

Unit 1. Simple Programs

| | |
|---------------------------------------|-----|
| 1.0 First Program | 13 |
| 1.1 Output..... | 24 |
| 1.2 Input & Variables | 35 |
| 1.3 Expressions..... | 46 |
| 1.4 Functions..... | 58 |
| 1.5 Boolean Expressions | 76 |
| 1.6 IF Statements..... | 86 |
| Unit 1 Practice Test | 99 |
| Unit 1 Project : Monthly Budget | 101 |

1.0 First Program

Sue is home for the Christmas holiday when her mother asks her to fix a “computer problem.” It turns out that the problem is not the computer itself, but some data their bank has sent them. Instead of e-mailing a list of stock prices in US dollars (\$), the entire list is in Euros (€)! Rather than perform the conversion by hand, Sue decides to write a program to do the conversion. Without referencing any books (they are back in her apartment) or any of her previous programs (also back in her apartment), she quickly writes the code to complete the task.

Objectives

By the end of this class, you will be able to:

- Use the provided tools (Linux, emacs, g++, styleChecker, testBed, submit) to complete a homework assignment.
- Be familiar with the University coding standards (Appendix A. Elements of Style).

Prerequisites

Before reading this section, please make sure you are able to:

- Type the code for a simple program (Chapter 0.2).

Overview of the process

The process of turning in a homework assignment consists of several steps. While these steps may seem unfamiliar at first, they will be well-rehearsed and second-nature in a week or two. The lab assistants (wearing green vests in the Linux lab) are ready and eager to help you if you get stuck on the way. The process consists of the following steps:

1. Log into the lab
2. Copy the assignment template using `cp`
3. Edit your file using `emacs`
4. Compile the program using `g++`
5. Verify your solution with `testBed`
6. Verify your style with `styleChecker`
7. Turn it in with `submit`

This entire process will be demonstrated in “Example – Hello World” at the end of the chapter.

0. Login

All programming assignments are done on the Linux system. This includes the pre-class assignments, the projects, and the in-lab tests. You can either go to the Linux Lab to use the campus computers, or connect remotely to the lab from your personal computer. Either way, you will need to log in. If you have not done this in Assignment 0.0, please re-visit the quiz for the default password. The lab assistants will be able to help you reset your password if necessary. Please see Appendix C: Lab Help for a description of what the lab assistants can and cannot do.

It is worthwhile to set up your computer so you do not need to come to the lab to do an assignment. The method is different for a Microsoft Windows computer than it is for an Apple Macintosh computer.

Remote access for Windows computers

1. Download the tool called PuTTY

[Setup - PuTTY](#)

2. Go to the lab and read the IP address (four numbers separated by periods) from any machine in the lab. They are 157.201.194.201 through 157.201.194.210. This will be the physical machine you are accessing when using remote access
3. Boot PuTTY and type in your IP address from step 2 and the port 215. You might want to save this session so you don't have to keep typing the numbers in.
4. Select [OPEN]. After you specify your username and password, you are now logged into that machine.

Remote access for Macintosh or Linux computers

If you are on a Macintosh or a Linux computer, bring up a terminal window and type the following command:

```
ssh <username>@<ip> -p 215
```

If, for example, you want to connect to machine 157.201.194.230 and your username is “sam”, then you would type:

```
ssh sam@157.201.194.230 -p 215
```

For more information, please see:

[Setup - Terminal](#)

1. Copy Template

Once you have successfully logged into the Linux system (either remotely or in the Linux Lab), the next step is to copy over the template for the assignment. All the assignments for this class start with a template file which has placeholders for the assignment name and the author (that would be you!). This file, and all other files pertaining to the course, can be found on:

```
/home/cs124
```

The assignment (and project and test) template is located on:

```
/home/cs124/template.cpp
```

On the Linux system, we type commands rather than use the mouse. The command used to copy a file is called “cp”. The syntax for the copy command is:

```
cp <source file> <destination file>
```

If, for example, you were to copy the template from `/home/cs124/template.cpp` into `hw10.cpp`, you would type the following command:

```
cp /home/cs124/template.cpp hw10.cpp
```

Most Linux commands do not display anything on the screen if they were successful. You will need to do a directory listing (`ls`) to see if the file copied. A list of other common Linux commands are the following:

| | | |
|--------------------|---------------------------------|--|
| Navigation tools | <code>cd</code> | Change Directory |
| | <code>ls</code> | List information about file(s) |
| | <code>cat</code> | Display the contents of a file to the screen |
| | <code>clear</code> | Clear terminal screen |
| | <code>exit</code> | Exit the shell |
| Organization tools | <code>yppasswd</code> | Modify a user password |
| | <code>mkdir</code> | Create new folder(s) |
| | <code>mv</code> | Move or rename files or directories |
| | <code>rm</code> | Remove files |
| Programming tools | <code>emacs</code> | Common code editor |
| | <code>vi</code> | More primitive but ubiquitous editor |
| | <code>g++</code> | Compile a C++ program |
| Homework tools | <code>styleChecker</code> | Run the style checker on a file |
| | <code>testBed</code> | Run the test bed on a file |
| | <code>submit</code> | Turn in a file |

For more commands or more details on the above, please see [Appendix D: Linux and Emacs Cheat-Sheet](#).



Sue's Tips

Be careful how you name your files. By the end of the semester, you could easily get lost in a sea of files. Spend a few moments thinking of how you will organize all your files as this will be a useful practice for the remainder of your career.

2. Edit with Emacs

Once the template has been copied to your directory, you are now ready to edit your program. There are many editors to choose from. Some editors are specialized to a specific task (such as Excel and Photoshop). The editor we use for programming problems is specialized for writing code. There are many editors you may use, including emacs and vi. For help with common emacs commands, please see “[Appendix D: Linux and Emacs Cheat Sheet](#).”

If you would like to write a program in `hello.cpp`, you can use emacs to edit create and edit the file with:

```
emacs hello.cpp
```

This will start emacs with a blank document named `hello.cpp`. From here you can type anything you like. However, if you wish this program to function correctly, you need to type valid C++. For your first program, you can make it say “Hello World” as we need to do for the first assignment:

```

/*****
 * Program:
 *   Assignment 10, Hello World
 *   Brother Helfrich, CS124
 * Author:
 *   Sam Student
 * Summary:
 *   This program is designed to be the first C++ program you have ever
 *   written. While not particularly complex, it is often the most difficult
 *   to write because the tools are so unfamiliar.
 *****/

#include <iostream>
using namespace std;

/*****
 * Hello world on the screen
 *****/
int main()
{
    // display
    cout << "Hello World\n";

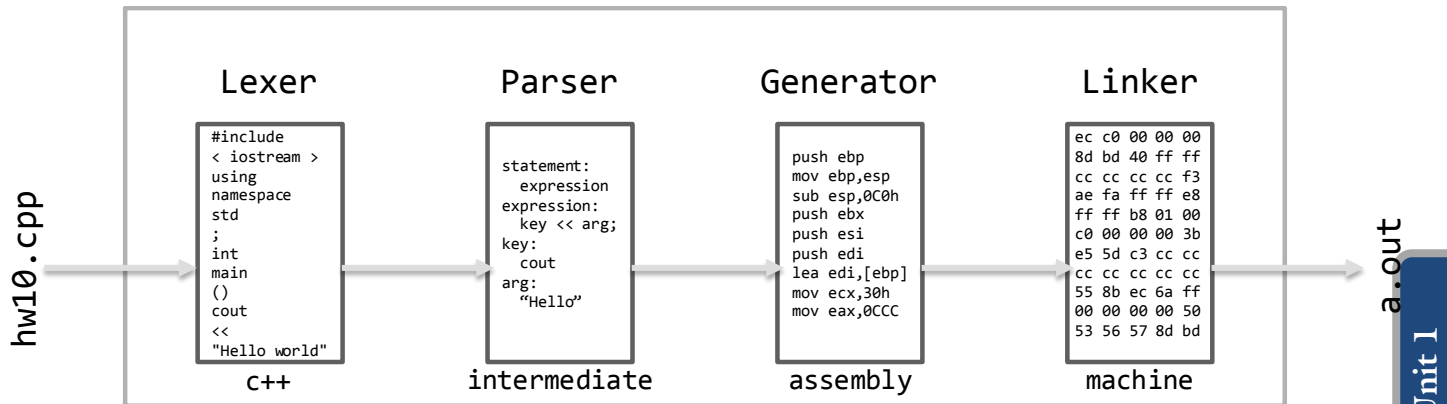
    return 0;
}

```

When you have finished writing the code for your program, save it and exit the editor. To save, first hit the `<control>` and `x` key at the same time, followed shortly with `<control>` and `s`. The shorthand for this key sequence is `C-x C-s`. You can then exit emacs with `C-x C-c`. More emacs keystrokes are presented in [Appendix D](#) at the back of this book.

3. Compile

After the program is saved in a file, the next step is compilation. Compilation is the process of translating the program from one format (C++ in this case) to another (machine language). This process is remarkably similar to how people translate text from French to English. There are four steps:



1. **Lexer:** Lexing is the process of breaking a list of text or sounds into words. When a non-speaker hears someone speak French, they are not even sure how many words are spoken. This is because they do not have the ability to lex. The end result of the lexing process is a list of tokens or words, each hopefully part of the source language.
2. **Parser:** Parsing is the process of fitting the words or tokens into the syntax of the language. In French, that is the process of recognizing which word is the subject, which is the verb, and which is the direct object. Once the process of parsing is completed, the listener understands not only what the words are, but what they mean in the context of the sentence.
3. **Generator:** After the meaning of the source language is understood through the parsing process, the next step is to generate text in the target language. In the case of the French to English translation, this means putting the parsed meaning from the French language into the equivalent English words using the English syntax. In the case of compiling C++ programs, the end result of this phase is assembly language similar to what we used in Chapter 0.2.
4. **Linker:** The final phase is to output the result from the code generator into a format understood by the listener. In the case of the French to English translation, that would involve speaking the translated text. In the case of compiling C++ code, that involves creating machine language which the CPU will be able to understand.

All four of these steps are done almost instantly with the compiler. The compiler we use in this class is `g++`. The syntax is:

```
g++ <source file>
```

If, for example, we are going to compile the file `hw10.cpp`, the following command will need to be typed:

```
g++ hw10.cpp
```

If the compilation is successful, then the file `a.out` will be created. If there was an error with the program due to a typographical error, then the compiler will state what the error was and where in the program the error was encountered.

4. Test Bed

After we have successfully passed the compilation process, it is then necessary to verify our solution. This is typically done in a two-step process. The first is to simply run the program by hand and visually inspect the output. To execute a newly-compiled program, type the name of the program in the terminal. Since the default name of a newly-compiled program is “a.out,” then type:

```
a.out
```

The second step in the verification process is to test the program against the key. This is done with a program called Test Bed. Test Bed compares the output of your program against what was expected. If everything behaves correctly, a message “No Errors” will be displayed. On the other hand, if the program malfunctions or produces different output than expected, then the difference is displayed to the user. In this way, Test Bed is a two-edged sword: you know when you got the right answer, but it is exceedingly picky. In other words, Test Bed will notice if a space was used instead of a tab even though it appears identical on the screen. The syntax for Test Bed is:

```
testBed <test name> <file name>
```

The first parameter to the Test Bed program is the test which is to be run. This test name is always present on a homework assignment, in-lab test, and project. The second parameter is the file you are testing. If, for example, your program is in the file `hw10.cpp` and the test is `cs124/assign10`, then the following code will be executed:

```
testBed cs124/assign10 hw10.cpp
```

It is important to note that you will not get a point on a pre-class assignment unless Test Bed passes without error.

5. Style Checker

Once the program has been written and passes Test Bed, it is not yet finished. Another important component is whether the code itself is human-readable and in a standard format. This is collectively called “style.” A programming style consists of many components, including variable names, indentations, and comments.

While style is an inherently subjective notion, we have a tool to help us with the process. This tool is called Style Checker. While Style Checker will certainly not catch all possible style mistakes, it will catch the most obvious ones. You should never turn in an assignment without running Style Checker first. The syntax for Style Checker is:

```
styleChecker <file name>
```

If, for example, you would like to run Style Checker on `hw10.cpp`, then the following command is to be executed.

```
styleChecker hw10.cpp
```

The main components to style include:

| | |
|-------------------------|--|
| Variable names | <p>Variable names should completely describe what each variable contains. Each should be camelCased: capitalize the first letter of every word in the name except the first word. We will learn about variables in Chapter 1.2:</p> <pre>numStudents</pre> |
| Function names | <p>Function names are camelCased just like variable names. Function names are typically verbs while variable names are nouns. We will learn about functions in Chapter 1.4.</p> <pre>displayBudget()</pre> |
| Indent | <p>Indentations are three spaces. No tabs please!</p> <pre>{ cout << "Hello world\n"; }</pre> |
| Line length | <p>Lines are no longer than 80 characters in length. If more space is needed for a comment, break the comment into two lines. The same is true for cout statements (Chapter 1.1) and function parameters (Chapter 1.4).</p> <pre>// Long comments can be broken into two lines // to increase readability. Start each new // line with “//”s</pre> |
| Program comments | <p>All programs have a program comment block at the beginning of the file. This can be found in the standard template. An example is:</p> <pre> /***** * Program: * Assignment 10, Hello World * Brother Helfrich, CS124 * Author: * Sam Student * Summary: * Display a message *****/ </pre> |
| Function comments | <p>Every function such as main() has a comment block describing what the function does:</p> <pre> /***** * MAIN * This program will display a simple message * on the screen *****/ </pre> |
| Space between operators | <p>All operators, such as addition (+) and the insertion operator (<<) are to have a single space on either side to set them apart:</p> <pre>sumOfSquares += userInput * userInput;</pre> |

For more details on the University's style guidelines, please see “Appendix A: Elements of Style” and look at the coding examples presented in this class.

6. Submit

The last step of turning in an assignment is to submit it. While we discuss this as the end of the homework process, you can submit an assignment as often as you like. In the case of multiple submissions, the last one submitted at the moment the assignment is graded is the one that will be used. It is therefore a good idea to submit your assignments frequently so your professor has the most recent copy of your work. The syntax for the program submission tool is:

```
submit <file name>
```

If, for example, your program is named “hw10.cpp,” then the following command is to be executed:

```
submit hw10.cpp
```

One word of caution with the Submit tool. The tool reads the program header to determine the professor name, the class number, and the assignment number. If any of these are incorrect, then the program will not be submitted correctly. For example, consider the following header:

```
/* *****  
 * Program:  
 *   Assignment 10, Hello World  
 *   Brother Helfrich, CS124  
 * Author:  
 *   Susan Bakersfield  
 * Summary:  
 *   This program is designed to be the first C++ program you have ever  
 *   written. While not particularly complex, it is often the most difficult  
 *   to write because the tools are so unfamiliar.  
 * ***** */
```

Here, Submit will determine that the program is an Assignment (as opposed to a Test or Project), the assignment number is 10, the professor is Br. Helfrich, and the class is CS 124. If any of these are incorrect, then the file will be sent to another location. To help you with this, submit tells the user what it read from the header:

```
submit homework to helfrich cs124 and assign10. (y/n)
```

It is worthwhile to read that message.

Sam's Corner



Submit is basically a fancy copy function. It makes two copies of the program: one for you and one for the instructor. If, for example, you submitted to “Assignment 10” for “CS 124”, then you will get a copy on.

```
/home/<username>/submittedHomework/cs124_assign10.cpp
```

Observe how the name of the file is changed to that of the assignment and class name. The second copy gets sent to the instructor. Here the filename is changed to the login ID. If, for example, your login is “eniacy”, then the file appears as eniacy.cpp in the instructor’s folder.

Please do not use a dot in the name of your file. If you submit hw1.0.cpp, for example, then it will appear as eniacy.0 instead of eniacy.cpp and the instructor will not grade it

Example 1.0 – Display “Hello World”

Demo This example will demonstrate how to turn in a homework assignment. All the tools involved in this process, including emacs, g++, testBed, styleChecker, and submit, will be illustrated.

Problem Write a program to prompt to display a simple message on the screen. This message will be the classic “Hello World” that we seem to always use when writing our first program with a new computer language.

The code for the solution is:

```

/*****
* Program:
*   Assignment 10, Hello World
*   Brother Helfrich, CS124
* Author:
*   Sam Student
* Summary:
*   This program is designed to be the first C++ program you have ever
*   written. While not particularly complex, it is often the most difficult
*   to write because the tools are so unfamiliar.
*****/

#include <iostream>
using namespace std;

/*****
* Hello world on the screen
*****/
int main()
{
    // display
    cout << "Hello World\n";

    return 0;
}

```

Of course the real challenge is using the tools...

Challenge As a challenge, modify this program to display a paragraph including your name and a short introduction. My paragraph is:

Hello, I am Br. Helfrich.

My favorite thing about teaching is interacting with interesting students every day. Some days, however, students have no questions and don't bother to come by my office. Those are long and lonely days...

See Also The complete solution is available at [1-0-firstProgram.cpp](#) or:

/home/cs124/examples/1-0-firstProgram.cpp



Problem 1

If your body was a computer, select all the von Neumann functions that the spinal cord would perform?

Answer:

Please see page 5 for a hint.

Problem 2

If a given processor were to be simplified to only contain a single instruction, which part would be most affected?

Answer:

Please see page 5 for a hint.

Problem 3

Which of the following does a CPU consume? {Natural language, C++, Assembly language, Machine }?

Answer:

Please see page 5 for a hint.

Problem 4

What is wrong with the following program:

```
#include <iostream>
using namespace std;

int main()
(
    cout << "Howdy\n";

    return 0;
)
```

Answer:

Please see page 7 for a hint.

Assignment 1.0

Write a program to put the text “Hello World” on the screen. Please note that examples of the code for this program are present in the course notes.

Example

Run the program from the command prompt by typing `a.out`.

```
$a.out
Hello World
$
```

Instructions

Please...

1. Copy template from: `/home/cs124/template.cpp`. You will want to use a command like:

```
cp /home/cs124/template.cpp assignment10.cpp
```

2. Edit the file using `emacs` or another editor of your choice. For example:

```
emacs assignment10.cpp
```

3. After you have typed your program, save it and compile with:

```
g++ assignment10.cpp
```

4. If there are no errors, you can run it with:

```
a.out
```

Please verify your solution against test-bed with:

```
testBed cs124/assign10 assignment10.cpp
```

5. Check the style to ensure it complies with the University’s style guidelines:

```
styleChecker assignment10.cpp
```

6. Turn your assignment in with the `submit` command. Don’t forget to submit your assignment with the name “Assignment 10” in the header

```
submit assignment10.cpp
```

Please see page 21 for a hint.

1.1 Output

Sam is sitting in the computer lab waiting for class to begin. He is bored, bored, bored! Just for kicks, he decides to dabble in [ASCII-art](#). His first attempt is to reproduce his school logo:

```
( _ \ ( _ \ ) ( _ ) ( _ )
| _ < \ / ) ( _ ) ( _ )
( _ / ( _ ) ( _ ) ( _ )
```

Objectives

By the end of this class, you will be able to:

- Display text and numbers on the screen.
- Left-align and right-align text.
- Format numbers to a desired number of decimal places.

Prerequisites

Before reading this section, please make sure you are able to:

- Type the code for a simple program (Chapter 0.2).
- Recite the major parts of a computer program (statements, headers, etc.) (Chapter 0.2).
- Use the provided tools to complete a homework assignment (Chapter 1.0).

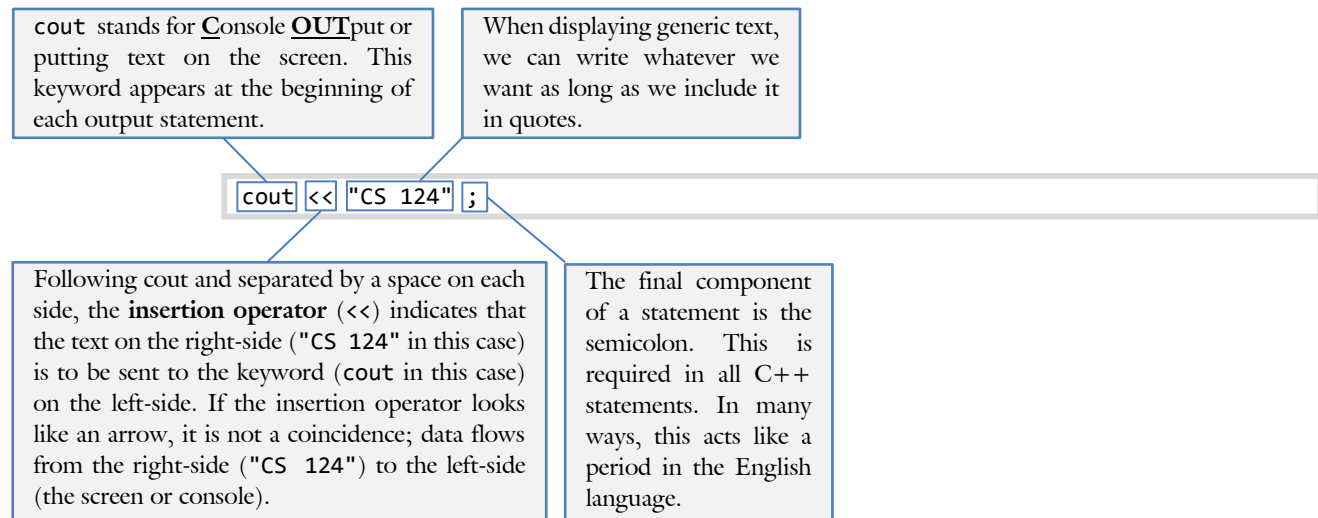
Overview of Output

There are two main methods for a computer to display output on the screen. The first is to draw the output with dots (pixels), lines, and rectangles. This is the dominant output method used in computer programs today. Any windowing operating system (such as Microsoft Windows or Apple Macintosh) favors programs using this method. While this does give the programmer maximum freedom to control what the output looks like, it is also difficult to program. There are dozens of drawing toolsets (OpenGL, DirectX, Win32 to name a few), each of which requires a lot of work to display simple messages.

The second method is to use streams. Streams are, in many ways, like a typewriter. An individual typing on a typewriter only needs to worry about the message that is to appear on the page. The typewriter itself knows how to render each letter and scroll the paper. A programmer using streams to display output specifies the text of the message as well as simple control commands (such as the end of the line, tabs, etc.). The operating system and other tools are left to handle the mechanics of getting the text to render on the screen. We will use stream output exclusively in CS 124.

COUT

As previously discussed, computer programs are much like recipes: consisting of a list of instructions necessary to produce some output. These instructions are called statements. One of the fundamental statements in the C++ language is `cout`: the statement that puts text on the screen. The syntax of `cout` is:



When you put this all together the above statement says "Put the text "CS 124" on the screen."

Displaying Numbers

Up to this point, all of our examples have been displaying text surrounded by double quotes. It is also possible to use `cout` to display numbers. Before doing this, we need to realize that computers treat integers (numbers without decimals) fundamentally differently than real numbers (numbers with decimals).

We can display an integer by placing the number after an insertion operator in a `cout` statement.

```
cout << 42;
```

Because this number is an integer, it will never be displayed with a decimal. On the other hand, if we are displaying a real number, then we add a decimal in the text:

```
cout << 3.14159;
```

In this example, the computer is not sure how many decimals of accuracy the programmer meant. To be clear on this point, it is useful to include the following code before displaying real numbers:

```
cout.setf(ios::fixed);           // no scientific notation please
cout.setf(ios::showpoint);       // always show the decimal for real numbers
cout.precision(2);               // two digits after the decimal
```

The first statement means we never want to see the number displayed in scientific notation. Unless the number is very big or very small, most humans prefer to see numbers displayed in "fixed" notation. The second statement indicates that the decimal point is required in all presentations of the number. The final statement indicates that two digits to the right of the decimal point will be displayed. We can specify any number of digits of course. Note that there is some interplay between these three statements; usually we use them together. These settings are "sticky." This means that once the program has executed these lines of code, all real numbers will be treated this way until the setting is changed again.

New Lines

Often the programmer would like to indicate that the end of a line has been reached. With a typewriter, one hits the Carriage Return to jump to the next line; it does not happen automatically. The same is true with stream output. The programmer indicates a newline is needed using two methods:

Both of these mean the same thing. They will output a new line to the screen.

```
cout << endl;  
cout << "\n";
```

The first method is called `endl`, short for “end of line.” This does not appear in quotes. Whenever a statement is executed with an `endl`, the cursor jumps down one line and moves to the left. The same occurs when the “`\n`” is encountered. Note that the `\n` must be in quotes. There can be many `\n`’s in a single run of text.

Observe that we have two different ways (`endl` and `\n`) to do the same thing. Which is best?

| | <code>endl</code> | <code>\n</code> |
|-------------------|---|--|
| Inside quotes | <pre>cout << "Hello"; cout << endl;</pre> | <pre>cout << "Hello\n";</pre> |
| Not inside quotes | <pre>cout << 5; cout << endl;</pre> | <pre>cout << 5; cout << "\n";</pre> |
| Use | When the previous item in the output stream is not in quotes, use the <code>endl</code> . | Most convenient when you want a newline and are already inside quotes. |

The Insertion Operator

As mentioned previously, the insertion operator (`<<`) is the C++ construct that allows the program to indicate which text is to be sent to the screen (through the `cout` keyword). It is also possible to send more than one item to the screen by stacking multiple insertion operators:

```
cout << "I am taking "  
    << "CS 124 "  
    << "this semester.\n";
```

By convention we typically align the insertion operators so they line up on the screen and are therefore easier to read. However, we may wish to put them in a single line:

```
cout << "I am taking " << "CS 124 " << "this semester.\n";
```

Both of these statements are exactly the same to the compiler; the difference lies in how readable they are to a human. There are three common reasons why one would want to use more than one insertion operator:

Set Width

Tabs work great for left-aligning text. However, often one needs to right-align text. This is performed with the `set width` command. `Set width` works by counting backwards from a specified numbers of spaces so the next text in the `cout` statement will be right-aligned. Consider the following code:

```
cout << setw(9) << "set\n";  
cout << setw(9) << "width\n";
```

The first statement will start at column zero, move 9 spaces to the right (by the number specified in the parentheses), then count to the left by three (the width of the word “set”). This means that the text “set” will start on column 6 (9 as specified in the `setw(9)` function minus 3 by the length of the next word). The next statement will again start at column zero (because of the preceding `\n`), move 9 spaces to the right, then count to the left by five (the width of the word “width”). This means that the text “width” will start on column 4 (9 minus 5). As a result, the two words will be right-aligned.

```
set
width
```

One final note: the `setw()` function is in a different library which needs to be included. You must `#include` the `iomanip` library:

```
#include <iomanip>
```

Sam's Corner

It turns out that there are many other formatting options available to programmers. You can output your numbers in hexadecimal, unset formatting flags, and pad with periods rather than spaces. Please see the following for a complete list of the options:

http://en.cppreference.com/w/cpp/io/ios_base.



Using Tabs and Set Width Together

Tabs and set width are commonly used together when displaying columns of figures such as money. Consider the following code:

```
#include <iostream>    // required for COUT
#include <iomanip>       // we will use setw() in this example
using namespace std;

int main()
{
    // configure the output to display money
    cout.setf(ios::fixed);    // no scientific notation except for the deficit
    cout.setf(ios::showpoint); // always show the decimal point
    cout.precision(2);        // two decimals for cents; this is not a gas station!

    // display the columns of numbers
    cout << "\t$" << setw(10) << 43.12 << endl;
    cout << "\t$" << setw(10) << 115.2 << endl;
    cout << "\t$" << setw(10) << 83299.3051 << endl;

    return 0;
}
```

In this example, the output is:

| | |
|----|----------|
| \$ | 43.12 |
| \$ | 115.20 |
| \$ | 83288.31 |

Observe how the second row displays two decimals even though the code only has one. This is because of the `cout.precision(2)` statement indicating that two decimals will always be used. The third row also displays two decimal places, rounding the number up because the digit in the third decimal place is a 5.



Sue's Tips

It is helpful to first draw out the output on graph paper so you can get the column widths correct the first time. When the output is complex (as it is for Project 1), aligning columns can become frustrating.

Special Characters

As mentioned previously, we always encapsulate text in quotes when using a `cout` statement:

```
cout << "Always use quotes around text\n";
```

There is a problem, however, when you actually want to put the quote mark (") in textual output. We have the same problem if you want to put the backslash \ in textual output. The problem arises because, whenever `cout` sees the backslash in the output text, it looks to the next character for instructions. These instructions are called **escape sequences**. Escape sequences are simply indications to `cout` that the next character in the output stream is special. We have already seen escape sequences in the form of the newline (\n) and the tab (\t). So, back to our original question: how do you display the quote mark without the text being ended and how do you display \n without a newline appearing on the screen? The answer is to escape them:

```
cout << "quote mark:\"    newline:\\n" << endl;
```

When the first backslash is encountered, `cout` goes into “escape mode” and looks at the next character. Since the next character is a quote mark, it is treated as a quote in the output rather than the marker for the end of the text. Similarly, when the next backslash is encountered after the newline text, the next backslash is treated as a backslash in the output rather than as another character. The output of the code would be:

```
quote mark:"    newline:\n
```

Up to this point, the following are the escape sequences we can use:

| Name | Character |
|--------------|-----------|
| New Line | \n |
| Tab | \t |
| Backslash | \\ |
| Double Quote | \" |
| Single Quote | \' |

There are many other lesser known and seldom used escape characters as well:

<https://msdn.microsoft.com/en-us/library/h21280bw.aspx>

Example 1.1 – Money Alignment

Demo

This example will demonstrate how to use tabs and `setw()` to align money. This is important in Assignment 1.1, Project 1, and many output scenarios.

Problem

Write a program to output a list of numbers on a grid so they can be easily read by the user.

| | | | | |
|----|---------|--|----|--------|
| \$ | 124.45 | | \$ | 321.31 |
| \$ | 1.74 | | \$ | 4.21 |
| \$ | 7439.12 | | \$ | 54.92 |

Diagram illustrating the alignment of numbers on a grid. The numbers are displayed in two columns, separated by a tab. The first column contains three numbers: \$ 124.45, \$ 1.74, and \$ 7439.12. The second column contains three numbers: \$ 321.31, \$ 4.21, and \$ 54.92. The numbers are right-aligned within their respective columns. The width of each column is 7 spaces, and there is 1 tab between the columns.

The first part of the solution is to realize that all the numbers are displayed as money. This requires us to format `cout` to display two digits of accuracy.

```
cout.setf(ios::fixed);           // no scientific notation
cout.setf(ios::showpoint);       // always show the decimal point
cout.precision(2);               // two digits for money
```

After the leading \$ the text is right-aligned to seven spaces. This will require code something like:

```
cout << "$" << setw(7) << 124.45;    // numbers not in quotes!
```

Following the first set numbers, we have another column separated by a tab.

```
cout << "\t";
```

Next, another column of numbers just like the first.

```
cout << "$" << setw(7) << 321.31;    // again, the numbers are not in quotes
```

Finally, we end with a newline

```
cout << endl;                        // instead, we could say "\n"
```

Put it all together:

```
// display the first row
cout << "$"
    << setw(7) << 124.45
    << "\t$"
    << setw(7) << 321.31
    << endl;
```

Challenge

As a challenge, try to increase the width of each column from 7 spaces to 10. How does this change the space between columns? Can you add a third column of numbers?

Finally, what is the biggest number you can put in a column before things start to get “weird.” What happens when the numbers are wider than the columns?

See Also

The complete solution is available at [1-1-alignMoney.cpp](#) or:

```
/home/cs124/examples/1-1-alignMoney.cpp
```



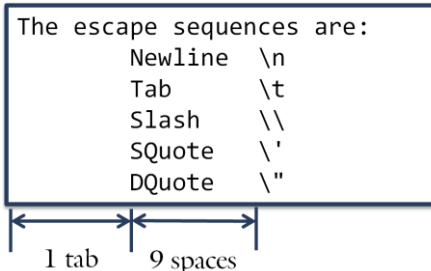
Example 1.1 – Escape Sequences

Demo

This example will demonstrate how to display special characters on the screen using escape sequences. Not only will we use escape sequences to get tabs and newlines on the screen, but we will use escapes to display characters that are normally treated as special.

Problem

Write a program to display all the escape sequences in an easy-to-read grid.



Solution

We need to start by noting that there are six lines in the output so we should expect to use six `\n` escape sequences. If we do not end each line with a newline, then all the text will run onto a single line.

Next, there needs to be a tab before each of the five lines in the list. This will be accomplished with a `\t` escape sequence. Each of the slashes in the escape sequence will need to be escaped. Consider the following code:

```
cout << "\tNewline \n\n";
```

This will result in the following output:

```
Newline
```

Notice how the “`\n`” was never displayed and we have an extra blank line. Instead, the following will be necessary:

```
cout << "\tNewline  \n\n";
```

Here, after the “Newline” text, the first “`\`” will indicate that the second is not be treated as an escape. The end result will display a `\` on the screen. Next the “`n`” will be encountered and displayed. The final “`\`” will indicate the following character is to be treated special. That character, the “`n`” will be interpreted as a newline.

The final challenge is the double quote at the end of the sequence. It, too, will need to be escaped or the compiler will think we are ending a string.

Challenge

As a challenge, try to reverse the order of the text so the escape appears before the label. Then try to right-align the label using `setw()`:

```
\n      Newline
\t      Tab
```

See Also

The complete solution is available at [1-1-escapeSequence.cpp](#) or:

```
/home/cs124/examples/1-1-escapeSequence.cpp
```



Problem 1

Write the code to put a newline on the screen:

Answer:

Please see page 26 for a hint.

Problem 2

How do you right-align numbers in C++?

```
5
555
```

Answer:

Please see page 28 for a hint.

Problem 3

If the tab stops are set to 8 spaces, what will be the output of the following code?

```
{
    cout << "\ta\n";
    cout << "a\ta\n";
}
```

Answer:

Please see page 27 for a hint.

Problem 4

Write the code to generate the following output:

```
/\\//\
\\//\
```

Answer:

Please see page 29 for a hint.

Problem 5

What is the output of the following code?

```
{  
    cout << "\t1\t2\t3\n\t4\t5\t6\n\t7\t8\t9\n";  
}
```

Answer:

Please see page 29 for question.

Problem 6

Write a program to put the following text on the screen:

```
I am taking  
    "CS 124"
```

Note that there is a tab at the start of the second line.

Answer:

Please see page 27 for a hint.

Problem 7

Write the code to generate the following output:

```
Bill:  
    $ 10.00 - Price  
    $  1.50 - Tip  
    $ 11.50 - Total
```

Answer:

Please see page 30 for a hint.

Assignment 1.1

Write a program to output your monthly budget:

| Item | Projected |
|---------|-----------|
| Income | \$1000.00 |
| Taxes | \$100.00 |
| Tithing | \$100.00 |
| Living | \$650.00 |
| Other | \$90.00 |

Example

```
Item          Projected
=====
Income        $ 1000.00
Taxes         $  100.00
Tithing       $   100.00
Living        $   650.00
Other         $    90.00
=====
Delta         $    60.00
```

Instructions

Please note:

- There is a single tab at the start of each line, but nowhere else.
- There are 13 '='s in the first column, 10 in the second. There are 2 spaces between the columns.
- The spacing between the '\$' and the right edge of the money is 9.
- You will need to set the formatting of the prices with the `precision()` command.
- Please display the money as a number, rather than as text. This means two things. First, the numbers should be outside the quotes (again, see the example above). Second, you will need to use the `setw()` function to get the numbers to line up correctly.
- Please verify your solution against:

```
testBed cs124/assign11 assignment11.cpp
```

Don't forget to submit your assignment with the name "Assignment 11" in the header.

Please see page 30 for a hint.

1.2 Input & Variables

Sue is excited because she just got a list of ancestor names from her grandmother. Finally, she can get some traction on her genealogy work! Unfortunately, the names are in the wrong order. Rather than being in the format of [LastName, FirstName MiddleInitial], they are [FirstName MiddleInitial LastName]. Instead of retyping the entire list, Sue writes a program to swap the names.

Objectives

By the end of this class, you will be able to:

- Choose the best data-type to represent your data.
- Declare a variable.
- Accept user input from the keyboard and store it in a variable.

Prerequisites

Before reading this section, please make sure you are able to:

- Type the code for a simple program (Chapter 0.2).
- Use the provided tools to complete a homework assignment (Chapter 1.0).
- Display text and numbers on the screen (Chapter 1.1).

Overview

Variables in computer languages are much like variables in mathematics:

Variables are a named location where we store data

There are two parts to this definition. The first part is the name. We always refer to variables by a name which the programmer identifies. It is always worthwhile to make the name as unambiguous as possible so it won't get confused with other variables or used later in the program. The second part is the data. A wide variety of data-types can be stored in a variable.

Variables

All the data in a computer is stored in memory. This memory consists of collections of 1's and 0's which are meant to represent numbers, letters, and text. There are two main considerations when working with variables: how to interpret the memory into something (like the number 3.8 or the text "Computer Science"), and what that something means (like your GPA or your major).

There is no intrinsic meaning for these 1's and 0's; they could mean or refer to just about anything. It is therefore the responsibility of the programmer to specify how to interpret these 1's and 0's. This is done through the data-type. A data-type can be thought of as a formula through which the program interprets the 1's and 0's in memory. An integer number, for example, is interpreted quite differently than a real number or a letter. Every computer has a built-in set of data-types facilitating working with text, numbers, and logical data. C++ facilitates these built-in data-types with the following type names:

| Data-type | Use | Size | Range of values |
|-----------------|-----------------------------------|------|---|
| bool | Logic | 1 | true, false |
| char | Letters and symbols | 1 | -128 to 127 ... or 'a', 'b', etc. |
| short | Small numbers, Unicode characters | 2 | -32,767 to 32,767 |
| int | Counting | 4 | -2 billion to 2 billion |
| long (long int) | Larger Numbers | 8 | $\pm 9,223,372,036,854,775,808$ |
| float | Numbers with decimals | 4 | 10^{-38} to 10^{38} accurate to 7 digits |
| double | Larger numbers with decimals | 8 | 10^{-308} to 10^{308} accurate to 15 digits |
| long double | Huge Numbers | 16 | 10^{-4932} to 10^{4932} accurate to 19 digits |

Thus when you declare a variable to be an integer (int), the 1's and 0's in memory will be interpreted using the integer formula and only integer data can be stored in the variable.

Sam's Corner

Under the covers, all data in a computer is represented as a charge stored in a very small capacitor on a chip. We call these bits, 1 indicating “true” (corresponding to a charge in the capacitor) and 0 corresponding to “false” (corresponding to no charge). Bits are stored differently on CDs, flash memory, and hard drives.



Integers

Integers are possibly the most commonly used data-type. Integers are useful for counting or for numbers that cannot have a decimal. For example, the number of members in a family is always an integer; there can never be 2.4 people in a family. You can declare a variable as an integer with:

```
int age = 42;
```

With this line of code, a new variable is created. The name is “age” which is how the variable will be referenced for the remainder of the program. Since the data-type is an integer (as specified by the int keyword), we know two things. First, the amount of memory used by the variable is 4 bytes (1 byte equals 8 bits so it takes a total of 32 bits to store one integer). Second, the value of the variable age must be between -2,147,483,648 and 2,147,483,647. Observe how the integer is initialized to the value of 42 in this example.

Sam's Corner

It is easiest to explain how integers are stored in memory by considering a special integer that is only positive and has 8 bits (this is actually an unsigned char). In this case, the right-most bit correspond to the 1's place, the next corresponds to the 2's place, the next corresponds to the 4's place, and so on. Thus the bits (00101010) is interpreted as:

$$\begin{array}{cccccccc}
 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 0 & + & 0 & + & 32 & + & 0 & + & 8 & + & 0 & + & 2 & + & 0 & = & 42
 \end{array}$$

In other words, each place has a value corresponding to it (as a power of two because we are counting in binary). You add that value to the sum only if there is a 1 in that place. Typically integers are 32 bits (4 bytes) in length. The left-most bit is special, indicating whether the number is positive or negative.



Floating point numbers

In mathematics, real numbers are numbers that can have a decimal. It is often convenient to represent very large or very small real numbers in scientific notation:

$$1888 = 1.888 \times 10^3$$

Observe how the decimal point position is specified by the exponent (10^3 in this case). In many ways, the decimal point can be said to “float” or move according to the exponent, the origin of the term “floating point numbers” in computer science. Floating point numbers are characterized by two parts: the precision part of the equation (1.888 in the above example) and the exponent (10^3). There are three floating point types available in the C++ language:

| Type name | Memory used | Exponent | Precision |
|-------------|-------------|-----------------------------|-----------|
| float | 4 | 10^{-38} to 10^{38} | 7 digits |
| double | 8 | 10^{-308} to 10^{308} | 15 digits |
| long double | 16 | 10^{-4932} to 10^{4932} | 19 digits |

Observe how the more data is used (measured in bytes), the more accurately the number can be represented. However, all floating point numbers are approximations. Examples of declaring floating point numbers include:

```
float gpa = 3.9;
double income = 103295.05;
long double pi = 3.14159265358979323;
```



Sue's Tips

While it is wasteful to use a larger data-type than is strictly necessary (who would ever want their GPA to be represented to 19 digits?), it is much worse to not have sufficient room to store a number. In other words, it is a good idea to leave a little room for growth when declaring a floating point number.

Characters

Another common data-type is a character, corresponding to a single letter, number, or symbol. We declare a character variable with:

```
char grade = 'A';
```

When making an assignment with chars, a single ' is used on each side of the character. This is different than the double quotes " used when denoting text. Each character in the char range has a number associated with it. This mapping of numbers to characters is called the ASCII table:

| Number | 46 | 47 | 48 | 49 | 50 | ... | 65 | 66 | 67 | 68 | ... | 97 | 98 | 99 | 100 |
|--------|----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|-----|
| Letter | . | / | 0 | 1 | 2 | | A | B | C | D | | a | b | c | d |

The complete ASCII table can be viewed in a variety of web sites:

<http://en.cppreference.com/w/cpp/language/ascii>



The `char` data-type is actually a form of integer where the size is 1 byte (or 8 bits). This means there are only 256 possible values, not four billion. Each number in the `char` range corresponds to a glyph (letter, number, or symbol) in the ASCII table. Thus you can treat a `char` like a letter or you can do math with it like any other integer. For example, `'A' + 1` is the same as `'B'`, which is 66.

Text

Text consists of a collection or string of characters. While all the data-types listed below can readily fit into a small slot in memory, text can be exceedingly long. For example, the amount of memory necessary to store your name is much less than that required to store a complete book. You declare a string variable with:

```
char text[256] = "CS 124";
```

There are a few things to observe about this declaration. First, the size of the buffer (or number of available slots in the string) is represented in square brackets `[]`. The programmer specifies this size at compile time and it cannot be changed. The second thing to note is how the contents of the string are surrounded in double quotes `"` just as they were with our `cout` examples.



Sue's Tips

The standard size to make strings is 256 characters in length. This is plenty long enough for most applications. It is usually more convenient (and bug-free) to have the same string length for an entire project than to have many different string buffer sizes (which would require us to keep track of them all!).

Logical Data

The final built-in data-type is a `bool`. This enables us to capture data having only two possible values. For example, a person is either pregnant or not, either alive or not, either male or not, or either a member of the church or not. For these data-types, we use a `bool`:

```
bool isMale = false;
```

There are only two possible values for a `bool`: `true` or `false`. By convention, we name `bool` variables in such a way that we know what `true` means. In other words, it would be much less helpful to have a variable called `gender`. What does `false` mean (that one *has* no gender like a rock)?



A `bool` takes a single byte of memory, consisting of 8 bits. Note that we really only need a single bit to capture Boolean (`true/false`) data. Why do we need 8 then? This has to do with how convenient it is for the computer to work with bytes and how awkward it is to work with bits. When evaluating a `bool`, any 1's in any of the bits will result in a `true` evaluation. Only when all 8 bits are 0 will the `bool` evaluate to `false`. This means that there are 255 true values ($2^8 - 1$) and 1 false value.

Input

Now that we know how to store data in a computer program using variables, it is possible to prompt the user for input. Note that without variables we would not have a place to store the user input so asking the user questions would be futile. The main mechanism with which we prompt users for input is the `cin` function. This function, like `cout`, is part of the `iostream` library. The code for prompting the user for his age is:

```
{
    int age;
    cin >> age;
}
```

In this example, we first declare a variable that can hold an integer. There are a couple important points here:

- Use `cin` rather than `cout`. This refers to Console INput, analogous to the Console OUtput of `cout`.
- The **extraction operator** `>>` is used instead of the **insertion operator** `<<`. Again, the arrow points the direction the data goes. In this case, it goes from the keyboard (represented by `cin`) to the variable (represented by `age`).
- There is always a variable on the right side of the extraction operator.

We can use `cin` with all built in data-types:

```
{
    // INTEGERS
    int age;           // integers can only hold digits.
    cin >> age;        // if a non-digit is entered, then age remains uninitialized.

    // FLOATS
    double price;      // able to handle zero or many digits
    cin >> price;

    // SINGLE LETTERS
    char letter;       // only one letter of any kind
    cin >> letter;      // anything but a white-space (space, tab, or newline).
    cin.get(letter);    // same as above, but will also get white-spaces

    // TEXT
    char name[256];     // any text up to 255 characters in length
    cin >> name;        // all user input up to the first white-space is accepted
}
```



A stream (the input from the keyboard into `cin`) can be thought of as a long list of characters moving from the keyboard into your program. The question is: how much input is consumed by a single `cin` statement? Consider the following input stream:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|--|
| 4 | 2 | C | e | l | s | i | u | s | |
|---|---|---|---|---|---|---|---|---|--|

And consider the code:

```
int temperature;
char units[256];
cin >> temperature;
cin >> units;
```

In this example, the input stream starts at the space before the 4. The first thing that happens is that all the white-spaces are skipped. This moves the cursor to the 4. Since a 4 is a digit, it can be put into the integer `temperature`. Thus the value in `temperature` is 4 and the cursor advances to the next spot. From here, 2 is recognized as a digit so the 4 value in `temperature` becomes 40 and 2 is added to yield 42. Again the cursor is advanced. At this point, `c` is not a digit so we stop accepting input in the variable `temperature`. The next `cin` statement is executed which accepts text. Recall that text accepts input up to the first white-space. Since the cursor is on the `c`, the entire word of “Celsius” will be put in the `units` variable and the cursor will stop at the white-space.

Multiple Extraction Operators

Often it is convenient to input data into more than one variable on a single input statement. This can be done by “stacking” the extraction operators much like we stacked the insertion operators:

```
{
    char name[256];
    int age;
    cin >> name >> age;
}
```

In this example, the first thing the user inputs will be put into the `name` variable and the second into `age`.


Whole Lines of Text

Recall how, when reading text into a variable using `cin`, only one word (or more accurately the characters between white-spaces) are entered. What do you do when you want to enter an entire line of text including the spaces? For this scenario, a new mechanism is needed:

```
{
    char fullName[256];           // store an individual's full name: Dr. Drake Ramoray
    cin.getline(fullName, 256);
}
```

Observe how we do not use the extraction (`>>`) operator which was part of our other input mechanisms. The `getline` function takes two parameters: the name of the variable (`fullName` in this example) and the length of the buffer (256 because that is how large `fullName` was when it was defined).

Example 1.2 – Many Prompts

| | |
|-----------|--|
| Demo | This example will demonstrate how to declare text, integer, floating point, and character variables. It will also demonstrate how to accept data from the user with each of these data types. |
| Problem | <p>Write a program to prompt the user for his first name, age, GPA, and the expected grade in CS 124. The information will then be displayed on the screen.</p> <pre>What is your first name: Sam What is your age: 19 What is your GPA: 3.91 What grade do you hope to get in CS 124: A Sam, you are 19 with a 3.9 GPA. You will get an A.</pre> |
| Solution | <p>The four variables are declared as follows:</p> <pre>char name[256]; int age; float gpa; char letterGrade;</pre> <p>To prompt the user for his age, it is necessary to display a prompt first so the user knows what to do. Usually we precede the prompt and the input with a comment and blank line:</p> <pre>// Prompt the user for his age cout << "What is your age: "; cin >> age;</pre> <p>Finally, we must not forget to format cout to display one digit after the decimal.</p> <pre>// configure the display to show GPAs: one digit of accuracy cout.setf(ios::fixed); cout.setf(ios::showpoint); cout.precision(1); // display the results cout << "\t" << name << ", you are " << age << " with a " << gpa << " GPA. You will get an " << letterGrade << ".\n";</pre> |
| Challenge | <p>As a challenge, try to accept an individual's full name (Such as "Sam S. Student") rather than just the first name.</p> <p>Also, try to configure the output to display two digits of accuracy rather than one.</p> |
| See Also | <p>The complete solution is available at 1-2-manyPrompts.cpp or:</p> <pre>/home/cs124/examples/1-2-manyPrompts.cpp</pre>  |

Problem 1

What is the output of the following line of code?

```
cout << "\\\"/\n";
```

Answer:

Please see page 29 for a hint.

Problem 2

How do you put a tab on the screen?

Answer:

Please see page 27 for a hint.

Problem 3

How do you output the following:

```
You will need to use '\n' a ton in this class.
```

Answer:

Please see page 29 for a hint.

Problem 4

How do you declare an integer variable?

Answer:

Please see page 36 for a hint.

Problem 5

How would you declare a variable for each of the following?

| Variable name | Declaration |
|---------------|-------------|
| yearBorn | |
| gpa | |
| nameStudent | |
| ageStudent | |

Please see page 36 for a hint.

Problem 6

Declare a variable to store the ratio of feet to meters.

Answer:

Please see page 37 for a hint.

Problem 7

What is the number of bytes for each data type?

```
{
    cout << sizeof(char) << endl;

    char a;
    cout << sizeof(a) << endl;

    cout << sizeof(bool) << endl;

    int b;
    cout << sizeof(b) << endl;

    float c;
    cout << sizeof(c) << endl;

    double d;
    cout << sizeof(d) << endl;

    long double e;
    cout << sizeof(e) << endl;
}
```

Please see page 36 for a hint.

Problem 8

Which of the following can store the largest number?

```
bool value;
char value[256];
int value
long double value;
```

Please see page 36 for a hint.

Problem 9

Declare a variable to represent the following number in C++: 8,820,198,883,463.39

Answer:

Please see page 37 for a hint.

Problem 10

Write the code to prompt a person for his first name.

Answer:

Please see page 39 for a hint.

Problem 11

Write the code to prompt a person for his two favorite numbers.

Answer:

Please see page 39 for a hint.

Problem 12

Write the code to prompt a person for his full name.

Answer:

Please see page 40 for a hint.

Assignment 1.2

Write a program that prompts the user for his or her income and displays the result on the screen. There will be two parts:

Get Income

The first part is code that prompts the user for his income. It will ask the user:

```
Your monthly income:
```

There will be a tab before “Your” and a single space after the “:”. There is no newline at the end of this prompt. The user will then provide his or her income as a float.

Display

The second part is code to display the results to the screen.

```
Your income is: $ 1010.99
```

Note that there is one space between the colon and the dollar sign. The money is right aligned to 9 spaces from the dollar sign.

Example

User input is underlined. Note that you will not be making the input underlined; this is just the notation used in the assignments to distinguish input from output.

```
Your monthly income: 932.16
Your income is: $ 932.16
```

Instructions

Please verify your solution against:

```
testBed cs124/assign12 assignment12.cpp
```

Don’t forget to submit your assignment with the name “Assignment 12” in the header.

Please see page 41 for a hint.

1.3 Expressions

Sam once spent a summer working as a cashier in a popular fast-food outlet. One of his responsibilities was to make change for customers when they paid with cash. While he enjoyed the mental exercise of doing the math in his head, he immediately started wondering how this could best be done with the computer. After a few iterations, he came up with a program to make light work of his most tedious task...

Objectives

By the end of this class, you will be able to:

- Represent simple equations in C++.
- Understand the differences between integer division and floating point division.
- See how to use the modulus operator to solve math and logic problems.

Prerequisites

Before reading this chapter, please make sure you are able to:

- Choose the best data-type to represent your data (chapter 1.2).
- Declare a variable (chapter 1.2).
- Display text and numbers on the screen (chapter 1.1).

Overview

Computer programs perform mathematical operations much the way one would expect. There are a few differences, however, owing to the way computers store numbers. For example, there is no distinction between integers and floating point numbers in Algebra. This means that dividing one by two will yield a half. However, in C++, integers can't store the number 0.5 or $\frac{1}{2}$. Also, a variable can update its value in C++ where in Algebra it remains constant through the entire equation. These challenges along with a few others makes performing math with C++ a little tricky.

In C++, mathematical equations are called **expressions**. An expression is a collection of values and operations that, when evaluated, result in a single value.

Evaluating Expressions

As you may recall from our earlier discussion of how computers work, a CPU can only perform elementary mathematical operations and these can only be done one at a time. This means that the compiler must break complex equations into simple ones for them to be evaluated correctly by the CPU. To perform this task, things are done in the following order:

1. Variables are replaced with the values they contain
2. The order of operations are honored: parentheses first and assignment last
3. When there is an integer being compared/computed with a float, it is converted to a float just before evaluation.

Step 1 - Variables are replaced with values

Every variable refers to a location of memory. This memory location is guaranteed to be filled with 1's and 0's. In other words, there is *always* a value in a variable and that value can always be accessed at any time. Sometimes the value is meaningless. Consider the following example:

```
{
    int number;
    cout << number << endl;           // the output is different every time because
                                        // the variable number was never initialized
}
```

Since the variable was never initialized, the value is not predictable. In other words, whoever last used that particular location in memory left data lying around. This means that there is some random collection of 1's and 0's in that location. We call this state uninitialized because the programmer never got around to assigning a value to the variable `number`. All this could be rectified with a simple:

```
int number = 0;
```

The first step in the expression evaluation process is to substitute the variables in the expression with the values contained therein. Consider the following code:

```
{
    int ageHumanYears = 4;
    int ageDogYears = ageHumanYears * 7;
}
```

In this example, the first step of evaluating the last statement is to substitute `ageHumanYears` with 4.

```
int ageDogYears = 4 * 7;
```

Step 2 - Order of Operations

The order of operations for mathematical operators in C++ is:

| Operator | Description |
|------------------|---|
| () | Parentheses |
| ++ -- | Increment, Decrement |
| * / % | Multiply, Divide, Modulo |
| + - | Addition, Subtraction |
| = += -= *= /= %= | Assign, Add-on, Subtract-from, Multiply onto, Divide from, Modulo from. |

This should be very familiar; it is similar to the order of operations for Algebra. There are, of course a few differences

Increment ++

Because it is possible to change the value of a variable in C++, we have an operator designed specifically for the task. Consider the following code:

```
{
    int age = 10;
    age++;
    cout << age << endl;        // the output is 11
}
```

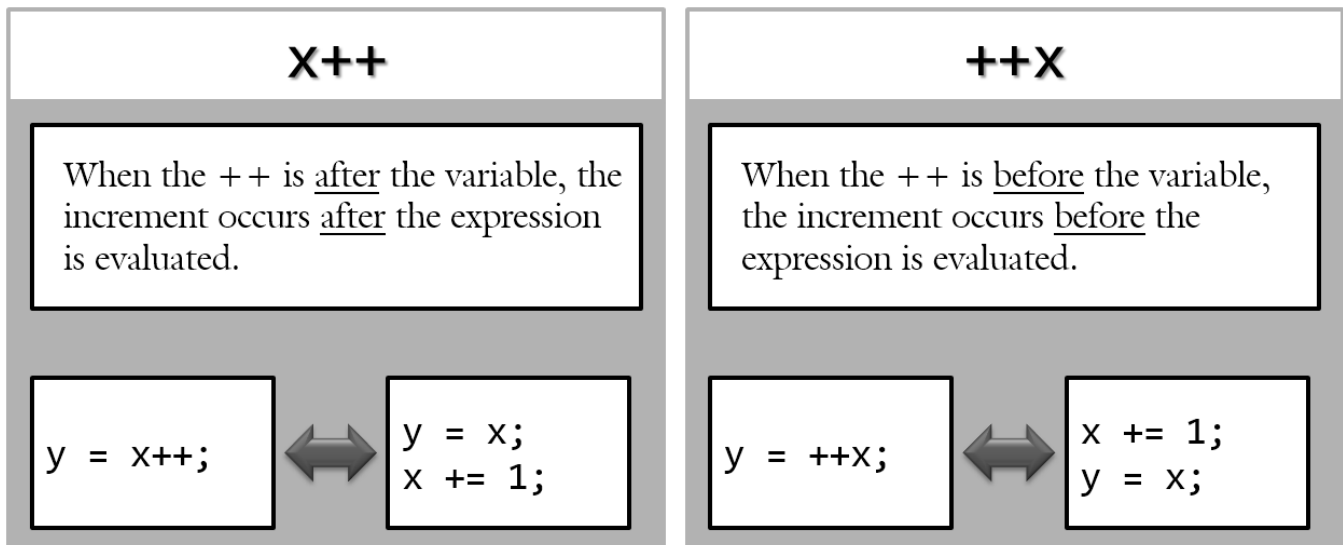
In this example, the `age++` statement serves to add one to the current value of `age`. Of course, `age--` works in the opposite way. There are two flavors of the increment (and decrement of course) operators: increment before the expression is evaluated and increment after. To illustrate, consider the following example:

```
{
    int age = 10;
    cout << age++ << endl;      // the output is 10 and the new value of age is 11
}
```

In this example, we increment the value of `age` *after* the expression is evaluated (as indicated by the `age++` rather than `++age` where we would evaluate *before*). Therefore, the output would be 10 although the value of `age` would be 11 at the end of execution. This would not be true with:

```
{
    int age = 10;
    cout << ++age << endl;      // the output is 11 and the new value of age is 11
}
```

In this case, `age` is incremented *before* the expression is evaluated and the output would be 11. In short:



Multiplication *

In C++ (and most other computer languages for that matter), the multiplication operator is an asterisk *. You cannot use the dot operator (ex: .), the multiplication x (ex: ×), or put a number next to a variable (ex: 7y) as you can in standard algebra notation.

```
{
    float answer1 = 1.2 * 2.3;    // the value of answer1 is 2.76
    int    answer2 = 2 * 3;        // the value of answer2 is 6
}
```

Division /

Floating point division (/) behaves the way it does in mathematics. Integer division, on the other hand, does not. The evaluation of integer division is always an integer. In each case, the remainder is thrown away. To illustrate this, consider the following:

```
{
    int    answer1 = 19 / 10;
    float  answer2 = 19.0 / 10.0;
    cout << answer1 << endl;        // the output is 1
        << answer2 << endl;        // the output is 1.9
}
```

In this case, the output of the first line is not 1.9 because the variable `answer1` cannot store a floating point value. When 19 is divided by 10, the result is 1 with a remainder of 9. Therefore, `answer1` will get the value 1 and the remainder is discarded. To get 1.9, we need to use floating point division.

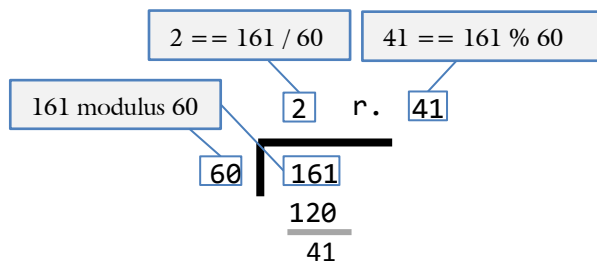
Modulus %

Recall that integer division drops the remainder of the division problem. What if you want to know the remainder? This is the purpose of the modulus operator (%). Consider the following code:

```
{
    int remainder = 19 % 10;
    cout << remainder;    // the output is 9
}
```

In this case, when you divide 19 by 10, the remainder is 9. Therefore, the value of `remainder` will be 9 in this case. For example, consider the following problem:

```
{
    int totalMinutes = 161;           // The movie "Out of Africa" is 161 minutes
    int numHours     = totalMinutes / 60; // The movie is 2 hours long ...
    int numMinutes   = totalMinutes % 60; // ... plus 41 minutes
}
```



Assignment =

In mathematics, the equals symbol $=$ is a statement of equality. You are stating that the right-side and the left-side are the same or balanced. In C++, the equals symbol is a statement of assignment. You are specifying that the evaluation of the right-side is to be assigned to the variable on the left-side. Consider the following code:

```
{
    int x = 2;
    x = x + 1;    // the value of x is updated from 2 to 3. We can
}               // change the value of variables in C++
```

The second statement would not be possible in mathematics; there is no value for x where $x=x+1$ is true. However, in C++, this is quite straightforward: the right-side evaluates to 3 and the variable on the left is assigned to that value. It turns out that adding a value to a variable is quite common. So common, in fact, that a shorthand is offered:

```
{
    int x = 2;
    x += 1;    // the new value of x is 3
}
```

The `+=` operator says, in effect, add the right-side to the variable on the left-side. The end result is the x being updated to the value of 3. The most common variants are:

| Operator | Description | Use |
|-----------------|---------------------|---------------|
| <code>+=</code> | Add and assign | Add onto |
| <code>-=</code> | Subtract and assign | Subtract from |
| <code>*=</code> | Multiply and assign | Multiply by |
| <code>/=</code> | Divide and assign | Subdivide |

Step 3 - Converting

The final step in evaluating an expression is to convert data from one type to another. This arises from the fact that you can't add an integer to a floating point number. You can add two ints or two floats, but not an int to a float. Consider the following code:

```
cout << 4 + 3.2 << endl;
```

In this example, there are two possibilities: either convert the integer 4 into the float 4.0 or convert the float 3.2 into the integer 3. C++ will always convert ints to floats and bools to ints in these circumstances. It is important to note, however, that this conversion will only happen immediately before the operator is evaluated.

Casting

Rather than allowing the compiler to convert integers or values from one data type to another, it is often useful to perform that conversion yourself explicitly. This can be done with casting. Casting is the process of specifying that a given value is to be treated like another data-type just for the purpose of evaluating a single expression. Consider the following code:

```
{
    int value = 4;
    cout << "float:  " << (float)value << endl;    // the output is "float:  4.0"
    cout << "integer: " << value << endl;         // the output is "integer: 4"
}
```

In this case, the output of the first cout statement will be 4.0 because the integer value 4 will be converted to a floating point value 4.0 in this expression. The value in the variable itself will not be changed; only the evaluation of that variable in that particular expression. The second cout statement will display 4 in this case.

There are a few quirks to casting. First, the variable you are casting does not change. Once you declare a variable as a given data-type, it remains that data-type for the remainder of the program. Casting just changes how that variable behaves for one expression.

Second, not all data-types convert in the most obvious way. Consider converting ints and bools:

```
{
    bool a = (bool)7;           // true    any number but 0 turns into true
    bool b = (bool)0;           // false   only zero turns to false
    int  c = (int>true;          // 1       true always becomes 1
    int  d = (int>false;         // 0       false always becomes 0
}
```

Sam's Corner

There are actually two notations for casting in C++. The older notation, presented above, was inherited from the C programming language and is somewhat deprecated. It still works, but purists will prefer the new notation. The new notation for casting has two main variants: static cast corresponding to casting that happens at compile time, and dynamic cast which happens at runtime. All the casting we do with procedural C++ can be static. We won't need to use dynamic casting until we learn about Object Oriented programming in CS 165.

```
{
    int value = 4;
    cout << "float: " << static_cast<float>(value) << endl;
}
```



Putting it all together

So how does this work together? Consider the following example:

```
{
    int f = 34;
    int c = 5.0 / 9 * (f - 32);
}
```

The most predictable way to evaluate the value of the variable `c` is to handle this one step at a time:

1. `int c = 5.0 / 9 * (f - 32);` // The original statement
2. `int c = 5.0 / 9 * (34 - 32);` // Step 1. Substitute the value `f` for 34
3. `int c = 5.0 / 9 * 2;` // Step 2. Perform subtraction: `2 == 34 - 32`
4. `int c = 5.0 / 9.0 * 2;` // Step 3. Convert 9 to 9.0 for floating point division
5. `int c = 0.555556 * 2;` // Step 2. Perform floating point division: `0.55555 == 5.0 / 9.0`
6. `int c = 0.555556 * 2.0;` // Step 3. Convert 2 to 2.0 for floating point multiplication
7. `int c = 1.111111;` // Step 2. Perform multiplication: `1.11111 == 0.555556 * 2.0`
8. `int c = 1;` // Step 3. Convert 1.111111 to the integer 1 for assignment



Sue's Tips

Seemingly simple expressions can be quite complex and unpredictable when data-type conversion occurs. It is far easier to use only one data-type in an expression. In other words, don't mix floats and ints!

Example 1.3 - Compute Change

Demo

This example will demonstrate how to evaluate simple expressions, how to update the value in a variable, casting, and how to use modulus.

Problem

Write a program to prompt the user for an amount of money. The program will then display the number of dollars, quarters, dimes, nickels, and pennies required to match the amount.

Solution

In this example, the user is prompted for a dollar amount:

```
// prompt the user
cout << "Please enter a positive dollar amount (ex: 4.23): ";
float dollars;
cin >> dollars;
```

Next it is necessary to find the number of cents. This is done by multiplying the dollar variable by 100. Note that dollars have a decimal so they must be in a floating point number. Cents, however, are always whole numbers. Thus we should store it in an integer. This requires conversion through casting.

```
// convert to cents
int cents = (int)(dollars * 100.00);
```

Finally we need to find how many Dollars (and Quarters, Dimes, etc) are to be sent to the user. We accomplish this by performing integer division (where the decimal is removed).

```
cout << "Dollars: " << cents / 100 << endl;
```

After we extract the dollars, how many cents are left? We compute this by finding the remainder after dividing by 100. We can ask for the remainder by using the modulus operator (`cents % 100`). Since we want to assign the new amount back to the cents variable, we have two options:

```
cents = (cents % 100);
```

This is exactly the same as:

```
cents %= 100;
```

Challenge

As a challenge, try to modify the above program so it will not only compute change with coins, but also for bills. For example, it will display the number of \$1's, \$5's, \$10's, and \$20's.

See Also

The complete solution is available at [1-3-computeChange.cpp](#) or:

```
/home/cs124/examples/1-3-computeChange.cpp
```



Problem 1

Please write the variable declaration used for each variable name:

- numberStudents: _____
- pi: _____
- hometown: _____
- priceApples: _____

Please see page 36 for questions

Problem 2

How much space in memory does each variable take?

- bool value; _____
- char value[256]; _____
- char value; _____
- long double value; _____

Please see page 36 for questions

Problem 3

Insert parentheses to indicate the order of operations:

a = a + b * c++ / 4

Please see page 47 for a hint.

Problem 4

What is the value of yard at the end of execution?

```
{  
    float feet = 7;  
    float yards = (1/3) feet;  
}
```

Answer:

yards == _____

Please see page 49 for a hint.

Problem 5

What is the value of a?

```
int a = (2 + 2) / 3;
```

Answer:

a == _____

Please see page 49 for a hint.

Problem 6

What is the value of b?

```
int b = 2 / 3 + 1 / 2;
```

Answer:

b == _____

Please see page 49 for a hint.

Problem 7

What is the value of c?

```
int f = 34;  
int c = 5 / 9 * (f - 32);
```

Answer:

c == _____

Please see page 49 for a hint.

Problem 8

What is the value of d?

```
int d = (float) 1 / 4 * 10;
```

Answer:

d == _____

Please see page 51 for a hint.

Problem 9

Write a program to prompt the user for a number of days, and return the number of days and weeks

Example:

```
How many days: 17
               weeks: 2
               days: 3
```

Please see page 49 for a hint.

Problem 10

What is the output?

```
{
    int dateOfBirth = 1987;
    int currentYear = 2006;

    cout << "age is "
          << currentYear++ - dateOfBirth
          << endl;

    cout << "age is "
          << currentYear++ - dateOfBirth
          << endl;
}
```

Answer:

Please see page 49 for a hint.

Temperature Conversion

Write a program to convert Fahrenheit to Celsius. This program will prompt the user for the Fahrenheit number and convert it to Celsius. The equation is:

$$C = 5/9(F - 32)$$

The program will prompt the user for the temperature, compute the Celsius value, and display the results.

Hint: If you keep getting zero for an answer, you are probably not taking integer division into account. Please review the text for insight as to what is going on.

Hint: If the last test fails, then you are probably not rounding correctly. Note that integers cannot hold the decimal part of a number so they always round down. If you use `precision(0)`, then the rounding will occur the way you expect.

Example

User input is in underline.

```
Please enter Fahrenheit degrees: 72
Celsius: 22
```

Assignment

The test bed is available at:

```
testBed cs124/assign13 assignment13.cpp
```

Don't forget to submit your assignment with the name "Assignment 13" in the header.

Please see page 49 for a hint.

1.4 Functions

Sue is working on a large project and is getting overwhelmed. How can she possibly keep all this code straight? To simplify her work, she decides to break the program into a collection of smaller, more manageable chunks. Each chunk can be individually designed, developed, and tested. Suddenly the problem seems much more manageable!

Objectives

By the end of this class, you will be able to:

- Create a function in C++.
- Pass data into a function using both pass-by-value and pass-by-reference.
- Be able to identify the scope of a variable in a program.

Prerequisites

Before reading this section, please make sure you are able to:

- Choose the best data-type to represent your data (Chapter 1.2).
- Declare a variable (Chapter 1.2).

Overview

A function is a small part of a larger program. Other terms (procedure, module, subroutine, subprogram, and method) mean nearly the same thing in the Computer Science context and we will use them interchangeably this semester.

There are two main ways to look at functions. The first is like a medical procedure. A medical procedure is a small set of tasks designed to accomplish a specific purpose. Typically these procedures are relatively isolated; they can be used in a wide variety of contexts or operations. Functions in C++ often follow this procedural model: breaking large programs into smaller ones.

The second way to look at functions is similar to how a mathematician looks at functions: an operation that converts input into output. The Cosine function is a great example: input in the form of radians or degrees is converted into a number between one and negative one. Frequently functions in C++ follow this model as programmers need to perform operations on data.

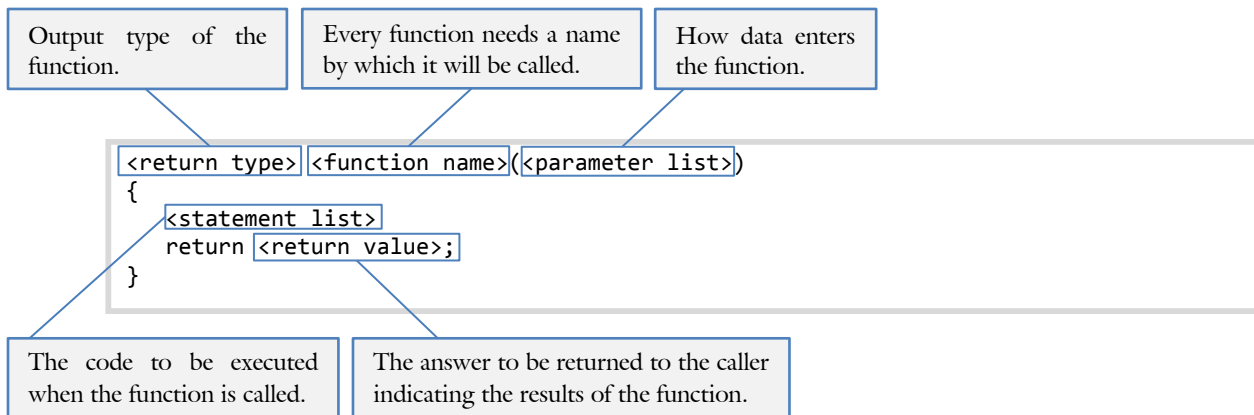
The syntax for both procedures and mathematical functions is the same in C++. The purpose of this chapter is to learn the syntax of functions so they can be used in our programs. All assignments, projects, and tests in this class will involve multiple functions from this time forward.

Function Syntax

There are two parts to function syntax: the syntax of declaring (or defining) a function, and the syntax of calling (or “running”) a function.

Declaring a Function

The syntax of a function is exactly the same as the syntax of `main()` because `main()` is a function!



Consider the following function to convert feet to meters:

```

/*****
 * CONVERT FEET TO METERS
 * Convert imperial feet to metric meters
 *****/
double convertFeetToMeters(double feet)
{
    double meters = feet * 0.3048;
    return meters;
}

```

Observe the function header. As the number of functions gets large in a program, it becomes increasingly important to have complete and concise function comment blocks.

Function names are typically verbs because functions do things. Similarly variable names are typically nouns because variables hold things. As with the function headers, strive to make the function names completely and concisely describe what the function does.

Finally, observe how one piece of information enters the function (`double feet`) and one piece of information leaves the function (`return meters;`). The input parameter (`feet`) is treated like any other variable inside the function.

Calling a Function

Calling a function is similar to looking up a footnote in the scriptures. The first step is to mark your current spot in the reading so you can return once the footnote has been read. The second step is to read the contents of the footnote. The third is to return back to your original spot in the reading. Observe that we can also jump to the Topical Guide or Bible Dictionary from the footnote. This requires us to remember our spot in the footnote as well as our spot in the scriptures. While humans can typically only remember one or two spots before their place is lost, computers can remember an extremely large number of places.

Computers follow the same algorithm when calling functions as we do when looking up a footnote:

```

{
    double heightFeet = 5.9;
    double heightMeters = convertFeetToMeters(heightFeet);
}

```

In this example, the user is converting his height in feet to the meters equivalent. To accomplish this, the function `convertFeetToMeters()` is called. This indicates the computer must stop working in the calling function and jump to the function `convertFeetToMeters()` much like a footnote in the scriptures indicates we should jump to the bottom of the page. After the computer has finished executing the code in `convertFeetToMeters()`, control returns to the calling function.

Example 1.4 – Simple Function Calling

Demo

This example will demonstrate how call a function and accept input from a function. There will be no parameter passing in this example.

Problem

Write a program to ask a simple question and receive a simple answer. This problem is inspired from one of the great literary works of our generation.

```
What is the meaning of life, the universe, and everything?  
The answer is: 42
```

The first function will return nothing. Hence, it will have the obvious name:

```
/* *****  
 * RETURN NOTHING  
 * This function will not return anything. Its only purpose is  
 * to display text on the screen. In this case, it will display  
 * one of the great questions of the universe  
 * ***** */  
void returnNothing()  
{  
    // display our profound question  
    cout << "What is the meaning of life, the universe, and everything?\n";  
  
    // send no information back to main()  
    return;  
}
```

The second function will return a single integer value back to the caller.

```
/* *****  
 * RETURN A VALUE  
 * This function, when called, will return a single integer value.  
 * ***** */  
int returnAValue()  
{  
    // did you guess what value we will be returning here?  
    return 42;  
}
```


The two functions can be called from main:

```
int main()  
{  
    // call the function asking the profound question  
    returnNothing();           // no data is sent to main()  
  
    // display the answer:  
    cout << "The answer is: "  
        << returnAValue()      // the return value of 42 is sent to COUT  
        << endl;  
  
    return 0;  
}
```

See Also

The complete solution is available at [1-4-simpleFunctionCalling.cpp](#) or:

```
/home/cs124/examples/1-4-simpleFunctionCalling.cpp
```

| Example 1.4 – Prompt Function | |
|-------------------------------|--|
| Demo | This example will demonstrate how create a simple prompt function. This function will display a message to the user asking him for information, receive the information using <code>cin</code> , and return the value through the function return mechanism. |
| Problem | <p>Write a program to prompt the user for his age. The user's age will then be displayed. User input is <u>bold and underlined</u>.</p> <pre>What is your age? <u>19</u> Your age is 19 years old.</pre> |
| Solution | <p>The prompt function follows the “return a value” pattern from the previous example:</p> <pre> /***** * GET AGE * Prompt the user for his age. First display a message stating * what information we hope the user will provide. Next receive * the user input. Finally, return the results to the caller. *****/ int getAge() { int age; // we need a variable to store the user input cout << "What is your age? "; // state what you want the user to give you cin >> age; // we need a variable to store the user input return age; // this sends data back to main() }</pre> <p>Next we will create <code>main()</code> to test our function.</p> <pre> /***** * MAIN * The whole purpose of main() is to test our getAge() function. *****/ int main() { // get the user input int age = getAge(); // store the data from getAge() in a variable // display the results cout << "Your age is " // note the space after "is" << age // the value from getAge() is stored here << " years old.\n"; // again a space before "year" return 0; // return "success" }</pre> |
| Challenge | As a challenge, try to add a new function to prompt for GPA. Note that this one will return a floating point number instead of an integer. What changes will you have to add to <code>main()</code> to test this function? |
| See Also | <p>The complete solution is available at 1-4-promptFunction.cpp or:</p> <pre>/home/cs124/examples/1-4-promptFunction.cpp</pre>  |

Parameter Passing

Parameter passing is the process of sending data between functions. The programmer can send only one piece of data from the callee (the function responding to the function call) and the caller (the function issuing or initiating the function call). This data is sent through the return mechanism. However, the programmer can specify an unlimited amount of data to flow from the caller to the callee through the parameter passing mechanism.

Multiple Parameters

To specify more than one parameter to a function in C++, the programmer lists each parameter as a comma-separated list. For example, consider the scenario where the programmer is sending a row and column coordinate to a display function. The display function will need to accept two parameters.

```

/*****
 * DISPLAY COORDINATES
 * Display the row and column coordinates on the screen
 *****/
void displayCoordinates(int row, int column) // two parameters are expected
{
    cout << "("
          << row                      // the row parameter is the first passed
          << ", "
          << column                  // the column parameter is the second
          << ")\n";
    return;
}

```

For this function to be called, two values need to be provided.

```
displayCoordinates(5, 10);
```

Parameter match-up occurs by order, not by name. In other words, the first parameter sent to `displayCoordinates()` will always be sent to the `row` variable. The second parameter will always be sent to the `column`.

Note that the two parameters do not need to be of the same data-type.

```

/*****
 * computePay
 * Compute pay based on wage and number of hours worked
 *****/
double computePay(float wage, int hoursWorked)
{
    return (double)(wage * hoursWorked);
}

```

Common mistakes when working with parameters include:

- Passing the wrong number of parameters. For example, the function may expect two parameters but the programmer only supplied one:

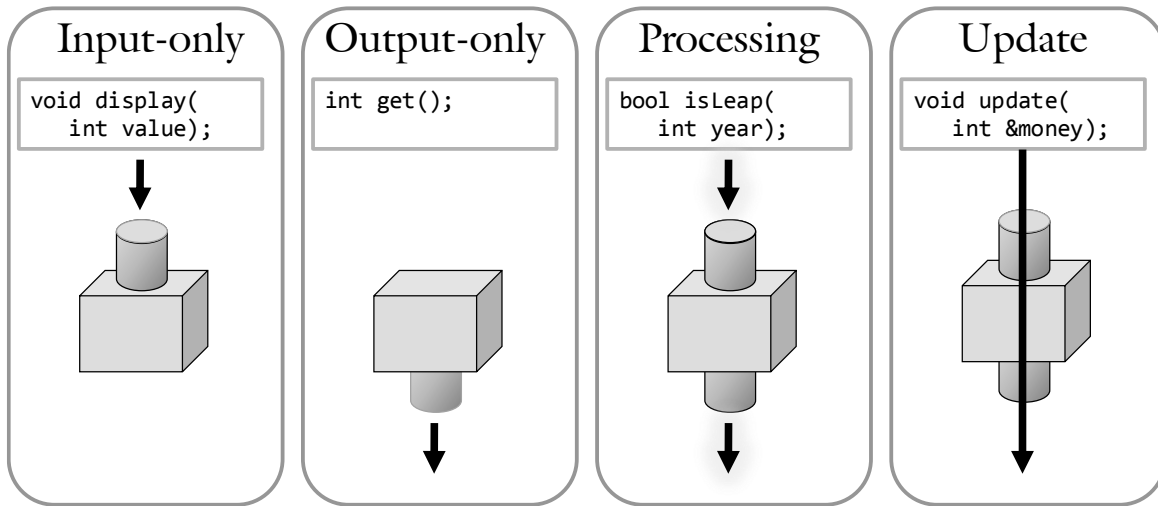
```
displayCoordinates(4); // two parameters expected. Where is the second?
```

- Getting the parameters crossed. For example, the function expects the first parameter to be `row` but the programmer supplied `column` instead:

```
displayCoordinates(column, row); // first parameter should be row, not column
```

Working with Parameters

There are four main ways to think of parameter passing in a C++ program:



Input Only: The first way involves data traveling one-way into a function. Observe how there is an input parameter (`int value`) but no return type (`void`). This is appropriate in those scenarios when you want a function to do something with the data (such as display it) but do not want to send any data back to the caller:

```
void display(int value);
```

Output Only: The second way occurs when data flows from a function, but not into it. An example would be a function prompting the user for information (such as `getIncome()` from Project 1). In this case, the parameter list is empty but there is a return value.

```
int get();
```

Processing: The third way occurs when a function converts data from one type to another. This model was followed in both our `computeSavings()` and `convertFeetToMeters()` examples. It is important to realize that you can have more than one input parameter (in the parentheses) but only one output parameter (the return mechanism).

```
bool isLeap(int year);
float add(float value1, float value2);
```

Update: The final way is when data is converted or updated in the function. This special case occurs when the input parameter and the return value are the same variable. In this case, we need a special indicator on the variable in the parameter list to specify that the variable is shared between the caller and the callee. We call this **call-by-reference**.

```
void update(int &money);
```

Example 1.4 – Compute Function

Demo

This example will demonstrate how to send data to a function and receive data from a function. This follows the “Processing” model of information flow through a function.

Problem

Write a program to compute how much money to put in a savings account. The policy is “half the income after tithing is removed.”

What is your allowance? 10.00
You need to deposit \$4.50

The function to compute savings takes income as input and returns the savings amount

```

/*****
 * For a given amount of earned income, compute the amount to be saved
 *****/
int computeSavings(int centsIncome)
{
    // first take care of tithing
    int centsTithing = centsIncome / 10;    // D&C 119:4
    centsIncome -= centsTithing;            // remove tithing from the income

    // next compute the savings
    int centsSavings = centsIncome / 2;     // savings are half the remaining
    return centsSavings;
}

```

Solution

This function will be called from main. It will provide the input centsIncome and present the results to the user through a cout statement.

```

/*****
 * Prompt the user for his allowance and display the savings component
 *****/
int main()
{
    // prompt the user for his allowance
    float dollarsAllowance;                // a float for decimal #s
    cout << "What is your allowance? ";
    cin >> dollarsAllowance;                // input is in dollars
    int centsAllowance = (int)dollarsAllowance * 100; // convert to cents

    // display how much is to be deposited
    int centsDeposit = computeSavings(centsAllowance); // call the function!
    cout << "You need to deposit $"
         << (float)centsDeposit / 100.0      // convert back to dollars
         << endl;
    return 0;
}

```

Challenge

As a challenge, create a function to convert a floating-point dollars amount (dollarsAllowance) into an integral cents amount (centsAllowance). Use the formula currently in main():

```
int centsAllowance = (int)dollarsAllowance * 100;
```

See Also

The complete solution is available at [1-4-computeFunction.cpp](#) or:

```
/home/cs124/examples/1-4-computeFunction.cpp
```



In the preceding example, there are a few things to observe about the function `computeSavings()`. First, integers were chosen instead of floating point numbers. This is because, though `floats` can work with decimals, they are approximations and often yield unwieldy answers containing fractions of pennies! It is much cleaner to work with integers when dealing with money. To make sure this is obvious, include the units in the variable name.

Data is passed from `main()` into the function `computeSavings()` through the `()`s after the function name. In this case, the expression containing the variable `centsAllowance` is evaluated (to the value 2150 if the user typed 21.50). This value (not the variable!) is sent to the function `computeSavings()` where a new variable called `centsIncome` is created. This variable will be initialized with the value from the calling function (2150 in this case). It is important to realize that a copy of the data from `main()` is sent to the function through the parameter list; the variable itself is not sent! In other words, the variable `centsIncome` in `computeSavings()` can be changed without the variable `centsAllowance` in `main()` being changed. This is because they are different variables referring to different locations in memory!

When execution is in the function `computeSavings()`, only variables declared in that function can be used. This means that the statements in the function only have access to the variables `centsIncome`, `centsTithing`, and `centsSavings`. The variables from the caller (`dollarsAllowance`, `centsAllowance`, and `centsDeposit`) are not visible to `computeSavings()`. To pass data between the functions, parameters must be used

Pass-By-Reference

Pass-by-reference, otherwise known as “call-by-reference” is the process of indicating to the compiler that a given parameter variable is shared between the caller and the callee. We use the ampersand `&` to indicate the parameter is pass-by-reference.

| Pass By Value | Pass By Reference |
|--|--|
| Pass-by-value makes a copy so two independent variables are created. Any change to the variable by the function will not affect the caller. | Pass-by-reference uses the same variable in the caller and the callee. Any change to the variable by the function will affect the caller. |
| <pre> /***** * Pass-by-value * No change to the caller *****/ void notChange(int number) { number++; } </pre> | <pre> /***** * Pass-by-reference * Will change the caller *****/ void change(int &number) { number++; } </pre> |

We use pass-by-reference to enable a callee to send more than one piece of data back to the caller. Recall that the return mechanism only allows a single piece of data to be sent back to the caller. An example of this would be:

```

void getCoordinates(int &row, int &column)           // data is sent back by row and column
{
    cout << "Specify the coordinates (r c): ";
    cin  >> row >> column;
    return;                                         // no data is sent using return
}
  
```

Example 1.4 – Passing By Reference

Demo

This example will demonstrate how to pass no data to a function, how to use pass-by-value, and how to use pass-by-reference.

The first function does not pass any data into or out of the function:

```
/* *****  
 * PASS NOTHING: No information is being sent to the function  
 * ***** */  
void passNothing()  
{  
    // a different variable than the one in MAIN  
    int value;  
    value = 0;  
}
```

The second passes data into the function, so the function gets a copy of what the caller sent it. This is called pass-by-value:

```
/* *****  
 * PASS BY VALUE: One-way flow of information from MAIN to the function.  
 *               No data is being sent back to MAIN  
 * ***** */  
void passByValue(int value)  
{  
    // show the user what value was sent to the function  
    cout << "passByValue(" << value << ")\n";  
  
    // this is a copy of the variable in MAIN. This will not  
    // influence MAIN in any way:  
    value = 1;  
}
```

The final uses pass-by-reference. This means that both the caller and the callee share a variable. This relationship is indicated by the '&' symbol beside the parameter in the function:

```
/* *****  
 * PASS BY REFERENCE: Two-way flow of data between the functions. Changes to  
 *               REFERENCE will also influence the variable in MAIN  
 * ***** */  
void passByReference(int &reference)  
{  
    // show the user what value was sent to the function  
    cout << "passByReference(" << reference << ")\n";  
  
    // this will actually change MAIN because there was the &  
    // in the parameter  
    reference = 2;  
}
```

The only difference between `passByReference(int &reference)` and `passByValue(int value)` is the existence of the & beside the variable name. When the & is specified, then pass-by-reference is used.

See Also

The complete solution is available at [1-4-passByReference.cpp](#) or:

```
/home/cs124/examples/1-4-passByReference.cpp
```

Unit 1

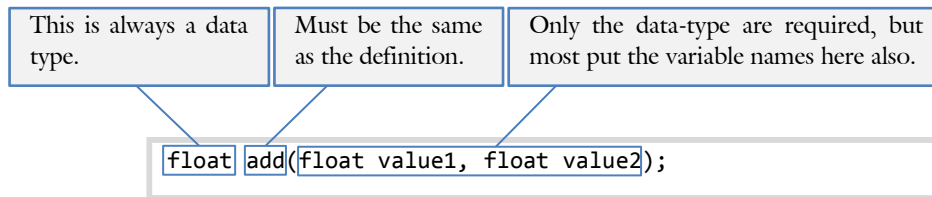
Solution

Prototypes

C++ programs are compiled linearly from the top of the file to the bottom. At the point in time when a given line of code is compiled, the compiler must know about all the variables and functions referenced in order for it to be compiled correctly. This means that the variables and functions must be defined above the line of code in which they are referenced.

One fallout of this model is that `main()` must be at the bottom of the file. This is required because any function referenced by `main()` must be defined by the time it is referenced in `main()`. As you may imagine, this can be inconvenient at times. Fortunately, C++ gives us a way to work around this constraint.

Prototypes are a way to give the compiler “heads-up” that a function will be defined later in the program. There are three required parts to a prototype: the return type, the function name, and the data-type of the parameters.



One nice thing about prototypes is that it allows us to put `main()` at the top of the program file, preceded by a list of all the functions that will appear later in the file.

```
#include <iostream>
using namespace std;

void displayInstructions();

/*****
int main()
{
    displayInstructions();

    // pause
    char letter;
    cin >> letter;
    return 0;
}

*****/
void displayInstructions()
{
    cout << "Please press the <enter>"
         << " key to quit the program"
         << endl;
    return;
}
```

Prototype of the function `displayInstructions()`. Note the absence of code; this is just an outline.

Here the function is called inside `main()` even though `displayInstructions()` has not been defined yet. Normally we need to define a function before we call it.

Now the function `displayInstructions()` is defined, we better have exactly the same name, return type, and parameter list as the prototype or the compiler will get grumpy.

Scope

A final topic essential to understanding how data passes between functions is Scope. Scope is the context in which a given variable is available for use. For example, if a variable is defined in one function, it cannot be referenced in another. The general rule of variable scope is the following:

A variable is only visible from the point where it is declared to the next closing curly brace }

Local Variables

The most common way to declare variables is in a function. This is called a “local variable” because the variable is local to (or restricted to) one function. Consider the following example:

```

/*****
 * PRINT NAME
 * Display the user's name. No data is shared with MAIN
 *****/
void printName()
{
    char name[256];          // Local variable only visible in the function printName
    cin >> name;
    cout << name << endl;    // last line of code where name is in scope
}

/*****
 * MAIN
 * Because there are no parameters being passed, there is
 * no communication between main() and printName()
 *****/
int main()
{
    printName();              // no variables are in scope here
    return 0;
}

```

Here name is a local variable. Its buffer is created when printName() is called. We know for a fact that it is not used or relevant outside printName().

IF Local

Though we have not learned about IF statements yet, consider the following code:

```

{
    int first = 20;
    int second = 10;

    if (first > second)
    {
        int temp = first;    // the variable temp is in scope from here...
        first = second;
        second = temp;       // ... to here. The next line has a } ending the scope
    }

    cout << first << ", "    // only first and second are "in scope" at this point
        << second << endl;
}

```

Here the variable temp is only relevant inside the IF statement. We know this because the variable *falls out of scope* once the } is reached after the statement “second = temp;”. Because the scope of temp is IF local, it is only visible inside the IF statement. Therefore, there is no possibility for side effects.

Blocks

A variable is only visible until program execution encounters the closing `}` in which it is defined. Note that you can introduce `{}`s at any point in the program. They are called blocks. Consider the following example:

```
{
    display();

    // pause
    {
        cout << "Press any key to continue";
        char something;
        cin.get(something);
    }
}
```

// the purpose of the {}s here are to limit scope
// only "in scope" for two lines of code

Since we are going to throw away something anyway and the value is irrelevant, we want to make sure that it is never used in a way different than is intended. The block ensures this.

Globals

A global variable is defined as a variable defined outside any function, usually at the top of a file.

```
#include <iostream>
using namespace std;

int input; // global variables are evil! Be careful

/*****
 * MAIN
 * Global variables are evil!
 *****/
int main()
{
    cout << "Enter your age: ";
    cin >> input;

    if (input > 25)
        cout << "Man you are old!\n";

    return 0;
}
```

These are very problematic because they are accessible by any function in the entire program. It therefore becomes exceedingly difficult to answer questions like:

- Is the variable initialized?
- Who set the variable last?
- Who will set the variable next?
- Who will be looking at this variable and depending on its value?

Unfortunately, these questions are not only exceedingly difficult to answer with global variables, but they are exceedingly important when trying to fix bugs. For this reason, global variables are banned for all classes in the BYU-Idaho Computer Science curriculum.

Problem 1

Write the C++ statements for the following:

$$c = 2\pi r$$

$$8 + 3 = x$$

$$e = mc^2$$

$$x = \frac{1}{2}y$$

Please see page 47 for a hint.

Problem 2

What is the value of c when the expression is evaluated:

```
int f = 34;  
float c = (f - 32) * 5 / 9;
```

Answer:

Please see page 49 for a hint.

Problem 3

Write a function to display "Hello World". Call it hello()

Answer:

Please see page 60 for a hint.

Problem 4

Write a function to return a number. Call it get()

Answer:

Please see page 64 for a hint.

Problem 5

What is the output?

```
int two()
{
    return 3;
}

int main()
{
    int one  = 2;
    int three = two() + one;

    cout << three << endl;

    return 0;
}
```

Answer:

Please see page 59 for a hint.

Problem 6

What is the output?

```
void a()
{
    cout << "a";
    return;
}

void b()
{
    cout << "bb";
    return;
}

int main()
{
    a();
    b();
    a();

    return 0;
}
```

Answer:

Please see page 59 for a hint.

Problem 7

What is the output?

```
double a(double b, double c)
{
    return b - c;
}

int main()
{
    float x = a(4.0, 3.0);
    float y = a(7.0, 5.0);

    cout << a(x, y) << endl;

    return 0;
}
```

Answer:

Please see page 62 for a hint.

Problem 8

What is the output?

```
double add(double n1, double n2)
{
    return n1 + n2;
}

int main()
{
    double n3 = add(0.1, 0.2);
    double n4 = add(n3, add(0.3, 0.4));

    cout << n4 << endl;

    return 0;
}
```

Answer:

Please see page 62 for a hint.

Problem 9

What is the output?

```
void weird(int a, int &b)
{
    a = 1;
    b = 2;
}

int main()
{
    int a = 3;
    int b = 4;

    weird(a, b);

    cout << a * b << endl;
    return 0;
}
```

Answer:

Please see page 65 for a hint.

Problem 10

What is the output?

```
void setTrue(bool a)
{
    a = true;
    return;
}

int main()
{
    bool a = false;

    setTrue(a);

    cout << (int)a << endl;

    return 0;
}
```

Answer:

Please see page 65 for a hint.

Problem 11

What is the output?

```
int main()
{
    cout << a(b()) << endl;
    return 0;
}

int a(int value)
{
    return value * 2;
}

int b()
{
    return 3;
}
```

Answer:

Please see page 67 for a hint.

Problem 12

What is the output?

```
char value = 'a';

int main()
{
    char value = 'b';

    if (true)
    {
        char value = 'c';
    }

    cout << value << endl;

    return 0;
}
```

Answer:

Please see page 67 for a hint.

Assignment 1.4

You should start this assignment by copying the file `/home/cs124/assignments/assign14.cpp` to your directory:

```
cp /home/cs124/assignments/assign14.cpp assignment14.cpp
```

There are two functions that you will need to write:

Display Message

Please create a function called `questionPeter()`. The function should not return anything but instead display the following message:

```
Lord, how oft shall my brother sin against me, and I forgive him?
Till seven times?
```

Display Answer

The second function called `responseLord()` will return the Lord's response: $7 * 70$. This function will not display any output but rather return a value to the caller.

Main

`main()` is provided in the file `/home/cs124/assignments/assign14.cpp`:

```
/* *****
 * Main will not do much here. First it will display Peter's question,
 * then it will display the Lord's answer
 * ***** */
int main()
{
    // ask Peter's question
    questionPeter();

    // the first part of the Lord's response
    cout << "Jesus saith unto him, I say not unto thee, Until seven\n";
    cout << "times: but, Until " << responseLord() << ".\n";

    return 0;
}
```

Example

```
Lord, how oft shall my brother sin against me, and I forgive him?
Till seven times?
Jesus saith unto him, I say not unto thee, Until seven
times: but, Until 490.
```

Instructions

Please verify your solution against:

```
testBed cs124/assign14 assignment14.cpp
```

Don't forget to submit your assignment with the name "Assignment 14" in the header.

Please see page 60 for a hint.

1.5 Boolean Expressions

Sam is reading his scriptures one day and comes across the following verse from 2 Corinthians:

Every man according as he purposeth in his heart, so let him give; not grudgingly, or of necessity: for God loveth a Cheerful giver. (2 Corinthians 9:7)

Now he wonders: is his offering acceptable to the Lord? To address this issue, he reduces the scripture to a Boolean expression.

Objectives

By the end of this class, you will be able to:

- Declare a Boolean variable.
- Convert a logic problem into a Boolean expression.
- Recite the order of operations.

Prerequisites

Before reading this section, please make sure you are able to:

- Represent simple equations in C++ (Chapter 1.3).
- Choose the best data-type to represent your data (Chapter 1.2).

Overview

Boolean algebra is a way to express logical statements mathematically. This is important because virtually all programs need to have decision making logic. There are three parts to Boolean algebra: Boolean variables (variables enabling the programmer to store the results of Boolean expressions), Boolean operators (operations that can be performed on Boolean variables), and Comparison operators (allowing the programmer to convert a number to a Boolean value by comparing it to some value). The most common operators are:

| Math | English | C++ | Example |
|------|--------------------------|-----|----------------------|
| ~ | Not | ! | !true |
| ^ | And | && | true && false |
| ∨ | Or | | true false |
| = | Equals | == | x + 5 == 42 / 2 |
| ≠ | Not Equals | != | graduated != true |
| < | Less than | < | age < 16 |
| ≤ | Less than or equal to | <= | timeNow <= timeLimit |
| > | Greater than | > | age > 65 |
| ≥ | Greater than or equal to | >= | grade >= 90 |

And, Or, and Not

The three main logical operators we use in computer programming are And, Or, and Not. These, it turns out, are also commonly used in our spoken language as well. For example, consider the following scripture:

Every man according as he purposeth in his heart, so let him give; not grudgingly, or of necessity: for God loveth a Cheerful giver. (2 Corinthians 9:7)

This can be reduced to the following expression:

acceptable = inHisHeart and not (grudgingly or necessity)

In C++, this will be rendered as:

```
bool isAcceptable = isFromHisHeart && !(isGrudgingly || isOfNecessity);
```

This Boolean expression has all three components: And, Or, and Not.

AND

The Boolean operator AND evaluates to true only if the left-side and the right-side are both true. If either are false, the expression evaluates to false. Consider the following statement containing a Boolean AND expression:

```
bool answer = leftSide && rightSide;
```

This can be represented with a truth-table:

| AND | | Left-side | |
|------------|-------|-----------|-------|
| | | true | false |
| Right-side | true | true | false |
| | false | false | false |

If leftSide = false and rightSide = false, then leftSide && rightSide evaluates to false. This case is represented in the lower-right corner of the truth table (observe how the column corresponding to that cell has false in the header corresponding to the leftSide variable. Observe how the row corresponding to that cell has false in the header corresponding to the rightSide variable).

The AND operator is picky: it evaluates to true only when both sides are true.

OR

The Boolean operator OR evaluates to `true` if either the left-side or the right-side are `true`. If both are `false`, the expression evaluates to `false`. Consider the following statement containing a Boolean OR expression:

```
bool answer = leftSide || rightSide;
```

The corresponding truth-table is:

| OR | | Left-side | |
|------------|-------|-----------|-------|
| | | true | false |
| Right-side | true | true | true |
| | false | true | false |

If `leftSide = false` and `rightSide = true`, then `leftSide || rightSide` evaluates to `true`. This case is represented in the middle-right cell of the truth table (observe how the column corresponding to that cell has `false` in the header corresponding to the `leftSide` variable. Also note how the row corresponding to that cell has `true` in the header corresponding to the `rightSide` variable).

The OR operator is generous: it evaluates to `true` when either condition is met.

NOT

The Boolean operator NOT is a unary operator: only one operand is needed. In other words, it only operates on the value to the right of the operator. NOT evaluates to `true` when the right-side is `false` and evaluates to `false` with the right-side is `true`. Consider the following statement containing a Boolean NOT expression:

```
bool wrong = ! right;
```

The corresponding truth-table is:

| NOT | |
|------------|-------|
| Right-side | |
| true | false |
| false | true |

If `right = false` then `!right` is `true`. If `right = true` then `!right` is `false`. In other words, the NOT operator can be thought of as the “opposite operator.”

Example

Back to our scripture from the beginning:

Every man according as he purposeth in his heart, so let him give; not grudgingly, or of necessity: for God loveth a Cheerful giver. (2 Corinthians 9:7)

This, as we discussed, is the same as:

```
bool isAcceptable = isFromHisHeart && !(isGrudgingly || isOfNecessity);
```

In this case, Sam is giving from his heart (`isFromHisHeart = true`) and is not giving of necessity (`isOfNecessity = false`). Unfortunately, he is a bit resentful (`isGrudgingly = true`). Evaluation is:

1. `bool isAcceptable = isFromHisHeart && !(isGrudgingly || isOfNecessity);`
2. `bool isAcceptable = true && !(true || false);` // replace variables with values
3. `bool isAcceptable = true && !(true);` // `true || false --> true`
4. `bool isAcceptable = true && false;` // `!true --> false`
5. `bool isAcceptable = false;` // `true && false --> false`

Thus we can see that Sam's offering is not acceptable to the Lord. The grudging feelings have wiped out all the virtue from his sacrifice.

Sam's Corner



The more transformations you know, the easier it will be to work with Boolean expressions in the future. Consider the distributive property of multiplication over addition:

$$a * (b + c) == (a * b) + (a * c)$$

Knowing this algebraic transformation makes it much easier to solve equations. There is a similar Boolean transformation called DeMorgan. Consider the following equivalence relationships:

$$!(p || q) == !p \&\& !q$$

$$!(p \&\& q) == !p || !q$$

It also works for AND/OR:

$$a || (b \&\& c) == (a || b) \&\& (a || c)$$

$$a \&\& (b || c) == (a \&\& b) || (a \&\& c)$$



Sue's Tips

Boolean operators also work with numbers as well. Recall that `0` \rightarrow `false` and all values other than `0` map to `true`. When evaluating Boolean expressions containing non-Boolean values, you convert the value to a `bool` immediately before the Boolean operator is evaluated:

$$(7 \&\& 0) \rightarrow (true \&\& false) \rightarrow false$$

$$!65 \rightarrow !true \rightarrow false$$

Comparison Operators

Boolean algebra only works with Boolean values, values that evaluate to either `true` or `false`. Often times we need to make logical decisions based on values that are numeric. Comparison operators allow us to make these conversions.

Equivalence

The first class of comparison operators consists of statements of equivalence. There are two such operators: equivalence `==` and inequality `!=`. These operators will determine whether the values are the same or not. Consider the following code:

```
int grade = 100;
bool isPerfectScore = (grade == 100);
```

In this example, the Boolean variable `isPerfectScore` will evaluate to `true` only when `grade` is 100%. If `grade` is any other value (including 101%), `isPerfectScore` will evaluate to `false`. It is also possible to tell if two values are not the same:

```
int numStudents = 21;
bool isClassHeldToday = (numStudents != 0);
```

Here we can see that we should go to class today. As long as the number of students attending class (`numStudents`) does not equal zero, class is held.

Relative Operators

The final class of comparison operators performs relative (not absolute) evaluations. These are greater than `>`, less than `<`, greater than or equal to `>=`, and less than or equal to `<=`. Consider the following example using integers:

```
int numBoys = 6;
int numGirls = 8;
bool isMoreGirls = (numGirls > numBoys);
```


This works in much the same way when we compare floating point numbers. Note that since floating point numbers (`float`, `double`, `long double`) are approximations, there is little difference between `>` and `>=`.

```
float grade = 82.5;
bool hasPassedCS124 = (grade >= 60.0); // passed greater than or equal to 60%
```

Finally, we can even use relative operators with chars. In these cases, it is important to remember that each letter in the ASCII table corresponds to a number. While we need not memorize the ASCII table, it is useful to remember that the letters are alphabetical and that uppercase letters are before lowercase letters:

```
char letterGrade = 'B';
bool goodGrade = ('C' >= letterGrade);
```

Example 1.5 – Decision Function

| | |
|-----------|---|
| Demo | This example will demonstrate how to write a function to help make a decision. This will be a binary decision (choosing between two options) so the return type will be a <code>bool</code> . |
| Problem | <p>Write a program to compute whether a user qualifies for the child tax credit. The rule states you qualify if you make less than \$110,000 a year (the actual rule is quite a bit more complex, of course!). Note that you either qualify or you don't: there are only two possible outcomes. If you do qualify, then the credit is \$1,000 per child. If you don't, no tax credit is awarded.</p> <pre>What is your income: <u>115000.00</u> How many children? 2 Child Tax Credit: \$ 0.00</pre> |
| Solution | <p>The key part of this problem is the function deciding whether the user qualifies for the child tax credit. The input is income as a <code>float</code> and the output is the decision as a <code>bool</code>.</p> <pre> /***** * QUALIFY * Does the user qualify for the tax credit? * This will return a BOOL because you either * qualify, or you don't! *****/ bool qualify(double income) { return (income <= 110000.00); }</pre> <p>Observe how the name of the function implies what true means. In other words, if <code>qualify()</code> returns true, then the user qualifies. If <code>qualify()</code> returns false, then the user doesn't. Always make sure the name of the function implies what true means when working with a <code>bool</code> function.</p> <p>The next part is computing the credit to be awarded. This will require an IF statement which will be discussed next chapter.</p> <pre> if (qualify(income)) cout << 1000.00 * (float)numChildren << endl; else cout << 0.00 << endl;</pre> <p>Notice how the return value of the <code>qualify()</code> function goes directly into the IF statement.</p> |
| Challenge | <p>It turns out that the child tax credit is actually more complex than this. The taxpayer gets the full \$1,000 for every child if the income is less than \$110,000 but it phases out at the rate of 5¢ for each \$1 after that. In other words, a family making \$120,000 will only receive \$500 per child. Thus there is no credit possible for families making more than \$130,000.</p> <p>As a challenge, modify the above example to more accurately reflect the law.</p> |
| See Also | <p>The complete solution is available at 1-5-decisionFunction.cpp or:</p> <pre>/home/cs124/examples/1-5-decisionFunction.cpp</pre>  |

Order of Operations

With all these Boolean operators, the order of operations table has become quite complex (a more complete version of this table is in [Appendix B](#)):

| | | | | | |
|-----------|-------------------------------|--|----------|-------|--------|
| () | Parentheses | | | | |
| ++ -- | Increment, decrement | | | Math | Unary |
| ! | Not | | | Logic | |
| * / % | Multiply, divide, modulo | | | Math | Binary |
| + - | Addition, subtraction | | | | |
| > >= < <= | Greater than, less than, etc. | | Relative | Logic | |
| == != | Equality | | Absolute | | |
| && | And | | AND | | |
| | Or | | OR | | |
| = += *= | Assignment | | | | |

There are a couple things to remember when trying to memorize the order of operations:

1. **Unary Before Binary:** When an operator only takes one operand (such as `x++` or `!true`), it goes at the top of the table. When an operator takes two (such as `3 + 6` or `grade > 60`), it goes at the bottom of the table.
2. **Math Before Logic:** Arithmetic operators (such as addition or multiplication) go before Boolean operators (such as AND or Greater-than). This means that operations evaluating to a `bool` go after operations evaluating to numbers.
3. **Relative Before Absolute:** Conditional operators making a relative comparison (such as greater-than `>`) go before those making absolute comparisons (such as not-equal `!=`).
4. **AND Before OR:** This is one of those things to just memorize. Possibly you can remember that they are in alphabetical order?



Sue's Tips

While it is useful (and indeed necessary!) to memorize the order of operations, please don't expect the readers of your code to do the same. It is far better to disambiguate your expressions by using many parentheses. This gives the bugs nowhere to hide!

Problem 1, 2

Write a function to multiply two numbers. Call the function `multiply()`.

Write `main()` to prompt the user for two numbers, call `multiply()`, and display the product.

Please see page 64 for a hint.

Problem 3

Write a function to represent the prerequisites for CS 165: you must pass CS 124 and Math 110.

Please see page 81 for a hint.

Problem 4

Write a function to represent how to pass this class: you can either earn a grade greater than or equal to 60% or you must bribe the professor. Realize, of course, that this is not how to pass the class...

Please see page 81 for a hint.

Problem 5-11

What is the value for each of the following variables?

| | |
|-------------------------------|----------------------|
| { | |
| bool a = ('a' < 'a'); | <input type="text"/> |
| bool b = ('b' > 'a'); | <input type="text"/> |
| bool c = (a * 4) && b; | <input type="text"/> |
| bool d = !(b (c true)); | <input type="text"/> |
| bool e = a && b && c && d; | <input type="text"/> |
| bool f = a b c d; | <input type="text"/> |
| bool g = (a != b) && true; | <input type="text"/> |
| } | |

Please see page 82 for a hint.

Problem 12-16

For each of the following, indicate where the parentheses goes to disambiguate the order of operations:

| Raw expression | With parentheses |
|--------------------------|----------------------|
| 1 + 2 > 3 * 3 | <input type="text"/> |
| ! a < b | <input type="text"/> |
| a + b && c d | <input type="text"/> |
| 2 * c++ > 2 + 7 == 9 % 2 | <input type="text"/> |
| a > b > c > d | <input type="text"/> |

Please see page 49 for a hint.

Assignment 1.5

Write a function to determine if an individual is a full tithe payer. This program will have one function that accepts as parameters the income and payment, and will return whether or not the user is a full tithe payer. The return type will need to be a Boolean value. Note that `main()` is already written for you. Also note that the skeleton of `isFullTithePayer()` is written, but there is more code to be written in the function for it to work as desired.

For this assignment, `main()` will be provided at:

```
/home/cs124/assignments/assign15.cpp
```

Please copy this file and use it as you did the templates up to this point.

Example

Two examples. The user input is in underline.

Example 1: Full Tithe Payer

```
Income: 100  
Tithe: 91  
You are a full tithe payer.
```

Example 2: Not a Full Tithe Payer

```
Income: 532  
Tithe: 40  
Will a man rob God? Yet ye have robbed me.  
But ye say, Wherein have we robbed thee?  
In tithes and offerings. Malachi 3:8
```

Instructions

The test bed is available at:

```
testBed cs124/assign15 assignment15.cpp
```

Don't forget to submit your assignment with the name "Assignment 15" in the header.

Please see page 81 for a hint.

1.6 IF Statements

Sue has just received her third text message in the last minute. Not only are her best friends text-aholics, but it seems that new people are texting her every day. Sometimes she feels like she is swimming in a sea of text messages. “If only I could filter them like I do my e-mail messages” thinks Sue as four new messages appear on her phone in quick succession. Deciding to put a stop to this madness, she writes a program to filter her text messages. This program features a series of IF statements, each containing a rule to route the messages to the appropriate channel.

Objectives

By the end of this class, you will be able to:

- Create an IF statement to modify program flow.
- Recognize the pitfalls associated with IF statements.

Prerequisites

Before reading this section, please make sure you are able to:

- Declare a Boolean variable (Chapter 1.5).
- Convert a logic problem into a Boolean expression (Chapter 1.5).
- Recite the order of operations (Chapter 1.5)

Overview

IF statements allow the program to choose between two courses of action depending on the result of a Boolean expression. In some cases, the options are “action” and “no action.” In other cases, the options may be “action A” or “action B.” In each of these cases, the IF statement is the tool of choice.

Action/No-Action

The first flavor of the IF statement is represented with the following syntax:

```
if (<Boolean expression>)  
    <body statement>;
```

For example:

```
{  
    if (assignmentLate == true)  
        assignmentGrade = 0;  
}
```

The Boolean expression, also called the controlling expression, determines whether the statements inside the body of the loop are to be executed. If the Boolean expression (`assignmentLate == true` in this case) evaluates to `true`, then control enters the body of the IF statement (`assignmentGrade = 0;`). Otherwise, the body of the IF statement is skipped.

Action-A/Action-B

The second flavor of the IF statement is represented with the following syntax:

```
if (<Boolean expression>)
    <body statement>;
else
    <body statement>;
```

For example:

```
{
    if (grade >= 60)
        cout << "Great job! You passed!\n";
    else
        cout << "I will see you again next semester...\n";
}
```

Much like the Action/No-Action IF statement, the Boolean expression determines whether the first set of statements is executed (`cout << "Great job! You passed!\n";`) or the second (`cout << "I will see you again next semester\n";`). The first statement is often called the “true condition” and the second the “else condition”.

Observe how the `else` component of the IF statement aligns with the `if`. Also, both the true-condition and the else-condition are indented the same: three additional spaces. Finally, note that there is no semicolon after the Boolean expression nor after the `else`. This is because the entire IF-ELSE statement is considered a single C++ statement.

Sam's Corner



IF statements in C++ are compiled into JUMPZ (or one of many conditional jump) assembly statements. When the CPU encounters these statements, execution could either proceed to the next instruction or jump to the included address, depending on whether the Boolean expression is TRUE or not. Since CPUs like to look ahead in an effort to optimize processor performance, a decision must be made: is it more likely the Boolean expression evaluates to TRUE or FALSE? As a rule, all CPUs optimize on the TRUE condition. For this reason, there is a slight performance advantage for the TRUE condition to be the “most likely” of the two conditions.

Consider, for example, the above code. Since the vast majority of the students will achieve a grade of greater than 60%, the “Great job!” statement should be in the true-condition and the “next semester” statement should be in the else-condition. This will be slightly more efficient than the following code:

```
if (grade < 60)
    cout << "I will see you again next semester...\n";
else
    cout << "Great job! You passed!\n";
```

Example 1.6 – IF Statements

Demo

This example will demonstrate both types of IF statements: the Action/No-Action type and the Action-A/Action-B type.

Problem

Write a program to prompt the user for his GPA and display whether the value is in the valid range.

```
Please enter your GPA: 4.01
Your GPA is not in the valid range
```

The first part of the program is a function determining whether the GPA is within the acceptable range. It will take a `float` GPA as input and return a `bool`, whether the value is valid.

```

/*****
 * Demonstrating an Action-A/Action-B IF statement
 *****/
bool validGpa(float gpa)
{
    if (gpa > 4.0 || gpa < 0.0)           // Boolean expression
        return false;                   // True condition
    else
        return true;                     // False or Else condition
}

```

Note how two options are presented, the invalid range and the valid range.

The second part of the program displays an error message only in the case where GPA is outside the accepted range. This is an example of the Action/No-Action flavor.

```

/*****
 * Demonstrating an Action/No-Action IF statement
 *****/
int main()
{
    float gpa;

    // prompt for GPA
    cout << "Please enter your GPA: ";
    cin >> gpa;

    // give error message if invalid
    if (!validGpa(gpa))                 // Boolean expression
        cout << "Your GPA is not in a valid range\n"; // Action or Body of the IF

    return 0;
}

```

Challenge

As a challenge, see if you can modify the IF statement in `main()` to the Action-A/Action-B variety by displaying a message if the user has entered a valid GPA.

Another challenge would be to remove the IF statement from the `validGpa()` function and replace it with a simple boolean expression similar to what was done in Chapter 1.5

See Also

The complete solution is available at [1-6-ifStatements.cpp](#) or:

```
/home/cs124/examples/1-6-ifStatements.cpp
```



Details

Anytime there is a place for a statement in C++, multiple statements can be added by using {}s. Similarly, whenever there is a place for a statement in C++, any statement can go there. For example, the body of an IF statement could contain another IF statement.

Compound Statements

Frequently we want to have more than one statement inside the body of the IF. Rather than duplicating the IF statement, we use {}s to surround all the statements going inside the body:

```
{
    if (!(classGrade >= 60))
    {
        classFail = true;
        classRetake = true;
        studentHappy = false;
    }
}
```

Each time an additional indentation is added, three more spaces are used. In this case, the IF statement is indented 3 spaces. Observe how the {}s align with the IF statement. The three assignment statements (such as `classFail = true;`) are indented an additional three spaces for a total of 6.

Nested Statements

Often we want to put an IF statement inside the body of another IF statement. This is called a nested statement because one statement is inside of (or part of) another:

```
{
    if (grade >= 90 && grade <= 100)
    {
        cout << "A";
        if (grade <= 93)
            cout << "-";
    }
}
```

Observe how the second COUT statement is indented 9 spaces, three more than the inner IF and six more than the outer IF. There really is no limit to the number of degrees of nesting you can use. However, when indentation gets too extreme (much more than 12), human readability of code often suffers.

Multi-Way

An IF statement can only differentiate between two options. However, often the program requires more than two options. This can be addressed by nesting IF statements:

```
{
    if (option == 1)
        cout << "Good!";
    else
    {
        if (option == 2)
            cout << "Better";
        else
            cout << "Best!";
    }
}
```

Observe how the inner {}s are actually not necessary. We only need to add {}s when more than one statement is used. Since a complete IF/ELSE statement (otherwise known as the Action-A/Action-B variant of an IF statement) is a single statement, {}s are not needed. Thus we could say:

```
{
    if (option == 1)
        cout << "Good!";
    else
        if (option == 2)
            cout << "Better";
        else
            cout << "Best!";
}
```

If we just change the spacing a little, we can re-arrange the code to a much more readable:

```
{
    if (option == 1)
        cout << "Good!";
    else if (option == 2)
        cout << "Better";
    else
        cout << "Best!";
}
```

This is the preferred style for a multi-way IF. Technically speaking, we can achieve a multi-way IF without resorting to ELSE statements.



Sue's Tips

Be careful and deliberate in the order in which the IF statements are arranged in multi-way IFs. Not only may a bug exist if they are in the incorrect order, but there may be performance implications as well. Make sure to put the most-likely or common cases at the top and the less-likely ones at the bottom.

Consider the following code:

```
{
    if (numberGrade >= 90 && numberGrade <= 100)
        letterGrade = 'A';

    if (numberGrade >= 80 && numberGrade < 90)
        letterGrade = 'B';

    if (numberGrade >= 70 && numberGrade < 80)
        letterGrade = 'C';

    if (numberGrade >= 60 && numberGrade < 70)
        letterGrade = 'D';

    if (numberGrade < 60)
        letterGrade = 'F';
}
```

Observe how each of the five IF statements stands on its own. This means that, every time the code is executed, each IF statement's Boolean expression will need to be evaluated. Also note how much of the Boolean

expressions are redundant. This statement has exactly the same descriptive power as the following multi-way IF:

```
{
    if (numberGrade >= 90)
        letterGrade = 'A';

    else if (numberGrade >= 80)
        letterGrade = 'B';

    else if (numberGrade >= 70)
        letterGrade = 'C';

    else if (numberGrade >= 60)
        letterGrade = 'D';

    else
        letterGrade = 'F';
}
```

Not only is this code much easier to read (simpler Boolean expressions) and less bug-prone (there is no redundancy), it is also much more efficient. Consider the case where `numberGrade == 93`. In this case, the first Boolean expression will evaluate to `true` and the body of the first IF statement will be executed. Since the entire rest of the multi-way IF is part of the ELSE condition of the first IF statement, it will all be skipped. Thus, far less code will be executed.

Pitfalls

The C++ language was designed to be as efficient and high-performance as possible. In other words, it was designed to facilitate making an efficient compiler so the resulting machine language executes quickly on the CPU. The C++ language was not designed to be easy to learn or easy to write code. Modern derivatives of C++ such as Java and C# were designed with that in mind. Taking this point into account, C++ programmers should always be on the look-out for various pitfalls that beset the language.

Pitfall: = instead of ==

Algebra treats the equals sign as a statement of equivalence, much like C++ treats the double equals sign. It is therefore common to mistakenly use a single equals when a double is needed:

```
{
    bool fail = false;
    if (fail = true)                // PITFALL: Assignment = used instead of ==
        cout << "You failed!\n";
}
```

In this statement, it may look like the program will display a message if the user has failed the class. Since the first statement sets `fail` to `false`, we will not execute the `cout` in the body of the IF. Closer inspection, however, will reveal that we are not *comparing* `fail` with `true`. Instead we are *setting* `fail` to `true`. Thus, the variable will change and the Boolean expression will evaluate to `true`.

Pitfall: Extra semicolon

Remember that the semicolon signifies the end of a statement. The end of an IF statement is the end of the statement inside of the body of the IF. Thus, if there is a semicolon after the Boolean expression, we are signifying that there is no body in the IF!

```
{
    if (false);                // PITFALL: Extra semicolon signifies empty body
        cout << "False!\n";
}
```

Because of the semicolon after the (false), the above code is equivalent to:

```
{
    if (false)                // If you mean to have an empty body,
        ;                    // then put it on its own line like this.
    cout << "False!\n";
}
```

In other words, the Boolean expression is ignored and the body of the IF is always executed!



Sue's Tips

Train your eye to look for extra semicolons on IF statements. They should never be there! If you meant to have an empty body (and you shouldn't!), then put the semicolon on its own line so the reader of the code knows it was intentional.

If the code editor tries to out-dent the body of your IF statement, pay attention: it is trying to tell you something important! The editor knows about semicolons and IF statements and is not fooled by this pitfall.

Pitfall: Missing {}s


In order to use a compound statement (more than one statement) in the body of an IF, it is necessary to surround the statements with {}s. A common mistake is to forget the {}s and just indent the statements. C++ ignores indentations (they are just used for human readability; the compiler throws away all white-spaces during the lexing process) and will not know the statements need to be in the body:

```
{
    if (classGrade < 60)
        classFail = true;
        classRetake = true;        // PITFALL: Missing {}s around the body of the IF
        studentHappy = false;
}
```

This is exactly the same as:

```
{
    if (classGrade < 60)
        classFail = true;
    classRetake = true;
    studentHappy = false;
}
```

Observe how only the first statement (classFail = true;) is part of the IF.

| Example 1.6 – Overtime | |
|------------------------|---|
| Demo | This example will demonstrate how to send data to a function and receive data from a function. This follows the “Processing” model of information flow through a function. |
| Problem | <p>Write a program to compute the hourly wage taking into account time-and-a-half overtime. In other words, if more than 40 hours are worked, then any additional hour benefits from a 50% increase in the wage. This can be solved only after the program makes a decision: are we using the regular formula (<code>hourlyWage * hoursWorked</code>) or the more complex overtime formula.</p> <pre>What is your hourly wage? <u>10</u> How many hours did you work? <u>41</u> Pay: \$ 415.00</pre> |
| Solution | <p>The function to compute pay taking the hourly wage and hours worked as input.</p> <pre> /***** * COMPUTE PAY * Compute the user's pay using time-and-a-half * overtime. *****/ float computePay(float hourlyWage, float hoursWorked) { float pay; // regular rate if (hoursWorked < 40) pay = hoursWorked * hourlyWage; // regular rate // overtime rate else pay = (40.0 * hourlyWage) + // first 40 are normal... ((hoursWorked - 40.0) * (hourlyWage * 1.5)); // ...the balance overtime return pay; } </pre> |
| Challenge | Some companies credit employees with an hour of work each month even if they only worked a few minutes. In other words, there are four pay rates: no pay for those who did not work, a full hour’s wage for those working less than an hour a week, regular wage, and the overtime wage. As a challenge, modify the above function to include this first-hour special case. |
| See Also | <p>The complete solution is available at 1-6-overtime.cpp or:</p> <pre>/home/cs124/examples/1-6-overtime.cpp</pre>  |

Problem 1

What is the output?

```
void function(int b, int a)
{
    cout << "a == " << a << '\t'
        << "b == " << b << endl;
}

int main()
{
    int a = 10;
    int b = 20;

    cout << "a == " << a << '\t'
        << "b == " << b << endl;

    function(a, b);

    return 0;
}
```

Answer:

Please see page 62 for a hint.

Problem 2-8

What are the values of the following variables?:

```
{
    bool a = false && true || false && true;
    bool b = false || true && false || true;
    bool c = true && true && true && false;
    bool d = false || false || false || true;
    bool e = 100 > 90 > 80;
    bool f = 90 < 80 || 70;
    bool g = 10 + 2 - false;
}
```

Please see page 79 for a hint.

Problem 9-13

For each of the following, indicate where the parentheses goes to disambiguate the order of operations:

| Raw expression | With parentheses |
|--------------------------------|------------------|
| 4 + 1 > 2 * 2 | |
| a++ < !b | |
| a * b + c && d e | |
| 3.1 * ! b > 7 * a++ == ++c + 2 | |
| a < b < c < d | |

Please see page 82 for a hint.

Problem 14

What is the output?

```
int subtract(int b, int a)
{
    return a - b;
}

int main()
{
    int c = subtract(4, 3);

    cout << c << endl;

    return 0;
}
```

Answer:

Please see page 62 for a hint.

Problem 15

Write a function to accept a number from the caller as a parameter and return whether the number is positive:

Please see page 87 for a hint.

Problem 16

What is the output?

```
{
    bool failedClass = false;
    int grade = 95;

    // pass or fail?
    if (grade < 60);
        failedClass = true;

    // output grade
    cout << grade << "%\n";

    // output status
    if (failedClass)
        cout << "You need to take "
            << "the class again\n";
}
```

Answer:

Please see page 92 for a hint.

Problem 17

What is the output?

```
{
    bool failedClass = false;
    int grade = 95;

    // pass or fail?
    if (grade = 60)
        failedClass = true;

    // output grade
    cout << grade << "%\n";

    // output status
    if (failedClass)
        cout << "You need to take "
            << "the class again\n";
}
```

Answer:

Please see page 91 for a hint.

Problem 18

What is the output when the user inputs the letter 'm'?

```
{
    char gender;

    // prompt for gender
    cout << "What is your gender? (m/f)";
    cin >> gender;

    // turn it into a bool
    bool isMale = true;
    if (gender == 'f');
        isMale = false;

    // output the result
    if (isMale)
        cout << "You are male!\n";
    else
        cout << "You are female!\n";
}
```

Answer:

Please see page 92 for a hint.

Problem 19

What is the output when the user inputs the number 5?

```
{
    int number;

    // prompt for number
    cout << "number? ";
    cin >> number;

    // crazy math
    if (number = 0)
        number += 2;

    // output
    cout << number << endl;
}
```

Answer:

Please see page 91 for a hint.

Assignment 1.6

Write a function (`computeTax()`) to determine which tax bracket a user is in. The tax tables are:

| If taxable income is over-- | But not over-- | Bracket |
|-----------------------------|----------------|---------|
| \$0 | \$15,100 | 10% |
| \$15,100 | \$61,300 | 15% |
| \$61,300 | \$123,700 | 25% |
| \$123,700 | \$188,450 | 28% |
| \$188,450 | \$336,550 | 33% |
| \$336,550 | no limit | 35% |

The yearly income is passed as a parameter to the function. The function returns the percentage bracket that the user's income falls in. The return value from the function will be an integer (10, 15, 25, 28, 33, or 35).

Next write `main()` so that it prompts the user for his or her income and accepts the result from the `computeTax()` function and displays the result to the screen with a “%” after the number.

Examples

Three examples. The user input is in underline.

Example 1: 28%

Income: 150000
Your tax bracket is 28%

Example 2: 35%

Income: 1000000
Your tax bracket is 35%

Example 3: 10%

Income: 5
Your tax bracket is 10%

Assignment

The test bed is available at:

testBed cs124/assign16 assignment16.cpp

Don't forget to submit your assignment with the name “Assignment 16” in the header.

Please see page 89 for a hint.

Unit 1 Practice Test

Practice 1.2

Write a program to prompt the user for his grade on a test, and inform him if he passed:

Example

User input is underlined.

```
What was your grade on the last test? 92  
You passed the test.
```

Another example

```
What was your grade on the last test? 59  
You failed the test.
```

Three Functions

Your program needs to have three functions: one to prompt the user for his grade, one to display the "passed" message, and one to display the "failed" message.

Assignment

Please copy into your home directory the course template from:

```
/home/cs124/tests/templateTest1.cpp
```

Use it to create the file for your test. Please:

- Write the program.
- Compile it.
- Run it to make sure it gives you the output you expect.
- Run the style checker and fix any style errors.
- Run the test bed and make sure that the output is exactly as expected:

```
testBed cs124/practice12 practice12.cpp
```

A sample solution is on:

```
/home/cs124/tests/practice12.cpp
```

Continued on the next page

Grading for Test1

Sample grading criteria:

| | Exceptional 100% | Good 90% | Acceptable 70% | Developing 50% | Missing 0% |
|--------------------------|---|---|---------------------------------------|---|---|
| Copy template 10% | Template is copied | | | Need a hint | Something other than the standard template is used |
| Compile 20% | No compile errors or warnings | One warning | One error | Two errors | Three or more compile errors |
| Modularization 20% | Functions used effectively in the program | No bugs exist in the declaration or use of functions | One bug exists in the function | Two bugs | All the code exists in one function |
| Conditional 10% | The conditional is both elegant and efficient | A conditional exists that determines if the grade is sufficient | A bug exists in the conditional | Elements of the solution are present | No attempt was made at the solution |
| I/O 20% | Zero test bed errors | Looks the same on screen, but minor test bed errors | One major test bed error | One or more tests pass test bed | Program input and output do not resemble the problem |
| Programming Style 20% | Well commented, meaningful variable names, effective use of blank lines | Zero style checker errors | One or two minor style checker errors | Code is readable, but serious style infractions | No evidence of the principles of "elements of style" in the program |

Unit 1 Project : Monthly Budget

Our first project will be to write a program to manage a user's personal finances for a month. This program will ask the user for various pieces of financial information then will then display a report of whether the user is on target to meet his or her financial goals.

This project will be done in three phases:

- Project 02 : Prompt the user for input and display the input back in a table
- Project 03 : Split the program into separate functions and do some of the budget calculations
- Project 04 : Determine the tax burden

Interface Design

The following is an example run of the program. An example of input is underlined.

```
This program keeps track of your monthly budget
Please enter the following:
```

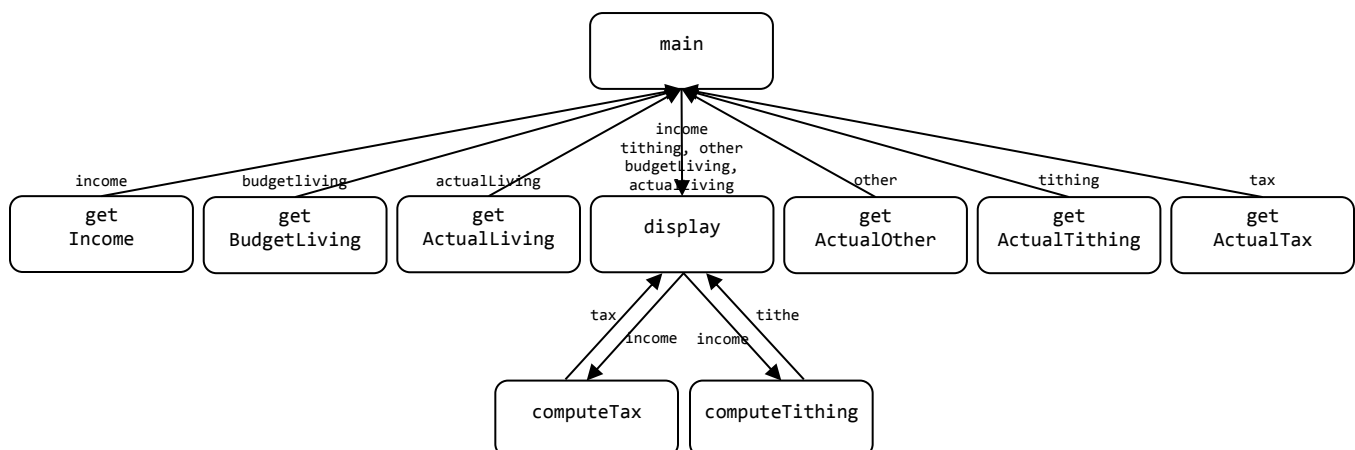
```
Your monthly income: 1000.00
Your budgeted living expenses: 650.00
Your actual living expenses: 700.00
Your actual taxes withheld: 100.00
Your actual tithe offerings: 120.00
Your actual other expenses: 150.00
```

```
The following is a report on your monthly expenses
```

| Item | Budget | Actual |
|------------|------------|------------|
| Income | \$ 1000.00 | \$ 1000.00 |
| Taxes | \$ 100.00 | \$ 100.00 |
| Tithing | \$ 100.00 | \$ 120.00 |
| Living | \$ 650.00 | \$ 700.00 |
| Other | \$ 150.00 | \$ 150.00 |
| Difference | \$ 0.00 | \$ -70.00 |

Structure Chart

You may choose to use the following functions as part of your design:



Algorithms

main()

Main is the function that signifies the beginning of execution. Typically `main()` does not do anything; it just calls other functions to get the job done. You can think of `main()` as a delegator. For this program, `main()` will call the get functions, call the `display()` function, and hold the values that the user has input. The pseudocode (described in [chapter 2.2](#)) for `main()` is:

```
main
  PUT greeting on the screen

  income ← getIncome()
  budgetLiving ← getBudgetLiving()
  actualLiving ← getActualLiving()
  actualTax ← getActualTax()
  actualTithing ← getActualTithing()
  actualOther ← getActualOther()

  display(income, budgetLiving, actualTax, actualTithing, actualLiving, actualOther)
end
```

getIncome()

The purpose of `getIncome()` is to prompt the user for his income and return the value to `main()`:

```
getIncome
  PROMPT for income
  GET income
  RETURN income
end
```

The pseudocode for the other get functions is the same. Note that all input from the users is gathered in the “get” functions. Also note that there is a tab before the “Your monthly income:”

display()

Display takes the input gathered from the other modules and puts it on the screen in an easy to read format. Display formats the output, displays some of the data to the user, and calls other functions to display the rest. The pseudocode for `display()` is the following:

```
display ( income, budgetLiving, actualTax, actualTithing, actualLiving, actualOther)
  SET budgetTax ← computeTax(income)
  SET budgetTithing ← computeTithing(income)
  SET budgetOther ← income – budgetTax – budgetTithing – budgetLiving
  SET actualDifference ← income – actualTax – actualTithing – actualLiving – actualOther
  SET budgetDifference ← 0

  PUT row header on the screen

  PUT income
  PUT budgetTax, actualTax,
  PUT budgetTithing, actualTithing,
  PUT budgetLiving, actualLiving,
  PUT budgetOther, actualOther,
  PUT budgetDifference, actualDifference
end
```

A few hints:

- A tab used for most of the indentations.
- The difference row is the difference between income and expense. Note that the difference for Budget should be zero: you plan on balancing your budget!
- Please follow the design presented in the Structure Chart (described in [chapter 2.0](#)) for your project. You may choose to add functions to increase code clarity (such as the `budgetOther` and `actualDifference` computation in `display()`).

`computeTithing()`

The purpose of `computeTithing()` is to determine amount that is required to be a full tithe payer. This does not include fast offerings and other offerings. The pseudocode for `computeTithing()` is:

And after that, those who have thus been tithed shall pay one-tenth of all their interest annually; and this shall be a standard law unto them forever, for my holy priesthood, saith the Lord. D&C 119:4

`computeTax()`

In order to determine the tax burden, it is necessary to project the monthly income to yearly income, compute the tax, and reduce that amount back to a monthly amount. In each case, it is necessary to determine the tax bracket of the individual and to then apply the appropriate formula. The tax brackets for the 2006 year are:

| If taxable income is over-- | But not over-- | The tax is: |
|-----------------------------|----------------|---|
| \$0 | \$15,100 | 10% of the amount over \$0 |
| \$15,100 | \$61,300 | \$1,510.00 plus 15% of the amount over 15,100 |
| \$61,300 | \$123,700 | \$8,440.00 plus 25% of the amount over 61,300 |
| \$123,700 | \$188,450 | \$24,040.00 plus 28% of the amount over 123,700 |
| \$188,450 | \$336,550 | \$42,170.00 plus 33% of the amount over 188,450 |
| \$336,550 | no limit | \$91,043.00 plus 35% of the amount over 336,550 |

The pseudocode for `computeTax()` is the following:

```

computeTax (monthlyIncome)
    yearlyIncome ← monthlyIncome * 12

    if ($0 ≤ yearlyIncome < $15,100)
        yearlyTax ← yearlyIncome * 0.10
    if ($15,100 ≤ yearlyIncome < $61,300)
        yearlyTax ← $1,510 + 0.15 *(yearlyIncome - $15,100)
    if ($61,300 ≤ yearlyIncome < $123,700)
        yearlyTax ← $8,440 + 0.25 *(yearlyIncome - $61,300)
    if ($123,700 ≤ yearlyIncome < $188,450)
        yearlyTax ← $24,040 + 0.28 *(yearlyIncome - $123,700)
    if ($188,450 ≤ yearlyIncome < $336,550)
        yearlyTax ← $42,170 + 0.33 *(yearlyIncome - $188,450)
    if ($336,550 ≤ yearlyIncome)
        yearlyTax ← $91,043 + 0.35 *(yearlyIncome - $336,550)

    monthlyTax ← yearlyTax / 12

    return monthlyTax
end

```

Note that this algorithm is vastly oversimplified because it does not take into account deductions and other credits. Please do not use this algorithm to compute your actual tax burden!

Project 02

The first submission point due at the end of Week 02 is to prompt the user for input and display the budget back on the screen:

```
This program keeps track of your monthly budget
Please enter the following:
```

```
Your monthly income: 1000.00
Your budgeted living expenses: 650.00
Your actual living expenses: 700.00
Your actual taxes withheld: 100.00
Your actual tithe offerings: 120.00
Your actual other expenses: 150.00
```

```
The following is a report on your monthly expenses
```

| Item | Budget | Actual |
|------------|------------|------------|
| Income | \$ 1000.00 | \$ 1000.00 |
| Taxes | \$ 0.00 | \$ 100.00 |
| Tithing | \$ 0.00 | \$ 120.00 |
| Living | \$ 650.00 | \$ 700.00 |
| Other | \$ 0.00 | \$ 150.00 |
| Difference | \$ 0.00 | \$ 0.00 |

A few hints:

- A tab used for most of the indentations and nowhere else.
- There are 15 '='s under Income, Budget, and Actual.
- The user's monthly income is used both for the Budget value and for the Actual value
- The Budget value for Taxes, Tithing, and Other will always be zero. Also the Difference, both Budget and Actual, will be zero. We will compute these in the next two parts of this project.

To complete this project, please do the following:

1. Copy the course template from:

```
/home/cs124/template.cpp
```

2. Write each function. Test them individually before “hooking them up” to the rest of the program.
3. Compile and run the program to ensure that it works as you expect:

```
g++ project02.cpp
```

4. Test the program with testbed and fix all the errors:

```
testBed cs124/project02 project1.cpp
```

5. Run the style checker and fix all the errors:

```
styleChecker project02.cpp
```

6. Submit it with “Project 02, Monthly Budget” in the program header:

```
submit project02.cpp
```

Project 03

This second part of the Monthly Budget project will be to divide the program into functions and perform some of the simple calculations:

```
This program keeps track of your monthly budget
Please enter the following:
```

```
Your monthly income: 1000.00
Your budgeted living expenses: 650.00
Your actual living expenses: 700.00
Your actual taxes withheld: 100.00
Your actual tithe offerings: 120.00
Your actual other expenses: 150.00
```

```
The following is a report on your monthly expenses
```

```
=====
Income      $    1000.00   $    1000.00
Taxes       $      0.00   $     100.00
Tithing      $    100.00   $     120.00
Living       $    650.00   $     700.00
Other        $    250.00   $     150.00
=====
Difference   $      0.00   $     -70.00
```

Notice how many of the values that were previously 0.00 now are computed. You will also need to calculate the values for Tithing, Budget Other, and Actual Difference. You can find the formula for these calculations earlier in the project description.

To complete this project, please do the following:

1. Start from the work you did in Project 02.
2. Fix any defects.
3. Write each function. Test them individually before "hooking them up" to the rest of the program.
4. Compile and run the program to ensure that it works as you expect:

```
g++ project03.cpp
```

5. Test the program with testbed and fix all the errors:

```
testBed cs124/project03 project03.cpp
```

6. Run the style checker and fix all the errors:

```
styleChecker project03.cpp
```

7. Submit it with "Project 03, Monthly Budget" in the program header:

```
submit project03.cpp
```

An executable version of the project is available at:

```
/home/cs124/projects/prj03.out
```

Project 04

This final part of the Monthly Budget project will be to add the compute tax component.

This program keeps track of your monthly budget
Please enter the following:

Your monthly income: 1000.00
Your budgeted living expenses: 650.00
Your actual living expenses: 700.00
Your actual taxes withheld: 100.00
Your actual tithe offerings: 120.00
Your actual other expenses: 150.00

The following is a report on your monthly expenses

| | | | | |
|------------|----|---------|----|---------|
| Income | \$ | 1000.00 | \$ | 1000.00 |
| Taxes | \$ | 100.00 | \$ | 100.00 |
| Tithing | \$ | 100.00 | \$ | 120.00 |
| Living | \$ | 650.00 | \$ | 700.00 |
| Other | \$ | 150.00 | \$ | 150.00 |
| ===== | | | | |
| Difference | \$ | 0.00 | \$ | -70.00 |

Notice how the taxes were computed to \$100.00 where in Project 02 they were set to 0.00.

To complete this project, please do the following:

1. Start from the work you did in Project 03.
2. Fix any defects and implement the `computeTax()` function.
3. Compile and run the program to ensure that it works as you expect.:

```
g++ project04.cpp
```

4. Test the program with testbed and fix all the errors:

```
testBed cs124/project04 project04.cpp
```

5. Run the style checker and fix all the errors:

```
styleChecker project04.cpp
```

6. Submit it with "Project 04, Monthly Budget" in the program header:

```
submit project04.cpp
```

An executable version of the project is available at:

```
/home/cs124/projects/prj04.out
```