets 501 times.
  y = sin ( x );                        i=i+1 can also be
  cout << x << "   " << y << "\n";      written "i++"
  x = x + 0.04;
}
cerr << "Program wrote 501 points to the file.\";
```

REDIRECT: Execution, with Output to the Screen

If we compile and run **sine_table.cpp**, we get our table:

```
./a.out
0   0
0.04   0.0399893
0.08   0.0799147
...
19.9601   0.89593
20.0001   0.912977
Program wrote 501 points to file.
```

Notice that both **cout** and **cerr** sent output to the screen.

(Also notice that 0.04 added 500 times gives me 20.0001, because of computer inaccuracies!)

REDIRECT: Execution, with Output Redirected to a File

But now let's redirect the program output to a file:

```
./a.out > sine_table.txt
Program wrote 501 points to file.
```

The useful message still prints to the screen, because **cerr** output can't be redirected. But we used **cout** for all the data, so that all went to the file.

You can examine the file using **gedit** or **kedit** from the program menu, or any of these terminal commands:

```
gedit sine_table.txt   <-- starts an editor
kedit sine_table.txt   <-- starts an editor
more sine_table.txt    <-- types out 20 lines at a time.
                           Terminate with 'q'
```

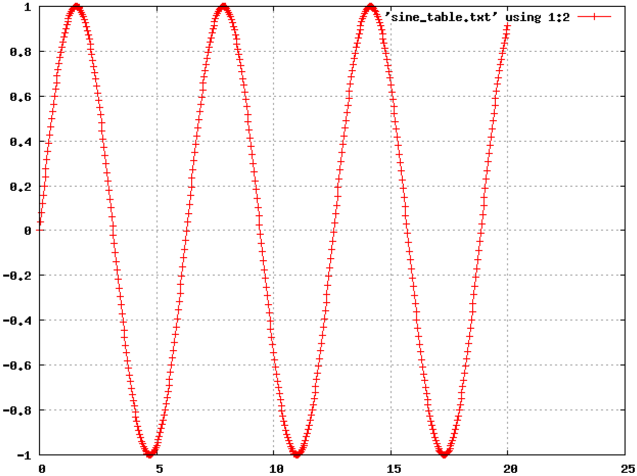
REDIRECT: Interactive Display

The reason we want our data in a file is that there are programs that can make a plot of it.

In particular, if the file **sine_table.txt** contains two columns of data separated by spaces (it does), then we make a nice plot by starting the **gnuplot** command and telling it to look at the file:

```
gnuplot
  set title "Sine Curve"           <-- label the plot
  set grid                         <-- draw grid
  set style data linespoints       <-- mark data points.
  plot "sine_table.txt" using 1:2  <-- X in column 1,
                                       Y is column 2
  quit                             <-- end gnuplot
```

REDIRECT: The GNUPLOT Display



REDIRECT: Save Graphics to File

If we wanted to save our graph to a file, use:
the **set term** command to pick the kind of file,
the **set output** command to name it.

```
gnuplot
  set term png                        <-- create PNG file
  set output "sine_table.png"         <-- name of PNG file
  set title "Sine Curve"
  set grid
  set style data linespoints
  plot "sine_table.txt" using 1:2
  quit
```

To view the image:

eog sine_table.png

REDIRECT: Redirecting Input

You can also redirect the input to a program. Instead of reading from the keyboard, you can tell a program to read a file. This might seem a very strange thing to do.

But let's suppose we're not good at typing interactively, or we have a bad memory. Suppose we put all the commands to gnuplot into a file called **gnu_input.txt**.

Then to make the plot, all we have to do is start up gnuplot, but tell it to get the commands from the file, not from us. As long as the last command is **quit**, gnuplot will do what we asked and then terminate gracefully:

```
gnuplot < gnu_input.txt
```

REDIRECT: Redirecting Input

The best thing about this approach is that the next time you need to graph a function, you can simply find the **gnu_input.txt** file, change the name of the data file you want to plot, and run gnuplot with the same input file as before.

Of course, once you get the plot, you might want to change some details, such as the title, or the style of plotting, but having a way to save your input commands and reuse them can be a big help!

If you don't have a way of displaying simple graphics on your own computer, you might be interested in installing **gnuplot**.

More information is available at **http://www.gnuplot.info/**

First C++ Programs

CONCLUSION: Summary

Today, I think I've repeated some of the points made last week about the structure of a simple C++ program, but I wanted to make sure that I explained to you why every line in the programs was where it was.

We've also begun to understand how a C++ program must define variables, in order to do numeric computing.

We looked at how a C++ program can write its output to the screen, which can be redirected to a file, which is one way to create graphical output.

Along the way, we've accidentally seen **for** statements, the **cerr** output operator, the include libraries <**iomanip**> and <**cmath**>, and the operator **setprecision(12)** for getting more digits of output. We will have a chance later to try to examine these topics more carefully.

Conclusion: Open Lab

Detelina has open lab hours Wednesday from 11:00 to 12:15.

Although we don't have any programming homework assignments yet, this might be a good time to come in and ask for a demonstration of some of the software Detelina knows about, such as NetBeans.

Detelina can also help you install software on your machine similar to what we have in the lab.

Conclusion: Coming up

- Reading: Deitel and Deitel, Chapter 2.1-2.4
- Thursday: Arithmetic, Logic, Integration
- Thursday: First Programming Homework will be assigned.

First C++ Programs

- Introduction
- The HELLO Program
- Add Integers
- Output Redirection
- Conclusion
- **In Class Exercise #2**

EXERCISE: Create Sine Table and Plot It

For today's in class exercise, I would like you to

- Type in the sine table program;
- Compile and run the program, and redirect the data to a file.
- Use the gnuplot program to view the data.

EXERCISE: Remember the Steps!

I suggest that you create a directory called **week2**.

You can use an editor such as **kedit** or **gedit** to type in the file **sine_table.cpp**, which you should store in the **week2** directory.

Start a terminal program, and then issue commands like this:

```
cd week2
g++ sine_table.cpp
./a.out > sine_table.txt
```

EXERCISE: The Whole Program

```
# include <iostream>
# include <cmath>
using namespace std;
int main ()
{
  int i;
  float x, y;
  x = 0.0;
  for ( i = 0; i <= 500; i = i + 1 )
  {
    y = sin ( x );
    cout << x << "   " << y << "\n";
    x = x + 0.04;
  }
  cerr << "Wrote 501 points to file.\n";
  return 0;
```

EXERCISE: Using GNUPLOT

Once your data file has been created, start the **gnuplot** program to display the data:

```
gnuplot
  set title "Sine Curve"                  <-- label the plot
  set grid                                <-- draw grid
  set style data linespoints              <-- mark data points.
  plot "sine_table.txt" using 1:2         <-- X in column 1,
                                              Y is column 2
  quit                                    <-- end gnuplot
```

EXERCISE: Reporting

Once you get the plot to display, please let Detelina know so she can give you credit for the exercise!

That's all for today!