Table of Contents

Course Overview

Procedural programming is the process of solving programming challenges by breaking large problems into smaller ones. These sub-problems are called procedures.

Goals

The goal of this class is that each student will be able to solve problems in C++ and have a solid foundation in software development methodology. By the end of the semester…

- You will be well prepared for CS 165 and the other course in computing majors. This class is the foundation on which all Computer Science (CS), Electrical Engineering (EE), and Electrical & Computer Engineering (ECEN) courses are based.
- You will have confidence in your ability to solve problems with C++. This class will include tons of opportunities for hands-on programming. By the end of the semester, you will have written more than thirty programs.
- You will possess a tool-bag of different approaches to solve software problems. Not only will we learn the mechanics of the C++ programming language, we will also learn how to solve programming problems using procedural tools.
- You will have a firm foundation in the basic constructs of procedural C++. All the components of procedural programming (except structures) will be learned in CS 124. Object Oriented Programming, the second half of the C++ language, is the topic of next semester.

These goals will be explored in the context of C++ using the Linux operating system.

Course Layout

The course will be broken into four sections:

1. **Simple Programs**. How to write your first program and begin thinking like a programmer. At the end of this section, you will be able to write a program with multiple functions, IF statements, and complex mathematical operations.
2. **Loops**. The goal of this section is to be able to solve problems using loops. Additionally we will learn several tools enabling us to solve hard programming problems and manage programs with large amounts of complexity.
3. **Arrays**. Next we will learn how to handle and manipulate large amounts of data in increasingly complex data structures. Specifically, we will learn about arrays, pointers, and file I/O.
4. **Advanced Topics**. The final section of the course will involve learning about a collection of specialized topics (dynamic memory allocation, multi-dimensional arrays, jagged arrays, and instrumentation to name a few) in the hopes they will help us understand the main topics of the semester better. In other words, the goal is not to learn these specialized topics so much as to make sure we have nailed the fundamentals.

How to Use This Textbook

This textbook is closely aligned with CS 124. All the topics, problems, and technology used in CS 124 are described in detail with these pages.

Sam and Sue

You may notice two characters present in various sections of this text. They embody two common archetypes representative of people you will probably encounter in industry.

Sue's Tips

Sue is a pragmatist. She cares only about getting the job done at a high quality level and with the minimum amount of effort. In other words, she cares little for the art of programming, focusing instead on the engineering of the craft.

Sue's Tips tend to focus on the following topics:

- Pitfalls: How to avoid common programming pitfalls that cause bugs
- Effort: How to do the same work with less time and effort
- Robustness: How to make code more resistant to bugs
- Efficiency: How to make code execute with fewer resources

Sam's Corner

Sam is a technology nerd. He likes solving hard problems for their own sake, not necessarily because they even need to be solved. Sam enjoys getting to the heart of a problem and finding the most elegant solution. It is easy for Sam to get hung up on a problem for hours even though he found a working solution long ago.

The following topics are commonly discussed in Sam's Corner:

- Details: The details of how various operations work, even though this knowledge is not necessary to get the job done
- Tidbits: Interesting tidbits explaining why things are the way they are

Neither Sue's Tips nor Sam's Corner are required knowledge for this course. However, you may discover your inner Sam or Sue and find yourself reading one of their columns.

Needing Help

Occasionally, each of us reaches a roadblock or feels like help is needed. This textbook offers two ways to bail yourself out of trouble.

If you find you are not able to understand the problems we do in class, work through them at home before class. This will give you time for reflection and help ask better questions in class.

If you find the programming problems to be too hard, take the time to type out all the examples by hand. Once you finish them, work through the challenge associated with each example. There is something about typing code by hand that helps it seep into our brains. I hope this helps.

Computers & Programs

Sam is talking with an old high school friend when he find out he is majoring in Computer Science. "You know," said his friend, "I have no idea how a computer works. Could you explain it to me?" Sam is stumped for a minute by this. None of the most common analogies really fit. A computer is not really like a calculator or the human brain. Finally, after much thought, Sam begins: "Computer programs are like recipes and the computer itself is like the cook following the instructions…"

Objectives

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

- Identify the major parts of a computer (CPU, main memory, etc.).
- Recite the major parts of a computer program (statements, headers, etc.).
- Type the code for a simple program.

Overview of Computers and Programs

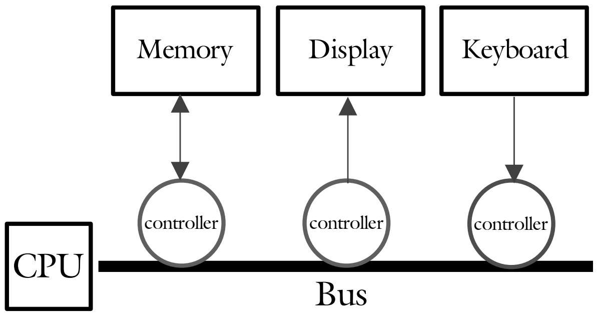
Consider a simple recipe for making cookies. The recipe consists of the ingredients (the materials used to make the cookies) as well as the instructions (such as "stir in the eggs"). If the recipe was precise enough then any two cooks would be able to produce an identical set of cookies from the provided ingredients. Most recipes, however, are ambiguous. This requires the cook to improvise or use his best judgment. This analogy holds well to the relationship between computers and programs with one exception: the computer is unable to improvise:

- **Cook**: The cook is the computer executing the instructions.
- **Recipe**: The recipe is the computer program to be executed.
- **Ingredients**: The ingredients are the input to the program.
- **Cookies**: The cookies correspond to the output of the program.
- **Chef**: The chef is the person who came up with the recipe. This is the role of the programmer.

Your job this semester is to derive the recipe (computer program) necessary to produce the desired cookies (output) from the provided ingredients (input).

Computers

Computers today are all based on the basic model developed by Dr. John von Neumann. They consist of a CPU (the part that executes instructions), the Memory (the part where programs and data are stored), the bus (the wire connecting all components), and the peripherals (input and output devices).



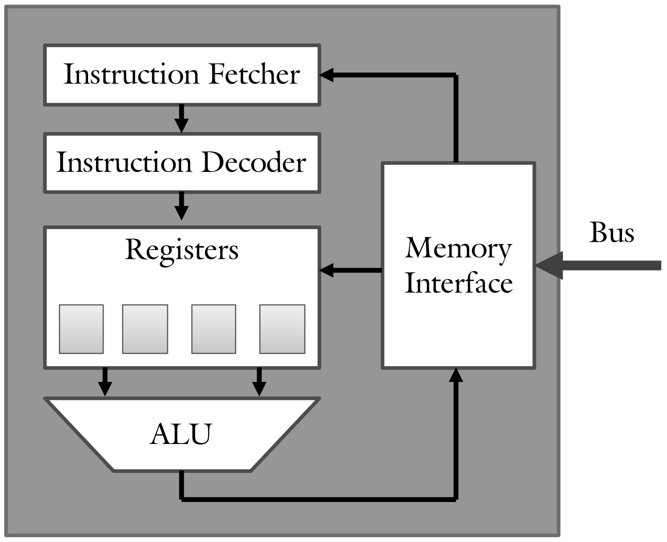
Please view the following movie to learn more about the parts of a computer:

Parts of a Computer

Possibly the most important part of a computer is the CPU. This has several components: the memory interface (operating much like a controller, it puts data on the bus and takes data off the bus), instruction fetcher (determines which instruction is next to be executed), the instruction decoder (interpreting what a given instruction does), the registers (keeping temporary data), and the ALU (Arithmetic Logic Unit, performing math and logical operations). Please view the following movie to learn more about the parts of a computer:

The CPU

Finally, programs consist of simple instructions that are executed by the CPU.



The following is a sample instruction set (real CPUs have hundreds of instructions but do essentially the same thing):

Name	Opcode	Description	Example
`NOOP`	0	Do nothing	`NOOP`
`LOAD`	1	Load a value from some memory location into the register	`LOAD M:3`
`SET`	2	Set the register to some value	`SET 1`
`SAVE`	3	Saves the value in a register to some memory location	`SAVE M:10`
`JUMP`	5	Sets the Next Instruction value to some memory location	`JUMP M:0`
`JUMPZ`	6	Same as JUMP except only sets the Next Instruction value if the register is set to zero	`JUMPZ M:0`
`ADD`	8	Adds a value to the register	`ADD 1`
`SUB`	9	Subtracts a value from what is currently in the register	`SUB 1`
`MULT`	10	Multiplies the current value in the register by some value	`MULT 2`
`DIV`	11	Integer division for some value in the register	`DIV 2`
`AND`	12	Returns 1 if the value and the register are both non-zero	`AND 1`
`OR`	13	Returns 1 if the value or the register is non-zero	`OR 1`
`NOT`	14	Returns 1 if the value in the register is zero	`NOT`

Please view the following movie to learn about how these instructions can be used to make a program:

Programs

The last item is an emulator (a program that pretends it is something else. In this case, it emulates the simple computer presented in the previous movies).

Emulator

To run a program:

1. Type or paste the program in the large edit control below the "Load Program" label. Make sure there is one instruction per line or it will not work correctly. You may also need to delete an extra space.
2. Press the Load button. This will load the program into memory and set the "Next Instruction" pointer to zero.
3. Step through the program, one instruction at a time, by pressing the Go button.

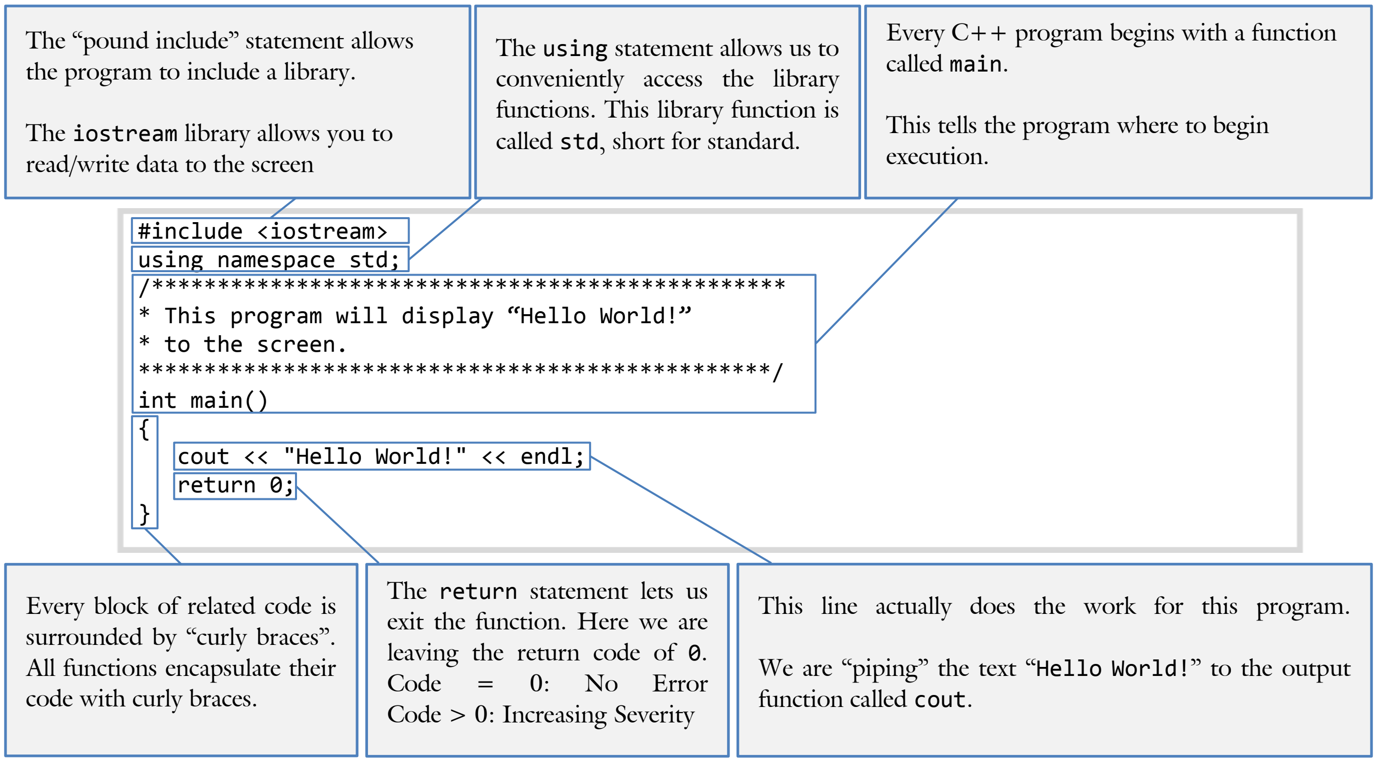
A couple programs you may want to run… what will they do?

```
SET 0xff
SAVE D:0
SET 0x00
SAVE D:0
JUMP M:0
```

```
SET 0xff
SAVE D:0
LOAD M:3
ADD 1
SAVE M:3
JUMP M:0
```

Programs

There are many computer languages, each enabling the programmer to write a computer program. In this class, we will be using the C++ language developed by Bjarne Stroustrup in the early 1980's. A C++ program consists of two parts: the header and the functions. The header describes what tools will be used in the program and the functions contain the recipes themselves. The functions consist of individual instructions called statements:



The "pound include" statement allows the program to include a library.

The `iostream` library allows you to read/write data to the screen

The `using` statement allows us to conveniently access the library functions. This library function is called `std`, short for standard.

Every C++ program begins with a function called `main`.

This tells the program where to begin execution.

```
#include <iostream>
using namespace std;
/*************************************************
* This program will display "Hello World!"
* to the screen.
************************************************/
int main()
{
   cout << "Hello World!" << endl;
   return 0;
}
```

Every block of related code is surrounded by "curly braces". All functions encapsulate their code with curly braces.

The `return` statement lets us exit the function. Here we are leaving the return code of `0`. Code = 0: No Error Code > 0: Increasing Severity

This line actually does the work for this program.

We are "piping" the text "`Hello World!`" to the output function called `cout`.

IOSTREAM: This simple program has two parts to the header. The first part indicates that the `iostream` library will be used. The `iostream` library contains many useful tools that we will be using in every program this semester, including the ability to display text on the screen. There are other libraries we will be introduced to through the course of the semester, including `iomanip`, `cassert`, and `fstream`.

NAMESPACE: The second line of code in the header indicates we are to be using the shorthand version of the tools rather than the longer and more verbose versions. This semester we will always have "using namespace std;" at the top of our programs, though in CS 165 we will learn when this may not be desirable.

CURLY BRACES: All the instructions or statements in a C++ program exist in functions. The `{}`s (called "curly braces") denote where the function begins and ends. Most programs have many functions, though initially our programs will only have one. Back to our recipe analogy, a program with many functions is similar to a recipe having sub-recipes to make special parts (such as the sauce). The functions can be named most anything, but there must be exactly one function called `main()`. When a program begins, execution starts at the head of `main()`.

COUT: The first instruction in the program indicates that the text "Hello world" will be displayed on the screen. We will learn more about how to display text on the screen, as well as how to do more complex and powerful tasks later in the semester (Chapter 1.1).

RETURN: The last statement in this example is the return statement. The return statement serves two purposes. The first is to indicate that the function is over. If, for example, a return statement is put at the beginning of a function, all the statements after return will not be executed. The second purpose is to send data from one function to another. Consider, for example, the absolute value function from mathematics. In this example, the output of the absolute value function would be sent to the client through the return mechanism. We will learn more about the use of return later in the semester (Chapter 1.4).

Comments

Strictly speaking, the sole purpose of a recipe is to give the cook the necessary information to create a dish. However, sometimes other chefs also need to be able to read and understand what the recipe is trying to do. In other words, it is often useful to answer "why" questions (example: "Why did we use whole wheat flour instead of white flour?") as well as "how" questions (example: "How do I adjust the recipe for high elevation?"). We use comment to address "why" questions.

Comments are notes placed in a program that are not read by the compiler. They are meant to make the program more human-readable and easier to adjust. There are two commenting styles in C++: line and block comments.

Line comments

Line comments indicate that all the text from the beginning of the line comment (two forward slashes `//`) to the end of the line is part of the comment:

```
{
   // line comments typically go inside functions just before a collection
   //     of related statements
   cout << "Display text\n";        // You can also put a line comment on the end
                                    // of a line of code like this.
}
```

Some general guidelines on comments:

- Comments should answer "Why" and "How" questions such as "Why is this variable set to zero?" or "How does this equation work?"
- Comments should not state the obvious.
- When there are multiple related statements in a function, set them apart with a blank line and a comment. This facilitates skimming the function to quickly gain a high-level understanding of the code.

Block comments

Block comments start with `/*` and continue until a `*/` is reached. They can span multiple lines, but often do not. We typically use block comments at the beginning of the program:

```
/************************************************************************
* 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.
************************************************************************/
```

All programs created for CS 124 need to start with this comment block. Observe how the comment blocks starts with the `/*` at the top and continues until the `*/` on the last line. We will start all of our programs with the same comment block. Fortunately there is a template provided so you won't have to type it yourself.

We also have a comment block at the beginning of every function. This comment block should state the name of the function and briefly describing what it is designed to do. If there is anything unusual or exceptional about the function, then it should be mentioned in the comment block. The following is an example of a function comment block:

```
/*************************************************************
 * MAIN
 * This program will display a simple message on the screen
 *************************************************************/
```

Problem 1

Which part of the CPU performs math operations?

Answer:

____

Please see page 5 for a hint.

Problem 2

If a program is like a recipe for cookies then which of the following is most like the data of the program?

- Ingredients
- Instructions
- Measurements
- Chef

Please see page 4 for a hint.

Problem 3

What is missing from this program?

```
#include <iostream>

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

   return 0;
}
```

Answer:

____

Please see page 7 for a hint.

Problem 4

Which of the following is a function?

```
int main()
```

```
#include <iostream>
```

```
return 0;
```

```
using namespace std;
```

Please see page 7 for a hint.

Problem 5

What is wrong with this program?

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

int main()
{
   cout << "Howdy\n";
}
```

Answer:

____

Please see page 7 for a hint.

Assignment 0.2

In I-Learn, please answer the following questions:

1. What does the following assembly program do?

```
SET 0xff
SAVE D:0
SET 0x00
SAVE D:0
JUMP M:0
```

2. What does the following assembly program do?

```
SET 0xff
SAVE D:0
LOAD M:3
ADD 1
SAVE M:3
JUMP M:0
```

3. Write the code to put "Hello world" on the screen:

4. What is the purpose of comments in a program?

5. Give an example of all the ways to write a comment in C++:

Unit 1. Simple Programs

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

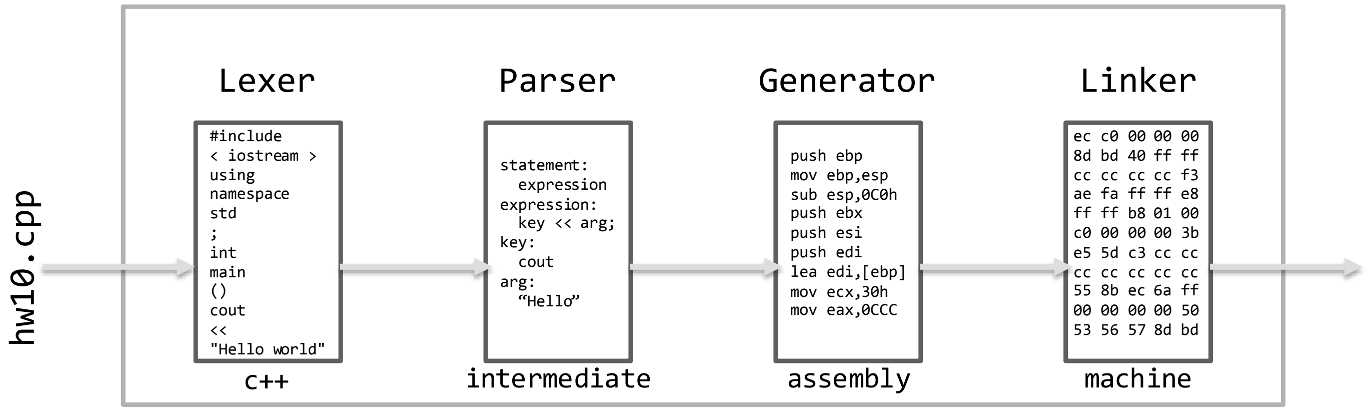
This will start emacs with a blank document named `hello.cpp`. From here you can type anything you like. However, if you wish this program to function correctly, you need to type valid C++. For your first program, you can make it say "Hello World" as we need to do for the first assignment:

```
/***********************************************************************
* 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 keystokes 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

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:

```
numStudents
```

Function names

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.

```
displayBudget()
```

Indent

Indentations are three spaces. No tabs please!

```
{
   cout << "Hello world\n";
}
```

Line length

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).

```
// Long comments can be broken into two lines
// to increase readability. Start each new
// line with "//"s
```

Program comments

All programs have a program comment block at the beginning of the file. This can be found in the standard template. An example is:

```
/*********************************************
* Program:
*    Assignment 10, Hello World
*    Brother Helfrich, CS124
* Author:
*    Sam Student
* Summary:
*    Display a message
*********************************************/
```

Function comments

Every function such as `main()` has a comment block describing what the function does:

```
/*********************************************
 * MAIN
 * This program will display a simple message
 * on the screen
 ********************************************/
```

Space between operators

All operators, such as addition (+) and the insertion operator (<<) are to have a single space on either side to set them apart:

```
sumOfSquares += userInput * userInput;
```

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.

Unit 1

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 "`eniac`", then the file appears as eniac.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 `eniac.0` instead of `eniac.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.

Solution

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
```

Unit 1

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).

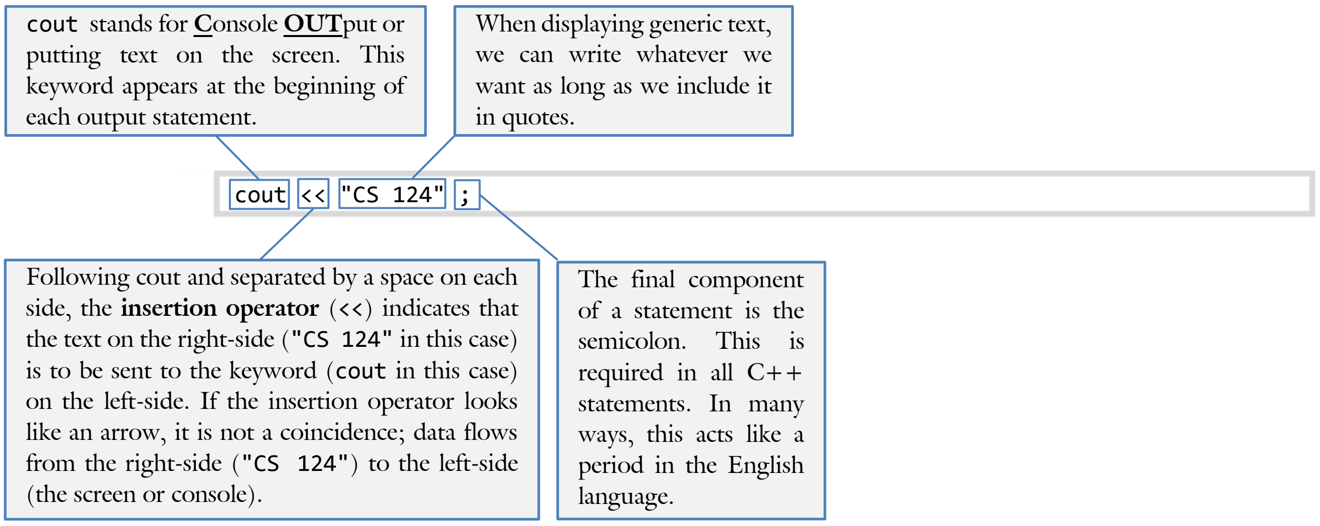
Overview of Output

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:



`cout` stands for <u>C</u>onsole <u>**OUT**</u>put or putting text on the screen. This keyword appears at the beginning of each output statement.

When displaying generic text, we can write whatever we want as long as we include it in quotes.

```
cout << "CS 124" ;
```

Following cout and separated by a space on each side, the **insertion operator** (`<<`) indicates that the text on the right-side (`"CS 124"` in this case) is to be sent to the keyword (`cout` in this case) on the left-side. If the insertion operator looks like an arrow, it is not a coincidence; data flows from the right-side (`"CS 124"`) to the left-side (the screen or console).

The final component of a statement is the semicolon. This is required in all C++ statements. In many ways, this acts like a period in the English language.

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 <u>not</u> 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?

	`endl`	`\n`

Inside quotes	`cout << "Hello";`
	`cout << endl;`
		`cout << "Hello\n";`

Not inside quotes	`cout << 5;`
	`cout << endl;`
		`cout << 5;`
		`cout << "\n";`

Use	When the previous item in the output stream is not in quotes, use the `endl`.	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:

Reason	Example	Explanation
Line Limit	`cout << "I want to make one "` `<< "very long line much "` `<< "more manageable.\n";`	Style checker limits the length of a line to 80 characters. It is often necessary to use multiple insertion operators to keep within this limit.
Mixing	`cout << "Mix text with "` `<< 42` `<< " numbers.\n";`	Variables need to be outside quotes, requiring separate insertion operators for each one.
Comments	`cout << "CS124"    // class` `<< "-1-"      // section` `<< "Bob";     // prof.`	Comments are more meaningful when they are on the same line as to what they are clarifying.

Alignment

It is often desirable to make output characters align in columns or tables. This is particularly useful when working with columns of numbers. In these cases, we have two tools at our disposal: tabs and set-width.

Tabs

When the typewriter was invented, it quickly becomes apparent that typists needed a convenient way to align numbers into columns. To this end, the tab key (also known as the tabular key) was invented. The tab key would skip the carriage (or cursor in the computer world) to the next tab stop. In the case of mechanical typewriters, tab stops were set every half inch. This meant that hitting the tab key would move the cursor to the next half inch increment. Sometimes this meant moving forward one space, other times the full half inch. The tab command (`\t`) in `cout` behaves exactly the same way as a typewriter. Each time the `\t` is encountered in textual data, the cursor moves forward to the next 8 character increment. Consider the following text:

```
cout << "\tOne\n";
cout << "Deux\tDeux\n";
cout << "\t\tTres\n";
```

The output from these statements is:

```
        One
Deux    Deux
                Tres
```

Observe how the word "`One`" is indented eight spaces. This is because the cursor started in the 0 column and, when the tab key was encountered, skipped to the right to the next tab stop (the 8 column).

The first word "`Deux`" is left aligned because, after the `\n` is encountered in the previous line, the cursor moves down one line and to the 0 column. After the first "`Deux`", the cursor is on the 3 column. From here the cursor skips to the next multiple of 8 (in this case the 8 column) when the tab is encountered. This makes the "`One`" and "`Deux`" left-aligned.

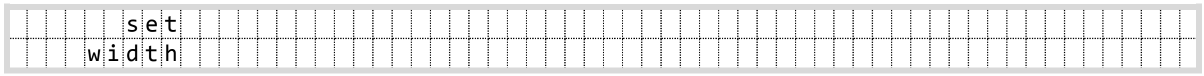
When the third statement is executed, the first tab moves the cursor to the 8 column. The second tab moves the cursor to the next multiple of 8 (the 16 column). From here, the text "`Tres`" is rendered.

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 backlash 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

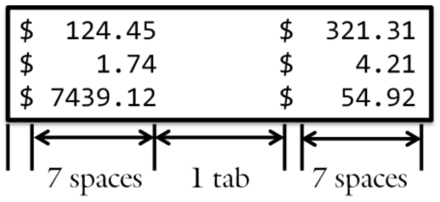
Example 1.1 – Money Alignment

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
```

Solution

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
```

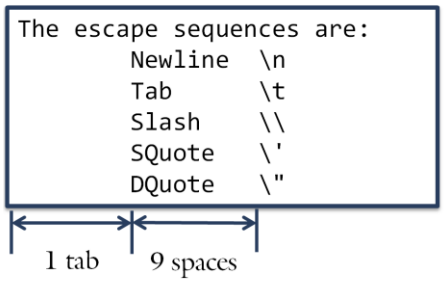
Example 1.1 – Escape Sequences

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.



```
The escape sequences are:
        Newline  \n
        Tab      \t
        Slash    \\
        SQuote   \'
        DQuote   \"
```

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, tool, 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	±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:

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

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

Sam's Corner

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)?

Sam's Corner

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 **C**onsole **IN**put, analogous to the **C**onsole **OUT**put 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
}
```

Sam's Corner

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 includng 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

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.

```
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.
```

Solution

The four variables are declared as follows:

```
char name[256];
int age;
float gpa;
char letterGrade;
```

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:

```

// Prompt the user for his age
cout << "What is your age: ";
cin  >> age;
```

Finally, we must not forget to format `cout` to display one digit after the decimal.

```
// 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";
```

Challenge

As a challenge, try to accept an individual's full name (Such as "Sam S. Student") rather than just the first name.

Also, try to configure the output to display two digits of accuracy rather than one.

See Also

The complete solution is available at [1-2-manyPrompts.cpp]() or:

```
/home/cs124/examples/1-2-manyPrompts.cpp
```

Unit 1

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 ½. 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
}
```

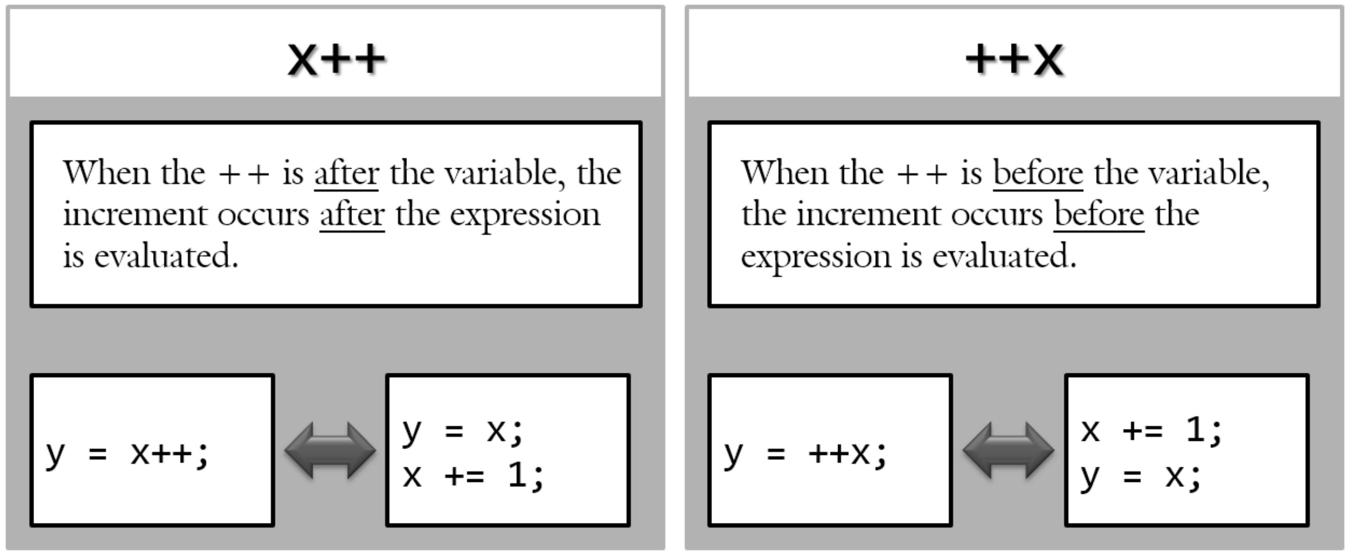
In this example, the `age++` statement serves to add one to the current value of `age`. Of course, `age--` works in the opposite way. There are two flavors of the increment (and decrement of course) operators: increment before the expression is evaluated and increment after. To illustrate, consider the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



x++

When the ++ is after the variable, the increment occurs after the expression is evaluated.

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

```
y = x;
x += 1;
```

++x

When the ++ is before the variable, the increment occurs before the expression is evaluated.

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

```
x += 1;
y = x;
```

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.

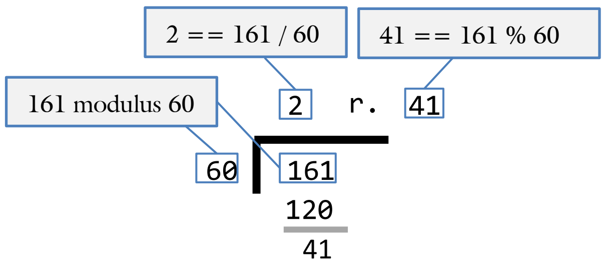
Modulus %

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
`+=`	Add and assign	Add onto
`-=`	Subtract and assign	Subtract from
`*=`	Multiply and assign	Multiply by
`/=`	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 `int`s or two `float`s, 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 `int`s to `float`s and `bool`s to `int`s 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 covert in the most obvious way. Consider converting `int`s and `bool`s:

```
{
   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 `float`s and `int`s!

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.

Assignment 1.3

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.

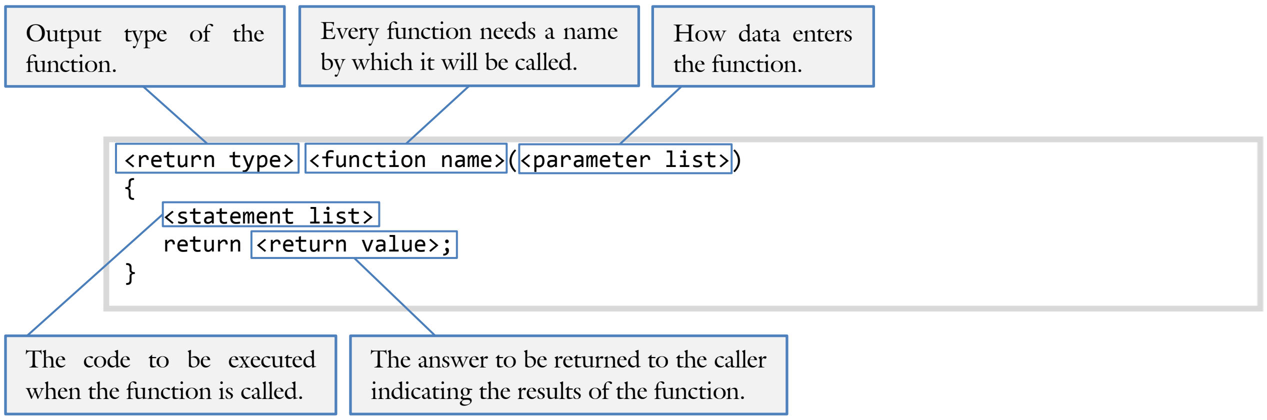
The syntax for both procedures and mathematical functions is the same in C++. The purpose of this chapter is to learn the syntax of functions so they can be used in our programs. All assignments, projects, and tests in this class will involve multiple functions from this time forward.

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!


Output type of the function.
Every function needs a name by which it will be called.
How data enters the function.
The code to be executed when the function is called.
The answer to be returned to the caller indicating the results of the function.

```
<return type> <function name>(<parameter list>)
{
   <statement list>
   return <return value>;
}
```

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
```

Unit 1

Solution

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 `cin`, and return the value through the function return mechanism.

Problem

Write a program to prompt the user for his age. The user's age will then be displayed. User input is **bold and underlined**.

```
What is your age? 19
Your age is 19 years old.
```

Solution

The prompt function follows the "return a value" pattern from the previous example:

```
/************************************************************
 * 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()
}
```

Next we will create `main()` to test our function.

```
/*******************************************************************
 * 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"
}
```

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 `main()` to test this function?

See Also

The complete solution is available at 1-4-promptFunction.cpp or:

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

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 matchup 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:

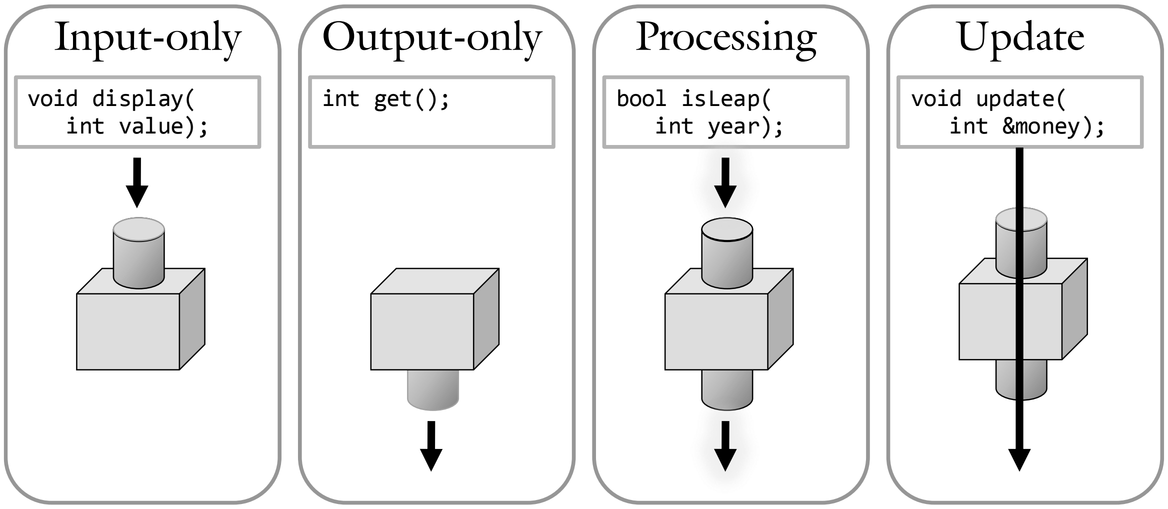
```
displayCoordinates(4);           // two parameters expected. Where is the second?
```

- 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);
```

Unit 1

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
```

Solution

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;
}
```

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.

`/****************************************`
` * Pass-by-value`
` *     No change to the caller`
` ****************************************/`
`void notChange(int number)`
`{`
`   number++;`
`}`
	`/******************************************`
	` * Pass-by-reference`
	` *     Will change the caller`
	` *****************************************/`
	`void change(int &number)`
	`{`
	`   number++;`
	`}`

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.

Solution

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
```

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.

This is always a data type.

Must be the same as the definition.

Only the data-type are required, but most put the variable names here also.

```
float add(float value1, float value2);
```

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
   {                              // the purpose of the {}s here are to limit scope
      cout << "Press any key to continue";
      char something;             // only "in scope" for two lines of code
      cin.get(something);
   }
}
```

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
$\sim$	Not	`!`	`!true`
$\wedge$	And	`&&`	`true && false`
$\vee$	Or	`||`	`true || false`
$=$	Equals	`==`	`x + 5 == 42 / 2`
$\neq$	Not Equals	`!=`	`graduated != true`
$<$	Less than	`<`	`age < 16`
$\leq$	Less than or equal to	`<=`	`timeNow <= timeLimit`
$>$	Greater than	`>`	`age > 65`
$\geq$	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` → `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) → (true && false) → false
!65 → !true → 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 `bool`.

Problem

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 quality, then the credit is $1,000 per child. If you don't, no tax credit is awarded.

```
What is your income: 115000.00
How many children? 2
Child Tax Credit: $ 0.00
```

Solution

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 `float` and the output is the decision as a `bool`.

```
/***************************************
 * 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);
}
```

Observe how the name of the function implies what true means. In other words, if `qualify()` returns `true`, then the user qualifies. If `qualify()` returns `false`, then the user doesn't. Always make sure the name of the function implies what `true` means when working with a `bool` function.

The next part is computing the credit to be awarded. This will require an IF statement which will be discussed next chapter.

```
   if (qualify(income))
      cout << 1000.00 * (float)numChildren << endl;
   else
      cout << 0.00 << endl;
```

Notice how the return value of the `qualify()` function goes directly into the IF statement.

Challenge

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.

As a challenge, modify the above example to more accurately reflect the law.

See Also

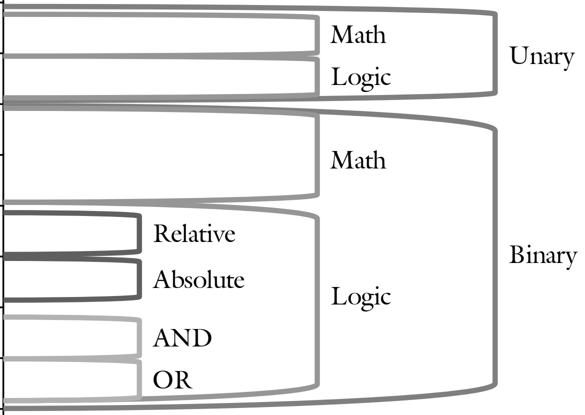
The complete solution is available at 1-5-decisionFunction.cpp or:

```
/home/cs124/examples/1-5-decisionFunction.cpp
```

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
`!`	Not
`* / %`	Multiply, divide, modulo
`+ -`	Addition, subtraction
`> >= < <=`	Greater than, less than, etc.
`== !=`	Equality
`&&`	And
`||`	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');
   bool b = ('b' > 'a');
   bool c = (a * 4) && b;
   bool d = !(b || (c || true));
   bool e = a && b && c && d;
   bool f = a || b || c || d;
   bool g = (a != b) && true;
}
```

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`	
`! a < b`	
`a + b && c || d`	
`2 * c++ > 2 + 7 == 9 % 2`	
`a > b > c > d`	

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
```

Solution

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 indention 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 indention 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

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 (`hourlyWage * hoursWorked`) or the more complex overtime formula.

```
What is your hourly wage? 10
How many hours did you work? 41
Pay: $ 415.00
```

Solution

The function to compute pay taking the hourly wage and hours worked as input.

```
/*****************************************
 * 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;
}
```

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

The complete solution is available at 1-6-overtime.cpp or:

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

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

Continued from previous 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

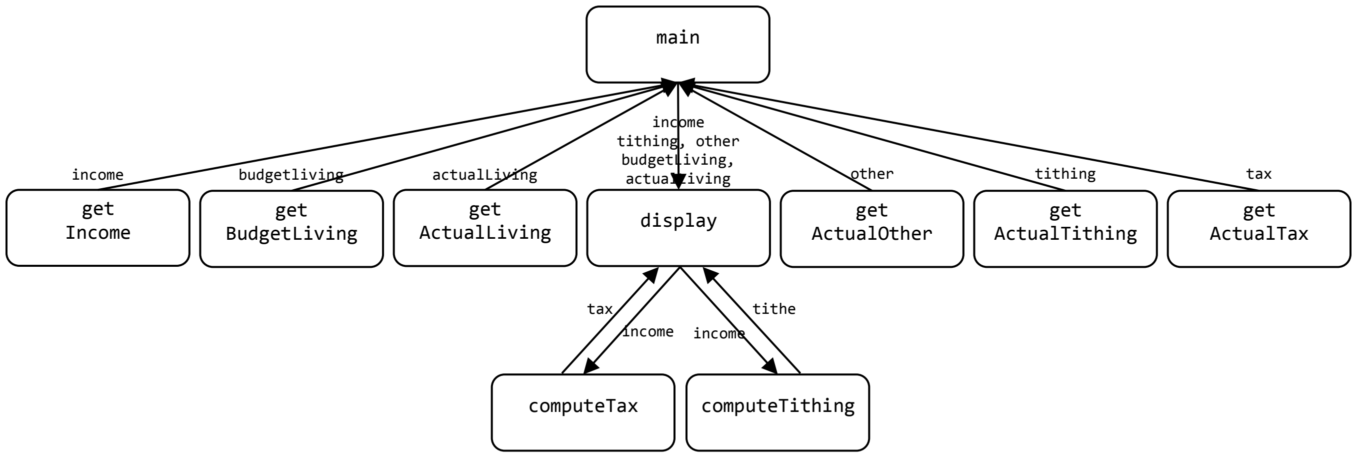
Interface Design

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
```

Unit 2. Design & Loops

2.0 Modularization

Sue is frustrated! She is working on a large project with a couple classmates where there must be a thousand lines of code and three dozen functions. Some functions are huge consisting of a hundred lines of code. Some functions are tiny and don't seem to *do* anything. How can she ever make sense of this mess? If only there was a way to map all the functions in a program and judge how large a function should be.

Objectives

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

- Measure the Cohesion level of a function.
- Measure the degree of Coupling between functions.
- Create a map of a program using a Structure Chart.
- Design programs that exhibit high degrees of modularization.

Prerequisites

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

- Create a function in C++ (Chapter 1.4).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).

Overview

Up to this point, our programs have been relatively manageable in size and complexity. In other words, one could conceivably keep the entire design in your head. Most interesting software problems, however, are far too large and far too complex for this. Very soon, this become so difficult that it is impossible for any one person or even group of people to understand everything. What is to be done?

One of the main techniques we have at our disposal to tame these size and complexity challenges is **modularization**. Modularization is a collection of tools metrics, and techniques that together enable us to reduce large problems into smaller ones.

The first tool we have at our disposal is the Structure Chart. This is a graphical representation of the functions in a program, including how they "talk" to each other. You may have noticed an example of a structure chart in the Unit 1 project.

The second modularization tool is a metric by which we measure the "strength" of a function. This will tell us the degree in which a given function is dedicated to a single task. We call this metric Cohesion.

The third and final modularization tool is a metric by which we measure the complexity of the information interchange between two functions. We call this metric Coupling.

These three tools (Structure Chart, Cohesion, and Coupling) together help a programmer to more effectively modularize a program so it is easier to write the code, easier to fix bugs, and easier to understand.

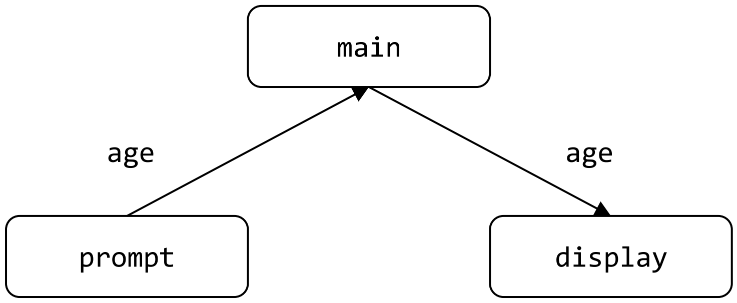
Structure Chart

A Structure Chart is a tool enabling us to design with functions without getting bogged down in the details of what goes on inside the functions. This is basically a graph containing all the functions in the program with lines indicating how the functions get called.

For example, consider a program designed to prompt the user for his age and display a witty message:

```
What is your age: 29
You are 29 again?  I was 29 for over a decade!
```

The Structure Chart would be:



There are three components to a Structure Chart: the function, the parameters, and how the functions call each other (program structure).

Functions

Each function in the Structure Chart is represented with a round rectangle. You specify the function by name (remember to camelCase the name as we do with all variable and function names). Since functions are typically verbs, there is typically a verb in the name. In the above example, there are three functions: `main`, `prompt`, and `display`.

Parameters

The second part of a Structure Chart is how information flows between functions. This occurs through parameters as well as through the return mechanism. If a function takes two parameters, then one would expect an arrow to flow into the function with two variables listed. In the above example, the function `prompt` accepts no data from `main` though it sends out a single piece of data: the `age`. Thus the following prototype for `prompt`:

```
int prompt();
```

The next function is `display`. It sends no data back to `main` but accepts a single piece of data, the `age`. Thus one would expect the following prototype for `display`:

```
void display(int age);
```

Program structure

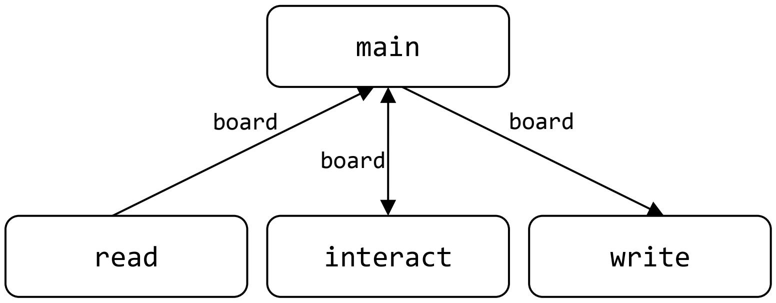
The final part of a Structure Chart is how the functions in a program call each other. Typically, we put `main` on top and, below `main`, all the functions that `main` calls. Note that you can have a single function that is called by more than one function. In this case, arrows will be reaching this function from multiple sources. Please see Project 1 for an example of a Structure Chart.

Designing with Structure Charts

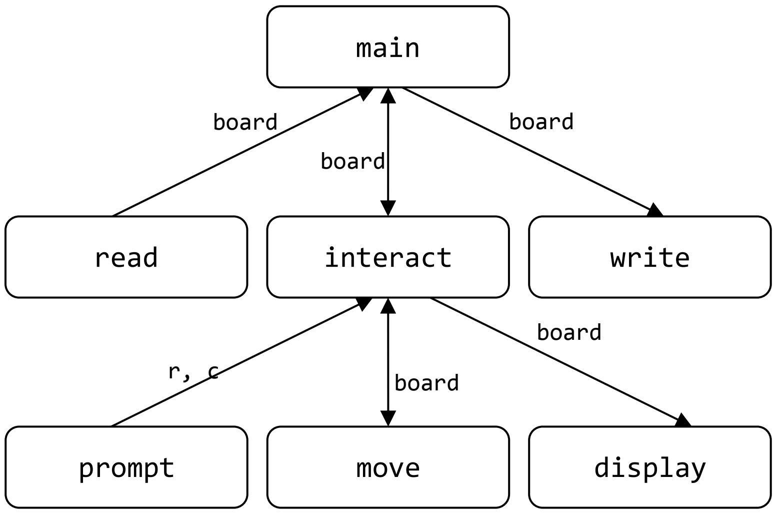
The Structure Chart is often the first step in the design process. Here we answer the big questions of the program:

- What will the program do?
- What are the big parts of the program?
- How will the various parts communicate with each other?

To accomplish this, we start with the top-down approach. We start with very general questions and slowly work to the details. Consider, for example, a program designed to play Tic-Tac-Toe. We would start with a very general design: the program will read a board from a file, allow the user to interact with the board, and then write the board back to the file.

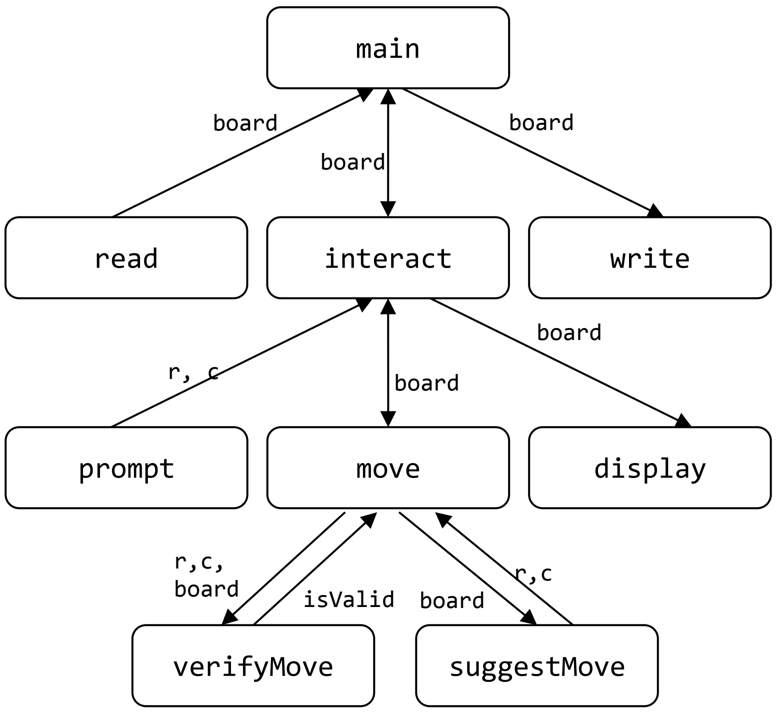


Note how each of the functions is Cohesive (does one thing and one thing only) and has simple Coupling (only one parameter is passed between the functions). That being said, the `interact()` function is probably doing too much work. We will delegate some of that work to three other functions:



Again, each of the functions (`prompt()`, `move()`, and `display()`) are Cohesive and have loose Coupling. However, it appears that the `move()` function is still too complex. While it is still Cohesive, we still might want to delegate some of the work to other functions.

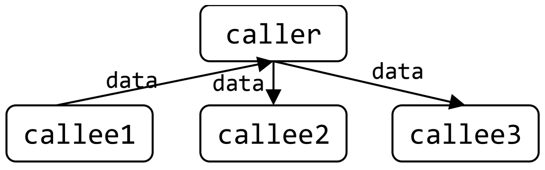
The final Structure Chart for our Tic-Tac-Toe program is the following:



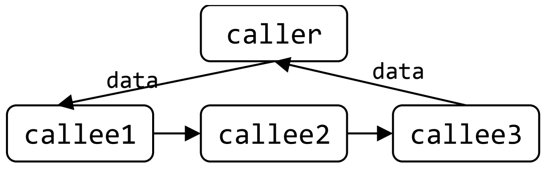
There are a few things to observe about this and all Structure Charts:

7. We always put `main()` on top. This means that control goes from the top down, rather than following the flow of the arrows. A common mistake new programmers make is to put `main()` in the center of the Structure Chart with arrows extending in all directions. We call this "spider" Structure Charts.
8. There are seldom more than three functions called from a single function. If too many arrows emanate from a given function, then that function is probably doing too much. There are exceptions from this rule-of-thumb of course. One example is when all the child functions do the same type of thing. The Structure Chart from Project 1 is an example. When in doubt, ask yourself if each function is Functionally cohesive.
9. A Structure Chart is a tree; there are no circuits. If a function (`interact()` in the above example) calls another function (`prompt()`), control returns to the caller when the callee is finished.

```
void caller()                  // caller will call three functions
{
   int data = callee1();       // first callee1 is called with the return value in data
   callee2(data);              // next data is sent to callee2.
   callee3(data);              // finally data is sent to callee3
}
```



Structure Chart of one function calling three in sequence



ERROR: there are no circuits in a Structure Chart! When `callee1()` is finished, data flows back to `caller()`. It cannot flow to `callee2()`

Example 2.0 – Count Factors

Demo

This example will demonstrate how to break a large and complex program into a set of smaller and more manageable functions. When considering how this will be done, the principles of Cohesion and Coupling will be considered. The resulting solution will be presented both as a structure chart and as a set of function prototypes.

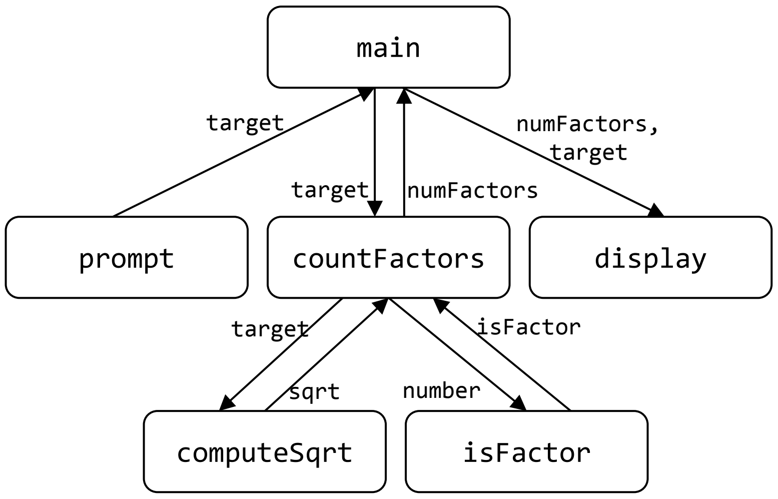
Problem

Write a program to find how many factors a given number has. To accomplish this, it is necessary to enumerate all the values between 1 and the square root of the target number. Each of these values will then need to be checked to see if it evenly divides into the target number.

```
What number would you like to find the factors for? 16
There are 4 factors in 16.
```

Solution

The first part of the solution is to identify the functions and how they will call each other. The following structure chart represents one possible solution.



The second part is to convert this map into a set of function prototypes.

```
int main();                        // calls prompt, countFactors, & display
int prompt();                      // called by main
int countFactors(int target);      // called by main, calls computeSqrt, isFactor
void display(int target, int num); // called by main
int computeSqrt(int target);       // called by countFactors
bool isFactor(int x, int y);       // called by isFactor
```

Observe how the structure chart tells you the function names, what parameters go into the functions, and who calls a given function. This allows us to create an outline for the program.

Challenge

As a challenge, try to fill in the functions for this program. The `computeSqrt()` function may look something like this:

```
#include <cmath>                          // for the SQRT function

int computeSqrt(int target)
{
   return (int)sqrt((double)target);
}
```

Unit 2

Cohesion

Recall that Cohesion is a metric by which we measure the "strength" of a function. A more formal definition of Cohesion is:

> ***Cohesion is a measurement of how well a function performs one task.***

This definition has two components:

- "a measurement:" Cohesion is a metric, reporting on the quality of one aspect of the design.
- "performs one task:" Cohesion is a property of a single function.

A well-designed function will be completely focused on a single task. There are four different levels of Cohesion (presented from best to worst): Strong, Extraneous, Partial, and Weak.

Strong Cohesion

The strongest and most desirable level of Cohesion is where all the code in a function is directed to one purpose. The formal definition of Strong Cohesion is:

> ***All aspects of a function are directed to perform a single task,***
> ***and the task is completely represented.***

There are two parts to this definition. The first is that the unit of software does nothing extra. Any extra code thrown into a function will forfeit its classification as Strong Cohesion. The second part of the definition is that the task or concept is completely represented. Anything that leaves part of the task undone or relies on the client to complete the work cannot be considered Strong. Of course, every function should strive for Strong cohesion. Observe that it does not matter how simple or complex the task is; it is Strongly Cohesive as long as only that task is being performed.

Consider the following function:

```
/*****************************************************************************
 * COMPUTE PAY
 * Determine an employee's pay based on hourly wage and number of hours worked
 *****************************************************************************/
float computePay(float hours, float wage)
{
   // regular pay
   if (hours < 40.0)
      return hours * wage;

   // overtime
   else
      return (wage * 40.0) +
             (wage * 1.5 * (hours - 40.0));
}
```

Observe how `computePay()` does one thing and one thing only: it computes pay given an employee's hourly wage and number of hours worked. This task is completely accomplished; no other work is needed in order to finish this task.

Extraneous Cohesion

The first weak form of Cohesion is Extraneous. Here exists something unnecessary in the unit of software. A political analogy would be a "rider" on a bill and a sports analogy would be a "bench warmer" on a team. The formal definition of Extraneous Cohesion is:

> ***At least one part of a function is <u>not</u> directed towards a single task.***
> ***However, the principle task is completely represented.***

Streamlined software should have no extraneous functionality. An example of an Extraneous compute-tax function would be one that correctly performs the calculation and then asks the user what to do with the refund. Any time the word "and" is used to completely describe a unit of software, it is probably Extraneous Cohesion.

Consider the following function:

```
/*************************************************************************
 * COMPUTE PAY
 * Determine an employee's pay based on hourly wage and number of hours worked.
 * This function also displays a warning message if too many hours are worked.
 *************************************************************************/
float computePay(float hours, float wage)
{
   // display error message if more than the maximum amount of work was done
   if (hours > 60.0)
      cout << "WARNING: Special permission is required to work more than 60 hours.\n";

   // regular pay
   if (hours < 40.0)
      return hours * wage;

   // overtime
   else
      return (wage * 40.0) +
             (wage * 1.5 * (hours - 40.0));
}
```

This function completely accomplishes the task of computing the pay for a worker. Unfortunately, it also does something that is not directly related to the task at hand. In this case, it warns if too much work is reported. As a general rule, a function designed to "compute" should not also "display."

Any time a function has code that is not directly related to the task at hand, there is a good chance that the function is Extraneous Cohesion (or worse!). Fortunately, the fix is very easy: move the extra code to a more appropriate location.

Partial Cohesion

Another weak form of cohesion is Partial, where a task is left incomplete. In other words, additional data or work needs to be stored or completed elsewhere for the concept or task to be completed. Note that Partial is not below Extraneous in the hierarchy but rather a peer. One does not improve a Partial class by making it Extraneous. Instead, one finished the job it was designed to do. The formal definition of Partial Cohesion is:

> ***All aspects of a function are directed to perform a single task,***
> ***but the task is not completely represented by the function.***

Partial Cohesion is particularly difficult for the client of a system because the onus is on the client to figure out how to finish the job. It is also difficult for the author of the software because, in order to thoroughly test the system, all possible implementations that complete the task needs to be discovered. Any time a description of a system necessitates a detailed description of the context in which it is used, that system is a candidate for Partial cohesion.

Consider the following functions:

```
/***********************************************************************
 * COMPUTE OVERTIME PAY
 * Determine an employee’s pay based on hourly wage and number of hours worked.
 * WARNING: Call computeNormalPay() if hours is less than 40
 ***********************************************************************/
float computeOvertimePay(float hours, float wage)
{
   return (wage * 40.0) +
          (wage * 1.5 * (hours - 40.0));
}

/***********************************************************************
 * COMPUTE NORMAL PAY
 * Determine an employee’s pay based on hourly wage and number of hours worked.
 * WARNING: Call computeOvertimePay() if hours is less than 40
 ***********************************************************************/
float computeNormalPay(float hours, float wage)
{
   return hours * wage;
}
```

Here each of the two functions accomplishes part of the task at hand. Anyone using one of these functions will also have to use the second to get the job done.

There are two ways to fix this problem and make the function(s) Strongly Cohesive: either combine the functionality into a single function or create a wrapper function that calls both of the components:

```
/***********************************************************************
 * COMPUTE PAY
 * Determine an employee’s pay based on hourly wage and number of hours worked
 ***********************************************************************/
float computePay(float hours, float wage)
{
   if (hours < 40.0)
      return computeNormalPay(hours, wage);
   else
      return computeOvertimePay(hours, wage);
}
```

This function is now Strongly Cohesive. It may be desirable to pursue a design like this when there is another part of the program that needs `computeNormalPay()` or `computeOvertimePay()` without the other part.

Weak Cohesion

The worst form of cohesion is Weak. One should never design for Weak Cohesion; it is a state that is to be generally avoided. The formal definition of Weak Cohesion is:

> ***At least one part of a function is not directed towards performing a single task. Additionally, the task is not completely represented by the function.***

In other words, weak cohesion is a combination of Extraneous and Partial. In theory, one should never come across Weak cohesion. Alas, if only this were true.

Consider the following function:

```
/*************************************************************************
 * COMPUTE PAY
 * Determine an employee’s pay based on hourly wage and number of hours worked.
 * This function also configures the display for money output.
 * WARNING: Call computeNormalPay() if hours is less than 40
 *************************************************************************/
float computePay(float hours, float wage)
{
   // set up the display for money
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);

   // regular pay
   if (hours < 40.0)
      cout << "ERROR: This only works for hours greater than 40\n";

   // compute overtime pay
   return (wage * 40.0) +
          (wage * 1.5 * (hours - 40.0));
}
```

This function exhibits Extraneous Cohesion. We can tell first because the function comment block contains the word “also.” A subsequent inspection of the code will reveal the code to configure output for money. Since this function does not display anything, the code clearly does not belong here.

This function exhibits Partial Cohesion because it only produces correct output if the employee’s wage is not less than 40 hours. In this case, the program will display an error message on the screen and still produce erroneous output.

Since this function is both Extraneous and Partial, it can be classified as Weakly Cohesive. It appears that the programmer threw code together hoping it would work, rather than properly designed the function. On the surface, it might seem that the best approach from this point is to add the missing functionality and remove the extraneous parts. In practice, a better approach is to start the design process from scratch with Cohesion in mind.

Coupling

Recall that Coupling is a metric of the complexity of the information interchange between two functions. A more formal definition of Coupling is:

Coupling is a measurement of the complexity of the interface between functions.

This definition has two components:

- "a measurement:" Coupling is a metric, reporting on the quality of one aspect of the design.
- "the complexity of the interfaces:" This can be summed up with the question: "how much does the programmer need to know to successfully use and how much does the resulting software need to do?"
- "between:" Coupling fundamentally is a measure of how different parts of a system communicate. It is not a property of an individual function, but rather how it interacts with the rest of the system.

A well-designed interface between functions will be easy to understand and use. There are seven different levels of Coupling (presented from best to worst): Trivial, Encapsulated, Simple, Complex, Document, Interactive, and Superfluous.

Trivial Coupling

Trivial is the weakest or best form of Coupling. Here the client of a unit of software needs to provide no information and receives no information from another unit of software. The formal definition of Trivial Coupling is:

There is no information interchange between functions.

In other words, one unit may instantiate, call, or activate another, but no information is passed. Similarly, no information can be gleaned from the timing of the function call. An example would be a function with no return value and no parameters.

Consider the following function:

```
/*****************************************************************************
 * DISPLAY INSTRUCTIONS
 * Inform the user about the functionality of this program
 *****************************************************************************/
void displayInstructions()
{
   // Inform the user about what this program will do
   cout << "This program will display the status of your monthly budget.\n"
        << "The produced report will include both your projected expenditures, "
        << "as well as what you actually spent last month.\n\n";

   // Inform the user about the type of data that will be requested
   cout << "To accomplish this, it is necessary to prompt you for several "
        << "confidential financial details.\n"
        << "Do not worry, no confidential data will be saved in the process.\n";
}
```

Note that a function may have no input parameters and have no return type yet still not be Trivially Coupled. Sometimes a hidden global variable is at play, making the function something other than Trivially Coupled.

```
void displayPay()
{
   cout << '$' << pay << endl;   // NOT trivially coupled!
}
```

Encapsulated Coupling

Encapsulated is a very weak form of Coupling defined by all of the parameters being in a trusted and accessible state. In other words, there are logical checks in place to ensure that no invalid data is sent between modules. This level makes no reference to the degree of complexity of the data nor the number of data items passed between modules.

The formal definition of Encapsulated Coupling is:

> ***All the information exchanged between functions***
> ***is in a convenient form and is guaranteed to be in a valid state.***

Note that Encapsulation is an Object-Oriented programming topic and is therefore not a topic for CS 124. That being said, there is a single example of encapsulated data that we have learned about thus far: Boolean data. Since a Boolean variable can have only two states and both are, by definition, valid, it is impossible to pass an invalid Boolean parameter. Enumerations are another validated data-type present in most languages. Here the compiler ensures that the data is always in a valid state. These will be discussed in more detail in CS 165. With modern languages and modern programs, the most common way to achieve the validated status is to use a class whose methods contain checks to guarantee data validity.

Consider the following function:

```
/************************************************************************
 * GET IS MALE
 * Prompt the user for his/her gender and return if he/she is male
 ************************************************************************/
bool getIsMale()
{
   char input;
   cout << "Please select your gender, ‘m’ for male and ‘f’ for female: ";
   cin  >> input;

   return (input == 'm') || (input == 'M');
}
```

This function takes no input parameters and returns a single Boolean value. The data is in a convenient form and is guaranteed to be valid. Note that Coupling makes no reference to information interchanges between the user and the program; it only concerns itself with information interchange between parts of the program.

Consider this function:

```
/************************************************************************
 * DISPLAY CHILD TAX CREDIT STATUS
 * Display to the user the status of their Child Tax Credit
 ************************************************************************/
void displayChildTaxCreditStatus(bool isEligible)
{
   if (isEligible)
      cout << "You are eligible for the child tax credit.";
   else
      cout << "You cannot claim the child tax credit this year.";
}
```

This is also Encapsulated Coupling because a single Boolean value is passed into the function. From this we can see that though a function may have Trivial input parameters, it is an Encapsulated Coupling if it has a single Encapsulated return value (and vice versa).

Simple Coupling

Another weak form of coupling is Simple, meaning that data can be selected, interpreted, and validated easily. Parameters consisting of simple built-in data-types such as integers or characters are often Simple, assuming that the use of the parameter is easily specified. The formal definition of Simple Coupling is:

All the information exchanged between functions is easy to select, interpret, and validate.

Perhaps this can be best captured with a few questions:

- Can I explain the purpose of this parameter with a few words?
- Is it intuitively obvious what this parameter represents and how it is used?
- Is it easy to validate the parameter with a simple IF statement?

Consider the following function:

```
/******************************************************************************
 * PROMPT FOR INCOME
 * Prompt the user for his/her monthly/yearly income.
 ******************************************************************************/
double promptForIncome(bool isMonthly)
{
   // different prompt according to the value of isMonthly
   if (isMonthly)
      cout << "Please specify your monthly income: ";
   else
      cout << "Enter your yearly income: ";

   // get the income value
   double income;
   cin >> income;
   return income;
}
```

In this case, the input parameter is a Boolean value. It may seem like this is an example of Encapsulated Coupling. Note, however, that the return value is a `double`. Clearly there are some values which are inappropriate (negative values or numbers which are smaller than a cent), meaning some trivial validation may be required. However, the return value is easy to select, interpret, and validate. This function is thus Simple Coupling.

Consider the following function:

```
/******************************************************************************
 * ELIGIBLE FOR CHILD TAX CREDIT
 * Return true if eligible for a child tax credit for a given child
 ******************************************************************************/
bool eligibleForChildTaxCredit(double income, int ageChild)
{
   // only eligible if the child is under 17 and income between $2,500 and $200,000
   return (ageChild < 17) && (income >= 2500.00) && (income <= 200000.00);
}
```

Note that the return type makes the function a candidate for Encapsulation Coupling. However, the income parameter as well as the ageChild parameter meet all the criteria for Simple Coupling. Since the lowest Coupling classification of any parameter (input or output) determines the overall Coupling of the function, this can be classified as Simple Coupling.

Complex Coupling

A tight or bad degree of coupling is Complex. Here the function callers need to know a great deal about the parameters in order to make a connection. The greater knowledge the callers need to have and the more work the callers need to accomplish to make sure the connection is done correctly, the more complex the level of coupling. While it might be worthwhile to enumerate sub-levels of Complex Coupling, it is sufficient to say that all are bad and should be avoided. The formal definition of Complex Coupling is:

At least one piece of information is non-trivial to create, validate, or interpret.

There are many things about a parameter which could cause it to be classified as Complex. A few examples:

- Input that is interpreted as a command, requiring the callee to act in response to the command. Understanding all possible commands is non-trivial.
- Two variables that must be in sync with each other for them to make sense, such as a list of names and a list of addresses. Here the 4th name on the first list corresponds to the 4th address on the second list. Creating and validating these two lists is non-trivial.
- Text that must be in a given format. Passing text consisting of nothing but spaces may constitute an invalid employee last name.

Consider the following function:

```
/************************************************************************
 * HANDLE ACTION
 * Execute one of a collection of actions depending on the “command” parameter
 ************************************************************************/
void handleAction(int command)
{
   if (command == 1)
      displayInstructions();
   else if (command == 2)
      openFile();
   else if (command == 3)
      playGame();
   else if (command == 4)
      exitProgram();
   else
      cout << "Invalid or unknown command: " << command << endl;
}
```

Notice how the parameter is a single integer. This may seem to be a candidate for Simple Coupling. However, the integer is not used in trivial way. Both the caller and the callee share a complex command language were actions are represented with numbers. If the language is expanded or reduced, both the caller and the callee will need to change their code to accommodate the change. This makes the two functions tightly Coupled.

Consider the following function:

```
/************************************************************************
 * DISPLAY FULL NAME
 * Display the user’s full name in the format: last name, first name.
 ************************************************************************/
void displayFullName(char nameLast[256], char nameFirst[256])
{
   cout << nameLast << ", " << nameFirst;
}
```

This seemingly simple function is actually incomplete. What if the last name string was empty? What if it contained a newline character or a comma? Dealing with all the formatting needs of a last name (no spaces, only limited punctuation, no newline characters, no numbers) is non-trivial to validate. That makes this function’s Coupling level Complex.

Document Coupling

Another tight level of Cohesion is Document. Here data conforming to some language is passed from a producer to a consumer. Note that the producer could be the caller or the callee, depending on the direction of information flow. The important thing to note is that both the producer and the consumer need to fully understand the intricacies of the shared language. The formal definition of Document Coupling is:

> ***At least one piece of information contains a rich language including syntactic and/or semantic rules.***

On the surface, this may seem rather difficult to understand and perhaps not an important case worth considering. In practice, however, it is more common than you may think.

In order to demonstrate Document Coupling, it is necessary to use C++ langauge constructs that will not be introduced for several chapters. Just pay attention to the commands and try to get the jist of the algorithm.

```
/*****************************************************************************
 * EXECUTE ASSEMBLY COMMAND
 * Take action according to an assembly opcode such as “ADD 4”
 *****************************************************************************/
void executeAssemblyCommand(string opcode, int & register)
{
   // get “ADD” from “ADD 4”
   string command = opcode.substr(0, 3);

   // get “4” from “ADD 4”
   string parameter = opcode.substr(4);

   // execute the command.
   if (command == "ADD")
      register += integerFromString(parameter);
   if (command == "SUB")
      register -= integerFromString(parameter);

   … and so on …
}
```

The above function needs to understand the complete assembly language and syntax in order to properly handle the input in the `opcode` parameter. Ths language certainly has “rich syntatic and sematic rules.”

Another example, this time taken from the Unit 3 project. Consider the game MadLib® where a story has placeholders in which the player of the game will insert his or her own answers to certain questions. The game starts with a raw story similar to the following:

```
I have a very :adjective pet :animal :.
```

Here the user will be prompted for an adjective and an animal:

```
        Adjective: silly
        Animal: great white shark
```

The result of this game will be a completed story with the user’s provided prompts:

```
I have a very silly pet great white shark.
```

A function interpreting the raw story and generating a completed story will need to exhibit Document Coupling because it will need to understand how to interpret all the prompts.

Interactive Coupling

Interactive Coupling is closely related to Document with an additional component: the information exchange between components is ongoing rather than a single one-time event. Interactive Coupling is usually characterized by a session where conversations begin, the participants react to each other, and conversations are terminated. There are usually syntactic and sematic rules that need to be followed to correctly maintain the conversation. Another way to look at Interactive coupling is a sequence of Document interactions involving maintenance of state by both parties. The formal definition of Interactive Coupling is:

> ***There exists a communication avenue between units of software involving non-trivial dialogs, sessions, or interactions.***

It is difficult to show an example of Interactive Coupling using the programming tools presented in CS 124. The main problem is that the two functions exhibiting this interaction need to maintain state. In other words, they both need to keep track of the status of the conversation. To demonstrate this, consider a function called `send()` which sends messages to another function or entity in the system.

```
/***************************************************************************
 * SEND EMAIL
 * Send an e-mail message using the SMTP protocol
 * Here sendEmail() is demonstrating Simple Coupling whereas send() is Interactive
 ***************************************************************************/
void sendEmail(char data[256])
{
   // initiate the conversation
   char *response = send("HELO");

   // from and to:
   response = send("MAIL FROM:<sender@sourcedomain.com>");
   response = send("RCPT TO:<recipient@destinationdomain.com>");

   // send body of the message
   response = send(data);

   // indicate we are done
   response = send("\n.\n");
   response = send("QUIT");
}
```

The SMTP e-mail protocol involves several many interactions between the client and the server. Notice that all of these interactions occur through the same `send()` function. Here the server responds to messages differently depending on the state of the message. Thus the `send()` function is exhibiting Interactive Coupling.

Superfluous Coupling

The tightest and therefore worst level of coupling is Superfluous. The formal definition of Interactive Coupling is:

> ***At least one piece of data or information is passed between functions unnecessarily.***

This is the only level of coupling that makes a reference to the amount of data passed between functions. Data passing between functions is only bad if that data is not necessary. There are two important parts to this definition: what is unnecessary and what is data/information passing. The unnecessary component is completely domain specific. For example, if a function has read/write access to an asset when read-only is required, then the write aspect is unnecessary and the coupling can be classified as superfluous. For example, consider the following function:

```
/**************************************************************************
 * PROMPT GRADE
 * Prompt the user for his/her GPA
 **************************************************************************/
float promptGrade(float grade)
{
   cout << "What is your GPA. Please enter a value between 0.0 and 4.0: ";
   cin  >> grade;
   return grade;
}
```

On the surface, this appears to be Simple Coupling: the grade parameter needs to be validated before it is used (what if the user selected -1.0 as the GPA?), but that validation can be easily accomplished. However, there is a problem. Data needs to leave this function, but it does not need to enter the function. For some reason, there is an input parameter called grade. This function would be Simple Coupling if grade was a local variable. However, because it is an unnecessary input parameter, this is Superfluous Coupling.

The "passing" component of the definition is a bit more difficult to explain. Consider the range of scope for a language like C++ language (from small to large): block, local variable, one-way parameter (by-value), two-way parameter (by-reference), and global variables. There are legitimate uses for each of these scope levels in many applications. However, if a function utilizes a scope larger than is necessary, then superfluous coupling exists. The degree of superfluosity depends on the number of scope levels beyond that which is necessary that was utilized in a given application. Consider the following function:

```
/**************************************************************************
 * DISPLAY USER HEIGHT
 * Display the height of the user in convenient units
 **************************************************************************/
void displayUserHeight(float heightFeet)
{
   // display the height in imperial units
   if (useImperialUnits)
      cout << heightFeet << " feet";

   // display the height in the metric system
   else
      cout << (heightFeet * 0.3048) << " meters";
}
```

This function may appear to be Simple Coupling because there are no output parameters and the single input parameter is easy to verify. However, there is another variable referenced (useImperialUnits) not declared in this function. This function is therefore a global variable, a scope much larger than necessary. The reference to this variable makes the entire function Superfluous Coupling.

Problems

Problem 1

Create a list of the levels of Cohesion from best on the top to worst on the bottom.

- ____
- ____
- ____
- ____

Please see page 117 for a hint.

Problem 2

Classify the form of Cohesion from the following example of code:

```
void displayGPA(float gpa)
{
   // configure the output so the GPA appears correctly
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(1);

   // display the GPA
   cout << gpa;
}
```

Answer:

____

Please see page 113 for a hint.

Problem 3

Classify the form of Cohesion from the following example of code:

```
float getGPA()
{
   // get the student’s GPA
   float gpa;
   cin >> gpa;

   // prompt for the number of credits
   cout << "Please enter the number of credits you are taking this semester: ";
   return gpa;
}
```

Answer:

____

Please see page 113 for a hint.

Problem 4

Create a list of the levels of Coupling from best on the top to worst on the bottom.

- ______________________________
- ______________________________
- ______________________________
- ______________________________
- ______________________________
- ______________________________
- ______________________________

Please see page 117 for a hint.

Problem 5

Classify the form of Coupling from the following example of code:

```
void displayGreeting()
{
   // prompt for name
   char name[256];
   cout << "What is your name? ";
   cin  >> name;

   // display the greeting
   cout << "Hello, " << name << " how are you today?\n";
}
```

Answer:

____

Please see page 113 for a hint.

Problem 6

Classify the form of Coupling from the following example of code:

```
void setGender()
{
   char input;
   cout << "Are you male? ";
   cin  >> input;

   isFemale = (input == 'n') || (input == 'N');
}
```

Answer:

____

Please see page 113 for a hint.

Problem 7

Match the Cohesion name with the definition:

Strong	At least one part of the function is unnecessary to the main task at hand
Extraneous	Components of the function are irrelevant and parts of the function are missing
Partial	Does one thing completely and one thing only
Weak	The main task of the function is not completely done

Please see page 113 for a hint.

Problem 8

Match the Coupling name with the definition:

Trivial	A dialog exists between functions that involves multiple interactions
Encapsulated	Information is passed between functions that involves a complex language
Simple	At least one parameter is non-trivial to create, validate, or interpret
Complex	All the parameters are in a convenient form and is guaranteed to be in a valid state
Document	At least one parameter is passed between functions unnecessarily
Interactive	There is no information interchange between functions
Superfluous	All the parameters are easy to select, interpret, and validate

Please see page 113 for a hint.

Problem 9

Classify the form of Cohesion from the following example of code:

```
float getGPA()
{
   // set up the display for GPA
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(1);

   // prompt for GPA
   float gpa;
   cout << "Enter your GPA: ";
   cin  >> gpa;
   return gpa;
}
```

Answer:

____

Please see page 113 for a hint.

Problem 10

Classify the form of Cohesion from the following example of code:

```
double getIncome()
{
   cout << "Enter your income: ";
   return 0.0;
}
```

Answer:

____

Please see page 113 for a hint.

Problem 11

Identify the type of Coupling that the following function exhibits:

```
float absoluteValue(float value)
{
   if (value >= 0)
      return value;
   else
      return -value;
}
```

Answer:

____

Please see page 117 for a hint.

Problem 12

Identify the type of Coupling that the following function exhibits:

```
char adjustedGrade(char grade, bool cheated)
{
   if (cheated)
      return 'F';
   else
      return grade;
}
```

Answer:

____

Please see page 117 for a hint.

Problem 13

Draw a Structure Chart to convert Fahrenheit to Celsius. The program has three functions besides main():

```
float getTemperature();
float convert(float fahrenheit);
void display(float celsius);;
```

Answer:

Please see page 112 for a hint.

Problem 14

Draw a Structure Chart to represent the following problem: prompt the user for his hourly wage and number of hours worked. From this, compute his weekly pay (taking time-and-a-half overtime into account). Deduct from his pay his tax and tithing. Finally, display to the user how much money he has left to spend:

Please see page 112 for a hint.

Problem 15

Sue wants to write a program to help her determine how much money she is spending on her car. Specifically, she wants to know how much she spends per day having the car sit in her driveway and how much she spends per mile driving it. This program will take into account periodic costs such as devaluation, insurance, and parking. It will also take into account usage costs such as gas, repair costs, and tires. Draw a Structure Chart to represent this program.

Please see page 112 for a hint.

Assignment 2.0

Your assignment is to create three Structure Charts representing how you would solve three programming problems. In these examples, you are not concerned with how you would implement each individual function. Instead, you want to identify what the function will do (Cohesion), how information will pass between them (Coupling), and how the functions call each other.

Problem 1: Compute grade

The first problem is to create a Structure Chart for a program to convert a number grade (ex: 88%) into a letter grade (ex: B+). Consider the following example (input is **Underlined**):

```
What is your grade in percent: 88
Your grade is B+
```

Problem 2: Compute tithing

The second problem is designed to help a child set aside part of his allowance for tithing. This program will prompt the user for his allowance, figure out how much is left after tithing is taken out, and display the results.

```
What is your allowance? $10.50
You get to spend: $9.45
```

Problem 3: Currency

The final problem is designed to help an international traveler convert his money to various currencies. After prompting him for the amount to be converted, the program will display how many British pounds, Euros, or Japanese Yen he will have:

```
How much money do you want to convert? $100.00
        British Pounds: £61.50
        Euros: €70.09
        Japanese Yen: ¥8079.06
```

Please bring these Structure Charts into class on a sheet of paper (face-to-face students) or take a picture and submit it electronically (online students). Don't forget to put your name on your assignment!

Please see page 112 for a hint.

2.1 Debugging

Sam has just spent an hour and a half in the lab tracking down a bug that turned out to be a small typo. What a waste of time! There are so many better things he could have been doing with that time (such as trying to get a date with that cute girl in his computer class named Sue). If only there was some way to get his program to tell him where the problems were, then this whole process would be much simpler!

Objectives

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

- Create asserts to catch many of the most common programmer problems.
- Use `#define` to move constants to the top of a program.
- Use `#ifdef` to create debug code in order to test a function.
- Write a driver program to verify the correctness of a function.
- Create stub functions to make an outline of a large program.

Prerequisites

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

- Create a function in C++ (Chapter 1.4).
- Convert a logic problem into a Boolean expression (Chapter 1.5).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).
- Measure the cohesion level of a function (Chapter 2.0).
- Measure the degree of coupling between functions (Chapter 2.0).
- Create a map of a program using structure charts (Chapter 2.0).

Asserts

When writing a program, we often make a ton of assumptions. We assume that a function was able to perform its task correctly; we assume the parameters in a function are set up correctly; and we assume our data structures are correctly configured. A diligent programmer would check all these assumptions to make sure his code is robust. Unfortunately, the vast majority of these checks are redundant and, to make matters worse, can be a drain on performance. A method is needed to allow a programmer to state all his assumptions, get notified when the assumptions are violated, and have the checks not influence the speed or stability of the customer's program. Assertions are designed to fill this need.

An assert is a check placed in a program representing an assumption the developer thinks is always true. In other words, the developer does not believe the assumption will ever be proven false and, if it does, definitely wants to be notified. An assert is expressed as a Boolean expression where the true evaluation indicates the assumption proved correct and the false evaluation indicates violation of the assumption. Asserts are evaluated at run-time verifying the integrity of assumptions with each execution of the check.

An assert is said to fire when, during the execution of the program, the Boolean expression evaluates to `false`. In most cases, the firing of an assert triggers termination of the program. Typically the assert will tell the programmer where the assert is located (program name, file name, function, and line number) as well as the assumption that was violated.

Assertions have several purposes:

- **Identify logical errors**. While writing a program, assertions can be helpful for the developer to identify errors in the program due to invalid assumptions. Though many of these can be found through more thorough investigation of the algorithm, the use of assertions can be a time saver.
- **Find special-case bugs**. Testers can help find assumption violations while testing the product because their copy of the software has the asserts turned on. Typically, developers love this class of bugs because the assert will tell the developer where to start looking for the cause of the bug.
- **Highlight integration issues**. During component integration activities or when enhancements are being made, well-constructed assertions can help prevent problems and speed development time. This is the case because the asserts can inform the programmer of the assumptions the code makes regarding the input parameters

Assertions are not designed for:

- **User-initiated error handling**. The user should never see an assert fire. Asserts are designed to detect internal errors, not invalid input provided by the user.
- **File errors**. Like user-errors, a program must gracefully recover from file errors without asserts firing.

Syntax

Asserts in C++ are in the `cassert` library. You can include asserts with:

```
#include <cassert>
```

Since asserts are simply C++ statements (more precisely, they are function calls), they can be put in just about any location in the code. The following is an example assert ensuring the value `income` is not negative:

```
assert(income >= 0);
```

If this assert were in a file called `budget.cpp` as part of a program called `a.out` in the function called `computeTithing`, then the following output would appear if the assumption proved to be invalid:

```
a.out: budget.cpp:164: float computeTithing(float income): Assertion `income >= 0'
failed.
Aborted
```

It is important that the client never sees a build of the product containing asserts. Fortunately, it is easy to remove all the asserts in a product by defining the `NDEBUG` macro. Since asserts are defined with pre-processor directives, the `NDEBUG` macro will effectively remove all assert code from the product at compilation time. This can be achieved with the following compiler switch:

```
g++ -DNDEBUG file.cpp
```

Sue's Tips

The three most common places to put asserts are:

1. At the top of a function to verify that the passed parameters are valid.
2. Just after a function is called, to ensure that the called function worked as expected.
3. Whenever any assumption is made.

Example 2.1 – Asserts

Unit 2

Demo

This example will demonstrate how to add asserts to an existing program. This will include how to brainstorm of where bugs might exist, common checkpoints where asserts may reside, and how to interpret the messages that asserts give us.

Problem

Given a program to compute how much tithing an individual should pay given an income, add asserts to catch common bugs.

```
What is the income? 100
Tithe for $100 is $10
```

Solution

The first part is to include the assert library:

```
#include <cassert>
```

Next, we will add asserts to the `computeTithe()` function.

```
float computeTithing(float income)
{
   assert(income >= 0.00);       // this only works for positive income

   // compute the tithing
   float tithe = income * 0.10;
   assert(tithe >= 0.00);        // The Lord doesn't owe us, right?
   assert(income > tithe);       // 10% should be less than 100%, right?

   // return the answer
   return tithe;
}
```

Observe how the asserts both validate the input (`income >= 0`) and perform sanity checks that the resulting output is not invalid (`tithe >= 0.0` as well as `income > tithe`). The first assert is designed to make sure the function is called correctly. The second is to make sure the math was performed correctly..

Challenge

As a challenge, add asserts to your Project 1 solution. Would any of these have caught bugs you ran into when you were writing your code?

See Also

The complete solution is available at 2-1-asserts.cpp or:

```
/home/cs124/examples/2-1-asserts.cpp
```

#define

The `#define` mechanism is a way to get the compiler to do a search-replace though your file before the program is compiled. This is useful for values that never change (like π). We will also use this to do more advanced things. An example of `#define` in action is:

Before expansion	After expansion

```
include <iostream>
using namespace std;

#define PI 3.14159

/**********************************
 * MAIN
 * Simple program to
 * demonstrate #define

*********************************/
int main()
{
   cout << "The value of pi is "
        << PI
        << endl;
   return 0;
}
```
	```
	include <iostream>
	using namespace std;



	/**********************************
	 * MAIN
	 * Simple program to
	 * demonstrate #define

	*********************************/
	int main()
	{
	   cout << "The value of pi is "
	        << 3.14159
	        << endl;
	   return 0;
	}
	```

Note that `#defines` are always `ALL_CAPS` according to our style guidelines.

Observe how the value in the `#define` can be used much like a variable. In many ways, it is like a variable with one important exception: it can't vary. In other words, the value represented by the `#define` is guaranteed to not change during the course of the program.

Sam's Corner

There is another way to make a variable that does not change: a constant variable:

```
const float PI = 3.14159;
```

Observe how the syntax is similar to any other variable declaration, including using the assignment operator with the semicolon. It is important to realize that this is not a global variable: the `const` modifier guarantees that the value in the variable does not change. Global variables are only dangerous because they can change in an unpredictable way

A few common uses of `#defines` are:

- **Constants**: Values that never change. Through the use of `#defines`, it is much easier to verify that all instances of the constant are the same in the program. We don't want to have more than one version of π for example.
- **Static file names**: Some file names, such as configuration files, are always in the same location. By using a #define for the file name, it is easy to see which files the program accesses and to ensure that all the parts of the code refer to the same file.

We can also put parameters in `#define` macros. Here, the syntax is similar to that of a function:

```
#define NEGATIVE(x) (-x)                    // also ALL_CAPS
```

Again, this will expand just before compilation just like the non-parameter `#define` does. Consider the following code:

Before expansion

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

#define ADD_TEN(x) (x + 10)

/*****************************
 * MAIN
 * #define expansion demo
 *****************************/
int main()
{
   int value = 5;
   cout << value
        << " + 10 = "
        << ADD_TEN(value)
        << endl;
   return 0;
}
```

After expansion

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



/*****************************
 * MAIN
 * #define expansion demo
 *****************************/
int main()
{
   int value = 5;
   cout << value
        << " + 10 = "
        << (value + 10)
        << endl;
   return 0;
}
```

For more information about the `#define` pre-processor directive, please see:

#define

#ifdef

Another pre-processor directive (along with `#define` and `#include`) is the `#ifdef`. The `#ifdef` preprocessor directive tells the compiler to optionally compile some code depending on the state of a condition. This makes it possible to have some code appear only in a Debug version of the program. Consider the following code:

```
/*****************************************
 * COMPUTE TAX
 * Compute the monthly tax
 *****************************************/
float computeTax(float incomeMonthly)
{
   float incomeYearly = incomeMonthly * 12.0;

#ifdef DEBUG                               // the code between #ifdef and #endif
   cout << "incomeYearly == "              //     only gets compiled if the
        << incomeYearly << endl;           //     DEBUG macro is defined
#endif // DEBUG

   float taxYearly;

   // tax code
   …

#ifdef DEBUG                              // observe how we format the output so we
   cout << "taxYearly == "                //     can tell which variable we are
        << taxYearly << endl;             //     looking at in the output stream
#endif // DEBUG

   return taxYearly / 12.0;
}
```

In this example, we have debug code displaying the values of key variables. Note that we don't always want this code to execute; test bed will certainly complain about the unexpected output. Instead, we only want the code to run when we are trying to fix a problem. The #ifdef mechanism allows this to occur. We can "turn on" the debug code with:

```
#define DEBUG
```

If this appears before the #ifdefs, then all the code will be included in the compilation and the couts will work as one expects. This allows us to have two versions in a single code file: the ship version containing code only for the customer to see, and the debug version containing tons of extra code to validate everything.

Sam's Corner

An #define can also be turned on at compilation time without ever touching the source code. We do this by telling the compiler we want the macro defined:

```
g++ -D<MacroName>
```

For example, if you want to turn on the DEBUG macro without using #define DEBUG, this can be accomplished with:

```
g++ -DDEBUG file.cpp
```

As an exercise, please take a close look at (/home/cs124/examples/2-1-debugOutput.cpp). Please see the following link for more detail on how #ifdef works.

#ifdef

Sam's Corner

The aforementioned #ifdef technique to display debug code can be tedious to write. Fortunately there is a more convenient way to do this. First, start with the following macro at the top of your program:

```
#ifdef DEBUG
#define Debug(x)  x
#else
#define Debug(x)
#endif
```

This macro actually does something quite clever. If DEBUG is defined in your program, then anything inside the debug() statement is executed. If DEBUG is not defined, then nothing is executed. Consider the following code:

```
void function(int input)
{
   Debug(cout << "input == " << input << endl);
}
```

If DEBUG is defined, then the above is expanded to:

```
void function(int input)
{
   cout << "input == " << input << endl;
}
```

If DEBUG is not defined, we get:

```
void function(int input)
{

}
```

Unit 2

Example 2.1 – Debug Output

Demo

This example will demonstrate how to add COUT statements to get some insight into how the program is behaving. It will do this by utilizing `#define` directives, `#ifdef` directives, and asserts.

Problem

Write a program to compute an individual's pay taking into account time-and-a-half overtime.

```
What is your hourly wage? 12
How many hours did you work? 39.5
Pay: $ 474.00
```

From this example, insert debug code to help discover the location of bugs.

```
What is your hourly wage? 12
How many hours did you work? 39.5
main: hourlyWage:  12
main: hoursWorked: 39.5
computePay(12.00, 39.50)
computePay: Regular rate
Pay: $ 474.00
```

Solution

The first thing to do is to add a mechanism to easily put debug code in the program.

```
#ifdef DEBUG
#define Debug(x) x
#else
#define Debug(x)
#endif // !DEBUG
```

Now, if `DEBUG` is not defined, none of the code in `Debug()` gets executed. If it is defined, then we can get all our debug output. The `computePay()` function with `Debug()` code is:

```
float computePay(float hourlyWage, float hoursWorked)
{
   Debug(cout << "computePay(" << hourlyWage << ", " << hoursWorked << ")\n");
   float pay;

   // regular rate
   if (hoursWorked < CAP)
   {
      Debug(cout << "computePay: Regular rate\n");
      pay = hoursWorked * hourlyWage;                      // regular rate
   }

   // overtime rate
   else
   {
      Debug(cout << "computePay: Overtime\n");
      pay = (CAP * hourlyWage) +                           // first 40 normal
         ((hoursWorked - CAP) * (hourlyWage * OVERTIME));  // balance overtime
   }
   return pay;
}
```

See Also

The complete solution is available at 2-1-debugOutput.cpp or:

```
/home/cs124/examples/2-1-debugOutput.cpp
```

Driver programs

Drivers are special programs designed to test a given function. This is an exceedingly important part of the programming process. An aerospace engineer would never put an untested engine in an airplane. He would instead mount the engine on a testing harness and run it through the paces. Only after exhaustive testing would he feel confident enough to put the engine in the airplane. We should also treat new functions with skepticism. When we validate functions before integrating them into the larger program, it is far easier to localize problems. After the function has been validated, then we can safely copy it to the project. Typically drivers consist of just the function `main()` and the function to be tested. Consider, for example, the prototype for the function `computePay()`:

```
float computePay(float hourlyWage, float hoursWorked);
```

A driver program for `computePay()` might be:

```
/**************************************
 * MAIN
 * Simple driver for computePay()
 **************************************/
int main()
{
   float wage;
   cout << "wage: ";                        // get the data as quickly as possible
   cin  >> wage;

   float hours;
   cout << "hours: ";                       // again, just the simplest prompt
   cin  >> hours;

   cout << "computePay("
        << "hourlyWage = " << wage << ", "   // show what was passed
        << "hoursWorked = " << hours
        << ") == "
        << computePay(wage, hours)          // show what was returned
        << endl;

   return 0;
}
```

Observe how the driver program is just a bare-bones program whose only purpose is to prompt the user for the data to pass to the function and to display the results. When you use the driver-program development methodology, you:

1. Start with a blank file. The only thing this program will do is test your function.
2. Write the function. As long as the coupling is loose, this should not be too complex.
3. Create a `main()` that only calls your function. This is typically done in three steps:
   a. First call your function with the simplest possible data.
   b. If your function requires any parameters, create simple cin statements in `main()` to fetch that data directly from the user.
   c. If your function returns something, display the results directly on the screen so it is easy to verify how the function responded to input.
4. Test your function with a variety of input. Start with simple input and work to more complex scenarios.

Example 2.1 – Driver

Demo

This example will demonstrate how to write a simple driver program to test a function we used in Project 1: `computeTax()`.

Solution

The drive program exists entirely in main(). We start with the prototype of the function we are testing:

```
double computeTax(double incomeMonthly);
```

There will be two parts: fetching the data from the user for the function parameters, and displaying the output of the function.

```
/******************************************************
 * MAIN
 *    A simple driver program for computeTax()
 ******************************************************/
int main()
{
   // get the income
   double income;              // the inputs to the function being
   cout << "Income: ";         //    tested is gathered directly from
   cin  >> income;             //    the user and sent to the function

   // call the function and display the results
   cout << "computeTax(" << income << ") == "  // what we are sending...
        << computeTax(income)                  // what the output is
        << endl;

   return 0;
}
```

Driver programs are very streamlined and simple. Once we have tested our function, we can safely throw them away.

Challenge

As a challenge, write a driver program for `computeTithe()` from your Project 1 code.

See Also

The complete solution is available at 2-1-driver.cpp or:

```
/home/cs124/examples/2-1-driver.cpp
```

This process works well when you are in the development phase of the project. You can also use the driver-program technique when you are in the testing and debugging phase of the project. This can be accomplished by modifying our `main()` to be a driver for any function in the program. Recall the `computeTax()` function from Project 1. We might think we have worked out all the bugs of the functions before they were integrated together. When running the program, however, it becomes apparent that something is broken.

Consider the following `main()` from Project 1 after it has been modified to test `computeTithing()`. Note how we use a `return` statement to ensure only the top part of the function is executed.

```
/***********************************************************
 * MAIN
 * Keep track of a monthly budget
 ***********************************************************/
int main()
{
   double incomeTest = getIncome();             // use the get function or a cin
   cout << computeTithing(incomeTest) << endl;  // simple display of the output
   return 0;                                    // return ensures we exit here and
                                                //    only test computeTithing()
                                                // rest of main below here

   // instructions
   cout << "This program keeps track of your monthly budget\n";
   cout << "Please enter the following:\n";
   // prompt for the various data
   double income          = getIncome();
   double budgetLiving    = getBudgetLiving();
   double actualLiving    = getActualLiving();
   double actualTax       = getActualTax();
   double actualTithing   = getActualTithing();
   double actualOther     = getActualOther();

   // display the results
   display(income, budgetLiving, actualTax, actualTithing,
           actualLiving, actualOther);
   return 0;
}
```

The driver program technique has been used for almost all the assignments we have done this semester.

Stub functions

A stub is a placeholder for a forthcoming replacement promising to be more complete. In the context of designing and building a program, a stub is a tool enabling us to put a placeholder for all the functions in our structure chart without getting bogged down with how the functions will work. In other words, stubs allow us to:

- **Get Started**: Stubs allow us to get the design from the Structure Chart into our code before we have figured out how to implement the functions themselves. This helps answer the question "how do I start on this project?"
- **Figure out data flow**. Because stubs include the parameters passed between functions, you can model information flow early in the development process.
- **Always have your program compile**. A program completely stubbed-out will compile even though it does not do anything yet. Then, as you implement individual functions, any compile errors you encounter are localized to the individual function you just implemented.

Consider the `computeTax` function from Project 1. The following would be an example of a stub:

```
float computeTax(float income)
{
   // stub for now...
   return 0.0;
}
```

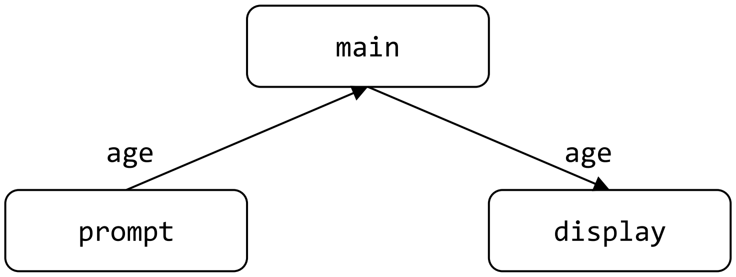
Example 2.1 – Stub Functions

Demo

This example will demonstrate how to turn a structure chart into stub functions.

Problem

Write stub functions for the following structure chart.



Solution

The stubbed version of this program would be:

```
int prompt()                 // don't bother with the function comment block
{                            //     with stubbed functions. We will do them later
   return 0;                 // make sure to return some value because the return
}                            //     type is not void in this function

void display(int age)        // all the parameters need to be present in the stub
{                            //     even if the body of the functions are empty.
}

int main()                   // make sure the stubs call all the children
{                            //     functions so we can make sure the data flow
   display(prompt());        //     works correctly. This should enable us to
   return 0;                 //     implement the functions in any order we choose
}
```

Observe how the stubbed functions consist of:

- **Prototypes**: the function name, return type, and parameters.
- **Empty body**: except for a return statement, the body is mostly empty.
- **Called functions**: include code to call the child functions. It is OK to use dummy parameters if none are know

Challenge

As a challenge, try to stub Project 1. A sample solution is available at 2-1-stubbed.cpp or:

```
/home/cs124/examples/2-1-stubbed.cpp
```

Problem 1

Which of the following most clearly illustrates the concept of coupling?

- The parameters passed to a function
- The task that the function performs
- The subdivision of components in the function or program
- The degree in which a function can be used for different purposes

Please see page 117 for a hint.

Problem 2

Identify the type of coupling that the following function exhibits:

```
float lastGrade = 0.0;

float getGPA()
{
   cout << "What is your grade? ";
   cin  >> lastGrade;

   return lastGrade;
} ;
```

Answer:

____

Please see page 117 for a hint.

Problem 3

Create a stub for the following function:

```
void displayIncome(float income)
{
   // configure output to display money
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);

   // display the income
   cout << "Your income is: $"
        << income
        << endl;
};
```

Answer:

Please see page 140 for a hint.

Problem 4

Create a stub for the following function:

```
float getIncome()
{
   float income;

   // prompt
   cout << "Please enter your income: ";
   cin  >> income;

   return income;
}
```

Answer:

Please see page 140 for a hint.

Problem 5

Create a driver program for the following function:

```
float sqrt(float value);
```

Answer:

Please see page 138 for a hint.

Problem 6

Create a driver program for the following function:

```
void displayTable(int numDaysInMonth, int offset);
```

Answer:

Please see page 138 for a hint.

Problem 7

Create an assert to verify the following variable is within the expected range:

```
float gpa
```

Answer:

Please see page 132 for a hint.

Problem 9

What asserts would you add to the beginning of the following function:

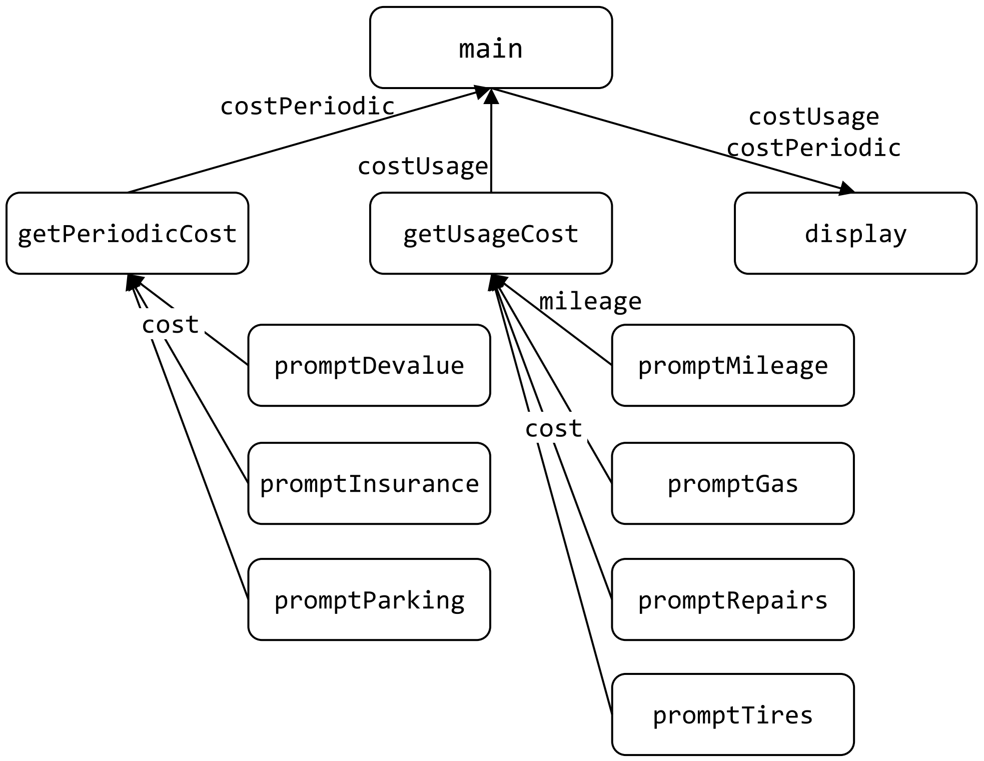
```
void displayDate(int day, int month, int year);
```

Answer:

Please see page 49 for a hint.

Assignment 2.1

Sue wants to write a program to help her determine how much money she is spending on her car. Specifically, she wants to know how much she spends per day having the car sit in her driveway and how much she spends per mile driving it. While working through this problem, she came up with the following structure chart:



Please create stub functions for all the functions in Sue's program. In other words, write a program to stub out every function represented in the above structure chart. If a function calls another function (ex: `getPeriodicCost()` calls `promptParking()`), then make sure that function call is in the stub. Finally, make sure all the parameters and return values from the structure chart are represented in the stub functions.

One final note: you do not need to have function headers for each individual function.

Please:

- Create stub functions for all the functions mentioned in the structure chart.
- If you were able to do this, then enter the following code in the function `display()`:

```
cout << "Success\n";
```

- Use the following testbed:

```
testBed cs124/assign21 assign21.cpp
```

- Submit to Assignment 21

Please see page 140 for a hint.

2.2 Designing Algorithms

Sue just spent a half hour writing the code for a certain function before she realized that she got the design all wrong. Not only was her code broken, but her entire approach to the problem was all wrong. Sue is frustrated: writing code is hard! If only there was a way to design a function without having to go through the work of getting it to compile…

Objectives

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

- Recite the pseudocode keywords.
- Understand the degree of detail required for a pseudocode design.
- Generate the pseudocode corresponding to a C++ function.

Prerequisites

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

- Create a map of a program using structure charts (Chapter 2.0).

Reading

All of the reading in this section will consist of the pseudocode videos:

- Design First: This video discusses why it is important to design a problem before the implementation is begun.
- When to Design: This video will discuss different times in the software development process when design activities are most effective.
- Different Design Approaches: This video will introduce four techniques to drafting a design: the paragraph method, flowchart, structure diagram, and pseudocode.
- Using Pseudocode: How pseudocode can be used as a tool to more effectively write software.
- Rules of Pseudocode: The conventions and rules of pseudocode.
- Pseudocode Keywords: The seven classes of keywords that are used in pseudocode.

Another design tool

Pseudocode, along with the Structure Chart, is one of our most powerful design tools. While the Structure Charts is concerned with what function we have in our program and how the functions interact with each other, Pseudocode is concerned with what goes on inside individual functions.

There are several reasons why we need to draft a solution before we start writing code. First, we need to be able to think big before getting bogged down in the details. Programming languages resist this process; they need you to always work at the most detailed level. This can be very frustrating; you are trying to solve a large problem but the language keeps forcing you to specify data types and work through syntax errors.

Using pseudocode

Pseudocode is a tool helping the programmer bridge the high-level English description of a problem and the C++ solution. To illustrate how this works, we will begin with a natural language definition of a problem. In this case, how to compute your tax liability with a simple 2 tier tax table.

Given a user's income, compute their tax burden by looking up the appropriate formula on the following table:

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

First, we will introduce structure in the problem definition by dividing the process into discrete steps. We are entering the realm of pseudocode here, but there is still far too much natural language to be much of a help.

Determine tax bracket the user's income according to the ranges in this table:

Min	Max	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%

Apply the appropriate formula:

Bracket	Formula
10%	10% of the amount over $0
15%	$1,510.00 plus 15% of the amount over $15,100
25%	$8,440.00 plus 25% of the amount over $61,300
28%	$24,040.00 plus 28% of the amount over $123,700
33%	$42,170.00 plus 33% of the amount over $188,450
35%	$91,043.00 plus 35% of the amount over $336,550

Return the results back to the caller.

Next we will be more precise with our equations and fill in more detail whenever possible. We are beginning to specify how things will be accomplished, not just what will be accomplished. Observe how our pseudocode is moving closer to our programming language with every step.

```
IF income is less than $15,100 then
    tax is 10% of the amount over $0
IF income between $15,100 and $61,300 then
    tax is $1,510 plus 15% of the amount over $15,100
IF income between $61,300 and $123,700 then
    tax is $8,440 plus 25% of the amount  over $61,300
IF income between $123,700 and $188,450 then
    tax is $24,040 plus 28% of the amount over $188,450
IF income is above $188,450 then
    tax is $91,043 plus 35% of the amount over $336,550
```

Finally, we reduce all operations to pseudocode keywords. Now we are ready to start writing code. The problem has been solved and now it is just a matter of translating the syntax-free pseudocode into the specific syntax of the high level language.

```
computeTax(income)

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

   RETURN tax
END
```

Always remember that pseudocode is just a design tool. With the exception of a few assignments in this class, your purpose is to develop great software rather than write pseudocode. Therefore, you should use pseudocode only as far as it helps you to develop software. This means that sometimes you will design right down to the pseudocode keywords while other times you will be able to stop designing before that point because the solution presents itself.

Pseudocode keywords

There are seven classes of keywords: receive, send, math, remember, compare, repeat, and call functions.

Receive

A computer can receive information from a variety of input sources, such as keyboard, mouse, a network, a sensor, or wherever.

Keyword	Description	Example
READ	Receive information from a file	READ studentGrade
GET	Receive input from a user, typically from the keyboard	GET income

Send

A computer can send information to the console, display it graphically, write to a file, or operate on a device.

Keyword	Description	Example
PRINT	When sending data to a permanent output device like a printer	PRINT full student name
WRITE	When writing data to a file	WRITE record
PUT	When sending data to the screen	PUT instructions
PROMPT	Just like PUT except always preceding a GET instruction	PROMPT for user name

Arithmetic

Most processors have the built-in capability to perform the following operations:

Keyword	Description	Example
()	Parentheses are used to override the default order of operations	c = (f – 32) * 5/9
* x / ÷ mod div	Any mathematical convention can be used	numWeeks = days div 7
+ -	Addition / subtraction	SET count = count + 1
|| √ π	Common mathematical values or operations	PUT √10
≤ ≠	Common comparison operations	IF grade ≥ 60

Remember

A computer can assign a value to a variable or a memory location

Keyword	Description	Example
SET = ←	Assign a value to a variable ←	SET answer ← 42

Compare

A computer can compare two values and select one of two alternate actions

Keyword	Description	Example

IF
ELSE
	For two possible outcomes	IF income / 10 ≤ tithingPaid
		    PUT full tithe message
		ELSE
		    PUT scripture from Malachi

SWITCH CASE	For multiple possible outcomes. This will be discussed in more detail in Chapter 3.5	SWITCH option
		    CASE 1
		        PUT great choice!
		    CASE 2
		        PUT could be better
		    CASE 3
		        PUT please reconsider

Repeat

A computer can repeat a group of actions.

Keyword	Description	Example
WHILE	When repeating through the same code more than once	WHILE studentGrade < 60 takeClass()
FOR	When counting	FOR count = 1 to 10 by 2s PUT count

Functions

A computer can call a function and pass parameters between functions.

Usage	Conventions	Example

Declaring	Functions are named.
	Input parameters are enumerated.
	Statements are indented.
	RETURN values.
	END.
		computeTithing(income)
		    SET tithing = income / 10
		    RETURN tithing
		END

Calling	Call by name Specify parameters	PUT computeTithing(income)

Sue's Tips

On the surface, it may seem like pseudocode is no different than C++. Why learn another language when C++ already does the job? The short answer is that pseudocode is less detailed than C++ so you can concentrate on more high-level design decisions without getting bogged down in the minutia of detail that C++ demands. The long answer is a bit more complicated.

In its truest form, pseudocode is syntax free. This means that anything goes! It starts very free-form much like natural language (English). As you refine your thinking and work out the program details, it begins to take on the structure of a high-level programming language. When you get down to the pseudocode keywords listed above, you have worked out all the design decisions. While it is not always necessary to develop pseudocode to this level of detail, it is an important skill to develop. This is why pseudocode is an important CS 124 skill to learn.

Example 2.2 – Compute Pay

Demo

This example will demonstrate the relationship between pseudocode and C++. Specifically, it will show what kinds of details are necessary in pseudocode (variable names, equations, and program logic) and which are not (variable declarations, C++ syntax, and comments).

Problem

Write the pseudocode corresponding to the following C++ function:

```
/**********************************************************
 * COMPUTE PAY
 *   Based on the user's wage and hours worked, compute the pay
 *   taking into account 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 normal
         ((hoursWorked - 40.0) * (hourlyWage * 1.5)); // balance overtime

   return pay;
}
```

Solution

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

```
computePay(hourlyWage, hoursWorked)
   IF hoursWorked < 40
      SET pay = hoursWorked x hourlyWage
   ELSE
      SET pay = (40 x hourlyWage) + ((hoursWorked - 40) x (hourlyWage x 1.5)
   RETURN pay
END
```

Observe how the pseudocode completely represents the logic of the program without worrying about data-types, declaring variables, or the syntax of the language.

Challenge

As a challenge, look at the Project 1 definition. The pseudocode for most of the functions is provided. Compare your C++ code with the provided pseudocode. See if you can write the pseudocode for the remaining functions (`computeTithing()`, `getActualTax()`, etc.).

Problem 1

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

```
bool vegas(bool b)
{
   b = false;
   return true;
}

int main()
{
   bool b;
   vegas(b);
   return 0;
}
```

Answer:

___

Please see page 65 for a hint.

Problem 2

What is the output when the user types 'a'?

```
void function(char &value)
{
   cin >> value;
   return;
}

int main()
{
   char input = 'b';
   function(input);
   cout << input << endl;
   return 0;
}
```

Answer:

___

Please see page 65 for a hint.

Problem 3

Which of the following is not a basic computer operation?

- Receive information
- Connect to the network
- Perform math
- Compare two numbers

Please see page 145 for a hint.

Problem 4

What is wrong with the following pseudocode?

```
IF age < 18
PUT message about not being old enough to vote
```

Answer:

___

Please see page 49 for a hint.

Problem 5

Which of the following is the correct pseudocode for sending a message to the user?

- DISPLAY
- `cout << "The following are the instructions\n";`
- WRITE instructions on the screen
- PUT instructions on the screen

Please see page 148 for a hint.

Problem 6

Which of the following is the correct pseudocode for computing time-and-a-half overtime?

- pay = (hours - 40) * pay * 1.5 + 40 * pay
- pay = ((float)hours - 40.0) * pay * 1.5 + 40 * (float)pay;
- pay = hours - 40 * pay * 1.5 + 40 * pay
- pay = hours over 40 times time-and-a-half plus regular pay for first 40 hours

Please see page 148 for a hint.

Problem 7

Which is the pseudocode command to differentiate between two options?

- IF
- COMPARE
- =
- same?

Please see page 149 for a hint.

Problem 8

Which pseudocode command displays data on the screen?

- PUT
- PROMPT
- OUTPUT
- SCREEN

Please see page 148 for a hint.

Problem 9

Write the pseudocode corresponding to the following C++:

```
float computeTithing(float income)
{
   float tithing;

   // Tithing is 10% of our income. Please
   // see D&C 119:4 for details or questions
   tithing = income * 0.10;

   return tithing;
}
```

Answer:

Please see page 49 for a hint.

Problem 10

Write the pseudocode for a function to determine if a given year is a leap year:

According to the Gregorian calendar, which is the civil calendar in use today, years evenly divisible by 4 are leap years, with the exception of centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

Answer:

Please see page 150 for a hint.

Problem 11

Write the pseudocode for a function to compute how many days are in a given year. Hint: the answer is 365 or 366.:

Answer:

Please see page 150 for a hint.

Problem 12

Write the pseudocode for a function to display the multiples of 7 under 100.

Answer:

Please see page 148 for a hint.

Assignment 2.2

Please create pseudocode for the following functions. The pseudocode is to be turned in by hand in class for face-to-face students and submit it as a PDF for online students:

Part 1: Temperature Conversion

```
int main()
{
   // Get the temperature from the user
   float tempF = getTemp();

   // Do the conversion
   float tempC = (5.0 / 9.0) * (tempF - 32.0);

   // display the output
   // I don't want showpoint because I don't want to show a point!
   cout.setf(ios::fixed);
   cout.precision(0);
   cout << "Celsius: " << tempC << endl;

   return 0;
}
```

Part 2: Child tax credit

```
int main()
{
   // prompt for stats
   double income      = getIncome();
   int    numChildren = getNumChildren();

   // display message
   cout.setf(ios::fixed);
   cout.precision(2);
   cout << "Child Tax Credit: $ ";
   if (qualify(income))
      cout << 1000.0 * (float)numChildren << endl;
   else
      cout << 0.0 << endl;

   return 0;
}
```

Part 3: Cookie monster

```
void askForCookies()
{
   // start with no cookies  :-(
   int numCookies = 0;

   // loop until the little monster is satisfied
   while (numCookies < 4)
   {
      cout << "Daddy, how many cookies can I have? ";
      cin  >> numCookies;
   }

   // a gracious monster to be sure
   cout << "Thank you daddy!\n";
   return;
}
```

Please see page 150 for a hint.

2.3 Loop Syntax

Sue's little brother is learning his multiplication facts and has asked her to write them on a sheet of paper. Rather than spend a few minutes to hand-write the table, she decides to write a program instead. This program will need to count from 1 to 10, so a FOR loop is the obvious choice.

Objectives

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop.
- Create a loop to solve a simple problem.

Prerequisites

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

- Convert a logic problem into a Boolean expression (Chapter 1.5).
- Generate the pseudocode corresponding to a C++ function (Chapter 2.2).

Overview

This is the first of a three part series on how to use loops to solve programming problems. The first part will focus on the mechanism of loops, namely the syntax.

Loops are mechanisms to allow a program to execute the same section of code more than once. This is an important tool for cases when an operation needs to happen repeatedly, when counting is required to solve a problem, and when the program needs to wait for an event to occur.

There are three types of loops in C++: WHILE, DO-WHILE, and FOR:

while	do-while	for

A WHILE loop is good for repeating through a given block of code multiple times.
`{`
`   while (x > 0)`
`   {`
`      x--;`
`      cout << x << endl;`
`   }`
`}`
	Same as WHILE except we always execute the body of the loop at least once.
	`{`
	`   do`
	`   {`
	`      x--;`
	`      cout << x << endl;`
	`   }`
	`   while (x > 0);`
	`}`
		Designed for counting, usually meaning we know where we start, where we end and what changes.
		`{`
		`   for (x = 10;`
		`        x > 0;`
		`        x--)`
		`   {`
		`      cout << x << endl;`
		`   }`
		`}`

WHILE

The simplest loop is the WHILE statement. The WHILE loop will continue executing the body of the loop until the controlling Boolean expression evaluates to `false`. The syntax is:

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

As with the IF statement, we can always have more than one statement in the body of the loop by adding curly braces {}s:

```
while (<Boolean expression>)
{
   <body statement1>;
   <body statement2>;
   ...
}
```

Observe how the body of the loop is indented three spaces exactly as the body of an IF statement is indented.

Sue's Tips

The WHILE loop keeps iterating as long as the Boolean expression evaluates to true. This may seem counter-intuitive at first. Some find it easier to think that the loop keeps iterating until the Boolean expression evaluates to true.

One way to keep this straight in your mind is to read the code as "while <condition> continue to <body>." For example consider the following code:

```
while (input < 0)
   cin >> input;
```

This would read "While input less-than zero, continue to prompt for new input."

It is possible to count with a WHILE loop. In this case, you initialize your counter before the loop begins, specify the continuation-condition (what must remain `true` as we continue through the loop) in the Boolean expression, and specify the increment logic in the body:

```
{
   int count = 1;
   while (count <= 10)                 // continue as long as this evaluates to true
   {                                   // use {}s because there is more than one
      cout << count << endl;           //      statement in the body of the loop
      count++;
   }
}
```

This code will display the numbers 1 through 10 on the screen, each number on its own line.

Example 2.3 – While Loop

Unit 2

Demo

This example will demonstrate the syntax of a simple WHILE loop.

Problem

Write a program to prompt the user for his letter grade. If the grade is not in the valid range (A, B, C, D, or F), then the program will display an error message and prompt again.

```
Please enter your letter grade: B
Your grade is B
```

…or…

```
Please enter your letter grade: E
Invalid grade. Please enter a letter grade {A,B,C,D,F} G
Invalid grade. Please enter a letter grade {A,B,C,D,F} C
Your grade is C
```

Solution

The most challenging part of this problem is the Boolean expression capturing whether the user input is in the valid range. Anything that is not an A, B, C, D, or F is classified as invalid.

```
char getGrade()
{
   char grade;   // the value we will be returning

   // initial prompt
   cout << "Please enter your letter grade: ";
   cin  >> grade;

   // validate the value
   while (grade != 'A' && grade != 'B' && grade != 'C' &&
          grade != 'D' && grade != 'F')
   {
      cout << "Invalid grade. Please enter a letter grade {A,B,C,D,F} ";
      cin  >> grade;
   }

   // return when done
   return grade;
}
```

Challenge

As a challenge, modify `getGrade()` so either uppercase or lowercase letter grades are accepted. This can be done by either doubling the size of the Boolean expression to include lowercase letters or by converting the grade to uppercase if it is lowercase.

See Also

The complete solution is available at 2-3-while.cpp or:

```
/home/cs124/examples/2-3-while.cpp
```

DO-WHILE

The DO-WHILE loop is the same as the WHILE loop except the controlling Boolean expression is checked after the body of the loop is executed. This is called a **trailing-condition** loop, while the WHILE loop is a **leading-condition** loop. As with the WHILE statement, the loop will continue until the controlling Boolean expression evaluates to `false`. The syntax is:

```
do
   <body statement>;
while (<Boolean expression>);
```

DO-WHILE loops are used far less frequently than the aforementioned WHILE loops. Those scenarios when the DO-WHILE loop would be the tool of choice center around the need to ensure the body of the loop gets executed at least once. In other words, it is quite possible the controlling Boolean expression in a WHILE loop will evaluate to `false` the first time through, removing the possibility the body of the loop is executed. This is guaranteed to not happen with the DO-WHILE loop because the body always gets executed *first*.

Example

Consider the following code prompting the user for his age:

```
{
   int age;
   do                                    // the "do" keyword on its own line
   {                                     // use {}s when there is more than one
      cout << "What is your age? ";      //      statement in the body of the loop
      cin  >> age;
   }                                     // keep the {}s on their own line
   while (age < 0);                      // continue until this evaluates to false
}
```

In this example, we want to prompt the user for his age at least once. The code will continue prompting the user for his age as long as the user enters negative numbers. Note how the `while` part of the DO-WHILE loop is on a separate line from the {}s. This is because the style guide specifies that {}s must be on their own lines.

Sam's Corner

It turns out that the WHILE loop and the DO-WHILE loop solve exactly the same set of problems. Any DO-WHILE loop can be converted to a WHILE loop by bringing one instance of the body outside the loop. In the above example, the equivalent WHILE loop would be:

```
{
   int age;
   cout << "What is your age? ";         // one copy of the body outside the loop
   cin  >> age;

   while (age < 0)                       // same condition as the DO-WHILE
   {
      cout << "What is your age? ";      // second copy in the body of the loop
      cin  >> age;
   }
}
```

Note how redundant it is to have the second copy of the body outside the loop. This is the primary advantage of the DO-WHILE: to reduce the redundancy.

Example 2.3 – Do-while Loop

Demo

This example will demonstrate how to use a DO-WHILE loop to validate user input.

Problem

Write a program to sum the numbers the user entered. The program will continue to prompt the user until the value zero (0) is typed.

```
Please enter a collection of integer values. When
        you are done, enter zero (0).
> 10
> 15
> -7
> 0
The sum is: 18
```

Solution

Because the user needs to be prompted at least once, a DO-WHILE loop is the right tool for the job.

```
int promptForNumbers()
{
   // display instructions
   cout << "Please enter a collection of integer values. When\n"
        << "\tyou are done, enter zero (0).\n";
   int sum = 0;
   int value;

   // perform the loop
   do
   {
      // prompt for value
      cout << "> ";
      cin  >> value;

      // add value to sum
      sum += value;
   }
   while (value != 0);
   // continue until the user enters zero

   // return and report
   return sum;
}
```

Challenge

As a challenge, can you modify the above function so the value zero is not added to the sum?

To take this one step further, modify the above problem so -1, not 0, is the terminating condition. This will require you to modify the instructions as well.

See Also

The complete solution is available at 2-3-doWhile.cpp or:

```
/home/cs124/examples/2-3-doWhile.cpp
```

FOR

The final loop is designed for counting. The syntax is:

```
for (<initialization statement>; <Boolean expression>; <increment statement>)
   <body statement>;
```

Here the syntax is quite a bit more complex than its WHILE and DO-WHILE brethren.

```
for (int count = 0; count < 5; count++)
   cout << count << endl;
```

Initialization:

The first statement to be executed in a loop.

- Can be any statement.
- We can declare and initialize a variable inside the loop:

```
for (int i = 0; …
```

- We can initialize more than one variable that is already defined elsewhere in the code:

```
for (j = 0, k = 0; …
```

- We can also leave it empty:

```
for (; i < 10; i++)
```

Boolean expression:

Is executed immediately before the body of the loop.

- Can be any expression.
- As long as the expression evaluates to `true`, the loop continues:
- If it is left empty, the expression evaluates to `true`. This means it will loop forever:

```
for (i = 0;  ; i++)
```

Increment:

Is executed immediately after the body of the loop.

- Can be any statement.
- Usually we put a `++` or `--` here:
- You can put more than one statement here:

```
for (… ; …; i++, j--)
```

- Can be left empty:

```
for (; i < 10; )
```

While the syntax of the FOR loop may look quite complex, it has the three things any counting problem needs: where to start (initialization), where to end (Boolean expression), and how much to count by (the increment statement). For example, a FOR loop to give a countdown from 10 to zero would be:

```
{
   // a countdown, just like what Cape Kennedy uses
   for (int countDown = 10; countDown >= 0; countDown--)
      cout << countDown << endl;
}
```

Sue's Tips

While it may seem difficult to remember what the three fields of a FOR loop (Initialization, Boolean expression, and increment) are for, there is a memory clue to help you remember. Consider the loop to count to four:

```
for (int count = 1; count < 5; count++)
   cout << count << endl;
```

This reads "For count equals one, *as long as* count is less than five, add one to count."

Example 2.3 – For Loop

Demo

This example will demonstrate how to use a FOR loop. It will allow the user to specify each of the three components of the loop (Initialization, Boolean expression, and Increment). In many ways, this is a driver program for the FOR loop.

Problem

Write a program prompt the user the parameters for counting. The program will then display the numbers in the specified range.

```
What value do you want to start at? 4
What value do you want to end at? 14
What will you count by (example: 2s): 3
        4
        7
        10
        13
```

Solution

The following program will count from `start` to `end`.

```
int main()
{
   // start
   cout << "What value do you want to start at? ";
   int start;
   cin  >> start;

   // end
   cout << "How high do you want to count? ";
   int end;
   cin  >> end;

   // increment
   cout << "What will you count by (ex: 2s): ";
   int increment;
   cin >> increment;

   // count it
   for (int count = start; count <= end; count += increment)
      cout << "\t" << count << endl;

   return 0;
}
```

Notice the three parts of a FOR loop: the starting condition (`int count = start`), the continuation condition (`count <= end`), and what changes every iteration (`count += increment`). As long as the continuation condition is met, the loop will continue and the body of the loop will execute.

Challenge

As a challenge, modify the above program to ensure that `start` ≤ `end` if `increment` is positive, and `start` ≥ `end` if `increment` is negative.

See Also

The complete solution is available at [2-3-for.cpp](2-3-for.cpp) or:

```
/home/cs124/examples/2-3-for.cpp
```

Problem 1

What is the output?

```
void function(int a, int &b)
{
   a = 0;
   b = 0;
}

int main()
{
   int a = 1;
   int b = 2;

   function(a, b);

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

}
```

Answer:

____

Please see page 65 for a hint.

Problem 2

What is the output?

```
int value = 1;

int main()
{
   int value = 2;
   cout << value;

   if (true)
   {
      int value = 3;
      cout << value;

      {
         int value = 4;
         cout << value;
      }
      cout << value;
   }
   cout << value << endl;
   return 0;
}
```

Answer:

____

Please see page 67 for a hint.

Problem 3

What is the output?

```
{
   int j = 10;

   for (int i = 1;
        i < 3;
        i++)
      j++;


   cout << i << endl;

   return 0;
}
```

Answer:

___

Please see page 161 for a hint.

Problem 4-7

Write the code to implement each of the following loops:

Count from 1 to 10	
Keep asking the user for input until he enters a value other than 'q'	
Display the powers of two below 2,000	
Sum the multiples of seven below 100	

Please see page 161 for a hint.

Problem 8

Which of the following has no syntax errors?

```
do while (true)
   cout << "Infinite loop!\n";
```

```
do
   cout << "Infinite loop!\n";
while (true);
```

```
do (true)
   cout << "Infinite loop!\n";
```

```
while (true)
   cout << "Infinite loop!\n";
do;
```

Please see page 159 for a hint.

Problem 9-10

Write a program that keeps prompting the user for a number until he inputs the number 0:

Use a WHILE loop

Use a DO-WHILE loop

Please see page 159 for a hint.

Unit 2

Problem 11

Write the code to count down from 100 to 10 in steps of 10:

```
100
90
80
70
60
50
40
30
20
10
```

Answer:

Please see page 162 for a hint.

Problem 12-13

Write the code to prompt the user for 5 numbers and display the sum

```
Please enter 5 numbers:
#1: 54
#2: 99
#3: 12
#4: 65
#5: 34
Sum: 264
```

Answer in pseudocode:

Answer in C++:

Please see page 161 for a hint.

Assignment 2.3

Sue's silly brother Steve has a teacher who loves to give tons of math homework. This week, the assignment is to add all the multiples of 7 that are less than 100. Last week, he had to add all the multiples of 3 that are less than 100. Sue wants to make sure that her brother gets a 100% on each assignment so she decided to write a program to validate each assignment.

Example

User input in **underline**.

```
What multiples are we adding? 5
The sum of multiples of 5 less than 100 are: 950
```

Another example:

```
What multiples are we adding? 7
The sum of multiples of 7 less than 100 are: 735
```

Assignment

Make sure you run test bed with:

```
testBed cs124/assign23 assignment23.cpp
```

Don't forget to submit your assignment with the name "Assignment 23" in the header.

Please see page 162 for a hint.

2.4 Loop Output

Sue can't seem to find the bug in her assignment. The code looks right and it compiles, but the loop keeps giving her different output than she expects. How can she ever hope to find the bug if everything executes so quickly? If only there was a way to step through the code one line at a time to see what each statement is doing…

Objectives

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

- Predict the output of a given block of code using the desk check technique.
- Recognize common pitfalls associated with loops.

Prerequisites

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop (Chapter 2.3).
- Create a loop to solve a simple problem (Chapter 2.3).
- Use `#ifdef` to create debug code in order to test a function (Chapter 2.1).

Desk Check

Please watch the following videos:

- Predicting Output: The purpose of this video is to illustrate the importance of being able to predict the output of code by inspection.
- Levels of Understanding: The purpose of this video is to illustrate different levels of understanding of an algorithm and the benefits of working at each level. There are three levels:
    1. **Concrete**: A low level understanding of the specific details of what the code is doing. It consists of a description of what happens at every step of the program execution.
    2. **Abstract**: Describes what parts of a program do, and how they relate to the larger whole.
    3. **Conceptual**: A high level comprehension that enables the programmer to explain in simple English what a program does.
- Dataflow: The purpose of this video is to illustrate that tracking dataflow is the most effective way to predict the output of a program.
- Desk Check Steps: The process of desk checking a program, from start to finish. It will describe how deck checking is a form of dataflow measurement, how the complete state of the program is captured at each level, and how every aspect of the algorithm is captured in the end. This chapter will describe the steps of performing a desk check, including line numbering, variable enumeration, and finally building the desk check table.
- Desk Check Table: The purpose of this video is to illustrate how to build and interpret a desk check table on a single-function problem.
- Desk Check with Functions: The purpose of this video is to illustrate how desk checking works across multiple functions.
- Online Desk Check: The purpose of this video is to illustrate how to desk check existing code.

Example 2.4 – Expression Desk Check

Problem

Desk check the following code:

```
{
   // get paid!
   double income = 125.37;

   // remove tithing, you keep 90%
   income *= 0.9;

   // don’t forget half goes to savings
   income /= 2.0;

   // a man has got to eat
   income -= 15.50;

   // display the results
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << "$" << income << endl; }
```

Solution

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
{
   double income = 125.37;        // (1)  extra code removed for brevity

   income *= 0.9;                 // (2)

   income /= 2.0;                 // (3)

   income -= 15.50;               // (4)

   cout << income << endl;        // (5)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution.

Line	income	output
1	125.37	
2	112.833	
3	56.4165	
4	40.9165	
5	40.9165	$40.92

Notice how we are able to track each step of execution of the code with the Desk Check table. Even though `income` changed many times in a few lines of code, we can always see the value at a given moment in time.

Example 2.4 – Conditional Desk Check

Problem

Desk check the following code:

```
int computeTax(double income)
{
   // 10%, 15%, 25%, 28%, and 33% brackets here.
   if (income < 0.00)
      return 0;
   else if (income <= 15100.00)
      return 10;
   else if (income <= 61300.00)
      return 15;
   else if (income <= 123700.00)
      return 25;
   else if (income <= 188450.00)
      return 28;
   else if (income <= 336550.00)
      return 33;

   return 35;
}
```

Solution

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
int computeTax(double income)
{
   if (income < 0.00)                // (1)
      return 0;                      // (2)
   else if (income <= 15100.00)      // (3)
      return 10;                     // (4)
   else if (income <= 61300.00)      // (5)
      return 15;                     // (6)
   else if (income <= 123700.00)     // (7)
      return 25;                     // (8)
   else if (income <= 188450.00)     // (9)
      return 28;                     // (10)
   else if (income <= 336550.00)     // (11)
      return 33;

   return 35;                        // (12)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution. We will start with income set to $50,000.00

Line	income	return
1	50000.0	
3	50000.0	
5	50000.0	
6	50000.0	15

Note how line (2) is not executed because the Boolean expression in line (1) evaluated to `false`. This means that we move to line (3). Since it too evaluated to `false`, we move on to line (5). Now since (`50000.0 <= 61300.00`) evaluates to true, we move on to line (6). Line (6) is a return statement, meaning it is the last line of the function we execute.

It is important to realize that line (2), (4), (7), (8), (9), (10), (11), and (12) are never executed.

Example 2.4 – Loop Desk Check

Problem

Desk check the following code:

```
{
   int i;
   int j = 0;

   for (i = 1; i < 10; i *= 2)
      j += i;

   cout << j << endl;
}
```

Solution

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
{
   int i;
   int j = 0;                     // (1) along with the preceding line

   for (i = 1; i < 10; i *= 2)    // (2)
      j += i;                     // (3)

   cout << j << endl;             // (4)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution.

Line	i	j
1	?	0
2	1	0
3	1	1
2	2	1
3	2	3
2	4	3
3	4	7
2	8	7
3	8	15
2	16	15
4	16	15

Observe how the desk-check is a record of the execution of the loop. You can always "look back in time" to see exactly what was happening at a given stage in execution. For example, the third execution of the loop (highlighted) occurred with `i` set to 4 and `j` set to 3. Since `i < 10` evaluated to `true` (because 4 is less than 10), the loop continued on to the body (step 3). From here, `i` remained unchanged, but `j` increased its value by 4. You can always read the current values of all the variables off the desk check table. As expected, the value of `j` jumped from 3 to 7 on this line of code

Challenge

Notice how the FOR loop has three components: the Initialization, the Boolean expression, and the Increment. As a challenge, create a desk check where step 2 is split into these three components:

2a: Initialization

2b: Boolean expression

2.c: Increment

Example 2.4 – Online Desk Check

Problem

Modify the following code to perform an online desk check.

```
{
   int i;
   int j = 0;

   for (i = 1; i < 10; i *= 2)
      j += i;

   cout << j << endl;
}
```

Solution

The first step is to insert a COUT statement labeling the variables that will be displayed in the various columns. When this is finished, COUT statements are inserted where the line numbers would be on the paper desk check. The resulting code is:

```
{
   int i = 99; // some value
   int j = 0;
   cout << "\ti\tj\n";                           // row header of Desk Check table

   cout << "1\t" << i << "\t" << j << endl;     // (1)

   for (i = 1; i < 10; i *= 2)
   {
      cout << "2\t" << i << "\t" << j << endl;  // (2)
      j += i;
      cout << "3\t" << i << "\t" << j << endl;  // (3)
   }

   cout << "4\t" << i << "\t" << j << endl;     // (4)
}
```

The output of this modified code should appear "very similar" to the desk check performed by hand:

```
        i       j
1       99      0
2       1       0
3       1       1
2       2       1
3       2       3
2       4       3
3       4       7
2       8       7
3       8       15
4       16      15
```

Observe how this table is the same as the paper desk check output.

Challenge

As a challenge, perform a paper and online desk check on the `computeTax()` function from Project 1. What kind of bugs would it help you find?

See Also

The complete solution is available at 2-4-deskCheck.cpp or:

```
/home/cs124/examples/2-4-deskCheck.cpp
```

Pitfalls

As with the pitfalls associated with IF statements, a few pitfalls are common among loops.

Pitfall: = instead of ==

Remember that '=' means assign, and '==' means compare. We almost always want '==' in loops:

```
{
   bool done = false;

   do
   {
      …
      if (x == 0)
         done = true;
   }
   while (done = false);  // PITFALL!  We probably want to compare done with false!
}
```

Pitfall: < instead of <=

Pay special attention to the problem you are trying to solve. Some loops require us to add "the numbers less than 100." This implies `count < 100`. Other loops require us to count "from 1 to 10." This implies `count <= 10`. This class of errors is called "off-by-one" errors:

```
{
   // count from 1 to 10
   for (int count = 1;
        count < 10;      // PITFALL!  The comment says 1 to 10 implying count <= 10
        count++)
      ;
}
```

Pitfall: Extra semicolon

The entire loop statement includes both the loop itself and the body. This means we do <u>not</u> put a semicolon on the FOR loop itself. If we do so, we are implying that there is no body of the loop (just like an IF statement):

```
{
   // count from 1 to 10
   for (int i = 1; i <= 10; i++); // PITFALL!  This signifies that there is no body
      cout << i << endl;          //       so this statement isn't part of the loop
}
```

Pitfall: Infinite loop

Please make sure that your loop will end eventually:

```
{
   // count from 1 to 10
   for (int i = 1; i > 0; i++)    // PITFALL!  I will always be greater than 0!
      cout << i << endl;
}
```

Problem 1

Which of the following is the definition of the conceptual level of understanding of an algorithm?

- What the program does, not how the solution is achieved
- What the components do and how they influence the program
- The value of every variable at every stage of execution
- Realization where the flaws or bugs are

Please see page 168 for a hint.

Problem 2

Where do you put the line numbers in a desk check table?

Answer:

____

Please see page 171 for a hint.

Problem 3

Given a program that converts feet to meters, create a desk check table for the input value of 2 feet.

```
convert
   PROMPT for feet
   GET feet
   SET meters = feet * 0.301
   PUT meters
END
```


Please see page 171 for a hint.

Problem 4

Desk check the following program.

```
addNumbers
   SET number = 1
   DOWHILE number ≤ 5
      SET number = number + number
   ENDDO
END
```


Please see page 171 for a hint.

Problem 5

What is the output?

```
{
   int i;
   for (i = 0; i < 4; i++)
      ;
   cout << "i == " << i;
}
```

Answer:

___

Please see page 161 for a hint.

Problem 6

What is the output?

```
{
   bool done = false;
   int n = 5;

   while (!done)
   {
      if (n = 2)
         done = true;
      n--;
   }
   cout << "n == " << n << endl;
}
```

Answer:

___

Please see page 157 for a hint.

Problem 7

What is the output for the input of 'a' and 'x'?

```
{
   char input;

   do
   {
      cout << "input: ";
      cin  >> input;

      cout << "\t"
           << input
           << endl;
   }
   while (input != 'x');
}
```

Answer:

___

Please see page 157 for a hint.

Problem 8

What is the output?

```
{
   int i;

   for (i = 0; i < 4; i++);
      cout << "H";
   cout << endl;
}
```

Answer:

____

Please see page 173 for a hint.

Problem 9

What is the output?

```
{
   int i;

   for (i = 0; i <= 4; i++)
      ;

   cout << "i == " << i << endl;
}
```

Answer:

____

Please see page 157 for a hint.

Problem 10

What is the output?

```
{
   int sum = 0;
   int count;

   for (count = 0;
        count < 4;
        count++)
      sum += count;

   cout << "sum == " << sum;
}
```

Answer:

____

Please see page 161 for a hint.

Problem 11

What is the output?

```
{
   bool done   = false;
   int  number = 1;

   while (!done)
   {
      cout << number << endl;
      number *= 2;

      if (number > 4)
         done = true;
   }
}
```

Answer:

____

Please see page 171 for a hint.

Problem 12

What is the output?

```
{
   int sum = 0;
   int count;

   for ( count = 1;
         count < 9;
         count *= 2)
      sum += count;

   cout << "sum == " << sum;
}
```

Answer:

____

Please see page 171 for a hint.

Problem 13

What is the output?

```
{
   int count = 0;

   while (count < 5)
      count++;

   cout << "count == " << count;
}
```

Answer:

____

Please see page 157 for a hint.

Problem 14

What is the output?

```
{
   int count = 10;

   while (count < 5)
      count++;

   cout << "count == " << count;
}
```

Answer:

____

Please see page 157 for a hint.

Problem 15

What is the output?

```
{
   int count = 0;

   do
      count++;
   while (count < 3);

   cout << "count == " << count;
}
```

Answer:

____

Please see page 159 for a hint.

Assignment 2.4

Please create a desk check for the following programs. There should be three tables, one for each of the three problems. Start by annotating the code with line numbers. Each will be turned in by hand at the beginning of class. Please print this sheet out and put your name on it:

Problem 1: Convert grade

```
{
   int numGrade = 70;
   char letter = 'F';

   if (numGrade >= 80)
   {
      if (numGrade >= 90)
         letter = 'A';
      else
         letter = 'B';
   }
   else
   {
      if (numGrade >= 70)
         letter = 'C';
      else
         letter = 'D';
   }
}
```


Problem 2: Prompt

The user input is 2, 0, 10 in the following code:

```
{
   int numCookies = 0;

   while (numCookies < 4)
   {
      cout << "Daddy, how many cookies "
              "can I have? ";
      cin  >> numCookies;
   }

   cout << "Thank you daddie!\n";
}
```


Continued on next page

Continued from previous page

Problem 3: Counter

```
{
   int iUp = 0;
   int iDown = 10;

   while (iUp < iDown)
   {
      cout << iUp << '\t'
           << iDown << endl;
      iUp++;
      iDown--;
   }
}
```

Please see page 171 for a hint.

2.5 Loop Design

Sam is upset because he is trying to find a copy of the ASCII Table containing both hexadecimal (base 16) values and decimal (base 10) values. None of his favorite C++ web sites have the right table and Google is turning out to be useless. Rather than stooping to ask the professor, Sam decides to write his own code to display the table. But how to start? What will the main loop look like? (For an example solution of this problem, please see `/home/cs124/examples/2-5-asciiTable.cpp`).

Objectives

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

- Recognize the three main types of loops.
- Use a loop to solve a complex problem.

Prerequisites

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop (Chapter 2.3).
- Create a loop to solve a simple problem (Chapter 2.3).

Three Types of Loops

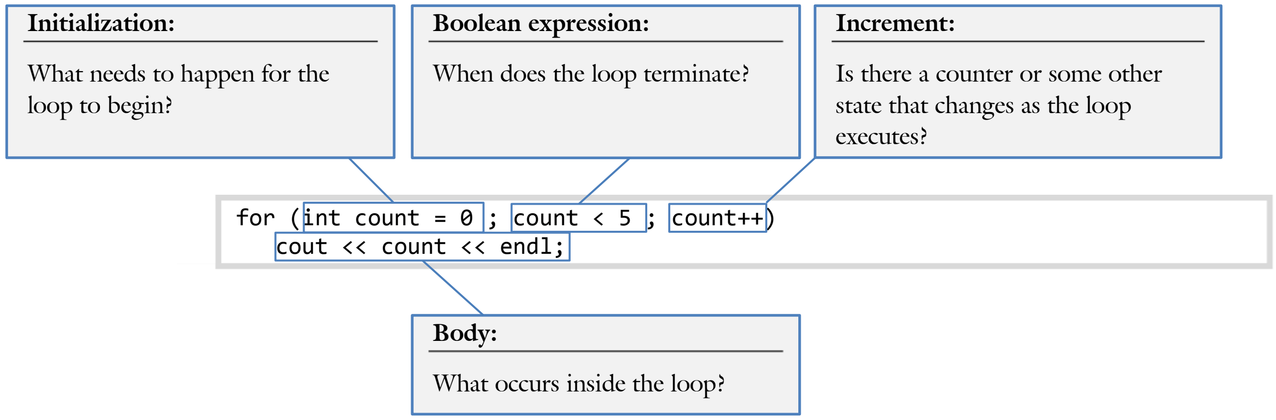
It takes a bit of skill to think in terms of loops. The purpose of this chapter is to give you design practice working with this difficult construct. Almost all looping problems can be broken into one of three categories: counter-controlled, event-controlled, and sentinel controlled. It is usually a good idea to identify which of the three types your problem calls for and design accordingly.

- **Counter Controlled**: Keep iterating a fixed number of times. The test is typically a single integer that changes with each iteration.
- **Event Controlled**: Keep iterating until a given event occurs. The number of iterations is typically unknown until the program runs.
- **Sentinel Controlled**: Keep iterating until a marker (called a sentinel) indicates the loop is done. The test is a given variable (typically a bool) that is set by any one of a number of events.

Counter-Controlled

A counter-controlled loop is a loop executing a fixed number of times. Typically that number is known before the loop starts execution. If, for example, I were to determine the number of students who had completed the homework assignment, a counter-controlled loop would be the right tool for the job.

In almost all circumstances a `for` statement is used for counter-controlled loops because `for` statement are designed for counting. Observe how the four parts to a FOR loop correspond to the four parts of the typical counting problem:


Initialization:
What needs to happen for the loop to begin?
Boolean expression:
When does the loop terminate?
Increment:
Is there a counter or some other state that changes as the loop executes?
Body:
What occurs inside the loop?

```
for (int count = 0; count < 5; count++)
   cout << count << endl;
```

Counter-controlled loops are readily identified by the presence of a single variable that moves through a range of values. In other words, counter-controlled loops do not exclusively increment by one: they might increment by 10 or powers of 3.

When designing with a counter-controlled loop, it is usually helpful to answer the following four questions:

- **How does the loop start**? In other words, what is the beginning of the range of values? This corresponds to the initialization part of the FOR loop.
- **How does the loop end**? In other words, by what condition do you know that you have yet to reach the end of the range of values? This corresponds to the Test or controlling Boolean expression part of the FOR loop.
- **What do you count by**? In other words, as you move through the range of values, how does the variable change? This corresponds to the Update part of the FOR loop.
- **What happens each iteration**? In other words, is some action taken at each step of the counting? Frequently no action is taken so this step can be skipped. If there is an action, then it goes in the Body section of the FOR loop.

Example 2.5 – Counter-Controlled Loop

Demo

This example will demonstrate how to design a counter-controlled loop.

Problem

Sue heard the following story in class one day:

> When Carl Friedrich Gauss was 6, his schoolmaster, who wanted some peace and quiet, asked the class to add up the numbers 1 to 100. "Class," he said, coughing slightly, "I'm going to ask you to perform a prodigious feat of arithmetic. I'd like you all to add up all the numbers from 1 to 100, without making any errors." "You!" he shouted, pointing at little Gauss, "How would you estimate your chances of succeeding at this task?" "Fifty-fifty, sir," stammered little Gauss, "no more..."

What the schoolmaster did not realize was that young Gauss figured out that adding the numbers from 1 to n is the same as: $sum = (n + 1)n / 2$. Sue wants to write a program to verify this. Her program will both add the numbers one by one, and use Gauss' equation.

Solution

With counter-controlled loops, four questions need to be answered:

- **Initialization**: The loop starts at 1: `int count = 1`
- **End**: The loop stops after the number is reached: `count <= n`
- **Update**: The loop counts by 1's: `count++`
- **Body**: Every iteration we increase the sum by count: `sum += count`

With these four answers, we can write the function.

```
int computeLoop(int n)
{
   int sum = 0;
   for (int count = 1; count <= n; count++)
      sum += count;

   return sum;
}
```

Challenge

As a challenge, find values where Gauss's equation does not work. Can you modify the program so it works with all integer values?

See Also

The complete solution is available at 2-5-counterControlled.cpp or:

```
/home/cs124/examples/2-5-counterControlled.cpp
```

Event-Controlled

An event-controlled loop is a loop that continues until a given event occurs. The number of repetitions is typically not known before the program starts. For example, if I were to write a program to prompt the user for her age but would re-prompt if the age was negative, an event-controlled loop is probably the right tool for the job. Typically event-controlled loops use `while` or `do-while` statements, depending if the loop needs to execute at least once.

Boolean expression:

How do you know when you are done? Event controlled loops keep going until an event has occurred. In this case, until the GPA is within an acceptable range.

```
while (gpa > 4.0 || gpa < 0.0)
{
   cout << "Enter your GPA: ";
   cin  >> gpa;
}
```

Body:

What changes every iteration? Typically something happens during event-controlled loops. This may be a re-prompt or a value is updated some other way. The programmer cannot predict how many iterations will be required; it depends on execution.

When designing with an event-controlled loop, it is usually helpful to answer the following two questions:

- **How do you know when you are done**? In other words, what is the termination condition? This condition maps directly to the controlling Boolean expression of a WHILE or DO-WHILE loop.
- **What changes every iteration**? Unlike counter-controlled loops where the counter changes in a predictable way, event controlled loops are typically driven by an external event. Usually there needs to be code in an event-controlled loop to re-query this external event. Often this comes in the form of re-prompting the user for input or reading more data from a file.

Sam's Corner

Notice how the two parts to an event-controlled loop (ending condition and what changes every iteration) look a lot like the two of the four parts of a counter-controlled loop (start, ending condition, counting, and what happens every iteration). This is because an event-controlled loop is a sub-set of a counter-controlled loop. If the counter-controlled loop does not have a starting condition or a counting component, then it probably is an event-controlled loop.

All FOR loops can be converted into a WHILE loop.

```
for ( i = 0;     // initialize
      i < 10;    // condition
      i++)       // increment
   cout << i << endl;
```

```
i = 0;             // initialize
while (i < 10)     // condition
{
   cout << i << endl;
   i++;            // increment
}
```

All WHILE loops can be converted into a FOR loop.

```
while (grade < 70.0)
   grade = takeClassAgain();
```

```
for (; grade < 70.0; )
   grade = takeClassAgain();
```

Example 2.5 – Event-Controlled Loop

Demo

This example will demonstrate how to design an event-controlled loop.

Problem

Write a program to keep prompting the user for his GPA until a valid number is entered.

```
Please enter your GPA (0.0 <= gpa <= 4.0): 4.1
Please enter your GPA (0.0 <= gpa <= 4.0): -0.1
Please enter your GPA (0.0 <= gpa <= 4.0): 3.9
GPA: 3.9
```

Solution

With event-controlled loops, two questions need to be answered:

- **End condition**: The loop ends when the condition is reached 0 ≤ gpa ≤ 4.0.
- **Update**: Every iteration, we re-prompt the user for the GPA.

With these two answers, we can write our pseudocode:

```
WHILE gpa > 4.0 or gpa < 0.0
   PROMPT for gpa
   GET gpa
```

With this pseudocode, it is straight-forward to write the function:

```
float getGPA()
{
   float gpa = -1.0;  // any value outside the expected range will do

   // loop until a valid value is received
   while (gpa > 4.0 || gpa < 0.0)
   {
      cout << "Please enter your GPA (0.0 <= gpa <= 4.0): ";
      cin  >> gpa;
   }

   // return with the loot
   assert(gpa <= 4.0 && gpa >= 0.0);  // paranoia will destroy-ya
   return gpa;
}
```

Observe how much more complex the C++ for `getGPA()` is than the pseudocode. This is because variable initialization, comments, the text of the prompt, and asserts are not required for pseudocode

Challenge

As a challenge, try to change the above function from a WHILE loop to a DO-WHILE loop. Both loops are commonly event-controlled.

See Also

The complete solution is available at 2-5-eventControlled.cpp or:

```
/home/cs124/examples/2-5-eventControlled.cpp
```

Sentinel-Controlled

A sentinel is a guardian or gatekeeper. In the case of a sentinel-controlled loop, the sentinel is a variable used to determine if a loop is to continue executing or if it is to terminate. We typically use a sentinel when multiple events could cause the loop to terminate. In these cases, the sentinel could be changed by any of the events. Typically event-controlled loops use `while` or `do-while` statements where the sentinel is a `bool` used as the condition. Perhaps, this is best explained by example.

Example 2.5 – Sentinel-Controlled Loop

Demo

This example will demonstrate how to design a sentinel-controlled loop.

Problem

Consider a professor trying to determine if a student has passed his class. There are many criteria to be taken into account (the grade and whether he cheated, to name a few). Rather than making a single highly-complex controlling Boolean expression, he decides to use a sentinel-controlled loop:

```
Welcome to CS 124!
What is your class grade? 109
Did you cheat in the class? (y/n) y

Welcome to CS 124!
What is your class grade? 81
Did you cheat in the class? (y/n) n
Great job!  Get ready for CS 165
```

Solution

The solution is to have one variable (passed) be set by a variety of conditions.

```
{
   bool passed = false;                          // the sentinel. Initially we have
                                                 //       not passed the class
   // the main loop
   while (!passed)                               // common sentinel, read
   {                                             //       "while not passed..."
      cout << "\nWelcome to CS 124!\n";

      // if you got a C or better, you may have passed...
      float grade;
      cout << "What is your class grade? ";
      cin  >> grade;
      if (grade >= 60.0)
         passed = true;                          // one of the ways the
                                                 //      sentinel may change
      // if you cheated, you did not pass
      char cheated;
      cout << "Did you cheat in the class? (y/n) ";
      cin >> cheated;
      if (cheated == 'y' || cheated == 'Y')
         passed = false;                         // another sentinel condition
   }
   cout << "Great job!  Get ready for CS 165\n";
}
```

See Also

The complete solution is available at 2-5-sentinelControlled.cpp or:

```
/home/cs124/examples/2-5-sentinelControlled.cpp
```

Problem 1

Write a program to put the alphabet on the screen. A.K.A. Sesame Street Karaoke.

```
A
B
C
…
Y
Z
```

Answer:

Please see page 182 for a hint.

Problem 2

Write a program to display the multiplication table of numbers less than 6.

```
1    2    3    4    5
2    4    6    8    10
3    6    9    12   15
4    8    12   16   20
5    10   15   20   25
```

Answer:

Please see page 182 for a hint.

Problem 3

Write a program to compute how many numbers under a user-specified value are both odd and a multiple of 5.

```
What is the number: 20
The number of values under 20 that are both odd and a multiple of 5 are: 2
```

Answer:

Please see page 182 for a hint.

Problem 4

Write a function to compute whether a number is prime:

```
bool isPrime(int number);
```

Answer:

Please see page 49 for a hint.

Assignment 2.5

Write a function (`displayTable()`) to display a calendar on the screen. The function will take two parameters:

- `numDays`: The number of days in a month.
- `offset`: The offset from Monday. If the offset is zero, then the month starts on Monday. If the offset is 2, the month starts on Wednesday. If the offset is 6, the month starts on Sunday.

This function will be "very similar" to the `displayTable()` function in Project 2. Please see the project for details on the spacing between the columns and a hint on how the algorithm might work. Next write `main()` so that it prompts the user for the number of days in the month and the offset.

Note that this is probably the most difficult assignment of the semester. Of course, this will also get you very far along on the project as well. Please, allocate a couple hours for this assignment.

Example

Three examples. The user input is **underlined**.

```
Number of days: 30
Offset: 3
  Su  Mo  Tu  We  Th  Fr  Sa
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30
```

```
Number of days: 28
Offset: 0
  Su  Mo  Tu  We  Th  Fr  Sa
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28
```

```
Number of days: 31
Offset: 6
  Su  Mo  Tu  We  Th  Fr  Sa
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
  22  23  24  25  26  27  28
  29  30  31
```

Assignment

The test bed is available at:

```
testBed cs124/assign25 assignment25.cpp
```

Don't forget to submit your assignment with the name "Assignment 25" in the header,

Please see page 182 for a hint.

2.6 Files

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. This is done by opening the file with the list of Euro prices, performing the conversion to US dollars, and writing the resulting values to another file.

Objectives

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

- Write the code to read data from a file.
- Write the code to write data to a file.
- Perform error checking on file streams.
- Understand the different ways the end-of-file marker can be found.

Prerequisites

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

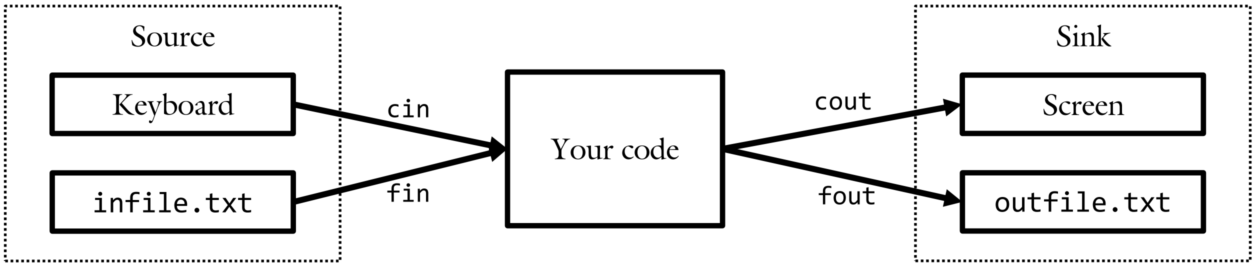
- Create a loop to solve a complex problem (Chapter 2.5).

Overview

After a program ends, all memory of its execution is removed. This fact is particularly unsatisfactory if the program was charged with maintaining the user's valuable data. To overcome this shortcoming, it is necessary to save to and retrieve data from a file.

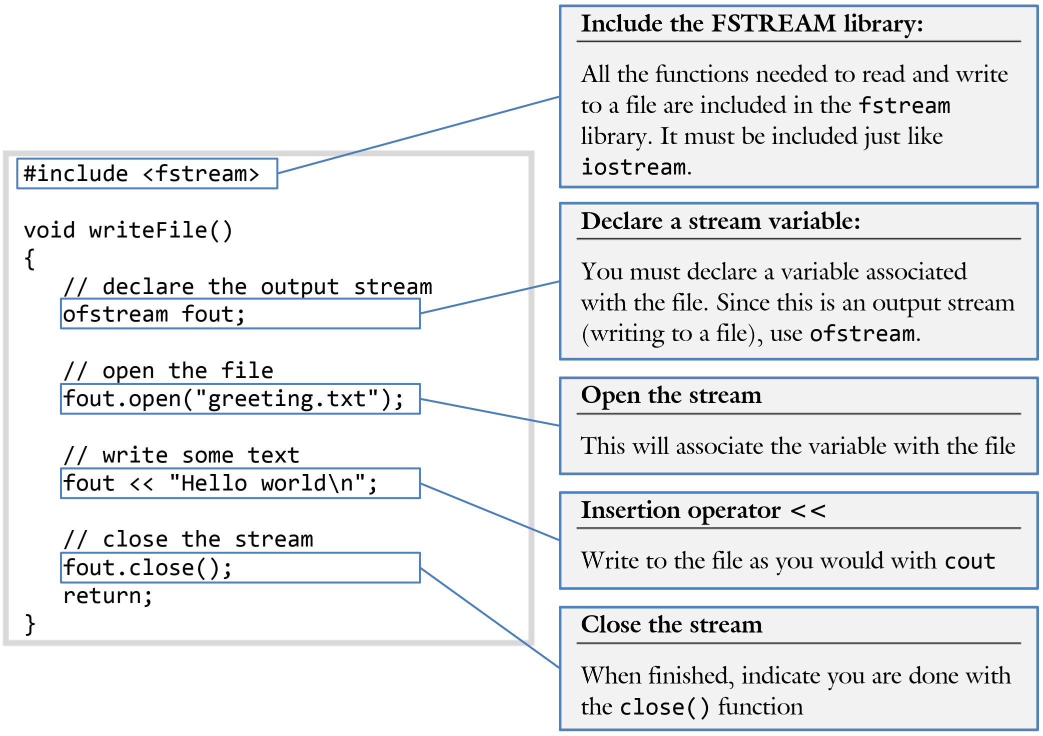
In many ways, writing data to a file is similar to writing data to the screen. In the file case, however, one needs to specify the target file instead of just using `cout`. In other words, `cout << number << endl;` would put the value of the variable `number` on the screen. If, on the other hand, the variable `fout` corresponded to a file, one could put the value of `number` in the file with `fout << number << endl;`.

Similarly, reading data from a file is similar to accepting user input from a keyboard. Here again, the programmer needs to specify the name of the source file. There is one additional difference, however. When reading data from the keyboard, the programmer can assume there is an infinite amount of data on hand (assuming an infinitely patient user!). Files, on the other hand, are of finite length. At some point, the end will be reached and the program needs to be ready to handle that event.



Writing to a File

When we write text to the screen, we use `cout`. This variable is defined in the `iostream` library and keeps track of where the cursor is on the screen. In other words, as we continue to send data to the screen, the cursor keeps moving to the right or down much the same way a typewriter advances. Sending data to a file is conceptually the same; we need a variable to keep track of the location of the cursor in the file so, as more data is sent to the file, the cursor advances. It follows that we would need a variable very similar to `cout` to do this. However, there is one key difference between writing to a file and writing to the screen: there is only one screen to write to while there may be many files. Therefore, it is necessary to also specify *which* file we are writing to. Consider the following code:



```
#include <fstream>

void writeFile()
{
   // declare the output stream
   ofstream fout;

   // open the file
   fout.open("greeting.txt");

   // write some text
   fout << "Hello world\n";

   // close the stream
   fout.close();
   return;
}
```

Include the FSTREAM library:
All the functions needed to read and write to a file are included in the `fstream` library. It must be included just like `iostream`.

Declare a stream variable:
You must declare a variable associated with the file. Since this is an output stream (writing to a file), use `ofstream`.

Open the stream
This will associate the variable with the file

Insertion operator <<
Write to the file as you would with `cout`

Close the stream
When finished, indicate you are done with the `close()` function

Simple file writing, in other words, consists of several components: the `fstream` library, declaring a stream variable (`fout`), opening the file, streaming data to the file, and closing it when finished.

Sam's Corner

We have previously mentioned that `cout` is a variable. While this is true, it is a special type of variable: an object. An object is a variable that contains both data (position of the cursor on the screen) and functions (think `cout.setw(5)` and `cout.getline(text, 256)`) associated with the data. Objects and the classes that define them are subjects of Object Oriented programming, the topic of CS 165. Don't worry about objects now; this is a topic for next semester.

FSTREAM library

When we were writing programs to display text on the screen, we needed to use the `iostream` library. This library defined two variables (`cout` and `cin`) and the functions associated with them. When writing to a file, we use the `fstream` library. This library contains two data types: `ofstream` and `ifstream`. OFSTREAM stands for <u>O</u>utput <u>F</u>ile <u>STREAM</u> used to send data out of a program and into a file. This is done with:

```
#include <fstream>  // need this line every time a file is used in the program
```

Declaring a stream variable

The next step is to declare a variable that will be used to send data to the file. Note that you may use any variable name you like (as long as it conforms to the Elements of Style guidelines). As a rule, you might want to use `fout` for a stream variable name because it looks and feels like "`cout`." The only difference is that `F` stands for File while `C` stands for Console (or screen). This is done with:

```
ofstream fout;       // declare an output file stream variable.
```

It is possible to have more than one output file stream active at once. A large and advanced program might, for example, write one file for user data, another for a log, and a third to save configuration data. This is not a problem; just have three `ofstream` variables:

```
{
   ofstream foutData;    // For user data
   ofstream foutLog;     // For a log
   ofstream foutConfig;  // For config. data. All 3 can be used at the same time
}
```

Opening a file

After a stream variable is declared, the next step is to connect that variable to a given file. This can be done with a "hard-coded" string (a string literal)…

```
   fout.open("data.txt");       // Always use the same file. We seldom do this.
```

… or it could be done with a variable:

```
{
   char fileName[256];          // string variable to hold the name of the file
   cin >> fileName;             // prompt user for the file name
   fout.open(fileName);         // open the file by referencing the variable.
}
```

It is also possible to both declare a stream variable and associate it with a file in one step:

```
{
   ofstream fout;
   fout.open(fileName);
}
```

```
{
   ofstream fout(fileName);

}
```

Both of the above lines of code do exactly the same thing; it is a matter of convenience which you choose. The final thing to note about opening a file is that, on occasion, there is no file to open. In other words, what would happen when the user attempts to write to a directory that does not exist, to a thumb drive that is full, or to a file where the user lacks the required permission? In each of these cases, an error message will need to be presented to the user. Thus, it is absolutely necessary to check for the error condition and quit the file writing process. Consider the following function:

```
/**********************************************************
 * Write GPA
 * This function will write the user's GPA to a file
 **********************************************************/
bool writeGPA(char fileName[], float gpa)
{
   // declare and open the stream
   ofstream fout(fileName);                // declare and initialize in one step
   if (fout.fail())                        // check for error with fail()
      return false;                        // indicate to caller that we failed!

   // write the data
   fout << gpa << endl;                    // actually do the work

   // quite
   fout.close();                           // don't forget to close when done
   return true;                            // report success to the caller
}
```

This function takes the filename as a parameter. To call the function, it is nessary to specify a string as the first parameter.

```
{
   writeGPA("myGrade.txt", 3.9);      // first parameter must be a string
}
```

Unit 2

Note how we check for error with the `fail()` function. If we are unable to open the file for writing for any reason (permissions, lack of space, general hardware error, etc.), then `fail()` will return `true`. This will mean that the function `writeGPA()` will be unable to do what it was asked to do: write to a file. We therefore commonly make file functions return Boolean values: `true` corresponds to success and `false` corresponds to failure.

Sam's Corner

By default, OFSTREAM will replace any file of the same name that is being written. This might be what the programmer intended if there is no other file or if data is to be updated. However, it is often necessary to append data onto the end of a file rather than replace it. This can be done by adding another parameter to the output stream declaration:

```
ofstream fout(fileName, ios::app);
```

In this case, the `ios::app` means to append the file rather than overwrite. Other modes include.

Mode	Meaning
`ios::app`	Append output to end of file (EOF)
`ios::ate`	Seek to EOF when the file is opened
`ios::binary`	Open file in binary mode
`ios::in`	Open file for reading. This happens automatically for `ifstream`
`ios::out`	Open file for writing. This happens automatically for `ofstream`
`ios::trunc`	Overwrite existing file instead of truncating. Default for `ofstream`

Streaming data to a file

We use `fout` to send data to a file in exactly the same way we use `cout` to send data to the screen. This means that all tools we had for screen display we also have for file writing. Consider the following code:

```
{
   // configure FOUT for displaying money, just like COUT
   fout.setf(ios::fixed);
   fout.setf(ios::showpoint);
   fout.precision(2);

   // display my budget
   fout << "\t$" << setw(9) << income   << endl;
   fout << "\t$" << setw(9) << spending << endl;
   fout << "\t ---------\n";
   fout << "\t$" << setw(9) << income - spending << endl;
}
```

The above code might be very familiar if `cout` were used instead of `fout`. The only real difference is that this data is sent to a file rather than the screen

Sue's Tips

While it is easy to verify the screen output of a program, it is often inconvenient to verify the file output. As a result, beginner programmers tend to forget details such as putting spaces between numbers. One easy way to work around this tendency is to first write your `writeFile()` function with `cout`s. After you have run the program a few times and you are sure of the formatting, turn your `cout`s in to `fout`s to send the same data to the file.

Closing the file

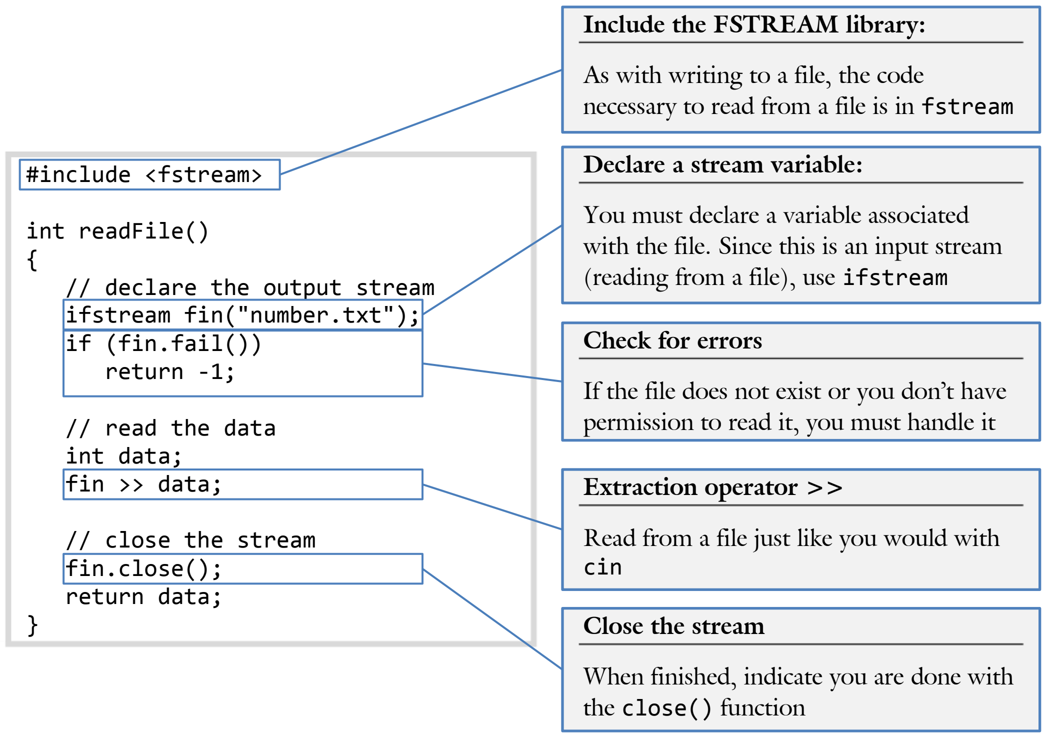
When we are finished writing data to a file, it is important to remember to close the file. On primitive operating systems (think MS-DOS), an un-closed file could never be reopened. Modern operating systems, however, will handle this step for you if you forget. However, it is "good form" to close a file as soon as the last data has been written to it:

```
fout.close();
```

Reading from a File

Just as writing text to a file with `fout` is similar to writing text to the screen with `cout`, reading text from a file has a `cin` equivalent: `fin`. There is, however, one important difference between reading text from the keyboard and reading text from a file. Eventually the end of the file will be reached. It is therefore necessary to make sure logic exists in the program to handle the unexpected end-of-file condition.

As with writing data to a file, several steps are involved: using the FSTREAM library (`#include <fstream>`), declaring the input file stream variable (`ifstream fin;`), checking for errors (`fin.fail()`), using the extraction operator (`fin >> data;`), and closing the file.



```
#include <fstream>

int readFile()
{
   // declare the output stream
   ifstream fin("number.txt");
   if (fin.fail())
      return -1;

   // read the data
   int data;
   fin >> data;

   // close the stream
   fin.close();
   return data;
}
```

Include the FSTREAM library:

As with writing to a file, the code necessary to read from a file is in `fstream`

Declare a stream variable:

You must declare a variable associated with the file. Since this is an input stream (reading from a file), use `ifstream`

Check for errors

If the file does not exist or you don't have permission to read it, you must handle it

Extraction operator >>

Read from a file just like you would with `cin`

Close the stream

When finished, indicate you are done with the `close()` function

FSTREAM library

As with writing to a file, it is necessary to remember to include the FSTREAM library. If this step is skipped, one can expect the following compile error:

```
example.cpp: In function "int readFile()":
example.cpp:6: error: aggregate "std::ifstream fin" has incomplete type and cannot be
defined
```

This cryptic compiler error means that `std::ifstream` is an unknown type. The reason, of course, is that IFSTREAM is defined in `fstream`. Therefore, don't forget:

```
#include <fstream>
```

Declaring a stream variable

Input stream variables are defined in much the same way as output stream variables. The most important difference, of course, is we use `ifstream` for Input File STREAM. Also, like output streams, we can declare and initialize the variable in a single line.

```
{
   ifstream fin;
   fin.open(fileName);
}
```

```
{
   ifstream fin(fileName);

}
```

Again, by convention, it is common to use `fin` for the variable name to emphasize the relationship with `cin`.

Check for errors

As with writing to a file, an essential part of reading from a file includes checking for errors. The same class of errors for writing to a file exists when reading from a file (no permissions, missing directory, general file-system error, etc.). Additionally, the potential exists that there might not be any data in the file to read. In all these cases, we can detect if an error occurred with the `fin.fail()` function call.

```
{
   ifstream fin(fileName);                // attempt to open the file
   if (fin.fail())                        // check for any type of error
   {
      cout << "Unable to open file "      // let the user know!
           << fileName << endl;           // he probably wants to know the file name
      return false;                       // report failure to the user
   }
}
```

Read the data

We write (to the screen or to a file) using the **insertion operator** (<<). Similarly, all read operations are done with the **extraction operator** (>>).

```
{
   float temperature;                 // first item is expected to be a number
   char  units[256];                  // next item is expected to be text
   fin >> temperature >> units;       // read both just like with cin
}
```

There are two ways we can tell if the read failed for any reason. The first is to check for a read failure. This can be accomplished with another `fin.fail()` function call. The second is to see if the extraction operator itself failed. The following two lines of code are equivalent:

```
{
   int value;
   fin >> value;
   if (!fin.fail())
      cout << "Success!\n";
}
```

```
{
   int value;
   if (fin >> value)
      cout << "Success!\n";

}
```

In other words, the extraction operator (>>) is actually a function call returning `false` when it fails for any reason. One reason may be that the file has been corrupted (or even erased!) during the read. Another may be that there is no more data in the file. Another way to state this last reason is that the "end-of-file" condition may have been met.

Reading to the end of the file

At the end of every file in a file system is a special marker indicating that there is no more data in the file. This can be thought of as the "end of road" marker on a highway. We can ask the file stream if we are at the end of file (EOF) with a function call:

```
{
   if (fin.eof())                          // returns TRUE if we are at the end
      cout << "There is no more data!\n";
}
```

This means that there are two ways to read all the data from a file. The first is to continue looping until the EOF marker is reached. The second is to read until an error has occurred on the read:

EOF	Read Failure
IF the end of the file character is encountered, the EOF flag will be set. You can check for this at any time:	If a read failure occurs, the extraction operator will return `false`. This can be checked on any read.

```
{
   ifstream fin("file.txt");

   while (!fin.eof())
   {
      char text[256];
      fin  >> text;
      cout << text << endl;
   }
   fin.close();
}
```

```
{
   ifstream fin("file.txt");

   char text[256];
   while (fin >> text)
   {
      cout << text << endl;
   }

   fin.close();
}
```

These two methods are not the same. Consider the case when there is a word and a space in the file.

w	o	r	d	space	EOF

In the first case, we will read the word on the first loop and display the text on the screen. On the second iteration, we will go into the body of the loop (because we are not yet at the end of the file: there is still a space left!). When we attempt to read the next word with `fin >> text`, we fail (there is no non-space data in the file after all). In this case, we will not change the value of `text` so the word will be repeated on the screen.

In the second case, we will successfully read the word on the first iteration of the loop. This, of course, will be displayed on the screen in the body of the loop. On the second iteration, we will fail to read (there is no non-space data in the file) so the loop will exit. This means the last word will not be repeated on the screen.

For more details on the aforementioned differences between using the EOF method and the read-failure method of reading from a file, please see Example – End of File on the following page.

Closing the file

As with writing data to a file, it is important to always remember to close the file that was read:

```
fin.close();
```

Sam's Corner

With most operating systems, the failure of a program to close a file is not catastrophic; the operating system will quietly close it for you once the program exits. This is not true on some mobile platforms or older operating systems. An improperly closed file or a file that has not been closed will remain locked and forever unavailable for use. Thus, it is good form to always close a file as soon as reading or writing has been completed.

Filenames

There is one final complication that arises when working with files: the necessity of dealing with filenames. Filenames are c-strings, something we learned about in Chapter 1.2 (page 38) but have done very little with since. The reason for this is that handling c-strings is a bit quirky. We cannot return a c-string from a function as we would any other data-type. Instead, we need to pass it as a parameter.

When passing a c-string, or any other array (which we will learn about in Unit 3), it comes in as pass-by-reference even though we don't have use the '&' operator. Thus the correct way to write a function to prompt the user for a filename is:

```
/**********************************************************
 * GET FILENAME
 * Prompt the user for a filename.
 **********************************************************/
void getFilename(char fileName[])                    // the fileName parameter behaves
{                                                    //    like pass-by-reference
   cout << "What is the name of the file? ";
   cin  >> fileName;                                 // text entered here will be sent
}                                                    //    back to the caller
```

If there are some things about this function that you don't understand, don't worry! We will learn more about this on page 245.

Example 2.6 – End of File

Demo

This example will demonstrate how to read all the content of the file using two techniques: either using the EOF method or the Read Failure method.

Problem

Write a program to read all the text out of a file and display the results on the screen. Consider, for example, the following text in a file in `2-6-eof.txt`:

```
I love software development!
```

The output is:

```
Filename? 2-6-eof.txt
Use the EOF method? (y/n): y
'I' 'love' 'software' 'development!' 'development!'
```

Solution

The first solution is to use EOF method.

```
void usingEOF(const char filename[])
{
   // open
   ifstream fin(filename);
   if (fin.fail())
   {
      cout << "Unable to open file " << filename << endl;
      return;
   }

   // get the data and display on the screen
   char text[256];
   // keep reading as long as:
   //   1. not at the end of file
   while (!fin.eof())
   {
      // note that if this fails to read anything (such as when there
      // is nothing but a white space between the file pointer and the
      // end of the file), then text will keep the same value as the
      // previous execution
      fin >> text;
      cout << "'" << text << "' ";
   }
   cout << endl;

   // done
   fin.close();
}
```

The second solution uses the Read Failure method. Everything is the same except the loop:

```
   // keep reading as long as:
   //   1. not at the end of file
   //   2. did not fail to read text into our variable
   //   3. there is nothing else wrong with the file
   while (fin >> text)
      cout << "'" << text << "' ";
```

See Also

The complete solution is available at 2-6-eof.cpp or:

```
/home/cs124/examples/2-6-eof.cpp
```

Example 2.6 – Read Data

Demo

This example will demonstrate how to read a small amount of data from a file. This will include two data types (a string and an integer). All error checking will be performed.

Problem

Consider the following file (`2-6-readData.txt`):

```
Sue 19
```

Read the file and display the results on the screen.

```
What is the filename? 2-6-readData.txt
The user Sue is 19 years old
```

Solution

The man work is performed by the `read()` function, taking a filename as a parameter.

```
bool read(char fileName[])                      // filename we will read from
{
   // open the file for reading
   ifstream fin(fileName);                      // connect to fileName
   if (fin.fail())                              // never forget to check for errors
   {
      cout << "Unable to open file "            // tell the user what happened
           << fileName << endl;
      return false;                             // return and report
   }

   // do the work
   char userName[256];
   int  userAge;
   fin >> userName >> userAge;                  // get two pieces of data at once
   if (fin.fail())
   {
      cout << "Unable to read name and age from "
           << fileName << endl;
      return false;
   }

   // user-friendly display
   cout << "The user "                          // display the data
        << userName
        << " is "
        << userAge << " years old\n";

   // all done
   fin.close();                                 // don't forget to close the file
   return true;                                 // return and report
}
```

Challenge

As a challenge, can you change the above program to accommodate the user's GPA. This will mean that there are three items in the file:

```
Sue 19 3.95
```

See Also

The complete solution is available at 2-6-readData.cpp or:

```
/home/cs124/examples/2-6-readData.cpp
```

Example 2.6 – Read List

Demo

This example will demonstrate how to read large amounts of data from a file. In this case, the file consists of a list of numbers. The program does not know the size of the list at compile time.

Problem

Write a program to sum all the numbers in a file. The numbers are in the following format:

```
34
25
10
43
```

The program will prompt the user for the filename and display the sum:

```
What is the filename: 2-6-readList.txt
The sum is: 112
```

Solution

The function `sumData()` does all the work in this example. It is important to note that the program does not need to remember all the files read from the file. Once the value is added to the `sum` variable, then it can be ignored.

```
int sumData(char fileName[])
{
   // open the file
   ifstream fin(fileName);
   if (fin.fail())
      return 0;                   // some error message

   // read the data
   int data;                      // always need a variable to store the data
   int sum = 0;                   // don.t forget to initialize the variable
   while (fin >> data)            // read: "while there was not an error"
      sum += data;                // do the work

   // close the stream
   fin.close();
   return sum;
}
```

Challenge

See if you can modify the above program (and the file that it reads from) to work with floating point numbers.

See Also

The complete solution is available at 2-6-readList.cpp or:

```
/home/cs124/examples/2-6-readList.cpp
```

Example 2.6 – Round Trip

Demo

A common scenario is to save data to a file then read the data back again next time the program is run. We call this "round-trip" because the data is preserved through a write/read cycle. This program will demonstrate that process.

Problem

Consider a file consisting of a single floating point number:

```
30.36
```

Write a program to read the file, prompt the user for a value to add to the value, and write the updated value back to the file.

```
Account balance: $30.36
Change: $5.20
New balance: $35.56
```

Solution

First, the program will read the balance from a file. If no balance is found, then return 0.0:

```
float getBalance()
{
   // open the file
   ifstream fin(FILENAME);                  // the filename is constant because
   if (fin.fail())                          //     it needs to be same every time
      return 0.0;                           // if no file found, start at $0.00

   // read the data
   float value = 0.0;
   fin >> value;                            // read the old value
   if (fin.fail())                          // if we cannot read this value for any
      return 0.0;                           //     reason, return $0.00

   // close and return the data
   fin.close();
   return value;                            // send the value back to main()
}
```

Then, after the user has been prompted for a new value and the balance has been updated, the new balance is written to the same file.

```
void writeBalance(float balance)
{
   // open the file for writing
   ofstream fout(FILENAME);                  // make sure it is the same file as
   …                                         //    we used for getBalance()

   // write the data
   fout.precision(2);                        // format fout for money just like we
   fout.setf(ios::fixed);                    //    would do for cout.
   fout.setf(ios::showpoint);
   fout << balance << endl;                  // it is "good form" to end with endl

   …
}
```

See Also

The complete solution is available at 2-6-roundTrip.cpp or:

```
/home/cs124/examples/2-6-roundTrip.cpp
```

Problem 1

What is in the file "data.txt"?

```
void writeData(int n)
{
   ofstream fout;
   fout.open("data.txt");

   for (int i = 0; i < n; i++)
     fout << i * 2 << endl;

   fout.close();
   return;
}

int main()
{
   writeData(4);

   return 0;
}
```

Answer:

Please see page 193 for a hint.

Problem 2

Given the following function:

```
bool writeFile(char fileName[])
{
   ofstream fout;
   fout.open(fileName);

   fout << "Hello World!\n";

   fout.close();

   return true;
}
```

Which would **not** cause the program to fail?

```
writeFile(data.txt);
```

```
writeFile("");
```

```
writeFile(10);
```

```
writeFile("data.txt");
```

Please see page 193 for a hint.

Problem 3

Write a function to put the numbers 1-10 in a file:

- Call the function numbers()
- Pass the file name in as a parameter
- Do error checking

Answer:

Please see page 193 for a hint.

Problem 4

What is the best name for this function?

```
void mystery(char f1[], char f2[])
{
   ofstream fout;
   ifstream fin;

   fin.open(f1);
   fout.open(f2);

   char text[256];
   while (fin.getline(text, 256))
      fout << text << endl;

   fin.close();
   fout.close();

   return;
}
```

Answer:

____

Please see page 196 for a hint.

Problem 5

Given the following file (`grades.txt`) in the format `<name> <score>`:

```
Jack 83
John 97
Jill 56
Jake 82
Jane 99
```

Write a function to read the data and display the output on the screen. Name the function `read()`.

Please see page 196 for a hint.

Problem 6

Write a function to:

- Open a file and read a number. Display the number to the user.
- Prompt the user for a new number.
- Save that number to the same file and quit.

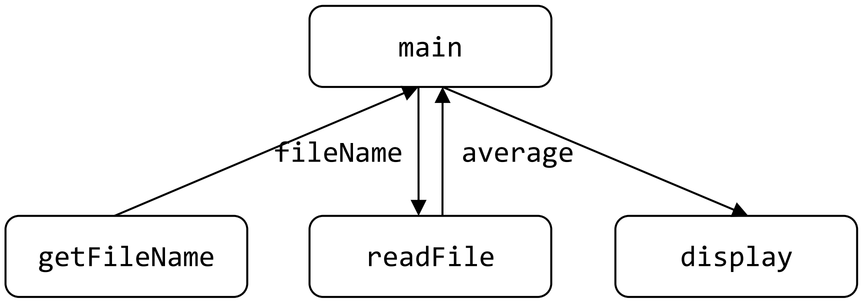
```
The old number is "42". What is the new number?  104
```

Answer:

Please see page 202 for a hint.

Assignment 2.6

Please write a program to read 10 grades from a file and display the average. This will include the functions `getFileName()`, `displayAverage()` and `readFile()`:



getFilename()

This function will prompt the user for the name of the file and return it. The prototype is:

```
void getFileName(char fileName[]);
```

Note that we don't return text the way we do integers or floats. Instead, we pass it as a parameter. We will learn more how this works in Section 3.

readFile()

This function will read the file and return the average score of the ten values. The prototype is:

```
float readFile(char fileName[]);
```

Hint: make sure you only read ten values. If there are more or less in the file, then the function must report an error. Display the following message if there is a problem with the file:

```
Error reading file "grades.txt"
```

display()

This function will display the average score to zero decimals of accuracy (rounded). The prototype is:

```
void display(float average);
```

Example

Consider a file called `grades.txt` (which you can create with emacs) that has the following data in it:

```
90 86 95 76 92 83 100 87 91 88
```

When the program is executed, then the following output is displayed:

```
Please enter the filename: grades.txt
Average Grade: 89%
```

Assignment

The test bed is available at:

```
testBed cs124/assign26 assignment26.cpp
```

Don't forget to submit your assignment with the name "Assignment 26" in the header.

Please see page 201 for a hint.

Unit 2 Practice Test

Practice 2.1

Mike's teacher told his class that a flipped coin lands on "heads" half the time and "tails" the other half. "That means that if I flip a coin 100 times, it should land on heads exactly fifty times!" said Mike. Mike's teacher knows that the law of probability does not quite work that way. To demonstrate this principle, she decides to write a program.

Program

Write a function to simulate a coin flip, returning true if the coin lands on "heads" and false if it lands on "tails." Next, prompt the user for how many trials will be needed for the experiment. Finally, display how many "heads" and "tails" where recorded in the experiment.

Example

User input is **underlined**.

```
How many coin flips for this experiment: 100
There were 49 heads.
There were 51 tails.
```

Assignment

Please:

- Start from the standard template we use for homework assignments:

```
/home/cs124/tests/templateTest2.cpp
```

- Make sure your professor's name is in the program header.
- Run test bed with

```
testBed cs124/practice21 test2.cpp
```

- Run style checker

Note that the following code might come in handy:

```
#include <stdlib.h>   // needed for the rand(), srand()
#include <ctime>      // needed for the time function
int main(int argc, char **argv)
{
   // this code is necessary to set up the random number generator. If
   // your program uses a random number generator, you will need this
   // code. Otherwise, you can safely delete it. Note: this must go in main()
   srand(argc == 1 ? time(NULL) : (int)argv[1][1]);

   // this code will actually generate a random number between 0 and 999
   cout << rand() % 1000 << endl;
}
```

Continued on the next page

Continued from previous page

Grading for Test2

Sample grading criteria:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
Expressions 10%	The expression for the equation is elegant and easy to verify	The expression correctly computes the equation	One bug exists	Two or more bugs exist	The expression is missing
Modularization 20%	Functional cohesion and loose coupling is used throughout	Zero syntax errors in the use of functions, but room exits for improvements in modularization	Data incorrectly passed between functions	At least one bug in the way a function is defined or called	All the code exists in one function
Loop 40%	The loop is both elegant and efficient	The loop is syntactically correct and used correctly	The loop is syntactically correct or is used correctly	Elements of the solution are present	No attempt was made to use a loop
Output 20%	Zero test bed errors	Looks the same on screen, but minor test bed errors	One major test bed error	The program compiles and elements of the solution exist	Program output does not resemble the problem or fails to compile
Style 10%	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

Solution available at:

```
/home/cs124/tests/practice21.cpp
```

Unit 2 Project : Calendar Program

Create a calendar for any month of any year from 1753 forward. This is similar to the `cal` utility on the Linux system. Prompt the user for the numeric month and year to be displayed as shown in the example below. Your calculations to determine the first day of the month shall start with January 1, 1753 as a Monday. The challenge here is to take into account leap years.

This project will be done in three phases:

- Project 05 : Design the calendar program
- Project 06 : Compute the offset for a given month and year
- Project 07 : Display the calendar table for a given month and year

Interface Design

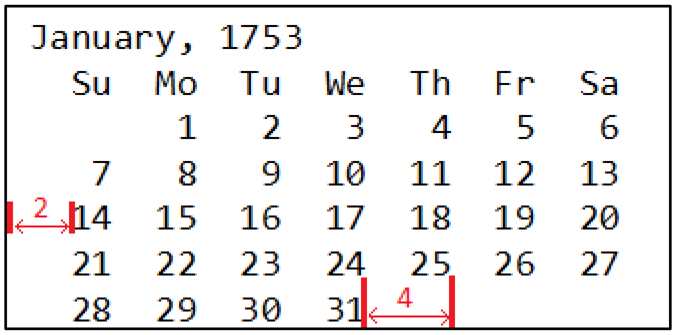
There are three parts of the interface design: the output (that which is displayed on the screen during normal operation), the input (that which the user provides), and the errors (when the user enters different data than the program expects).

Output

The following is a sample run of the program. The input is **underlined**.

```
Enter a month number: 1
Enter year: 1753

January, 1753
  Su  Mo  Tu  We  Th  Fr  Sa
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31
```

The table is formatted in the following way:



```
January, 1753
  Su  Mo  Tu  We  Th  Fr  Sa
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31
```

Input

First the user will be prompted for the month number:

```
Enter a month number: 1
```

Next the user will be prompted for the year:

```
Enter year: 1990
```

Errors

When the program prompts the user for a month, only the values 1 through 12 are accepted. Any other input will yield a re-prompt:

```
Enter a month number: 13
Month must be between 1 and 12.
Enter a month number: 0
Month must be between 1 and 12.
Enter a month number: 1
```

The same is true with years; the user input must be greater than 1752:

```
Enter year: 90
Year must be 1753 or later.
Enter year: 1990
```

Project 05

Write a structure chart for the calendar program. Make sure that all the functions exhibit the highest level of cohesion and the lowest level of coupling.

On campus students are required to attach this rubric to your design document. Please self-grade.

Project 06

The second part of the Calendar Program project (the first part being the structure part due earlier) is to write the pseudocode for two functions: computeOffset() and displayTable().

1. Write the pseudocode for the function `computeOffset`. This function will determine the day of the week of the first day of the month by counting how many days have passed since the 1st of January, 1753 (which is a Monday and `offset == 0`). That number (`numDays`) divided by 7 will tell us how many weeks have passed. The remainder will tell us the offset from Monday. For example, if the month begins on a Thursday, then `offset == 3`. The prototype for the function is:

```
int computeOffset(int month, int year);
```

Please do not plagiarize this from the internet; you must use a loop to solve the problem. The output for this function is the following:

Day	offset
Sunday	6
Monday	0
Tuesday	1
Wednesday	2
Thursday	3
Friday	4
Saturday	5

2. Write the pseudocode for the function `displayTable`. This function will take the number of days in a month (`numDays`) and the offset (`offset`) as parameters and will display the calendar table. For example, consider `numDays == 30` and `offset == 3`. The output would be:

```
  Su  Mo  Tu  We  Th  Fr  Sa
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30
```

There are two problems you must solve: how to put the spaces before the first day of the month, and how to put the newline character at the end of the week. The prototype of the function is:

```
void displayTable(int offset, int numDays);
```

Project 07

The final part of the Calendar Program project is to write the code necessary to make the calendar appear on the screen:

```
Enter a month number: 1
Enter year: 1753

January, 1753
  Su  Mo  Tu  We  Th  Fr  Sa
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31
```

A few hints:

- Check the case where `offset == 6` when the month begins with Sunday. July 2001 is an example of such a month. You do not want to have a blank line between the column headers (the days of the week) and the day numbers.
- Check the case where the last day of the month is on a Saturday. March 2001 is an example of such a month. You do not want a blank row at the bottom of the calendar.
- Use the `cal` program built into the Linux system when you test the program by hand.

An executable version of the project is available at:

```
/home/cs124/projects/prj07.out
```

Please do the following:

1. Start with the code written in Project 06.
2. Fix any bugs that escaped your notice the first time through.
3. Verify your solution with testBed:

```
testBed cs124/project07 project07.cpp
```

4. Submit you code with "`Project 07, Calendar`" in the program header.

Unit 3. Pointers & Arrays

3.0 Array Syntax

Sam is working on a function to compute a letter grade from a number grade. While this can be easily done using IF/ELSE statements, he feels there must be an easier way. There is a pattern in the numbers which he should be able to leverage to make for a more elegant and efficient solution. While mulling over this problem, Sue introduces him to arrays…

Objectives

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

- Declare an array to solve a problem.
- Write a loop to traverse an array.
- Predict the output of a code fragment containing an array.
- Pass an array to a function.

Prerequisites

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop (Chapter 2.3).
- Create a loop to solve a simple problem (Chapter 2.3).

Overview

In the simplest form, an array is a "bucket of variables." Rather than having many variables to represent the values in a collection, we can have a single variable representing the bucket. There are many instances when working with buckets is more convenient than working with individual data items. One example is text:

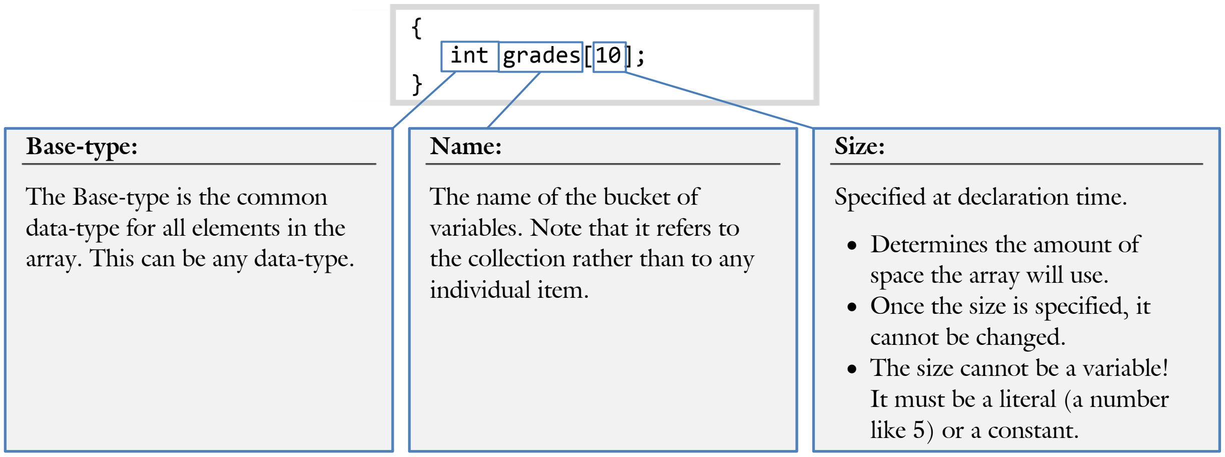
```
char text[256];
```

In this example, the important unit is the collection of characters rather than any single character. It would be extremely inconvenient to have to manage 256 individual variables to store the data of a single string. There are three main components of the syntax of an array: the syntax for declaring an array, the syntax for referencing an individual item in an array, and the syntax for passing an array as a parameter:

Declaring an array	Referencing an array	Passing as a parameter
Syntax: `<Type> <variable>[size]`	Syntax: `<variable>[<index>]`	Syntax: `(<Type> <variable>[])`
Example: `int grades[200];`	Example: `cout << grades[i];`	Example: `void func(int grades[])`
A few details: • Any data-type can be used. • The size must be a natural number {1, 2, etc.}, not a variable.	A few details: • The index starts with 0 and must be within the valid range.	A few details: • You must specify the base-type. • No size is passed in the square brackets `[]`.

Declaring an Array

A normal variable declaration asks the compiler to reserve the necessary amount of memory and allows the user to reference the memory by the variable name. Arrays are slightly different. The amount of memory reserved is computed by the size of each member in the list multiplied by the number of items in the list.



```
{
   int grades[10];
}
```

Base-type:

The Base-type is the common data-type for all elements in the array. This can be any data-type.

Name:

The name of the bucket of variables. Note that it refers to the collection rather than to any individual item.

Size:

Specified at declaration time.

- Determines the amount of space the array will use.
- Once the size is specified, it cannot be changed.
- The size cannot be a variable! It must be a literal (a number like 5) or a constant.

The first part is the base-type. This is the type of data common to all items in the list. In other words, we can't have an array where some items are integers and others are characters. We can use any data-type as the base-type. This includes built-in data-types (`int`, `float`, `bool`, etc.) as well as custom data-types we will create in future semesters.

The second part is the name. Since the array name refers to the collection of elements (as opposed to any individual element), this name is commonly plural. Another common naming convention is to have the "`list`" prefix (`int listGrade[10];`).

The final part is the size. It is important to note that the compiler needs to know the size of the array at compilation time. In other words, we cannot make this a variable that the user provides the value for. It is legal to have a literal (example: `10`), a constant earlier defined (example: `const int SIZE = 10;`) or a `#define` resolving to a constant or a literal (example: `#define SIZE 10`). One final note: once the size has been specified, it cannot be changed. We will learn how to specify the size at run-time using a variable later in the semester (Chapter 4.1)

Sam's Corner

While it is illegal to have a variable in the square brackets of an array declaration, our compiler lets it slide. It is a very bad idea to rely on a given compiler's non-adherence to the language standard: it will make it difficult to port (or move) the code to another compiler.

That being said, the new C++ standard (called C++11) makes an allowance on this front. Please read about generalized constant expressions: C++11 Generalized constant expressions.

Initializing

When declaring a simple variable, it is possible to initialize the variable at the same time:

```
{
   int variable1;             // declared but uninitialized
   variable1 = 10;            // now it is initialized

   int variable2 = 10;        // declared and initialized in one step
}
```

It is also possible to declare and initialize an array variable in one step:

```
{
   char grades[4] =
   {
      'A', 'C', 'B', 'A'
   };
}
```

Observe how the list of items to initialize are delimited with curly braces (`{}`). Since the Elements of Style demands that each curly brace be on its own line, we align them with the base-type. Finally, the individual items in the array are presented in a comma-separated list.

Declaration	In memory	Description
`int array[6];`	`? ? ? ? ? ?`	Though six slots were set aside, they remain uninitialized. All slots are filled with unknown values.

`int array[6] =`
`{`
`   3, 6, 2, 9, 1, 8`
`};`
	`3 6 2 9 1 8`	The initialized size is the same as the declared size so every slot has a known value.

`int array[6] =`
`{`
`   3, 6`
`};`
	`3 6 0 0 0 0`	The first 2 slots are initialized, the balance are filled with 0. This is a partially filled array.

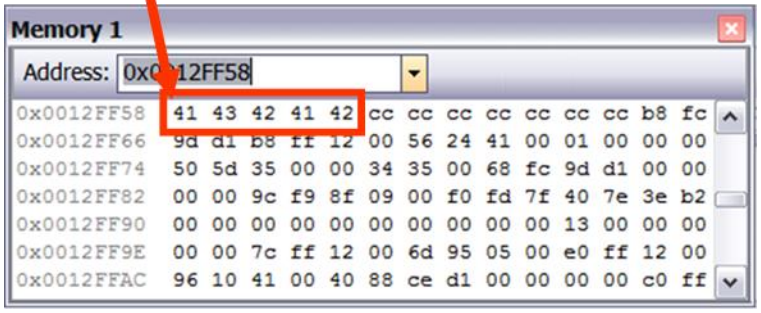
`int array[] =`
`{`
`   3, 6, 2, 9, 1, 8`
`};`
	`3 6 2 9 1 8`	Declared to exactly the size necessary to fit the list of numbers. The compiler will count the number of slots

`int array[6] = {};`	`0 0 0 0 0 0`	This is the easiest way to initialize an array with zeros in all the slots

Sam's Corner

Arrays are stored in memory in the most efficient way imaginable: just a continuous block of adjacent memory locations. The array variable itself refers to (or "points to") the first item in that block of memory. Consider the following memory-dump of a simple declaration of an array of characters:

```
char grades[5] = {'A', 'C', 'B', 'A', 'B'};
```



```
Address: 0x0012FF58
0x0012FF58  41 43 42 41 42 cc cc cc cc cc cc cc b8 fc
0x0012FF66  9d d1 b8 ff 12 00 56 24 41 00 01 00 00 00
0x0012FF74  50 5d 35 00 00 34 35 00 68 fc 9d d1 00 00
0x0012FF82  00 00 9c f9 8f 09 00 f0 fd 7f 40 7e 3e b2
0x0012FF90  00 00 00 00 00 00 00 00 00 00 13 00 00 00
0x0012FF9E  00 00 7c ff 12 00 6d 95 05 00 e0 ff 12 00
0x0012FFAC  96 10 41 00 40 88 ce d1 00 00 00 00 c0 ff
```

In this example, the memory starts at location `0x0012FF58`.

Declaring an array of strings

Since arrays of characters are called strings, how do we make arrays of strings? In essence, we will need an array of arrays. We call these multi-dimensional arrays and they will be the topic of Chapter 4.0.

```
{
   char listNames[10][256];      // ten strings
}
```

Observe the two sets of square brackets. The first set (`[10]`) refers to the number of strings in the array of strings. The second set (`[256]`) refers to the size of each individual string in the list. As a result, we will have ten strings, each 256 bytes in length. This means the total size of `listNames` is `sizeof(char) * 10 * 256`.

We can also initialize an array of strings at declaration time:

```
{
   char listNames[][256] =       // the number of strings is not necessary,
   {                             //   the compiler can count
      "Thomas",
      "Edwin",                   // use either "quotes" or {'E', 'd', 'w', … },
      "Ricks"
   };
}
```

Referencing an Array

In mathematics, we can define a sequence of numbers much like we define arrays of numbers in C++. We refer to individual members of a sequence in mathematics with the subscript notation: X_2 is the second element of the sequence *X*. We use the square bracket notation in C++:

```
cout << list[3] << endl;         // Access the fourth item in the list
```

One important difference between arrays and mathematical sequences is that the indexing of arrays starts at zero. In other words, the first item is `list[0]` and the second is `list[1]`. The reason for this stems from how arrays are declared. Recall that the array variable refers to (or points to) the first item in the list. The array index is actually the offset from that first item. Thus, when one references `list[2]`, one is actually saying "move 2 spots from the first item."

Loops

Since indexing for arrays starts at zero, the valid indices for an array of 10 items is 0 … 9. This brings us to our standard FOR loop for arrays:

```
for (int i = 0; i < num; i++)
   cout << list[i];
```

From this loop we notice several things. First, the array index variable is commonly the letter `i` or some version of it (such as `iList`). This is one of the few times we can get away with a one letter variable name.

The second point is that we typically start the list at zero. If we start at one, we will skip the first item in the list.

The Boolean expression is (`i < num`). Observe how we could also say (`i <= num - 1`). This, however, is needlessly complex. We can read the Boolean expression as "as long as the counter is less than the number of items in the list."

Sue's Tips

It is very bad to index off the end of an array. If, for example, we have an array of 10 numbers, what happens when we attempt to access the 20th item?

```
{
   int list[10];
   int i = 20;
   list[i] = -1;              // error!  Off the end of the list
}
```

The compiler will not prevent such program logic mistakes; it is up to the programmer to catch these errors. In the above example, we will assign the value `-1` to some random location of memory. This will probably cause the program to malfunction in an unpredictable way. The best way to prevent this class of problems is to use asserts to verify our referencing:

```
{
   int list[10];
   int i = 20;

   assert(i >= 0 && i < 10);  // much easier bug to fix
   list[i] = -1;
}
```

Always use an assert to verify that the index is in the range of legal values!

Referencing an array of strings

Arrays of strings are also referenced with the square brackets. However, the programmer needs to specify whether an individual character from a string is to be referenced, whether an entire string is to be referenced, or whether we are working with the collection of strings. Consider the following example:

```
{
   char listNames[][256] =
   {
      "Thomas",
      "Edwin",
      "Ricks"
   };

   cout << listNames[0][0] << endl;      // the letter ‘T’
   cout << listNames[0]    << endl;      // the string “Thomas”
   cout << listNames       << endl;      // ERROR: COUT can’t accept an array of
}                                        //        strings. Write a loop!
```

Example 3.0 – Array Copy

Demo

It is possible to copy the data from one integer variable into another with a simple assignment operator. This is not true with an array. To perform this task, a loop is required. This example will demonstrate how to copy an array of integers.

Problem

Write a program to prompt the user for a list of ten numbers. After the user has entered the values, copy the values into another array and display the list.

Solution

It is not possible to perform the array copy with a simple assignment statement:

```
arrayDestination = arraySource;   // this will not work
```

Instead, it is necessary to write a loop and copy the items one at a time.

```
{
   const int SIZE = 10;       // we can use SIZE to declare because it is a CONST
   int listDestination[SIZE]; // copy data to here. It starts uninitialized
   int listSource[SIZE] =     // copy data from here
   {
      6, 8, 2, 6, 1, 7, 2, 9, 0
   };

   // a FOR loop is required to copy the data from one array to another.
   for (int i = 0; i < SIZE; i++)
      listDestination[i] = listSource[i];
}
```

Challenge

As a challenge, modify the program to copy an array of floating point numbers rather than an array of integers. Make sure you format the output to one or two decimals of accuracy.

As an additional challenge, try to display the list backwards.

See Also

The complete solution is available at 3-0-arrayCopy.cpp or:

```
/home/cs124/examples/3-0-arrayCopy.cpp
```

Arrays as Parameters

Passing arrays as parameters is quite different than passing other data types. The reason for this is a bit subtle. When passing an integer, a copy of the value is sent to the callee. When passing an array, however, the data itself does not move. Instead, only the address of the data is sent. This means, in effect, that passing arrays is always pass-by-reference.

Passing strings

As mentioned previously, strings are just arrays. Thus, passing a string as a parameter is the same as passing an array as a parameter. For any parameter-passing scenario, there are two parts: the callee (the function being called) and the caller (the function initiating the function call).

The parameter declaration in the callee looks much like the declaration of an array. There is one exception however: there is no number inside the square brackets. The reason for this may seem a bit counter-intuitive at first: the callee does not know the size of the array. Consider the following example:

```
/********************************************
 * DISPLAY NAME
 * Display a user's name on the screen
 ********************************************/
void displayName(char lastName[], bool isMale) // no number inside the brackets!
{
   if (isMale)
      cout << "Brother ";
   else
      cout << "Sister ";
   cout << lastName;                           // treated like any other string
   return;
}
```

In this example, the first parameter (`lastName`) is a string. For the rest of the function, we can treat `lastName` like any local variable in the function.

Notice that we use the parameter mechanism to pass data back to the caller rather than using the return mechanism. We do this because array parameters behave like pass-by-reference variables.

```
/*******************************************
 * GET NAME
 * Prompt the user for this last name
 *******************************************/
void getName(char lastName[])              // Even though this is pass-by-reference
{                                          //     there is no & in the parameter
   cout << "What is your last name? ";
   cin  >> lastName;
   return;                                 // Do not use the return mechanism
}                                          //     for passing arrays
```

Observe how both the input parameter (demonstrated in the function `displayName()`) and the output parameter (demonstrated in the function `getName()`) pass the same way.

Calling a function accepting an array as a parameter is accomplished by passing the entire array name. Observe how we do not use square brackets when passing the string.

```
/**********************************************
 * MAIN
 * Driver function for displayName and getName
 **********************************************/
int main()
{
   char name[256];                              // this buffer is 256 characters
   getName(name);                               // no []s used when passing arrays

   displayName(name,    true  /*isMale*/);      // again, no []s
   displayName("Smith", false /*isMale*/);      // we can also pass a string literal.
                                                // this buffer is not 256 chars!
   return 0;
}
```

The complete program for this example is available on 3-0-passingString.cpp:

```
/home/cs124/examples/3-0-passingString.cpp
```

When calling a function with an array, do not use the square brackets (`[]`s). If you do, you will be sending only one element of the array (a `char` in this case). Observe how we can pass either a string variable (`name`) or a string literal (`"Smith"`). In the former case, the buffer is 256 characters. In the latter case, the buffer is much smaller. Therefore, the caller specifies the buffer size, not the callee. This is why the callee omits a number inside the square brackets (`[]`s) in the parameter declaration.

Sam's Corner

Recall that we should only be passing parameters by-reference when we want the callee to change the value. This gets a bit confusing because passing arrays as parameters is much like pass-by-reference. How can we avoid this unnecessarily tight coupling? The answer is to use the `const` modifier.

The `const` modifier allows the programmer to say "This variable will never change." When used in a parameter, it is a guarantee that the function will not alter the data in the variable. It would therefore be more correct to declare `displayName()` as follows:

```
void displayName(const char lastName[], bool isMale);
```

If the programmer made a mistake and actually tried to change the value in the function, the following compiler error message would be displayed:

```
example.cpp: In function "void displayName(const char*, bool)":
example.cpp:15: error: assignment of read-only location "* lastName"
```

Passing arrays

Passing arrays as parameters is much like passing strings with one major exception. While strings are frequently (but not always!) 256 characters in length, the size of array buffers are difficult to predict. For this reason, we almost always pass the size of an array along with the array itself:

```
/******************************************
 * FILL LIST
 * Fill a list with the user input
 ******************************************/
void fillList(float listGPAs[], int numGPAs)
{
   cout << "Please enter " << numGPAs << " GPAs\n";
   for (int iGPAs = 0; iGPAs < numGPAs; iGPAs++)
   {
      cout << "\t#" << iGPAs + 1 << " : ";
      cin  >> listGPAs[iGPAs];
   }
}
```

In this case, the function would not know the number of items in the list if the caller did not pass that value as a parameter.

Sue's Tips

There are commonly three variables in the typical array loop: the array to be looped through, the number of items in the array, and the counter itself. Each of these is related yet each fulfill a different role. It is a good idea to choose variable names to emphasize the differences and similarities:

- `listGPA`: The "`list`" prefix indicates it is an array, the "`GPA`" suffix indicates it pertains to GPAs.
- `numGPA`: The "`num`" prefix indicates it is the number of items, the "`GPA`" suffix again indicates what list it pertains to.
- `iGPA`: The "`i`" prefix indicates it is an incrementer (or iterator).

Using consistent and predictable names makes it easier to spot bugs.

Observe how the caller can then specify the size of the array and, by doing so, control the number of iterations through the loop:

```
/*******************************************
 * MAIN
 * Driver program for fillList
 *******************************************/
int main()
{
   float listSmall[5];
   float listBig[500];

   fillList(listSmall, 5);   // make sure the passed size equals the true size
   fillList(listBig, 500);   // this time, many more iterations will be performed

   return 0;
}
```

A common mistake is to pass the wrong size of the list as a parameter. In the above example, it would be a mistake to pass the number `10` for the size of `listSmall` because it is only 5 slots in size.

Example 3.0 – Passing an Array

Demo

This example will demonstrate how to pass arrays of numbers as parameters. Included will be how to pass an array as an input parameter (to be filled) and how to pass an array as an output parameter (to be displayed).

Problem

Write a program to prompt the user for 4 prices in Euros, and display the dollar amount.

```
Please enter 4 prices in Euros
        Price # 1: 1.00
        Price # 2: 3.75
        Price # 3: .17
        Price # 4: 104.54
The prices in US dollars are:
        $1.38
        $5.17
        $0.23
        $144.04
```

Solution

The function to prompt the user for the prices is an output parameter. Notice how we do not use the ambersand (`&`) when specifying an output parameter if the parameter is an array.

```
void getPrices(float prices[], int num)
{
   cout << "Please enter " << num << " prices in Euros\n";

   for (int i = 0; i < num; i++)
   {
      cout << "\tPrice # " << i + 1 << ": ";
      cin  >> prices[i];
   }
}
```

When the parameter is input-only, it is a good idea to include the `const` modifier to the array parameter declaration to indicate that the function will not modify any of the data.

```
void display(const float prices[], int num)
{
   // configure the output for money
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);

   // display the prices
   cout << "The prices in US dollars are:\n";
   for (int i = 0; i < num; i++)
      cout << "\t$" << prices[i] << endl;
}
```

Challenge

As a challenge, modify the `display()` function so both the Euro and the Dollar amounts are displayed. This will require you to make a copy of the `price` array so both arrays can be passed to the display function.

See Also

The complete solution is available at 3-0-passingArray.cpp or:

```
/home/cs124/examples/3-0-passingArray.cpp
```

Sue's Tips

Always pass the size of the array as a parameter. The least problematic way to do this is to let the compiler compute the number of elements. Consider the following code:

```
float listSmall[5];
cout << sizeof(listSmall) << endl;      // sizeof(float) * 5 == 20
cout << sizeof(listSmall[0]) << endl;   // sizeof(float) == 4
cout << sizeof(listSmall) / sizeof(listSmall[0]) << endl;
                                        // 20 / 4 == 5 NumElements!
```

Thus, it is very common to pass the size of a list using the following convention:

```
fillList(listSmall, sizeof(listSmall) / sizeof(listSmall[0]) );
```

This expression is worth memorizing.

Passing an array of strings

Passing arrays of strings is a bit more complex than passing single strings. While the reason for the differences won't become apparent until we learn about multi-dimensional arrays later in the semester (Chapter 4.0), the syntax is as follows:

```
/********************************************
 * DISPLAY NAMES
 * Display all the names in the passed list
 *******************************************/
void displayNames(char names[][256], int num)  // second [] has the size in it
{
   for (int i = 0; i < num; i++)               // same as with other arrays
      cout << names[i] << endl;                // access each individual string
}
```

Observe that, like with simple strings, the first set of square brackets near the `names` variable is empty. The second set, however, needs to include the size of each individual string. When we call this function, one does not include the square brackets. This ensures the complete list of names is passed, not an individual name.

```
/*******************************************
 * MAIN
 * Simple driver program for displayNames
 *******************************************/
int main()
{
   char fullName[3][256] =
   {
      "Thomas",
      "Edwin",
      "Ricks"
   };
   displayNames(fullName, 3);
   return 0;
}
```

Example 3.0 – Array of Strings

Demo

This example will demonstrate how to pass an array of strings as a parameter.

Problem

In this final example, we will prompt the user for his five favorite scripture heroes and display the results in a comma-separated list:

```
Who are your top five scripture heroes?
        #1: Joseph
        #2: Paul
        #3: Esther
        #4: Nephi
        #5: Mary
Your five heroes are: Joseph, Paul, Esther, Nephi, Mary
```

Solution

The function to prompt the user for the strings. Observe how the double square-bracket notation is used and the size of each buffer is in the brackets:

```
void getNames(char listNames[][256], int numNames)
{
   cout << "Who are your top " << numNames << " scripture heroes?\n";

   // standard FOR loop
   for (int iNames = 0; iNames < numNames; iNames++)
   {
      cout << "\t#" << iNames + 1 << ": ";
      cin  >> listNames[iNames];            // one element of an array of strings
   }                                        //    is simply a string!
}
```

To call the function, we specify that the entire list of names is to be used by passing the `names` variable without square brackets.

```
int main()
{
   // declare the array of strings
   char names[5][256];

   // prompt the user for the data
   getNames(names, 5 /*numNames*/);    // send the entire array of strings

   // display the list of names
   display(names, 5 /*numNames*/);

   return 0;
}
```

Challenge

As a challenge, modify the program so that each name only can contain 32 characters rather than 256. Also, change the list size to 10 items instead of five. How much code needs to change to make this work?

See Also

The complete solution is available at 3-0-arrayStrings.cpp or:

```
/home/cs124/examples/3-0-arrayStrings.cpp
```

Passing Characters, Strings, and List of Strings

Consider the following functions:

```
void displayListNames(const char names[][256], int num)  // second [] has the size
{
   for (int i = 0; i < num; i++)               // same as with other arrays
      cout << '\t' << names[i] << endl;        // access each individual string
}

void displayName(const char name[])  // no NUM variable, no value in the[]s
{
   cout << "One single name: " << name << endl;
}

void displayLetter(char letter)
{
   cout << "One single letter: " << letter << endl;
}
```

Given a list of names (`fullName`), we can call each of the above functions:

```
{
   char fullName[3][256] =
   {
      "Thomas",
      "Edwin",
      "Ricks"
   };

  displayListNames(fullName, 3);       // display all the members of fullName
  displayName(fullName[2]);            // display just the 3rd string in the list
  displayLetter(fullName[2][0]);       // display first letter of 3rd name
}
```

Given a single string (`word`), we can call only `displayName()` and `displayLetter()`:

```
{
   char word[256] = "BYU-Idaho";

   displayName(word);                        // pass a variable with the data "BYU-Idaho"
   displayLetter(word[4]);                   // pass the letter 'I'

   displayName("Vikings");                   // pass a string literal "Vikings"
}
```

Finally, given a single letter (`letter`), we can call only `displayLetter()`:

```
{
   char letter = 'C';

   displayLetter(letter);                    // pass the variable 'C'

   displayLetter('K');                       // pass the literal 'K'
}
```

The complete solution is available at 3-0-passing.cpp or:

```
/home/cs124/examples/3-0-passing.cpp
```

Unit 3

Problem 1

What is the output?

```
{
   float nums[] = {1.9, 5.2, 7.6};

   cout << nums[1] << endl;
}
```

Answer:

____

Please see page 218 for a hint.

Problem 2

What is the output?

```
{
   int a[] = {2, 4, 6};
   int b = 0;

   for (int c = 0; c < 3; c++)
     b += a[c];

   cout << b << endl;
}
```

Answer:

____

Please see page 218 for a hint.

Problem 3

What is the output?

```
{
   char letters[] = "FFFFFFDCBAA";
   int  number = 87;

   cout << letters[number / 10]
        << endl;
}
```

Answer:

____

Please see page 219 for a hint.

Problem 4

What is the output?

```
{
   int one[] = {6, 2, 1, 5};
   int two[] = {3, 9, 9, 7, 12};

   for (int i = 0; i < 2; i++)
      one[0] = one[i] * two[i];

   cout << one[0] << endl;
}
```

Answer:

____

Please see page 218 for a hint.

Problem 5

What is the size of the following variables?

Declaration	Size of
`char a;` `cout << sizeof(a) << endl;`	
`char b[10];` `cout << sizeof(b) << endl;`	
`cout << sizeof(b[0]) << endl;`	
`int c;` `cout << sizeof(c) << endl;`	
`int d[20];` `cout << sizeof(d) << endl;`	
`cout << sizeof(d[1]) << endl;`	

Please see page 36 for a hint.

Problem 6

Write the code to put the numbers 1-10 in an array and display the array backwards:

Answer:

Please see page 220 for a hint.

Problem 7

Given the following function declaration:

```
void fill(int array[])
{
   for (int i = 0; i < 10; i++)
      array[i] = 0;
}
```

Write the code to call the function with a variable called `array`.

Answer:

Please see page 221 for a hint.

Problem 8

Given the following code:

```
{
   float numbers[32];

   display(numbers, 32);
}
```

Write a function prototype for `display()`.

Answer:

Please see page 223 for a hint.

Problem 9

Write the code to compute and display the average of the following numbers: 54.1, 18.6, 32.7, and 7:

```
Average is 28.1
```

Answer:

Please see page 220 for a hint.

Unit 3

Problem 10, 11

Write a function to prompt the user for 10 names. The resulting array should be sent back to `main()`.

Write a driver function `main()` to call the function and display the names on the screen.

Please see page 226 for a hint.

Unit 3

Assignment 3.0

This program will consist of three parts: `getGrades()`, `averageGrades()`, and a driver program.

getGrades

Write a function to prompt the user for ten grades and return the result back to `main()`. Note that any variables declared in `getGrades()` will be destroyed when the function returns. This means that `main()` will need to declare the array and pass it as a parameter to `getGrades()`.

averageGrades

Write another function to find the average of the grades and return the answer. Of course, the grades array will need to be passed as a parameter. The return value should be the average.

Driver Program

Finally, create `main()` that does the following:

- Has the grades array as a local variable
- Calls `getGrades()`
- Calls `averageGrades()`
- Displays the result

Please note that for this assignment, integers should be used throughout.

Example

The user input is **underlined**.

```
Grade 1: 90
Grade 2: 86
Grade 3: 95
Grade 4: 76
Grade 5: 92
Grade 6: 83
Grade 7: 100
Grade 8: 87
Grade 9: 91
Grade 10: 0
Average Grade: 80%
```

Assignment

The test bed is available at:

```
testBed cs124/assign30 assignment30.cpp
```

Don’t forget to submit your assignment with the name “Assignment 30” in the header.

Please see page 49 for a hint.

3.1 Array Design

Sue was again enlisted by her mother to help her make sense of some stock data. While it is easy to determine the starting price of the stock (the first item on the list) or the ending price of the stock (the last item on the list), it is much more tedious to find the high and low values. Rather than laboriously search the list by hand, Sue writes a program to find these values.

Objectives

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

- Search for a value in an array.
- Look up a value in an array.

Prerequisites

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

- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Predict the output of a code fragment containing an array (Chapter 3.0).
- Pass an array to a function (Chapter 3.0).

Overview

Arrays are used for a wide variety of tasks. The most common usage is when a collection of homogeneous data is needed. Common examples include text (arrays of characters), lists of numbers (a set of grades), and lists of more complex things (lists of addresses or students for example). In each case, individual members of the list can be referenced by index. Problems involving this usage of arrays typically involve filling, manipulating, and extracting data from a list.

Another use for an array would be to look up a value from a list. Consider selecting an item off a menu or looking up the price on an order sheet. In both of these cases, equivalent logic can be written with IF/ELSE statements. Storing the data in a table, however, often takes less memory, requires less code, and is much easier to update. Problems involving this usage of arrays typically involve looking up data in tables.

Lists of Data

When solving problems involving lists of data, it is common to need to write a loop to visit every element of the list. Most of these problems can be reduced to the following two questions:

1. How do you iterate through the list (usually using a standard FOR loop)?
2. What happens to each item on the list?

To illustrate this principle, three examples will be used: filling a list from a file of data, finding an item from a list of unsorted data, and finding an item from a list of sorted data.

Example 3.1 – Read From File

Demo

This example will demonstrate how to fill an array from a list of numbers in a file. This is a common function to write: fill an array from a given file name, an array to be filled, and the number of items in the array.

Problem

Fill the array `data` with the values in the following file:

```
1   3   6   2   9   3
```

Solution

In order to write a function to read this data into an array, it is necessary to answer the question "what needs to happen to each item in the list?" The answer is: read it from the file (using `fin >>`) and save it in the array(using `fout <<`). To accomplish this, our function needs to take three parameters: `fileName` or the location from which we will be reading the data, `data` or where we will be placing the data, and `max` or the size or capacity of the array `data`.

Observe how we need to send some information back to the caller, namely how many items we successfully read. This is most conveniently done through the return type where 0 indicates a failure. Consider the following function:

```
int readFile(const char fileName[],    // use const because it will not change
             int data[],               // the output of the function
             int max)                  // capacity of data, it will not change
{
   // open the file for reading
   ifstream fin(fileName);             // open the input stream to fileName
   if (fin.fail())                     // never forget the error checking
   {
      cout << "ERROR: Unable to read file "
           << fileName << endl;        // display the filename we tried to read
      return 0;                        // return the error condition: none read
   }

   // loop through the file, reading the elements one at a time
   int numRead = 0;                    // initially none were read
   while (numRead < max &&             // don't read more than the list holds
          fin >> data[numRead])        // read and check for errors
      numRead++;                       // increment the index by one

   // close the file and return
   fin.close();                        // never forget to close the file
   return numRead;                     // report the number successfully read
}
```

Observe how we make sure to check that we are not putting more items in the list than there is room. If the list holds 10 but the file has 100 items, we should still only read 10.

Challenge

We did not traverse the array using the standard FOR loop even though all three parts (initialization, condition, and increment) are present. As a challenge, try to modify the above function so a FOR loop is used to read the data from the file instead of a WHILE loop. Which solution is best?

See Also

The complete solution is available at 3-1-readFile.cpp or:

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

Example 3.1 – Searching an Unsorted List

Demo

Another common array problem is to find a given item in an unsorted list. In this case, the ordering of the list is completely random (as unsorted lists are) so it is necessary to visit every item in the list.

Problem

Write a function to determine if a given search value is present in a list of integers:

```
bool linearSearch(const int numbers[], int size, int search);
```

If the value `search` is present in `numbers`, return `true`, otherwise, return `false`.

Solution

The first step to solving this problem is to answer the question "what needs to happen to each item in the list?" The answer is: compare it against the sought-after item. This will be accomplished by iterating through the array of numbers, comparing each entry against the `search` value.

```
bool linearSearch(const int numbers[],    // the list to be searched
                  int size,               // how many items are in the list
                  int search)             // the term being searched for
{
   // walk through every element in the list
   for (int i = 0; i < size; i++)         // standard FOR loop for an array
      if (search == numbers[i])           // compare each against the search item
         return true;                     // if found, then leave with true

   // not found if we reached the end
   return false;
}
```

Observe how the larger the list (`size`), the longer it will take. We call this a "linear search" because the cost of the search is directly proportional to the size of the list.

Challenge

Finding if an item exists in a list is essentially the same problem as finding the largest (or smallest) item in a list. As a challenge, modify the above function to return the largest number:

```
int findLargestValue(const int numbers[], int size);
```

To accomplish this, declare a variable that contains the largest value currently found. Each item is compared against this value. If the largest number currently found is smaller than the current item being compared, then update the value with the current item. After every item in the list has been compared, the value of the largest is returned.

See Also

The complete solution is available at 3-1-linearSearch.cpp or:

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

Example 3.1 – Searching a Sorted List

Demo

It turns out that people rarely perform linear searches. Imagine how long it would take to look up a word in the dictionary! This example will demonstrate how to do a binary search.

Problem

Write a function to determine if a given search value is present in a list of integers:

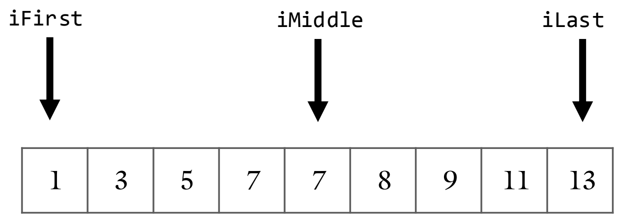
```
bool binarySearch(const int numbers[], int size, int search);
```

If the value `search` is present in `numbers`, return `true`, otherwise, return `false`.

Solution

The binary search algorithm works much like searching for a hymn in the hymnal:

1. Start in the middle (`iMiddle`) by opening the hymnal to the center of the book.
2. If the hymn number is greater, then you can rule out the first half of the book. Thus the first possible page (`iFirst`) it could be on is the middle (`iMiddle`), the last is the end (`iLast`).
3. If the hymn number is smaller then you can rule out the second half of the book.
4. Repeat steps 1-3. With each iteration, we either find the hymn or rule out half of the remaining pages. Thus `iFirst` and `iLast` get closer and closer together. If `iFirst` and `iLast` are the same, then our hymn is not present and we quit the search.



Observe how the question "what needs to happen to each item in the list?" is answered with "decide if we should focus on the top half or bottom half of the list."

```
bool binarySearch(const int numbers[], int size, int search)
{
   int iFirst = 0;                        // iFirst and iLast represent the range
   int iLast = size – 1;                  // of possible values: the whole list

   while (iLast >= iFirst)                // as long as the range is not empty
   {
      int iMiddle = (iLast + iFirst) / 2; // find the center (step (1) above)

      if (numbers[iMiddle] == search)     // if we found it, then stop
         return true;
      else if (numbers[iMiddle] > search) // if middle is bigger, focus on the
         iLast  = iMiddle - 1;            //     beginning of the list (step (2))
      else                                // otherwise (smaller), focus on the
         iFirst = iMiddle + 1;            //     end of the list (step (3))
   }                                      // continue (step (4))

   // only got here if we didn't find it
   return false;                          // failure
}
```

See Also

The complete solution is available at 3-1-binarySearch.cpp or:

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

Table Lookup

Arrays are also a very useful tool in solving problems involving looking up data from a table or a list of values. This class of problems is typically solved in two steps:

1. Create a table of the data to be referenced.
2. Write code to extract the data from the table.

This is best illustrated with an example. Consider the following code to convert a number grade into a letter grade:

```
/************************************
 * COMPUTE LETTER GRADE
 * Compute the letter grade from the
 * passed number grade
 ************************************/
char computeLetterGrade(int numberGrade)
{
   assert(numberGrade >= 0 && numberGrade <= 100);

   // table to be referenced
   char grades[] =
   { //0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
      'F', 'F', 'F', 'F', 'F', 'F', 'D', 'C', 'B', 'A', 'A'
   };

   assert(numberGrade / 10 >= 0);
   assert(numberGrade / 10 <  sizeof(grades) / sizeof(grades[0]);
   return grades[numberGrade / 10];         // Divide will give us the 10's digit
}
```

When using this technique, it is important to spend extra time and thought on the representation of the data in the table. The goal is to represent the data as clearly (read: error-free) as possible and to make it as easy to extract the data as possible. This programming technique is called **data-driven design**.

Observe how we do not have a FOR loop to iterate through the list. Since we were careful about how the list was ordered (where the index of the `grades` array correspond to the first 10's digit of the `numberGrade` array), we can look up the letter grade directly.

Finally, while it may seem excessive to have three asserts in a function containing only two statements, these asserts go a long way to find bugs and prevent unpredictable behavior.

Unit 3

Example 3.1 – Tax Bracket

Demo

This example will demonstrate a table-loopup design for arrays. In this case, the tax table will be put in a series of arrays.

Problem

Consider the following tax table:

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

Compute a user's tax bracket based on his income.

Solution

The first part of the solution is to create three arrays representing the lower part of the tax bracket, the upper part of the tax bracket, and the taxation rate. The second part is to loop through the brackets, seeing if the user's income falls withing the upper and lower bounds. If it does, the corresponding tax rate is returned.

```
int computeTaxBracket(int income)
{
   int lowerRange[] =                          // the 1st column of the tax table
   {  // 10%     15%     25%     28%     33%     35%
          0,  15100,  61300, 123700, 188450, 336550
   };
   int upperRange[] =                          // the 2nd column
   {  // 10%     15%     25%     28%     33%     35%
      15100,  61300, 123700, 188450, 339550, 999999999
   };
   int bracket[]                               // the bracket
   {
         10,     15,     25,     28,     33,     35
   };

   for (int i = 0; i < 6; i++)                 // the index for the three arrays
      if (lowerRange[i] <= income && income <= upperRange[i])
         return bracket[i];

   return -1;                                  // not in range (negative income?)!
}
```

Challenge

As a challenge, modify this function to compute the actual income. This will require a fourth array: the fixed amount. See if you can put your function in Project 1 and get it to pass testBed.

See Also

The complete solution is available at 3-1-computeTaxBracket.cpp or:

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

Problem 1

What is the output of the following code?

```
{
   int a[4];

   for (int i = 0; i < 4; i++)
      a[i] = i;

   for (int j = 3; j >= 0; j--)
      cout << a[j];

   cout << endl;
}
```

Answer:

____

Please see page 218 for a hint.

Problem 2

What is the output of the following code?

```
{
   char a[] = {'t', 'm', 'q'};
   char b[] = {'a', 'z', 'b'};
   char c[3];

   for (int i = 0; i < 3; i++)
      if (a[i] > b[i])
         c[i] = a[i];
      else
         c[i] = b[i];

   for (int i = 0; i < 3; i++)
      cout << c[i];
   cout << endl;
}
```

Answer:

____

Please see page 237 for a hint.

Problem 3

Complete the code to count the number of even and odd numbers:

```
void displayEvenOdd(const int values[],
                    int num)
{
   //determine even/odd
   int numEvenOdd[2] = {0, 0};









   // display
   cout << "Number even: "
        << numEvenOdd[0] << endl;
   cout << "Number odd:  "
        << numEvenOdd[1] << endl;
}
```

Please see page 237 for a hint.

Problem 4

Fibonacci is a sequence of numbers where each number is the sum of the previous two:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Write the code to complete the Fibonacci sequence and store the results in an array.

```
void fibonacci(int array[], int num)
{















}
```

Please see page 237 for a hint.

Assignment 3.1

Start with Assignment 3.0 and modify the function `averageGrades()` so that it does not take into account grades with the value -1. In this case, -1 indicates the assignment was not completed yet so it should not factor in the average.

Examples

Two examples… The user input is **underlined**.

Example 1

```
Grade 1: 90
Grade 2: 86
Grade 3: 95
Grade 4: 76
Grade 5: 92
Grade 6: 83
Grade 7: 100
Grade 8: 87
Grade 9: 91
Grade 10: -1
Average Grade: 88%
```

Notice how the -1 for the 10th grade is not factored into the average.

Example 2

```
Grade 1: -1
Grade 2: -1
Grade 3: -1
Grade 4: -1
Grade 5: -1
Grade 6: -1
Grade 7: -1
Grade 8: -1
Grade 9: -1
Grade 10: -1
Average Grade: ---%
```

Since all of the grades are -1, there is no average. You will need to handle this condition.

Assignment

The test bed is available at:

```
testBed cs124/assign31 assignment31.cpp
```

Don't forget to submit your assignment with the name "Assignment 31" in the header.

Please see page 235 for a hint.

3.2 Strings

Sue has just received an e-mail from her Grandma Ruth. Grandma, unfortunately, is new to computers (and typing for that matter) and has written the entire message using only capital characters. This is very difficult to read! Rather than suffer through the entire message, she writes a program to convert the all-caps text to sentence case.

Objectives

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

- Understand the role the null character plays in string definitions.
- Write a loop to traverse a string.

Prerequisites

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

- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Predict the output of a code fragment containing an array (Chapter 3.0).
- Pass an array to a function (Chapter 3.0).

Overview

While we have been using text (`char text[256];`) the entire semester, we have yet to discuss how it works. Through this chapter, we will learn that strings have a little secret: they are just arrays. Specifically, they are arrays of characters with a null-character at the end.

```
{
   char text[] =
   {
      'C', 'S', ' ', '1', '2', '4', '\0'
   };
   cout << text << endl;
}
```

One common task to perform on strings is to write a loop to traverse them. Consider the following loop prompting the user for text and displaying it one letter at a time on the screen:

```
{
   char text[256];                // strings are arrays of characters
   cout << "Enter text: ";        // using the double quotes creates a string literal
   cin  >> text;                  // CIN puts the null-character at the end of strings

   for (int i = 0; text[i]; i++)  // almost our second standard FOR loop
      cout << text[i] << endl;    // we can access strings one character at a time
}
```

This chapter will discuss why strings are defined as arrays of characters with a null-character, how to declare a string and pass one to a function, and how to traverse a string using a FOR loop.

Implementation of Strings

An integer is a singular value that can always fit into a single 4-byte block of memory. Text, however, is fundamentally different. Text can be any length, from a few letters to a complete novel. Because text is an arbitrary length, it cannot be stuck into a single location of memory as a number can. It is therefore necessary to use a different data representation: an array.

Text is fundamentally a one-dimensional construct. Each letter in a book can be uniquely addressed from its offset (index) from the beginning of the manuscript. For this reason, text is fundamentally an array of characters. We call these arrays of characters "**strings**" because they are "strings of characters" somewhat like a collection of characters attached by a string.

I	n		t	h	e		b	e	g	i	n	n	i	n	g		…

Because text can be a wide variety of lengths, it is necessary to have some indicator to designate its size. There are two fundamental strategies: keep an integral variable to specify the length, or specify an end-of-string marker. The first method is called **Pascal-strings** because the programming language Pascal uses it as the default string type. This method specifies that the first character of the string is the length:

6	C	S		1	2	4

The second method, called **c-strings**, puts an end-of-string marker. This marker is called the null-character:

C	S		1	2	4	null

While either design is viable, c-strings are used exclusively in C++ and are the dominant design in programming languages today. The main reason for this is that each slot in a character array is, well, a character. This means that the maximum value that can be put in a character slot is 255. Thus, the maximum length of a Pascal-string is 255 characters. C-strings do not have this limitation; they can be any length.

Null-character

The null-character (`'\0'`) is a special character used to designate the end of a c-string. We can assign the null-character to any `char` variable:

```
{
   char nullCharacter = '\0';        // single character assigned a null

   char text[256];                   // text is just an array of characters
   text[0] = '\0';                   // putting the null at the beginning signifies
}                                    //       an empty string
```

The value of the null-character is `0` on the ASCII Table:

```
assert('\0' == 0);
```

There is a special reason why the null-character was given the first slot on the ASCII table: it is the only character that equates to `false` when cast to a `bool`. In other words:

```
assert('\0' == false);
```

Since zero is the only integer mapping to `false`, we can assume that the null-character is the only `false` character in the ASCII table.

C-Strings

All strings are c-strings by default in C++. To illustrate, consider the following code:

```
cout << "Hello world" << endl;
```

One may think that this string uses eleven bytes of memory (the space character is also a character). However, it actually takes twelve:

H	e	l	l	o		w	o	r	l	d	null

We can verify this with the following code:

```
cout << sizeof("Hello world") << endl;
```

The output, of course, will be `12`.

String Syntax

In all ways, we can treat strings as simple arrays. There are two characteristics of strings, however, which make them special. The first characteristic is the existence of the null-character. While all strings are character arrays, not all character arrays are strings. In other words, an array of letter grades for a class will probably not be a string because there is no null-character. The existence of the null eliminates the need to pass a `length` parameter when calling a function. However, we will still probably want to pass a buffer size variable.

The second characteristic is the double-quotes notation. Consider the following two strings with equivalent declarations:

```
{
   char text1[] =
   {
      'C', 'S', ' ', '1', '2', '4', '\0'    // the hard way. Don't do this!
   };
   char text2[] = "CS 124";                 // ah, much better...
}
```

Clearly the double-quote notation greatly simplifies the declaration of strings. Whenever we see these double-quotes, however, we must always remember the existence of the hidden null-character.

Declaring a string

The rules for declaring a string are very similar to those of declaring an array:

Declaration	In memory	Description
`char text[10];`	`? ? ? ? ? ? ? ? ? ?`	None of the slots are initialized
`char text[7] = "CS 124";`	`C S   1 2 4 \0`	The initialized size is the same as the declared size
`char text[10] = "CS 124";`	`C S   1 2 4 \0 \0 \0 \0`	The first 7 are initialized, the balance are filled with `0`
`char text[] = "CS 124";`	`C S   1 2 4 \0`	Declared to exactly the size necessary to fit the text

Sue's Tips

A common mistake when declaring a string is to fail to leave space for the null character. Remember that the size of the declared string must be one greater than the maximum number of characters that could possibly be in the string.

Passing a string

Since strings are just arrays, exactly the same rules apply to them as they apply to arrays. This means that they can always be treated like pass-by-reference parameters. Consider the following function:

```
/**************************************
 * GET NAME
 * Prompt the user for his first name
 **************************************/
void getName(char name[])
{
   cout << "What is your first name? ";
   cin  >> name;
}
```

Observe how there is no return statement or ampersand (`&`) on the `name` parameter. Because we are passing an array, we still get to fill the value. Now, consider the following code to call this function:

```
{
   getName("String literal");      // ERROR! There is no variable passed here!

   char name[256];
   getName(name);                  // this works, a variable was passed
}
```

What went wrong? The function asked for an array of characters and the caller provided it! The answer to this conundrum is a bit subtle.

A string literal (such as `"String literal"` above) refers to data in the read-only part of memory. This data is not associated with any variable and cannot be changed. The data-type is therefore not an "array of characters," but rather a "constant array of characters." In other words, it cannot be changed.

When authoring a function where the input text is treated as a read-only value, it is very important to add a `const` prefix. This prefix enables the caller of the function to pass a string literal instead of a string variable:

```
/************************************
 * Determine if a given file exists
 ************************************/
bool checkFile(const char fileName[])    // CONST modifier allows caller to
{                                        //     pass a literal
   ifstream fin(fileName);

   if (fin.fail())
      return false;

   fin.close();
   return true;
}
```

Now, due to the `const` modifier, either of the following is valid:

```
{
   char fileName[] = "data.txt";         // string variable that is read/write
   checkFile(fileName);                  // the CONST modifier has no effect here

   checkFile("data.txt");                // this would be a compile error without
}                                        //     the CONST modifier in checkFile()
```

Comparing Strings

Recall that strings are arrays of characters terminated with a null character. Consider two arrays of integers. The only way to compare if two lists of numbers are the same is to compare the individual members.

```
{
   int list1[] = { 4, 8, 12 };
   int list2[] = { 3, 6, 9  };
   if (list1 == list2)                  // this compares the location of the
      cout << "Same!";                  //    two lists in memory. It does _NOT_
   else                                 //    compare the contents of the lists!
      cout << "Different!";
}
```

Instead, a loop is required:

```
{
   int list1[] = { 4, 8, 12 };
   int list2[] = { 3, 6, 9  };
   bool same = true;
   for (int i = 0; i < 3; i++)           // we must go through each item in the
      if (list1[i] != list2[i])          //    two lists to see if they are the same.
         same = false;                   //    There is no other way!
}
```

The only way to see if the lists are the same is to write a loop. Similarly, c-strings cannot be compared with a single `==` operator. To compare two strings, it is necessary to write a loop to traverse the strings!

```
{
   char text1[256] = "Computer Science";
   char text2[256] = "Electrical Engineering";
   if (text1 == text2)                               // this will _NOT_ compare
      cout << "Same!"                                //    the contents of the two
   else                                              //    strings. It will only
      cout << "Different!";                          //    compare the addresses!
}
```

Traversing a String

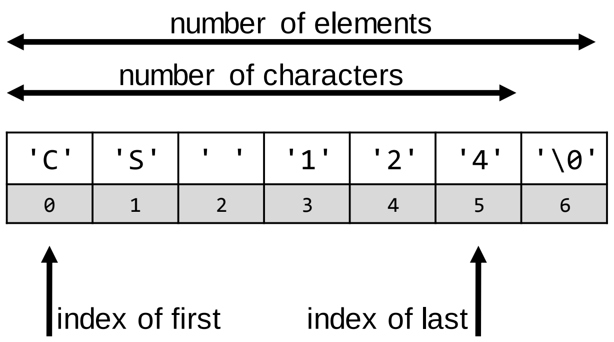
Strings are just arrays of characters where the end is marked with the null character (`'\0'`). This means we can use a FOR loop to walk through a string much the same way we do with an integer array. The only difference is how we know we have reached the end of the array. With arrays of numbers, we typically need to pass another parameter (ex: `numElements`) to tell us when we are at the end:

```
{
   for (int i = 0; i < numElements; i++)
      cout << array[i] << endl;
}
```

So, how big is the string? Consider the following string:

```
char name[] = "CS 124";
```

This corresponds to the following layout in memory:



From here, we have the following relationships:

Amount of memory used	`sizeof(name)`
Number of elements in the array	`sizeof(name) / sizeof(name[0])`
Number of characters in the string (minus null)	`(sizeof(name) / sizeof(name[0]) - 1`
Index of the last item	`(sizeof(name) / sizeof(name[0]) - 2`

Therefore, to walk through an array backwards, you will need:

```
{
   char name[] = "CS 124";
   for (int i = (sizeof(name) / sizeof(name[0]) - 2); i >= 0; i--)
      cout << name[i] << endl;
}
```

It is important to remember that this only works when the size of the buffer is the same as the size of the string. This is not commonly the case!

Traversing using the null-character

With strings, we do not typically have a variable indicating the length of the string. Instead, we look for the null character:

```
{
   for (int i = 0; text[i] != '\0'; i++)
      cout << text[i];
}
```

In this case, we are going to keep iterating until the null-character (`'\0'`) is encountered:

0	1	2	3	4	5	6
C	S		1	2	4	\0

Note that the null character (`'\0'`) has the ASCII value of zero. This is convenient because, when it is casted to a `bool`, it is the only `false` value in the ASCII table. Therefore, we can loop through a string with the following code:

```
{
   for (int i = 0; text[i]; i++)
      cout << text[i];
}
```

Observe how the condition in the FOR loop checks if the `i`th character in the string is `true` or, in other words, not the null-character.

0	1	2	3	4	5	6
C	S		1	2	4	\0
true	true	true	true	true	true	false

Consider the case where a string of text exists with spaces between the words. The problem is to convert the spaces to underscores. In other words "`Introduction to Software Development`" becomes "`Introduction_to_Software_Development`":

```
{
   char text[]  = "Software Development";       // target string, any will do
   for (int i = 0; text[i]; i++)                // loop through all items in the string
      if (text[i] == ' ')                       // check each item against a space
         text[i] = '_'                          // replace with an underscore
}
```

Example 3.2 – Toggle Case

Demo

This example will demonstrate how to walk through a string, modifying each character one at a time. This will be done by using an index to loop through the string until the null character is encountered.

Problem

Consider the scenario where the unfortunate user typed his entire e-mail message with the CAPS key on. This will not only capitalize most characters, but it will un-capitalize the first letter of each sentence. Write a function to correct this error.

```
Please enter your text: sOFTWARE eNGINEERING
Software Engineering
```

Solution

The function to convert the text will take a string as a parameter. Recall that arrays are always pass-by-reference. This means that the parameter can serve both as the input and output of the function.

```
void convert(char text[]);
```

To traverse the string we will loop through each item with an index. Unlike with a standard array, we need to use the condition `text[i]`, not `i < num`:

```
for (int i = 0; text[i]; i++)
   ;
```

With this loop, we can look at every character in the string. Now we will need to determine if a character is uppercase or lowercase. If the character is uppercase (`isupper()`), then we need to lowercase it (`tolower()`). Otherwise, we need to uppercase it (`toupper()`). Note that the functions `isupper()`, `tolower()`, and `toupper()` are in the `cctype` library

```
/******************************************
 * CONVERT
 * Convert uppercase to lowercase and vice versa
 *****************************************/
void convert(char text[])
{
   for (int i = 0; text[i]; i++)        // loop through all the elements in text
      if (isupper(text[i]))             // check each element's case
         text[i] = tolower(text[i]);    // convert to lower if it is upper
      else
         text[i] = toupper(text[i]);    // otherwise convert to upper
}
```

Challenge

As a challenge, try to modify the above program to count the number of uppercase letters in the input stream. The prototype is:

```
int countUpper(const char text[]);
```

See Also

The complete solution is available at 3-2-toggleCase.cpp or:

```
/home/cs124/examples/3-2-toggleCase.cpp
```

Unit 3

Example 3.2 – Sentence Case

Demo

This example will demonstrate how to traverse a string, processing each character individually.

Problem

Write a program to to perform a sentence-case conversion (where only the first letter of a sentence is capitalized) on an input string.

```
Please enter your input text: this IS sOME tExT. soME More TeXt.
The sentence-case version of the input text is:
        This is some text. Some more text.
```

Solution

Sentence-casing is the process of converting every character to lower-case except the first character in the sentence. In other words, after a period, exclaimation point, or question mark is encountered, the next letter needs to be uppercase. We capture this condition with the `isNewSentence` variable. If a sentence-ending character is found, we set `isNewSentence`. This variable remains set until an alphabetic character is found (`isalpha()`). In this case, we convert the letter to uppercase (`toupper()`) and set `isNewSentence` to `false`. Otherwise, we set the letter to lowercase (`tolower()`).

```
void convert(char text[])
{
   // the first letter of the input is the start of a sentence
   bool isNewSentence = true;

   // traverse the string
   for (int i = 0; text[i]; i++)
   {
      if (text[i] == '.' || text[i] == '!' || text[i] == '?')
         isNewSentence = true;

      // convert the first letter to uppercase
      if (isNewSentence && isalpha(text[i]))
      {
         text[i] = toupper(text[i]);
         isNewSentence = false;
      }

      // everything else to lowercase
      else
         text[i] = tolower(text[i]);
   }
}
```

Challenge

As a challenge, modify the above program to handle Title Case the text. This is done by converting every character to lowercase except the first letter of the word. In other words, the first letter after every space is uppercase (as opposed to the first letter after every sentence-ending punctuation).

See Also

The complete solution is available at 3-2-sentenceCase.cpp or:

```
/home/cs124/examples/3-2-sentenceCase.cpp
```

Example 3.2 – Backwards String

Demo

This example will demonstrate how to traverse a string backwards. This will require two loops through the string: forward to find the end of the string, and reverse to display the string backwards.

Problem

Write a program to prompt the user for some text and display the text backwards:

```
Please enter some text: Software Engineering
The text backwards is "gnireenignE erawtfoS"
```

Solution

This function will consist of two parts: looping through the stirng to find the null-character, then progressing backwards to display the contents of the string.

```
/**************************************
 * DISPLAY BACKWARDS
 * Display a string backwards
 **************************************/
void displayBackwards(const char text[])   // const for string literals
{
   // first find the end of the string
   int i = 0;                              // needs to be a local variable
   while (text[i])                         // as long as the null is not found
      i++;                                 // keep going through the string

   // now go backwards
   for (i--; i >= 0; i--)                  // back up one because we went too far
      cout << text[i]                      // display each individual character
   cout << endl;
}
```

When we are finished with the first loop (the forward moving one), the index will be referring to the location of the null-character. We don't want to display the null-character. Therefore, the first step of the display loop is to back the index by one slot. From here, it displays each character including first character of the string (with the `0` index).

Challenge

Notice how a WHILE loop as used instead of a FOR loop in the first part of the function. As a challenge try to re-write this loop as a standard FOR loop.

Again, notice how the second loop uses a FOR loop. Can you re-write it to use a WHILE loop instead?

See Also

The complete solution is available at 3-2-backwards.cpp or:

```
/home/cs124/examples/3-2-backwards.cpp
```

Problem 1

How much memory is reserved by each of the following?

`char text[8]`	
`char text[] = "CS 124";`	
`char text[10] = "124";`	

Please see page 244 for a hint.

Problem 2

What is the output of the following code?

```
{
   char text1[] = "Text";
   char text2[] = "Text";

   if (text1 == text2)
      cout << "Equal";
   else
      cout << "Different"
}
```

Answer:

___

Please see page 246 for a hint.

Problem 3

What is the output of the following code?

```

   char text1[] = "this";
   char text2[] = "that";

   if (text1[0] == text2[0])
      cout << "Equal";
   else
      cout << "Different";
}
```

Answer:

___

Please see page 246 for a hint.

Problem 4

What is the output of the following code?

```
{
  char text[5] = "42";

  cout << text[4] << endl;
}
```

Answer:

___

Please see page 244 for a hint.

Problem 5

Which of the following is the correct prototype of a string passed to a function as a parameter?

```
void displayText(char text);
```

```
void displayText(char text []);
```

```
void displayText(text);
```

```
void displayText(char [] text);
```

Please see page 221 for a hint.

Problem 6

Consider the following function prototype from #5:

How would you call the function `displayText();`?

Answer:

___

Please see page 221 for a hint.

Problem 7

What is the output of the following code?

```
{
   char text[] = "Hello";

   for (int i = 0; i < 6; i++)
   {
      bool value = text[i];
      cout << value;
   }

   cout << endl;
}
```

Answer:

___

Please see page 243 for a hint.

Assignment 3.2

Traversing a string (or any other type of array for that matter) is a common programming task. This assignment is the first part in a two-part series (the other is Assignment 3.5) where we will learn different techniques for visiting every member of a string. Your assignment is to write the function `countLetters()` then a driver program to test it.

countLetters

Write a function to return the number of letters in a string. This involves traversing the string using the array notation (with an index as we have been doing all semester). We will re-write this function in Assignment 3.5 to do the same thing using a pointer.

Driver Program

Create a `main()` that prompts the user for a line of input (using `getline`), calls `countLetters()`, and displays the number of letters.

Note that since the first `cin` will leave the stream pointer on the newline character, you will need to use `cin.ignore()` before `getline()` to properly fetch the section input. Take the first example below, the input buffer will look like:

z	\n	N	o	Z	'	s	H	e	r	e	!	\n

If the program inputs a character followed by a line, the code will look like this:

```
cin >> letter;
cin.getline(text, 256);
```

After the first `cin`, the input pointer will point to the 'z'. When the `getline` statement gets executed next, it will accept all input up to the next newline character. Since the pointer is already on the newline character, the result will be an empty string. To skip this newline character, we use `cin.ignore()`.

Example

Two examples… The user input is **underlined**.

Example 1:

```
Enter a letter: z
Enter text: NoZ'sHere!
Number of 'z's: 0
```

Example 2:

```
Enter a letter: a
Enter text: Brigham Young University - Idaho
Number of 'a's: 2
```

Assignment

The test bed is available at:

```
testBed cs124/assign32 assignment32.cpp
```

Don't forget to submit your assignment with the name "Assignment 32" in the header.

Please see pages 40 and 249 for a hint.

3.3 Pointers

Sue has just finished working on her résumé and begins the arduous task of posting the update. She puts one copy on her LinkedIn page, another in her electronic portfolio, and another in the school's career site. Unfortunately, she also sent a number of copies to various prospective employers across the country. How can she update them? Rather than sending copies everywhere, it would have been much easier if she just sent links. This way, she would only have to update one location and, when people follow the link, they would also get the most recent version.

Objectives

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

- Declare a pointer variable.
- Point to an existing variable in memory with a pointer variable.
- Get the data out of a pointer.
- Pass a pointer to a function.

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).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).

Overview

Up to this point, our variables have been working exclusively with data. This is familiar and straight-forward: if you want to make a copy of a value then use the assignment (=) operator. Often, however, it is more convenient to work with *addresses* rather than with data. Consider Sue's aforementioned scenario. If Sue would have distributed the address of her résumé (link to the document) rather than the data (the physical résumé), then the multiple-copy problem would not exist.

There are several reasons why working with addresses can be more convenient:

- **One master copy**: Often we want to keep one master copy of the data to avoid versioning and update problems like Sue's.
- **Size**: When working with large amounts of data (think arrays and strings), it can be expensive to make a copy of the data every time a function is called. For this reason, arrays are always passed by pointers to functions (more on that later).
- **References & citations**: Consider a lawyer citing a legal case. Rather than bringing the entire case into the courtroom to make a point, he instead cites the relevant case. This way, any interested party can go look at the original case if they are unsure of the details.

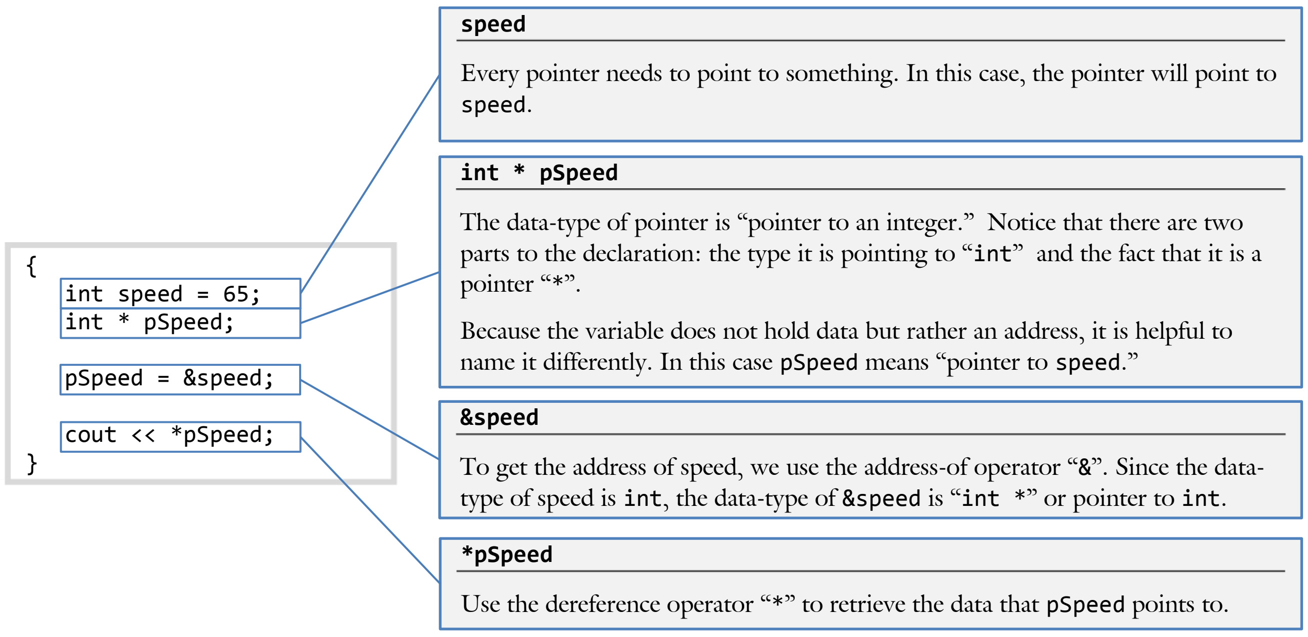
We use addresses in everyday life. Other names include links, references, citations, and pointers. For something to be considered a pointer, it must exhibit the following properties:

- Points: Each pointer must point to something. This could be data, a physical object, an idea, or whatever.
- Mapping: There must be a unique mapping. In other words, for every pointer, there must be exactly one object it is referring to.
- Addressing scheme: There must be some way to retrieve the data the pointer is referring to.

In a nutshell, a pointer is the address of data, rather than the data itself. For example, your debit card does not actually hold any money in it (data), instead it holds your account number (pointer to data). Today we will discuss the syntax of pointers and using pointers as parameters in functions. During the course of our discussion, we will also reveal a little secret of arrays: they are accessed using pointers!

Syntax

Pointer syntax can be reduced to just three parts: declaring a pointer with the `*`, using the address-of operator (`&`) to derive an address of a variable, and the dereference operator (`*`) to retrieve the data from a given address.



```
{
   int speed = 65;
   int * pSpeed;

   pSpeed = &speed;

   cout << *pSpeed;
}
```

`speed`

Every pointer needs to point to something. In this case, the pointer will point to `speed`.

`int * pSpeed`

The data-type of pointer is "pointer to an integer." Notice that there are two parts to the declaration: the type it is pointing to "`int`" and the fact that it is a pointer "`*`".

Because the variable does not hold data but rather an address, it is helpful to name it differently. In this case `pSpeed` means "pointer to `speed`."

`&speed`

To get the address of speed, we use the address-of operator "`&`". Since the data-type of speed is `int`, the data-type of `&speed` is "`int *`" or pointer to `int`.

`*pSpeed`

Use the dereference operator "`*`" to retrieve the data that `pSpeed` points to.

Declaring a pointer

When declaring a normal data variable, it is necessary to specify the data-type. This is required so the compiler knows how to interpret the 1's and 0's in memory and how to evaluate expressions. Pointers are no different in this regard, but there is one extra degree of complexity. Not only is it necessary to specify the data-type of the data that is pointed to, it is also necessary to identify the fact that the variable is a pointer. Therefore, pointer declaration has two parts: the data-type and the `*`.

```
<data-type> * <pointer variable>;
```

The following is an example of a pointer to a float:

```
float * pGPA;
```

The first part of the declaration is the data-type we are pointing to (`float`). This is important because, after we dereference the pointer, the compiler needs to know what type of data we are working with.

Sue's Tips

We need to remember to treat pointers differently than data variables because we need the dereference operator (`*`) when accessing data. To help remember, always prefix a pointer variable with a `p`. This will make it less likely to confuse a pointer variable with a regular data variable.

Getting the address of a variable

Every variable in C++ resides in some memory location. With the address-of operator (`&`), it is possible to query a variable for its address at any point in the program. The result is always an address, but the data-type depends on the type of the data being queried. Consider the following example:

```
{
   // a bunch of normal data variables
   int    valueInteger;                    // integer
   float  valueFloatingPoint;              // floating point number
   bool   valueBoolean;                    // Boolean
   char   valueCharacter;                  // character

   // a bunch of pointer variables
   int   * pInteger;                       // pointer to integer
   float * pFloatingPoint;                 // pointer to a floating point number
   bool  * pBoolean;                       // pointer to a Boolean value
   char  * pCharacter;                     // pointer to a character

   // assignments
   pInteger       = &valueInteger;         // both sides of = are pointers to integers
   pFloatingPoint = &valueFloatingPoint;   // both sides are pointers to floats
   pBoolean       = &valueBoolean;         // both sides are pointers to Bools
   pCharacter     = &valueCharacter;       // both sides are pointers to characters
}
```

In the first assignment (`pInteger = &valueInteger`), the data-type of `valueInteger` is `int`. When the address-of operator is added to the expression, the data-type becomes "pointer to `int`." The left-side of the assignment is `pInteger` which is declared as a "pointer to an `int`." Since both sides are the same data-type (pointer to an `int`), then there is not a problem with the assignment. If we tried to assign `&valueInteger` to `pFloatPoint`, we would get the following compile error:

```
example.cpp: In function "int main()":
example.cpp:20: error: cannot convert "int*" to "float*" in assignment
```

Sam's Corner

All pointers are the same size, regardless of what they point to. This size is the native size of the computer. If, for example, you are working on a 32 bit computer, then a pointer will be four bytes in size:

```
assert(sizeof(int  *) == 4);       // only true for 32 bit computers
assert(sizeof(char *) == 4);       // all addresses are the same size
```

However, if the computer was 64 bits, then the pointer would be eight bytes in size:

```
assert(sizeof(int  *) == 8);       // only true for 64 bit computers
assert(sizeof(char *) == 8);       // again, what they point to does not matter
```

While this may seem counterintuitive, it is actually quite logical. The address to my college apartment in GPS coordinates was exactly the same size as the address to a nearby football stadium. The addresses were the same size, even though the thing they pointed to were not!

Retrieving the data from a pointer

We can always retrieve the data from a given address using the dereference operator (*). For this to be accomplished the compiler must know the data-type of the location in memory and the address must be valid. Since the data-type is part of the pointer declaration, the first part is not a problem. It is up to the programmer to ensure the second constraint is met. In other words, the compiler ensures that a data variable is always referring to a valid location in memory. However, with pointers, the programmer needs to set up the value. Consider the following example:

```
{
   int speed = 65;                   // the location in memory we will be pointing to
   int * pSpeed;                     // currently uninitialized. Don't dereference it!

   pSpeed = &speed;                  // now it is initialized to the address of speed

   cout << *pSpeed << endl;          // need to use the * to get the data
}
```

If we removed the dereference operator (*) from the `cout` statement: `cout << pSpeed << endl;`, then we would pass a "pointer to an integer" to `cout`. This would display not the value 65, but rather the location where that value exists in memory:

```
0x7fff9d235d74
```

If we skipped the initialization step in the above code (`pSpeed = &speed`), then the variable `pSpeed` would remain un-initialized. Thus, when we dereference it, it would refer to a location in memory (segment) the program does not own. This would cause a segmentation fault (a.k.a "crash") at run-time:

```
Segmentation fault (core dumped)
```

Example 3.3 – Variable vs. Pointer

Demo

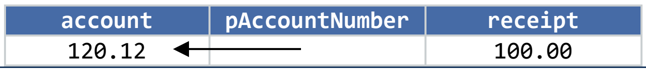
This example will demonstrate the differences between working with pointers and working with variables. Pointers do not hold data so they can only be said to share values with other variables. Variables, on the other hand, make copies of values.

Solution

In the following example, we will have a standard variable called `account` which stores my current account balance. We will also have a pointer to my account called `pAccountNumber`. Observe how we add the ‘`p`’ prefix to pointer variables to remind us they are special. Finally, we will modify the account balance both through manipulating the `account` variable and the `pAccountNumber` variable.

```
{
   // Standard variable holding my account balance. I opened the account with
   // birthday money from granny (I love granny!).
   double account = 100.00;

   // Visiting the ATM, I get a receipt of my current account balance ($100.00)
   double receipt = account;

   // Pointer variable not currently pointing to anything. I have just declared
   // it, but it is still not initialized. We declare a pointer variable by
   // specifying the data type we are pointing to and the special ‘*’ character
   double * pAccountNumber;

   // Now my account number refers to the variable account. We do this by
   // getting the address of the account variable. This is done with the
   // address-of operator ‘&’
   pAccountNumber = &account;

   // From here I can either access the account variable directly…
   account += 0.12;  // interest from the bank

   // … or I can access it through the pAccountNumber pointer. In this case, I
   // went to the ATM machine and added $20.00. Observe how I can access the
   // data of a pointer with the dereference operator ‘*’
   *pAccountNumber += 20.00;

   // Now I will display the differences
   cout << "Receipt:   $" << receipt << endl;    // the old value: $100.00
   cout << "Balance:   $" << account << endl;    // updated value: $120.12
}
```

The receipt is a standard variable, not changing to reflect the latest copy of the variable. My debit card contains my account number, a pointer! Thus going to the ATM machine (dereferencing the account number pointer) always gets me the latest copy of my account balance.



Challenge

As a challenge, change the value `receipt` to reflect a modest “adjustment” to your account.

```
recipt = 1000000.00;        // I wish this actually worked!
```

Notice how it does not influence the value in the variable `account`. Only pointers can do that!

See Also

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

```
/home/cs124/examples/3-3-variableVsPointer.cpp
```

Unit 3

Example 3.3 – Pointer to Master Copy

Demo

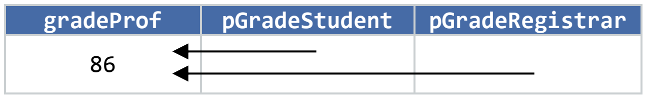
This example will demonstrate how more than one pointer can refer to a single location in memory. This will allow multiple variables to be able to make updates to a single value.

Problem

Professor Bob keeps the "master copy" of everyone's grades. He also encourages his students to keep track of their grades so they know how they are doing. Finally, the registrar needs to have access to the grades. To make sure that the various versions remain in sync, he distributes pointers instead of copies. The student got an 86 for his grade. Later, the professor noticed a mistake and updates the grade to 89.

Solution

In this example, there are three variables: an integer representing the professor's master copy of the grade (`gradeProf`), a pointer representing the student's ability to access the grade (`pGradeStudent`), and a pointer representing the registrar's copy of the grades (`pGradeRegistrar`).



The code for this examples is:

```
{
   // initial grade for the student
   int gradeProf = 86;                          // start with a normal data variable

   // two people have access
   const int * pGradeStudent   = &gradeProf;  // a const so student can't change
   const int * pGradeRegistrar = &gradeProf;  // registrar can't change it either

   // professor updates the grade.
   gradeProf = 89;

   // report the results
   cout << *pGradeRegistrar
        << endl;
   cout << *pGradeStudent
        << endl;
}
```

Observe how neither `pGradeRegistrar` nor `pGradeStudent` actually contain a grade; they only contain the address of the professor's grade. Thus, when `gradeProf` changes to 89, they will reflect the new value when `*pGradeRegistrar` and `*pGradeStudent` are dereferenced. This is one of the advantages of using pointers: sharing.

Challenge

You may notice how `pGradeStudent` and `pGradeRegistrar` both have the const prefix. This guarentees that `*pGradeStudent` won't inadvertently change gradeProf. As a challenge, try to change the value with the following code:

```
*pGradeStudent = 99;                 // Yahoo! Easy 'A'!
```

Now, remove the `const` prefix from the `pGradeStudent` declaration. Does the compile error go away?

See Also

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

```
/home/cs124/examples/3-3-pointerToMasterCopy.cpp
```

Example 3.3 – Point to the "Right" Data

Demo

This example will demonstrate how to conditionally assign a pointer to the "right" data. In this case, the pointer will either refer to one variable or another, depending on a condition.

Problem

Imagine Sam and Sue are on a date and they are trying to figure out who should pay. They decide that whoever has more money in their account should pay. They use that person's debit card. Recall that the debit card points to the person's account. In other words, a debit card does not contain actual money. Instead, it carries the account number or a pointer to the account:

Solution

We have two variables here (accountSam and accountSue). We will also have a pointer which will refer to one of the accounts or the other depending on the current balance:

accountSam	pAccount	accountSue
120.01	──→	398.21

In this case, `pAccount` will point to Sue's account because she happens to have more money today.

```
{
   // start off with money in the accounts
   float accountSam = 120.01;       // original account of Sam
   float accountSue = 398.21;       // Sue does a better job saving
   float * pAccount;                // uninitialized pointer

   // who will pay...
   if (accountSam > accountSue)     // warning: do not try this on an actual date
      pAccount = &accountSam;       // the & gets the address of the account.
   else                             //     This is much like a debit card number
      pAccount = &accountSue;

   // use the debit card to pay
   float priceDinner = 21.65;       // not an expensive dinner!
   *pAccount -= priceDinner;        // remove price of dinner from Sue's account
   *pAccount -= priceDinner * 0.15; // don't forget the tip

   // report
   cout << accountSam << endl;      // since pAccount points to accountSue,
   cout << accountSue << endl;      //     only accountSue will have changed
}
```

Challenge

Just before walking into the restaurant, Sam remembered that he has a saving account that his mother told him was "only for a rainy day." With Sue pulling out her debit card, the skies are definitely looking cloudy!

```
float accountSamsMother = 562.09;   // for emergency use only!
```

Modify the above program to make it work with a third account.

See Also

The complete solution is available at 3-3-pointerToRightData.cpp or:

```
/home/cs124/examples/3-3-pointerToRightData.cpp
```

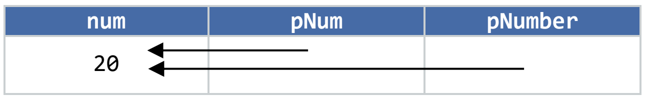
Pointers and functions

In the C programming language (the predecessor to C++), there is no pass-by-reference parameter passing mechanism. As you might imagine, this is quite a handicap! How else can you return more than one variable from a function if you cannot use pass-by-reference? It turns out, however, that this is no handicap at all. In C, we use pass-by-pointer instead.

Passing a pointer as a parameter to a function enables the callee to have the same access to the value as the caller. Any changes made to the dereferenced pointer are reflected in the caller's value. Consider the following example:

```
/************************************
 * SET TWENTY
 * Change the value to 20
 ************************************/
void setTwenty(int * pNumber)             // pass-by-pointer
{
   *pNumber = 20;                         // by changing the dereference value, we
}                                         //      change who pNumber points to: num

/************************************
 * MAIN
 * Simple driver program for setTwenty
 ***********************************/
int main()
{
   // the value to be changed
   int num = 10;                           // the value to be changed
   int * pNum = &num;                      // get the address of num

   // the change is made
   setTwenty(pNum);                        // this could also be:  setTwenty(&num);

   // display the results
   cout << num << endl;                    // num == 20 due to setTwenty()
   return 0;
}
```

How does this work? Consider the following diagram:



The variable `num` in `main()` starts at the value `10`. Also in `main()` is `pNum` which points to `num`. This means that any change made to `*pNum` will be reflected in `num`. When the function `setTwenty()` is called, the parameter `pNumber` is passed. This variable gets initialized with the same address that `pNum` sent. This means that both `pNum` and `pNumber` contain the same address. Thus, any change made to `*pNumber` will be the same as making a change to `*pNum` and `num`. Therefore, when we set `*pNumber = 20` in `setTwenty()`, it is exactly the same thing as setting `num = 20` in `main()`.

Pass-by-pointer

Up to this point in time, we have had two ways to pass a parameter: **pass-by-value** which makes a copy of the variable so the callee can't change it and **pass-by-reference** which links the variable from the callee to that of the caller so the callee can change it. Now, with pointers, we can use pass-by-pointer. **Pass-by-pointer** is sending a copy of the address of the caller's variable to the callee. With this pointer, the callee can then make a change that will be reflected in the caller's data. To illustrate this point, consider the following code:

```
/**********************************************************************
 * FUNCTION
 * Demonstrate pass by value, pass by reference, and pass by pointer
 **********************************************************************/
void function(int value, int &reference, int * pointer)
{
   cout << "value:     " << value     << " &value:     " << &value     << endl;
   cout << "reference: " << reference << " &reference: " << &reference << endl;
   cout << "*pointer:  " << *pointer  << " pointer:    " << pointer    << endl;
}

/***********************************************************************
* Just call a function. No big deal really.
***********************************************************************/
int main()
{
   int number;
   cout << "Please give me a number: ";
   cin  >> number;
   cout << "number:    " << number
        << "\t&number: " << &number
        << endl << endl;


   function(number  /*by value    */,
            number  /*by reference*/,
            &number /*by pointer  */);


   return 0;
}
```

This program is available at 3-3-passByPointer.cpp or `/home/cs124/example/3-3-passByPointer.cpp`. The output of this program is:

```
Please give me a number: 100
number:    100  &number: 0x7fff5fcbf07c

value:     100 &value:     0x7fff5fcbf54c
reference: 100 &reference: 0x7fff5fcbf07c
*pointer:  100 pointer:    0x7fff5fcbf07c
```

Observe the value of number in `main()` (`100`) and the address in main (`0x7fff5fcbf07c`). The first parameter is **pass-by-value.** Here, we would expect the value to be copied which it is. Since pass-by-value creates a new variable that is independent of the caller's variable, we would expect that to be in a different address. Observe how it is. The address of `value` is `0x7fff5fcbf54c`, different than that of `number`.

The second parameter is **pass-by-reference**. The claim from Chapter 1.4 was that the caller's variable and the callee's variable are linked. Now we can see that the linking is accomplished by making sure both refer to the same location in memory. In other words, because they have the same address (`0x7fff5fcbf07c`), any change made to `reference` should also change `number`.

The final parameter is **pass-by-pointer**. A more accurate way of saying this is we are passing a pointer by value (in other words, making a copy of the address). Since the address is duplicated in the `pointer` variable, it follows that the value of `*pointer` should be the same as that of `number`.

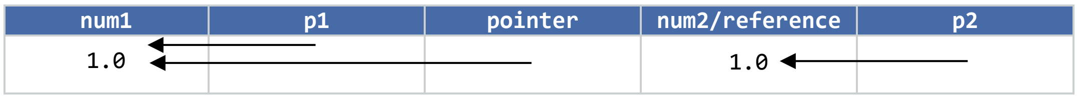
The only difference between pass-by-pointer and pass-by-reference is notational. With pass-by-pointer, we need to use the address-of operator (`&`) when passing the parameter and the dereference operator (`*`) when accessing the value. With pass-by-reference, we use the ampersand (`&`) in the callee's parameter. Aside from these differences, they behave exactly the same. This is why the C programming language does not have pass-by-reference: it doesn't need it!

Pitfall: Changing pointers

As mentioned above, pass-by-pointer is the same thing as passing an address by value. This means that we can change what the pointer is referring to in the function, but we cannot change the pointer itself. To illustrate, consider the following function:

```
/************************************
 * CHANGE POINTER
 * Change what pointer is referring to
 ***********************************/
float *changePointer(float * pointer,     // pass-by-pointer. Initially &num1 or p1
                     float &reference)    // pass-by-reference. Tied to num2
{
   *pointer = -1.0;                       // can change *pointer. This will then change
                                          //     num1 in main().
   pointer = &reference;                  // this function's copy of p1 will change. It
                                          //     will not change p1 in main because the
                                          //     address was pass-by-value
   return pointer;                        // send reference's address (the same address
}                                         //     as num2) back to the caller.

/************************************
 * MAIN
 * Simple driver program for changePointer
 ***********************************/
int main()
{
   float num1 = 1.0;
   float num2 = 2.0;

   float *p1 = &num1;                     // p1 gets the address of num1
   float *p2;                             // p2 is initially uninitialized

   p2 = changePointer(p1, num2);          // because a copy of p1 is sent, it does not
                                          //     change. However, the copy's value is
                                          //     returned which will change p2
   assert(*p1 == -1.0);                   // changed in changePointer() to -1.0
   assert(p1 == &num1);                   // not changed since it was initialized
   assert(p2 == &num2);                   // assigned when function returned to the
                                          //     address of num2

   return 0;
}
```

Notice how the values `num1` and `num2`, as well as the the pointers `p1` and `p2` are in `main()` while `pointer` and `reference` are in `changePointer()`. Since `reference` is pass-by-reference to `num2`, they share the same slot in memory. The variable `pointer`, on the other hand, refers to `num1` by pass-by-pointer.



Sam's Corner

If we want to change a pointer parameter in a function (not what it points to), we have three options:

- Return the value and have the caller make the assignment (`float * change();`)
- Pass a pointer to a pointer. This is called a handle. (`void change(float ** handle);`)
- Pass the pointer by reference. (`void change(float * &pointer);`)

Arrays are pass-by-pointer

As you may recall, arrays are pointers. Specifically, the array variable points to the first item in the range of memory locations. It thus follows that passing an array as a parameter should be pass-by-pointer. In fact, it is. This is why we said in Chapter 3.0 that passing an array is *like* pass-by-reference. The reason is that it is actually pass-by-pointer. Consider the following example:

```
/***************************
 * INITIALIZE ARRAY
 * Set the values of an array
 * to zero
 ***************************/
void initializeArray(int * array,              // we could say int array[] instead, it
                     int size)                 //     means basically the same thing
{
   for (int i = 0; i < size; i++)              // standard array loop
      array[i] = 0;                            // use the [] notation even though we
}                                              //     declared it as a pointer

/***************************
 * MAIN
 * Simple driver program
 ***************************/
int main()
{
   const int SIZE = 10;                        // const variables are ALL_CAPS
   int list[SIZE];                             // can be declared as a CONST
   assert(SIZE == sizeof(list) / sizeof(*list));// *list is the same as list[0]

   // call it the normal way
   initializeArray(list, SIZE);                // call the function the normal way

   // call it with a pointer
   int *pList = list;                          // list is a pointer so this is OK
   initializeArray(pList, SIZE);               // exactly the same as the first time
                                               //       we called initializeArray
   return 0;
}
```

The square bracket `[]` notation (as opposed to the pointer `*` notation) is convenient because we can forget we are working with pointers. However, they are just a notational convenience. We can remove them and work with pointers to get a more clear indication of what is going on. This program is available at:

```
/home/cs124/examples/3-3-arrays.cpp
```

Problem 1

What is the output of the following code?

```
{
   float accountSam = 100.00;
   float accountSue = 200.00;
   float * pAccount1 = &accountSam;
   float * pAccount2 = &accountSue;
   float * pAccount3 = &accountSam;
   float * pAccount4 = &accountSue;

   *pAccount1 += 10.00;
   *pAccount2 *=  2.00;
   *pAccount3 -= 15.00;
   *pAccount4 /=  4.00;

   cout << accountSam << endl;
}
```

Answer:

____

Please see page 258 for a hint.

Problem 2

What is the output of the following code?

```
{
   int  a = 10;
   for (int * b = &a; *b < 12; (*b)++)
      cout << ".";
   cout << endl;
}
```

Answer:

____

Please see page 258 for a hint.

Problem 3

What is the output of the following code?

```
{
   int   a = 16;
   int * b = &a;
   int   c = *b;
   a  = 42;
   int * d = &c;
   a  = *b;
   d  = &a;
   *d = 99;
   cout << "*b == " << *b <<
endl;
}
```

a	b	c	d

Answer:

____

Please see page 258 for a hint.

Problem 4

What is the output of the following code?

```
{
   int   a = 10;
   int * b = &a;

   while (*b > 5)
      (*b)--;

   cout << "Answer: "
        << a
        << endl;
}
```

Answer:

____

Please see page 258 for a hint.

Problem 5

What is the output of the following code?

```
void funky(int * a)
{
   *a = 8;
   return;
}

int main()
{
   int b = 9;
   funky(&b);
   cout << b << endl;

   return 0;
}
```

Answer:

____

Please see page 262 for a hint.

Assignment 3.3

Write a program to ask two people for their account balance. Then, based on who has the most money, all subsequent purchases will be charged to his or her account.

Example

User input is **underlined**:

```
What is Sam's balance? 229.12
What is Sue's balance? 241.45
Cost of the date:
	Dinner:    32.19
	Movie:     14.50
	Ice cream: 6.00
Sam's balance: $229.12
Sue's balance: $188.76
```

Note that there is a tab before "`Dinner`," "`Movie`," and "`Ice cream`." There are spaces after the colons.

Challenge

As a challenge, try to write a function to reduce the value of the debit card according to the cost of the date:

```
void date(float *pAccount);
```

This function will contain the three prompts (Dinner, Movie, and Ice Cream) and reduce the value of `pAccount` by that amount.

Assignment

Write the program to store the two account balances. Create a pointer to the account that has the largest balance. Then, for the three item costs on the date, reduce the balance of the appropriate account using the pointer. The testbed is:

```
testBed cs124/assign33 assignment33.cpp
```

Don't forget to submit your assignment with the name "Assignment 33" in the header.

Please see page 261 for a hint.

3.4 Pointer Arithmetic

The bishop asked Sam to deliver a stack of flyers to all the apartments in his complex. Remembering that an apartment address is like a pointer (where the dereference operator takes you to the apartment) and the apartment block is like an array (a contiguous set of addresses referenced by the first address), Sam realizes that this is a pointer arithmetic problem. He starts at the first address (the pointer to the first item in the array), delivers the flyer (dereferences the pointer with the `*`), and increments the address (adds one to the pointer with `p++`). This is continued (with a FOR loop) until the last address is reached.

Objectives

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

- Understand how to advance a pointer variable with the ++ operator.
- Traverse a string with the second standard FOR loop.
- Explain the difference between a constant pointer and a pointer to a constant value.

Prerequisites

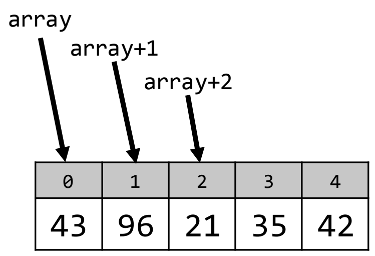
Before reading this section, please make sure you are able to:

- Declare a pointer variable (Chapter 3.3).
- Point to an existing variable in memory with a pointer variable (Chapter 3.3).
- Get the data out of a pointer (Chapter 3.3).
- Pass a pointer to a function (Chapter 3.3).
- Understand the role the null character plays in string definitions (Chapter 3.2).
- Write a loop to traverse a string (Chapter 3.2).

Overview

Pointer arithmetic is the process of manipulating pointers to better facilitate accessing arrays of data in memory. Though the term "arithmetic" implies that a whole range of arithmetic operations can be performed, we are normally restricted to addition and subtraction.

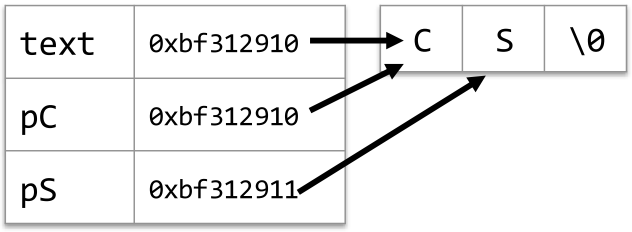
Recall from Chapter 3.3 that a pointer is a reference to a location in memory. We typically do not know where this memory is located until run-time; the operating system places the program in memory and often puts it in a different location every time. Recall from Chapter 3.0 that arrays are collections of data guaranteed to reside in contiguous blocks of memory. From these two observations it should be clear that, given some `array[i]`, the location `array[i + 1]` should be in the adjacent location in memory. Pointer arithmetic is the process of levering this proximity to access array data.



Consider the following code:

```
{
   char text[] = "CS";     // some buffer in memory
   char * pC = text;       // text refers to the first item, so pC does as well
   char * pS = text + 1;   // the next location in memory is one item beyond the first
}
```

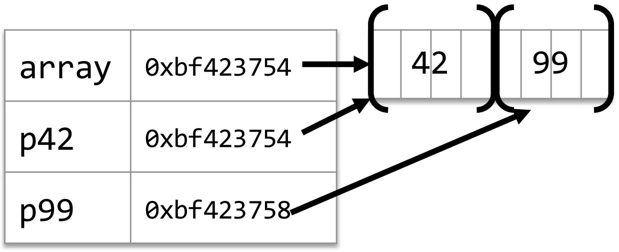
In this example, `text` points to the string "`CS`" or, more explicitly, to the first letter of the string. Next, the variable `pC` inherits its address from `text` which points to '`C`' (hence the name `pC`). Finally, since the letter '`S`' is one letter beyond '`C`' in the string, it follows it has an address one greater than the '`C`'. Thus, we can find the address of '`S`' by taking the address of '`C`' (as specified by the variable `text`) and adding one:



It turns out that integer pointers work the same way. The only difference is that integers are 4 bytes in length where characters are one. However, the pointer arithmetic is the same:

```
{
   int array[] =
   {
      42, 99                     // two numbers using 8 bytes of memory
   };
   int * p42 = array;            // just like characters, points to the first item
   int * p99 = array + 1;        // add one to move forward four bytes!
}
```

Just like the first example, `array` points to the first number in the list. Next, the variable `p42` has the same value (the address of `42`) as array. Finally, since the number `99` is next to the number `42`, it follows it will have an address one greater. Thus, we can find the address of `99` by taking the address of `42` and adding one.



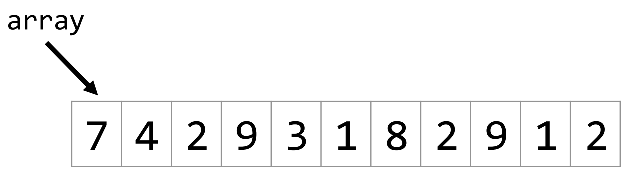
Because arrays (including strings) are just pointers, it is often most convenient to traverse an array with a pointer than with an index. This involves incrementing the pointer variable rather than an index.

Arrays

As discussed before, arrays are simply pointers. This gives us two different notations for working with arrays: the square bracket notation and the star notation. Consider the following array:

```
int array[] =
{
   7, 4, 2, 9, 3, 1, 8, 2, 9, 1, 2
};
```

This can be represented with the following table:



Consider the first element in an array. We can access this item two ways:

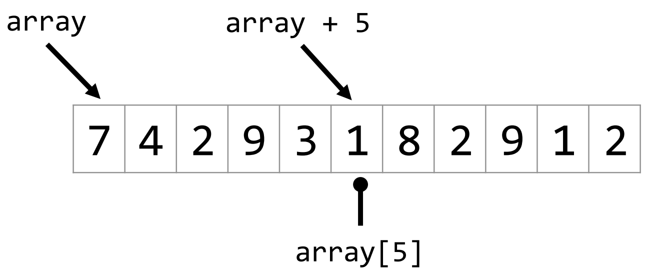
```
cout << "array[0] == " << array[0] << endl;
cout << "*array   == " << *array   << endl;
assert(array[0] == *array);
```

The first output line will of course display the value `7`. We learned this from Chapter 3.0. The second will dereference the `array` pointer, yielding the value it points to. Since pointers to arrays always point to the first item, this too will give us the value `7`. In other words, there is no difference between `*array` and `array[0]`; they are the same thing!

Similarly, consider the 6th item in the list. We can access with:

```
cout << "array[5]     == " << array[5]     << endl;
cout << "*(array + 5) == " << *(array + 5) << endl;
assert(array[5] == *(array + 5));
```

This is somewhat more complicated. We know the 6th item in the list can be accessed with `array[5]` (remembering that we start counting with zero instead of one). The next statement (with `*(array + 5)` instead of `array[5]`) may be counterintuitive. Since we can point to the 6th item on the list by adding five to the base pointer (`array + 5`), then by dereferencing the resulting pointer we get the data:



Therefore we can access any member of an array using either the square bracket notation or the star-parentheses notation.

Sam's Corner

It turns out that the square bracket array notation (`[]`) is actually a macro expansion for an addition and a dereference. Consider the following code:

```
cout << array[5] << endl;
```

We have already discussed how this is the same as adding five to the base pointer and dereferencing:

```
cout << *(array + 5) << endl;
```

Now, from the commutative property of addition, we should be able to re-order the operands of an addition without changing the value:

```
cout << *(5 + array) << endl;
```

From here, it gets a bit dicey. The claim is that the addition and dereference operator combined are the same as the square bracket operator. If this is true (and it is!), then the following should be true:

```
cout << 5[array] << endl;
```

Thus, under the covers, arrays are just pointers. The square brackets are just used to make it more intuitive and easy to understand for novice programmers. Please see the following code for an example of how this works at 3-4-notationAbuse.cpp or:

```
/home/cs124/examples/3-4-notationAbuse.cpp
```

Pointers as Loop Variables

Up to this point, all the loops we have written to access individual members of a string or array have used index variables and the square-bracket notation. It turns out that we can write an equivalent pointer-loop for each index-loop. These loops tend to perform better than their index counterparts because fewer assembly instructions are required. The two main applications for pointers as loop variables are array traversing loops and string traversing loops.

Array traversing loop

Recall that the standard way to use a `for` loop to walk through an array is:

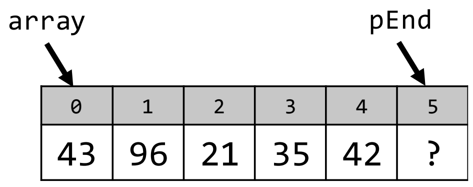
```
for (int i = 0; i < num; i++)
   cout << array[i];
```

It turns out we can use a pointer to loop through an array of integers if the length of the array is known.

Consider the following array:

```
   int array[] =
   {
      43, 96, 21, 35, 42
   };
```

In this example, the pointer to the beginning of the list is `array` and the pointer to the item off the end of the list is `array + num`:



This allows us to write a loop to walk through the list:

```
   int * pEnd = array + num;
   for (int * p = array; p < pEnd; p++)
      cout << *p << endl;
```

Observe how, with each iteration, the pointer variable `p` advances by one address. This continues until `p` is no longer less-than the item off the end of the list `pEnd`. Since we are working with arrays, we can dereference each item in the list with `*p`. For an example of this loop in action, please see 3-4-pointerArray.cpp or:

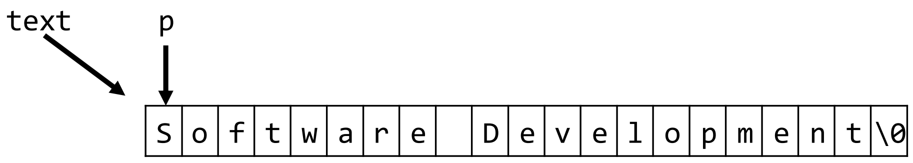
```
/home/cs124/examples/3-4-pointerArray.cpp
```

String traversing loop

With strings, the end of the string is defined as the null-character. This leads us to the second standard `for` loop: traversing a string with a pointer.

```
for (char * p = text; *p; p++)
   cout << *p;
```

Just like with the aforementioned array example, we advance the pointer rather than an index into the string. The big difference is the controlling Boolean expression: a null-terminator check rather than looking for a pointer to the end of the string.



In this example, `text` points to the first item in the string ('S'). The loop starts by assigning `p` to also point to the first item. The loop continues by advancing `p` through the string. The loop terminates when `*p` is no longer `true`. This occurs when `p` points to the null-character.

Example 3.4 – String Traverse

Demo

This example will demonstrate how to traverse a string using the pointer notation. It will use the second standard FOR loop:

```
for (char * p = text; *p; p++)
   ;
```

Problem

Write a function to display the contents of a string, one character on each line:

```
Please enter the text: Point
        P
        o
        i
        n
        t
```

Solution

The function will take a pointer to a character as a parameter. Recall that arrays are pointers to the first item in the list. The same thing is true with strings. String variables are actually pointers to the first character. Thus the prototype is:

```
void display(const char * text);
```

From here, we will use the second standard FOR loop to iterate through each item in the string. Recall that the dereference operator `*` is needed to retrieve the data from the string.

```
/*************************************
 * DISPLAY
 * Display the passed text one character
 * at a time using a pointer
 ************************************/
void display(const char * text)
{
   // second standard for loop
   for (const char *p = text;         // p will point to each item in the string
        *p;                           // as long as *p is not the NULL character
        p++)                          // increment one character at a time
   {
      cout << '\t' << *p << endl;     // access each item with the dereference *
   }
}
```

Challenge

As a challenge, can you modify the program so the function displays every other character in the string? What happens when there are an odd number of characters in the string? How can you detect that condition so the program does not malfunction?

See Also

The complete solution is available at 3-4-traverse.cpp or:

```
/home/cs124/examples/3-4-traverse.cpp
```

Example 3.4 – Convert Case

Demo

This example will demonstrate how to walk through a loop using the pointer notation. In this case, processing is performed on every character in the string.

Problem

Write a program to convert uppercase characters to lowercase and vice-versa.

```
Please enter your text: sOFTWARE eNGINEERING
Software Engineering
```

Solution

There are two components to this problem. The first is to use the second standard FOR loop to walk through the string. The second part is to use the pointer notation instead of the array notation to reference each member of the string.

Array notation

```
void convert(char text[])
{
   for (int i = 0; text[i]; i++)
      if (isupper(text[i]))
         text[i] = tolower(text[i]);
      else
         text[i] = toupper(text[i]);
}
```

Pointer notation

```
void convert(char * text)
{
   for (char * p = text; *p; p++)
      if (isupper(*p))
         *p = tolower(*p);
      else
         *p = toupper(*p);
}
```

Notice how the array notation uses the square bracket `[]` notation to declare the function parameter while the pointer notation uses the `*`. Both notations mean mostly the same thing, "a pointer to the first item in the string."

In the array notation solution, the individual items in the string are referenced with `text[i]`. With the pointer notation, we use the `*p` notation. Both produce the same results but using a different mechanism. In general, the pointer notation is preferred because it is simpler and more efficient.

Challenge

As a challenge, try to modify the above program to count the number of uppercase letters in the input stream. The prototype is:

```
int countUpper(const char *text);
```

See Also

The complete solution is available at 3-4-toggleCase.cpp or:

```
/home/cs124/examples/3-4-toggleCase.cpp
```

Example 3.4 – String Length

Demo

This example will demonstrate how to walk through a string using the pointer notation.

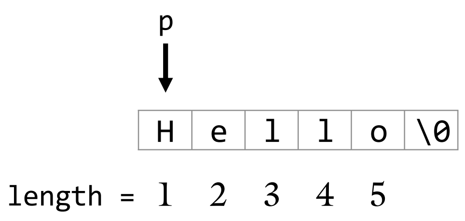
Problem

Write a program to find the length of a string. This will be done using the array notation, the pointer notation using the second standard FOR loop, and the optimal solution:

```
Please enter your text: Software
        Array notation:   8
        Pointer notation: 8
        Optimal version:  8
```

Solution

The most straight-forward way to do this is to walk through the string using the second standard FOR loop. Here, the pointer `p` will serve as the loop control variable. Inside the body of the loop, we will have a `length` variable incrementing with each iteration.



Since we only increment `length` when we have not yet encountered the null character, the value at the end of the loop should be the length of the string.

```
/************************************
 * STRING LENGTH : pointer version
 * Increment the length variable with
 * every iteration
 ************************************/
int stringLength(char * text)
{
   int length = 0;                          // declared out of FOR loop scope

   for (char * p = text; *p; p++)           // second standard FOR loop
      length++;                             // increment length every iteration

   return length;                           // return the length variable
}
```

Notice how two variables (`length` and `p`) increment with each iteration of the loop. A more efficient solution would not have this redundancy. Please see the video and associated code for the better solution.

See Also

The complete solution is available at 3-4-stringLength.cpp or:

```
/home/cs124/examples/3-4-stringLength.cpp
```

Example 3.4 – String Compare

Demo

This example will demonstrate how to iterate through two strings at the same time. This will be done using the array notation with an index, the pointer notation using the second standard FOR loop, and an optimal solution.

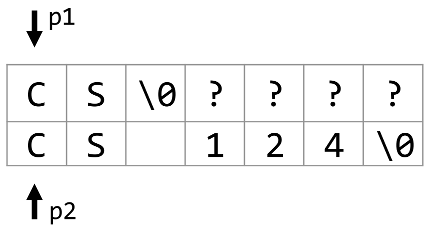
Problem

Write a program to determine if two strings are the same. Note that the equivalance operator alone is insufficient (`text1 == text2`) because it will just compare two addresses. We also cannot use the dereference operator alone (`*text1 == *text2`) because it will just compare the first two items in the string. It will be necessary to compare each item in both strings using a loop.

```
Please enter text 1: Software
Please enter text 2: Software
        Array notation:   Equal
        Pointer notation: Equal
        Optimal version:  Equal
```

Solution

To solve the problem, it is necessary to have two pointer variables moving in tandem through the two strings:



The hardest problem here is to know when to terminate the loop. There are three conditions that could terminate the loop:

1. When `*p1 != *p2` or when the currently considered character is different.
2. When the end of the first string is reached. `*p1 == '\0'`
3. When the end of the second string is reached. `*p2 == '\0'`

Thus, when either of these conditions are met, the loop terminates. If the first condition terminates the loop, then the strings are different. Otherwise, the strings are the same.

```
bool isEqual(char * text1, char * text2)
{
   char * p1;                         // for text1
   char * p2;                         // for text2

   for (p1 = text1, p2 = text2;       // two pointers require two initializations
       *p1 == *p2 &&                  // same logic as with array index version but
       *p1 && *p2;                    //    somewhat more efficient
        p1++, p2++)                   // both pointers need to be advanced
      ;

   return (*p1 == *p2);
}
```

See Also

The complete solution is available at 3-4-stringCompare.cpp or:

```
/home/cs124/examples/3-4-compareString.cpp
```

Constant Pointers

Earlier in the semester, we introduced the notion of the `const` modifier:

```
const int SIZE = 10;                           // SIZE can never change
```

The `const` modifier allows the programmer to specify (and the compiler to enforce!) that a given variable will never change. Along with this, we can also point to constant data. The most common time we do this is when passing arrays as parameters:

```
void displayName(const char * name);          // displayName() cannot change name
```

In this example, the function `displayName()` is not able to change any of the data in the string `name`.

Another class of constant data is a constant pointer. A constant pointer refers to a pointer that must always refer to a single location in memory. Consider the following code:

```
   int array[4];                        // declare an array of integers
   array[0] = 6;                        // legal. The data is not constant.
   array++;                             // ERROR!  We cannot change the variable 'array'!
```

When we declare an array, the array pointer will always refer to the same location of memory. We cannot perform pointer arithmetic on this variable.

It turns out that whenever we declare an array using the square bracket notation (`[]`), whether in a function parameter or as a local variable, that variable is a constant pointer. Consider the following code:

```
void function(      char * parameter1,      // can change both pointer and value
              const char * parameter2,      // can change pointer but not value
                    char   parameter3[],    // can change value but not pointer
              const char   parameter4[])    // can change neither value nor pointer
{
   // change the value
   *parameter1 = 'x';                  // legal because parameter 1 is not a const
   *parameter2 = 'x';                  // ERROR! parameter2 is a const!
   *parameter3 = 'x';                  // legal because parameter 3 is not a const
   *parameter4 = 'x';                  // ERROR! parameter4 is a const

   // change the pointer
   parameter1++;                       // legal; not a const pointer
   parameter2++;                       // legal; not a const pointer
   parameter3++;                       // ERROR! parameter3 is a const pointer
   parameter4++;                       // ERROR! parameter4 is a const pointer
}
```

When a parameter is passed with the square brackets or when an array is declared in a function, it is still possible to do pointer arithmetic. Consider the following example:

```
{
   int array[4];      // constant pointer.
   int * p = array;   // not a constant pointer!

   p++;               // we cannot do this with array. array++ would be illegal!
}
```

Even though `array` and `p` refer to the same location in memory, we can increment `p` but not `array` because `array` is a constant pointer whereas `p` is not.

Sam's Corner

The most common way to make a parameter a constant pointer is to use the `[]` notation in the parameter declaration. This, unfortunately, implies that the pointer is an array (which it may not be!). Another way to make a variable a constant pointer is to use the `const` modifier after the `*`:

```
void function(      int *                pointer,
              const int *                pointerToConstant,
                    int * const constantPointer,
              const int * const constantPointerToConstant);
```

This is the same as:

```
void function(      int *                pointer,
              const int *                pointerToConstant,
                    int         constantPointer[],                // note the []s
              const int         constantPointerToConstant[]);     // more []s
```

Problem 1

What is the output of the following code?

```
{
   char * a;
   char * b = a;
   char   c = 'x';
   char   d = 'y';
    b = &c;
    a = &d;
   *b = 'z';
   *a = *b;
   cout << "d == " << d << endl;
}
```

a	b	c	d

Answer:

____

Please see page 258 for a hint.

Problem 2

How much memory does each of the following reserve?

```
char text[2];
```

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

```
char text[] = "Point";
```

```
char * text[6];
```

Please see page 216 for a hint.

Problem 3

What is the output of the following code?

```
{
   char x[] = "Sam";
   char y[] = "Sue";
   char * z;

   if (x == y)
      z = x;
   else
      z = y;

   cout << *z << endl;
}
```

Answer:

____

Please see page 277 for a hint.

Problem 4

What is the output of the following code?

```
{
   char text[] = "Software";

   for (int i = 4;
        i < 7;
        i++)
     cout << text[i];

   cout << endl;
}
```

Answer:

____

Please see page 219 for a hint.

Problem 5

What is the output of the following code?

```
{
   char a1[10] = "42";
   char a2[10];

   int i;
   for (i = 0; a1[i]; i++)
      a2[i] = a1[i];
   a2[i] = '\0';

   cout << a2 << endl;
}
```

Answer:

____

Please see page 277 for a hint.

Unit 3

Problem 6

What is the output of the following code?

```
{
   char text[] = "Banana";

   char * pA = text + 2;

   cout << *pA << endl;
}
```

Answer:

____

Please see page 148 for a hint.

Problem 7

What is the output of the following code?

```
{
   int array[] = {1, 2, 3, 4};

   cout << *(array + 2) << endl;
   cout << array + 2    << endl;
   cout << array[3]     << endl;
   cout << *array + 3   << endl;
}
```

Answer:

Please see page 271 for a hint.

Problem 8

What is the output of the following code?

```
{
   char * a = "Software";
   char * b;

   while (*a)
      b = a++;

   cout << *b << endl;
}
```

Answer:

Please see page 49 for a hint.

Unit 3

Problem 9

What is the output of the following code?

```
{
   char text[] = "Software";

   int a = 0;
   for (char * p = text; *p; p++)
       a++;

   cout << a << endl;
}
```

Answer:

Please see page 49 for a hint.

Assignment 3.4

Start with Assignment 3.2. Modify `countLetters()` so that it walks through the string using pointers instead of array indexes. The output should be exactly the same as with Assignment 3.2.

Example

Two examples… The user input is **underlined**.

Example 1:

```
Enter a letter: z
Enter text: NoZ'sHere!
Number of 'z's: 0
```

Example 2:

```
Enter a letter: a
Enter text: Brigham Young University - Idaho
Number of 'a's: 2
```

Assignment

The test bed is available at:

```
testBed cs124/assign34 assignment34.cpp
```

Don't forget to submit your assignment with the name "Assignment 34" in the header.

Please see page 261 for a hint.

3.5 Advanced Conditionals

Sue is working on a program that has a simple menu-based user interface. The main menu has six options, each one associated with a letter on the keyboard. She could create a large IF/ELSE statement to implement this menu, but that solution seems tedious and inelegant. Fortunately she has just learned about SWITCH statements which seem like the perfect tool for the job.

Objectives

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

- Create a SWITCH statement to modify program flow.
- Create a conditional expression to select between two expressions.

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).
- Create an IF statement to modify program flow (Chapter 1.6).
- Recognize the pitfalls associated with IF statements (Chapter 1.6).

Overview

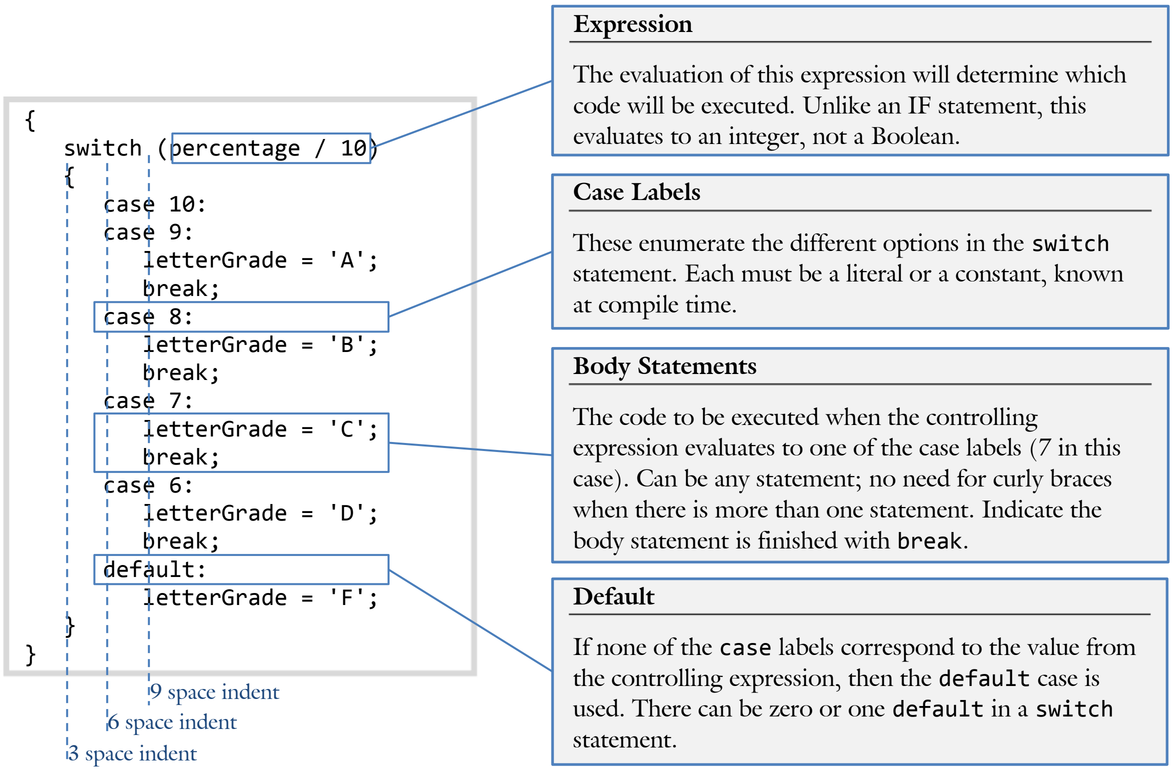
A fundamental component of any programming language is decision-making logic. Up to this point, the only decision-making mechanism we have learned is the IF statement. While this is a useful and powerful construct, it is actually rather limited: it will only help you choose between two options. We have worked around this restriction by nesting IF statements (Chapter 1.6) and using arrays (Chapter 3.0). This chapter will introduce three new decision making tools: SWITCH statements, conditional expressions, and bitwise operators.

Switch	Conditional Expression	Bitwise Operators

Alllows the programmer to choose between more than two options. Each option is specified with an integral value.
`switch(option)`
`{`
`   case 'Q':`
`      return true;`
`   case 'D':`
`      display(data);`
`      break;`
`   …`
`}`
	Useful for selecting between two expressions or values (rather than two statements as an IF statement does).
	`cout << "Greetings "`
	`     << (isMale ?`
	`         "Mr." : "Mrs.")`
	`     << lastName`
	`     << endl;`
		Enables a single integer to store 32 Boolean values worth of data for a more compact representation of an array of `bools`.
		`int value =`
		`  0x0001 |    // 1st bit`
		`  0x0004 |    // 3rd bit`
		`  0x0200;     // 10th bit`
		
		`if (value & 0x001)`
		`   cout << "1st bit";`

Switch

Most decisions in programming languages are made with IF statements. This works great if there are only two possible decisions. While it is possible to use multiple IF statements to achieve more than two possibilities (think of the tax function from Project 1), the answer is less than elegant at times. The SWITCH statement provides the ability to specify more than two possibilities in an elegant and efficient way.



```
{
   switch (percentage / 10)
   {
      case 10:
      case 9:
         letterGrade = 'A';
         break;
      case 8:
         letterGrade = 'B';
         break;
      case 7:
         letterGrade = 'C';
         break;
      case 6:
         letterGrade = 'D';
         break;
      default:
         letterGrade = 'F';
   }
}
```

Expression

The evaluation of this expression will determine which code will be executed. Unlike an IF statement, this evaluates to an integer, not a Boolean.

Case Labels

These enumerate the different options in the `switch` statement. Each must be a literal or a constant, known at compile time.

Body Statements

The code to be executed when the controlling expression evaluates to one of the case labels (7 in this case). Can be any statement; no need for curly braces when there is more than one statement. Indicate the body statement is finished with `break`.

Default

If none of the `case` labels correspond to the value from the controlling expression, then the `default` case is used. There can be zero or one `default` in a `switch` statement.

Expression

The SWITCH statement is driven by a controlling expression. This expression will determine which CASE will be executed. The best way to think of the controlling expression is like a switch operator for a railroad track. There are a finite number of tracks down which the operator can send a train, each identified by a track number. The operator's job is to match the physical track with the requested track number. Similarly, the controlling expression in a SWITCH statement determines which CASE label to execute.

The controlling expression must evaluate to an integer. Evaluation to a `bool`, `char`, or `long` works as well because each can be readily converted to an `int`. You cannot use a floating point number or pointer as the data-type of the controlling expression.

For example, consider the following code to implement a user interface menu. The following menu options are presented to the user:

```
Please enter an option:
        S ... Save the game
        N ... New game
        P ... Play the current game
        D ... Display the current status
        Q ... Quit
        ? ... Display these options again
```

The following code will implement the menu:

```
/******************************
 * EXECUTE COMMAND
 * Execute command as specified
 * by the caller
 ******************************/
void executeCommand(char option) // note that this is a char, not an integer
{
   switch (option)               // because a char readily converts to an integer, this
   {                             //    is not a problem
      case 'S':                  // ‘S’ corresponds to 83 on the ASCII table
         saveGame(game);         // execute the function corresponding to the ‘S’ input
         break;                  // finished with this case
      case 'N':
         newGame(game);
         break;
      …                          // there are more options here of course
    }                            // don’t forget the closing curly brace
}
```

The controlling expression can do more complex arithmetic. This is commonly the case when working with floating point numbers. In this second example, we are converting a GPA into a more workable medium:

```
/*******************************
 * DISPLAY GRADE
 * Display the letter grade version
 * of a GPA
 *******************************/
void displayGrade(float gpa)     // value is originally a float
{
   switch ((int)(gpa * 10.0))    // note how we cast to an integer
   {
      case 40:                   // corresponding to a 4.0 GPA
         cout << "A+";
         break;
      case 39:                   // corresponding to a 3.9 --> 3.3 GPA
      case 38:
      case 37:
      case 36:
      case 35:
      case 34:
      case 33:
         cout << "A";
         break;
      …
   }
}
```

When the controlling expression cannot be converted to an integer, then we will get a compile error:

```
{
   char name[] = "CS 124";
   switch (name)                 // ERROR! Pointers cannot be converted to an integer
   {
      …
   }
}
```

The problem here is that we cannot readily convert a pointer into an integer. Typically some non-trivial processing needs to happen before this can be done. Of course, the programmer could just cast the address into an integer, but that would probably be a bug!

Floating point numbers also cannot be used for the controlling expression. This makes sense when you realize that floating points are approximations.

```
{
   float pi = 3.14159;
   switch (pi)                     // ERROR! Floating point numbers cannot be converted
   {                               //        to an integer without casting
      …
   }
}
```

Case labels

Back to our train-track analogy, the CASE label in a SWITCH statement corresponds to a track number. Note that, aside from Harry Potter's wizarding world, there is no such thing as platform 9¾. Similarly, each CASE label must be an integer.

There is an additional constraint. In the C++ language, the compiler must know at compile time each CASE label. This means that the value must be a constant (`const int VALUE = 4;`) or a literal (`4`), it cannot be a variable. The final constraint is that each label must be unique. Imagine the confusion of the train operator trying to determine *which* "track 12" the train wants!

The first example corresponds to standard integer literal case numbers. In this case, we are converting a class number into a class name:

```
/********************************************
 * DISPLAY CLASS NAME
 * Convert a class number (124) into a
 * name (Introduction to Software Development)
 ********************************************/
void displayClassName(int classNumber)
{
   switch (classNumber)               // classNumber must be an integer
   {
      case 124:
         cout << "Introduction to Software Development\n";
         break;
      case 165:
         cout << "Object Oriented Software Development\n";
         break;
      case 235:
         cout << "Data Structures\n";
         break;
      case 246:
         cout << "Software Design & Development\n";
         break;
   }
}
```

In the second example, we have a character literal as the case label. In this case, we will be displaying the name for a typed letter:

```
/*******************************************
 * DISPLAY LETTER NAME
 * Display the full name (“one”) corresponding
 * to a given letter (‘1’)
 *******************************************/
void displayLetterName(char letter)
{
   switch (letter)                     // though letter is a char, it readily converts
   {                                   //     to an integer
      case 'A':                        // character literal, corresponding to 65
         cout << "Letter A";
         break;
      case '1':                        // character literal 48
         cout << "Number one";
         break;
      case 32:                         // corresponding to the character literal ' '
         cout << "Space";
         break;
   }
}
```

Finally, we can use a constant for a case label. Here, the compiler guarantees the value cannot be changed.

```
const int GOOD   1;
const int BETTER 2;
const int BEST   3;

/*****************************************
 * DISPLAY
 * convert a value into the name
 *****************************************/
void display(int value)
{
   switch (value)
   {
      case BEST:
         cout << "Best!\n";
         break;
      case BETTER:
         cout << "Better!\n";
         break;
      case GOOD:
         cout << "Good!\n";
        break;
   }
}
```

There are three common sources of errors with case labels. The first is to use something other than an integer. This will never correspond to the integer resulting from the controlling expression. The second is to use a variable rather than a literal or constant. The final is to duplicate a case label:

```
{
   int x = 3;
   switch (4)
   {
      case "CS 124":      // ERROR! Must be an integer and this is a pointer to a char
         break;
      case 3.14159:       // ERROR! This is a float and needs to be an integer
         break;

      case x:             // ERROR! This is a variable and it needs to be a literal
         break;

      case 2:
         break;
      case 2:             // ERROR! Duplicate value
         break;
   }
}
```

Sue's Tips

It turns out that a SWITCH statement is much more efficient than a collection of IF/ELSE statements. The reason for this has to do with how compilers treat SWITCH statements. Typically, the compiler creates something called a "perfect hash" which allows the program to jump to exactly the right spot every time. Thus, a SWITCH with 100 CASE labels is just as efficient as one with only 3. The same cannot be said for IF/ELSE statements!

Default

The DEFAULT keyword is a special CASE label corresponding to "everything else." In other words, it is the catch-all. If none of the other CASE labels correspond to the result of the controlling expression, then the DEFAULT is used. There can be either zero or one default in a switch statement. Typically we put the default at the end of the list but it could be anywhere.

The first example illustrates using a DEFAULT where multiple CASEs would otherwise be needed:

```
switch (numberGrade / 10)
{
   case 10:
   case 9:
      cout << "Perfect job";
      break;
   case '8':
   case '7':
      cout << "You passed!\n";
      break;
   default:                          // covering 6, 5, 4, 3, 2, 1, and 0
      cout << "Take the class again\n";
}
```

In these cases, the DEFAULT is used to minimize the size of the source code and to increase efficiency. It is useful to put a comment describing what cases the DEFAULT corresponds to.

The second example uses the DEFAULT to handle unexpected input. In these cases, there is typically a small number of acceptable input values and a very large set of invalid input values:

```
{
   char input;
   cout << "Do you want to save your file (y/n)? ";
   cin >> input;                         // though a char is accepted, there are only
                                         //     two valid inputs: ‘y’ and ‘n’
   switch (input)                        // 256 possible values, but only 2 matter
   {
      case 'Y':                          // we are treating ‘Y’ and ‘y’ the same here
      case 'y':
         save(data);
         break;
      case 'n':                          // the “do-nothing” condition
      case 'N':
         break;
      default:                           // everything else
         cout << "Invalid input '"
              << input
              << "'. Try again\n";
   }
}
```

The most common error with DEFAULT statements is to try to define more than one.

Body statements

You can put as many statements inside the CASEs as you choose. The `{}`s are only needed if you are declaring a variable. To leave a switch statement, use the BREAK statement. This will send execution outside the SWITCH statement. Note that the BREAK statement is optional.

```
/******************************
 * EXECUTE COMMAND
 * Execute command as specified
 * by the caller
 ******************************/
void executeCommand(char option)
{
   switch (option)
   {
      case 's':             // When there are two case labels like this, the
                            //    fall-through is implied
      case 'S':             // Save and Quit
         saveGame(game);
         // fall-through… We don’t want a break because we want the ‘Q’ condition next
      case 'q':
      case 'Q':             // Quit
         quit(game);
         break;
    }
}
```

A common error is to forget the BREAK statement between cases which yields an unintentional fall-through. If a fall-through is needed, put a comment to that effect in the code.

Example 3.5 – Golf

Demo

This example will demonstrate a simple SWITCH statement to enable the program to select between six different options.

Problem

In the game of golf, each hole has a difficulty expressed in terms of how many strokes it takes for a standard (read "a very good") golfer to complete. This standard is called "par." If a golfer completes a hole in one fewer strokes than par, he is said to achieve a "birdie" (no actual birds are used in this process). If he does two better, he achieves an "eagle." Finally, if he takes one more stroke than necessary, he gets a "bogie." Write a function to convert a score into the corresponding label.

```
What is your golf score? 3
What is par for the hole? 5
You got an eagle
```

Solution

The important work is done in the following function:

```
/************************************
 * DISPLAY
 * Translate the golfer performance into
 * a "bogie," "par," or whatever
 ************************************/
void display(int score, int par)
{
   // translate the golfer performance into a "bogie", or "par" or whatever
   switch (score - par)
   {
      case 2:
         cout << "You got a double bogie\n";
         break;
      case 1:
         cout << "You got a bogie\n";
         break;

      case 0:
         cout << "You got par for the hole\n";
         break;

      case -1:
         cout << "You got a birdie\n";
         break;
      case -2:
         cout << "You got an eagle\n";
         break;

      default:
         cout << "Your score was quite unusual\n";
   }

   return;
}
```

See Also

The complete solution is available at 3-5-golf.cpp or:

```
/home/cs124/examples/3-5-golf.cpp
```

Unit 3

Conditional Expression

While an IF statement chooses between two *statements*, a conditional expression chooses between two *expressions*. For example, consider the following code inserting a person's title before their last name according to their gender:

```
cout << "Hello "
     << (isMale ? "Mr. " : "Mrs. ")
     << lastName;
```

Even though we have a single `cout` statement, we can embed the conditional expression right in the middle of the statement.

Up to this point, we have used **unary operators** (operators with a single operand) such as increment (`++a`), logical not (`!a`), address-of (`&a`) and dereference (`*a`). We have also used **binary operators** (operators with two operands) such as addition (`a + b`), modulus (`a % b`), logical and (`a && b`), greater than (`a > b`), and assignment (`a += b`). There is exactly one **ternary operator** (operator with three operands) in the C++ language: the conditional expression:

```
<Boolean expression> ? <true expression> : <false expression>
```

Like all operators, the result is an expression. In other words, the evaluation of the conditional expression is either the true-expression or the false-expression.

Note that the conditional expression operator resides midway on the order of operations table. Because the insertion (`<<`) and the extraction (`>>`) operator are above the conditional expression in the order of operations, we commonly need to put parentheses around conditional expressions.

Example 1: Absolute value

Consider, for example, the absolute value function. In this case, we return the value if the value is already positive. Otherwise, we return the negative of the value. In other words, we apply the negative operator only if the value is already negative.

```
number = (number < 0) ? -number : number;
```

Of course this could be done with an IF statement, but the conditional expression version is more elegant.

Example 2: Minimum value

In another example, we would like to find the smaller of two numbers:

```
lower = (number1 > number2) ? number2 : number1;
```

Here we are choosing between two values. The smaller of the two will be the result of the conditional expression evaluation and subsequently assigned to the variable `lower`.

Sue's Tips

Conditional expressions have comparable performance characteristics to IF statements; compilers typically treat them in a similar way. Some programmers avoid conditional expressions because they claim it makes the code more difficult to read. Some programmers favor conditional expressions because they tend to make the code more compact. In general, this is a stylistic decision. As with all stylistic issues, favor a design that increases code clarity and exposes potential bugs.

Example 3.5 – Select Tabs or Newlines

Demo

This example will demonstrate how to use conditional expressions to choose between options. While an IF statement could do the job, conditional expressions are more elegant.

Problem

Write a program to display the multiplication tables between 1 and n. We wish to put a tab between each column but a newline at the end of the row. In other words, we put a tab after every number except the number at the end of the row:

```
How big should your multiplication table be?  4
1       2       3       4
2       4       6       8
3       6       9       12
4       8       12      16
```

Solution

To display two-dimensional data such as a multiplication table, it is necessary to have two counters: one for the row and one for the column. After each number in the table, we will have either a newline `'\n'` or a tab `'\t'`. We choose which is to be used based on the column number.

```
void displayTable(int num)
{
   for (int row = 1; row <= num; row++)           // count through the rows
      for (int col = 1; col <= num; col++)        // count through the columns
         cout << (row * col)                      // display the product
              << (col == num ? '\n' : '\t');     // tab or newline, depending
}
```

Notice how the function could have been done with an IF statement, but it would not have been as elegant:

```
void displayTable(int num)
{
   for (int row = 1; row <= num; row++)           // count through the rows
      for (int col = 1; col <= num; col++)        // count through the columns
      {
         if (col == num)                         // are we on the last column?
            cout << (row * col) << endl;          // display an endl
         else
            cout << (row * col) << '\t';          // display a tab
      }
}
```

Challenge

As a challenge, modify the above function to handle negative values in the `num` parameter. If a negative value is passed, make it a positive value using absolute value. Implement absolute value using a conditional expression.

See Also

The complete solution is available at 3-5-selectTabOrNewLine.cpp or:

```
/home/cs124/examples/3-5-selectTabOrNewline.cpp
```

Bitwise Operators

To understand bitwise operators, it is first necessary to understand a bit. As you may recall from Chapter 1.2, data is stored in memory through collections of bits. There are 8 bits in a byte and an integer consists of 4 bytes (32 bits). With computers, we represent numbers in binary (base 2 where the only possible values are 0 and 1), decimal (base 10 where the only possible values are 0…9), and hexadecimal (base 16 where the possible values are 0…9, A…F). Consider, for example, the binary value 00101010. The right-most bit corresponds to 2^0, the next corresponds to 2^1, and the next corresponds to 2^2. Thus 00101010 is:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1	0

$$0 + 0 + 32 + 0 + 8 + 0 + 2 + 0 = 42$$

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. This is how we conver the binary 00101010 in the decimal 42. Hexadecimal is similar to decimal except we are storing a nibble (4 bits for 2^4 possible values) into a single digit. Thus there are 3 ways to count to 16, the binary way, the decimal way, and the hexadecimal way:

Binary	`0000`	`0001`	`0010`	`0011`	`0100`	`0101`	`0110`	`0111`	`1000`	`1001`	`1010`	`1011`	`1100`	`1101`	`1110`	`1111`	`10000`
Decimal	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`	`8`	`9`	`10`	`11`	`12`	`13`	`14`	`15`	`16`
Hexadecimal	`0x00`	`0x01`	`0x02`	`0x03`	`0x04`	`0x05`	`0x06`	`0x07`	`0x08`	`0x09`	`0x0A`	`0x0B`	`0x0C`	`0x0D`	`0x0E`	`0x0F`	`0x10`

In many ways, a byte is an array of eight bits. Since each bit stores a `true`/`false` value (similar to a `bool`), we should be able to store eight Boolean values in a single byte. The problem, however, is that computers find it easier to work with bytes rather than bits (hence `sizeof(bool) == 1`). Wouldn't it be great if we could access individual bits in a byte? We can with bitwise operators.

Bitwise operators allow us to perform Boolean algebra not on Boolean variables but on individual bits. We have six bitwise operators:

Operator	Description	Example
`~`	Bitwise NOT	`0101 == ~1010`
`&`	Bitwise AND	`1000 == 1100 & 1010`
`|`	Bitwise OR	`1110 == 1100 | 1010`
`^`	Bitwise XOR	`0110 == 1100 ^ 1010`
`<<`	Left shift	`0110 == 0011 << 1`
`>>`	Right shift	`0110 == 1100 >> 1`

One common use of bitwise operators is to collapse a collection of Boolean values into a single integer. If, for example, we have a variable called `settings` containing these values, then we can turn on bits in settings with the bitwise OR operator `|`. We can then determine if a setting is on with the bitwise AND operator `&`.

Consider, for example, the following daily tasks Sue may need to do during her morning routine:

```
#define takeShower     0x01
#define eatBreakfast   0x02
#define getDressed     0x04
#define driveToSchool  0x08     // observe how each literal refers to a single bit
#define driveToChurch  0x10
#define goToClass      0x20
#define doHomework     0x40
#define goOnHike       0x80
```

Observe how each value corresponds to turning on a single bit. We can next identify common tasks:

```
#define weekDayRoutine  = takeShower | eatBreakfast  |
                          getDressed | driveToSchool |
                          doHomework                         // use the bitwise OR
#define saturdayRoutine = eatBreakfast | getDressed |        //    to combine settings.
                          goOnHike                           //    This will set many
#define sundayRoutine   = takeShower | eatBreakfast |        //    individual bits
                          getDressed | driveChurch
```

With the bitwise OR operator, we are adding individual bits to the resulting value. Now when the code attempts to perform these tasks, we use the bitwise AND to see if the bits are set:

```
{
   unsigned char setting = sundayRoutine;

   // take a shower?
   if (setting & takeShower)                          // use the bitwise AND to check
      goTakeAShower();                                //     if an individual bit
                                                      //     is set. Be careful to
   // eat breakfast?                                  //     not use the && here… it
   if (setting & eatBreakfast)                        //     will always evaluate
      goEatBreakfast();                               //     to TRUE

   // get dressed?
   if (setting & getDressed)
      goGetDressed();
}
```

Bitwise operators are rarely used in typical programming scenarios. They can be seen in system programming where programs are talking to hardware devices that use individual bits for control and status reporting. However, when you encounter them, you should be familiar with what they do.

Sam's Corner

It turns out that we could have been using bitwise operators since the very beginning of the semester. Remember how we format floating point numbers for output:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

It turns out that the `setf()` method of `cout` uses bitwise operators to set configuration data:

```
cout.setf(ios::fixed | ios::showpoint);
cout.precision(2);
```

Since `ios::fixed` $==$ 4 (2^2) and `ios::showpoint` $==$ 1024 (2^{10}), we could also be truly cryptic and say:

```
cout.setf((std::ios_base::fmtflags)1028);  // need to cast it to fmtflags
cout.precision(2);
```

Example 3.5 – Show Bits

Demo

This example will demonstrate how to look at the individual bits of a variable. This will be accomplished by looping through all 32 bits of an integer, masking away each individual bit with the bitwise and `&` operator.

Problem

Write a program to display the bits of a number, one at a time.

```
Please enter a positive integer: 42
The bits are: 00000000000000000000000000101010
```

Solution

The first step to solving this problem is to create a number (called `mask`) with a single bit in the left-most place. When this mask is bitwise ANDed against the target number (`mask & value`), the resulting expression will evaluate to `true` only if there is a 1 in that place in the target number. Next, the 1 in the mask is shifted to the right by one space (`mask = mask >> 1`) and the process is repeated.

```
/***************************************
 * DISPLAY BITS
 * Display the bits corresponding to an integer
 ***************************************/
void displayBits(unsigned int value)
{
   unsigned int mask = 0x80000000;            // only the left-most bit is set

   for (int bit = 31; bit >= 0; bit--)        // go through all 32 bits
   {
      cout << ((mask & value) ? '1' : '0');   // check the current bit
      mask = mask >> 1;                       // shift the mask by one space
   }
   cout << endl;
}
```

Challenge

As a challenge, make a char version of the above function. How big must the mask be? How many bits will it have to loop through?

See Also

The complete solution is available at 3-5-showBits.cpp or:

```
/home/cs124/examples/3-5-showBits.cpp
```

Problem 1

What is the output of the following code?

```
{
   int   a =  0;
   int   b =  1;
   int * c = &b;
   *c = 2;
   int * d = &a;
    b =  3;
    d = &b;
   *d =  4;
   cout << "b == " << b << endl;
}
```

a	b	c	d

Answer:

____

Please see page 258 for a hint.

Problem 2

What is the output of the following code?

```
{
   char a[] = "Banana";
   char b[] = "Bread";
   char * c;

   if (a == b)
      c = a;
   else
      c = b;

   cout << c << endl;
}
```

Answer:

____

Please see page 277 for a hint.

Unit 3

Problem 3

What is the output of the following code?

```
{
   int number = 5;

   switch (number)
   {
      case 4:
         cout << "four!\n";
         break;
      case 5:
         cout << "five!\n";
         break;
      case 6:
         cout << "six!\n";
         break;
      default:
         cout << "unknown!\n";
   }
}
```

Answer:

____

Please see page 285 for a hint.

Problem 4

What is the syntax error in the following code?

```
{
   int input = 20;

   switch (input)
   {
      case 'a':
         cout << "A!\n";
         break;
      case true:
         cout << "B!\n";
         break;
      case 2.0:
         cout << "C!\n";
         break;
      default:
         cout << "unknown!\n";
   }
}
```

Answer:

____

Please see page 287 for a hint.

Problem 5

What is the output of the following code?

```
{
   float grade1 = 3.7;
   int   grade2 = 60;

   switch ((int)(grade1 * 2.0))
   {
      case 8:
         grade2 += 5;
      case 7:
         grade2 += 5;
      case 6:
         grade2 += 5;
      case 5:
         grade2 += 5;
      case 4:
         grade2 += 5;
      default:
         grade2 += 10;
   }

   cout << grade2 << endl;
}
```

Answer:

____

Please see page 290 for a hint.

Problem 6

Write a function that take a letter as input and displays a message on the screen:

Letter	Message
A	Great job!
B	Good work!
C	You finished!
All other	Better luck next time

Answer:

Please see page 285 for a hint.

Assignment 3.5

The purpose of this assignment is to demonstrate `switch` statements and conditional operators. Though of course it is possible to complete this assignment without using either, it will defeat its purpose.

Your assignment is to write two functions (`computeLetterGrade()` and `computeGradeSign()`) and a single driver program to test them.

computeLetterGrade

Write a function to return the letter grade from a number grade. The input will be an integer, the number grade. The output will be a character, the letter grade. You must use a `switch` statement for this function. Please see the syllabus for the meaning behind the various letter grades.

computeGradeSign

Write another function to return the grade sign (+ or -) from a number grade. The input will be the same as with `computeLetterGrade()` and the output will be a character. If there is no grade sign for a number grade like 85%=B, then return the symbol '*'. **You must use at least one conditional expression**. Please see the syllabus for the exact rules for applying the grade sign.

Driver Program

Create a `main()` that prompts the user for a number graded then displays the letter grade.

Example

Three examples… The user input is **underlined**.

Example 1: 81%

```
Enter number grade: 81
81% is B-
```

Example 2: 97%

```
Enter number grade: 97
97% is A
```

Example 3: 77%

```
Enter number grade: 77
77% is C+
```

Assignment

The test bed is available at:

```
testBed cs124/assign35 assignment35.cpp
```

Don't forget to submit your assignment with the name "Assignment 35" in the header.

Please see page 49 for a hint.

Unit 3 Practice Test

Practice 3.3

We are all habitual writers; our word choice and the way we phrase things are highly individualistic. Write a program to count the frequency of a given letter in a file.

Example

Given a file that contains the sample solution for this problem, count the frequency of usage of a given letter in the file.

User input is **underlined**.

```
What is the name of the file: /home/cs124/tests/practice33.cpp
What letter should we count: t
There are 125 t's in the file
```

Another execution on the same file:

```
What is the name of the file: /home/cs124/tests/practice33.cpp
What letter should we count: i
There are 66 i's in the file
```

Assignment

Write the program to:

- Prompt the user for the filename
- Read the data from the file one letter at a time
- Compare each letter against the target letter
- Display the count of instances on the screen
- Use proper modularization of course.

Please use the following test bed to validate your answers:

```
testBed cs124/practice33 practice33.cpp
```

You can validate your answers against:

```
/home/cs124/tests/practice33.cpp
```

Continued on the next page

Continued from previous page

Grading for Test 3

Sample grading criteria:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
Syntax of the array 30%	All references of the array are elegant and optimal	Array correctly declared and referenced	One bug exists	Two or more bugs	An array was not used in the problem
File interface 30%	Solution is elegant and efficient	All the data is read and error checking is performed	Able to open the file and read from it	Elements of the solution are present	No attempt was made to open the file
Problem solving 20%	Solution is elegant and efficient	Zero test bed errors	There exist only one or two flaws in the approach to solve the problem	Elements of the solution are present	Input and output do not resemble the problem
Modularization 10%	Functional cohesion and loose coupling is used throughout	Zero bugs with function syntax but there exist modularization errors	One bug exists in the syntax or use of a function	Two or more bugs	All the code exists in one function
Programming Style 10%	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 3 Project : MadLib

Write a program to implement Mad Lib®. According to Wikipedia,

> Mad Libs is a word game where one player prompts another for a list of words to substitute for blanks in a story; these word substitutions have a humorous effect when the resulting story is then read aloud.

The program will prompt the user for the file that describes the Mad Lib®, and then prompt him for all the substitute words. When all the prompts are complete, the program will display the completed story on the screen.

This project will be done in three phases:

- Project 08 : Design the MadLib program
- Project 09 : Read the file and display all the prompts to the user
- Project 10 : Display a MadLib for a given file and set of user input

Interface Design

The program will prompt the user for the filename of his Mad Lib®, allow him to play the game, then ask the user if he/she wants to play another. Consider the following Mad Lib® with the filename `madlibZoo.txt`:

```
Zoos are places where wild :plural_noun are kept in pens or cages :! so
that :plural_noun can come and look at them :. There is a zoo :! in the park
beside the :type_of_liquid fountain :. When it is feeding time, :! all
the animals make :adjective noises. The elephant goes :< :funny_noise
:> :! and the turtledoves go :< :another_funny_noise :. :> My favorite
animal is the :! :adjective :animal :, so fast it can outrun a/an
:another_animal :. :! You never know what you will find at the zoo :.
```

An example of the output is:

```
Please enter the filename of the Mad Lib: madlibZoo.txt
        Plural noun: boys
        Plural noun: girls
        Type of liquid: lemonade
        Adjective: fuzzy
        Funny noise: squeak
        Another funny noise: snort
        Adjective: hungry
        Animal: mouse
        Another animal: blue-fin tuna

Zoos are places where wild boys are kept in pens or cages
so that girls can come and look at them. There is a zoo
in the park beside the lemonade fountain. When it is feeding time,
all the animals make fuzzy noises. The elephant goes "squeak"
and the turtledoves go "snort." My favorite animal is the
hungry mouse, so fast it can outrun a/an blue-fin tuna.
You never know what you will find at the zoo.

Do you want to play again (y/n)? n
Thank you for playing.
```

Note that there is a tab before each of the questions (ex: "`Plural noun:`")

File Format

Consider the following user's file called `madLibExample.txt`:

```
his is one line with a newline at the end. :!
Here we have a comma :, and a period :. :!
We can have :< quotes around our text :> if we want :!
This will prompt for :< My favorite cat :> is :my_favorite_cat :. :!
```

Notice the following traits of the file:

- Every word, keyword, or punctuation is separated by a space or a newline. These are called tokens.
- Tokens have a colon before them. They are:

Symbol	Meaning
!	Newline character. No space before or after.
<	Open double quotes. No space after.
>	Close double quotes. No space before.
.	Period. No space before.
,	Comma. No space before.
anything else	A prompt

- If a prompt is encountered, convert the text inside the prompt to a more human-readable form. This means:

    1. Sentence-case the text, meaning capitalize the first letter and convert the rest to lowercase.
    2. Convert underscores to spaces.
    3. Proceed the prompt with a tab.
    4. Put a colon and a space at the end.
    5. The user's response to the text could include spaces.

Your program will not need to be able to handle files of unlimited length. The file should have the following properties (though you will need to do error-checking to make sure):

- There are no more than 1024 characters total in the file.
- There are no more than 32 lines in the file.
- Each line has no more than 80 characters in it.
- There are no more than 256 words in the file.
- Each word is no more than 32 characters in length.

Hint: to see how to declare and pass an array of strings, please see page 227.

Hint: when displaying the story, you will need to re-insert spaces between each word. Either the word before or the word after a given space negotiate whether there is a space between the words. This means that either word can remove the space.

Project 08

The first part of the project is the design document. This consists of three parts:

1. Create a structure chart describing the entire Mad Lib® program.

2. Write the pseudocode for the function `readFile` a function to read the Mad Lib® file into some data structure (examples: a string, and array of something). You will need to include the logic for reading the entire story into the data-structure and describe how the story will be stored.

3. Write the pseudocode for the function `askQuestion`. This need to describe how to turn `":grandma's_name"` into `"\tGrandma's name: "` and also describe how to put the user's response back into the story. If, for example, the file had the tags "`:favorite_car`" and "`:first_pet's_name`" then the following output would result:

```
        Favorite car: Ariel Atom 3
        First pet's name: Midnight
```

On campus students are required to attach this rubric to your design document. Please self-grade.

Project 09

The second part of the Mad Lib project (the first part being the design document due earlier) is to write the code necessary read the Mad Lib from a file and prompt the user:

```
Please enter the filename of the Mad Lib: madlibZoo.txt
        Plural noun: boys
        Plural noun: girls
        Type of liquid: lemonade
        Adjective: fuzzy
        Funny noise: squeak
        Another funny noise: snort
        Adjective: hungry
        Animal: mouse
        Another animal: blue-fin tuna
```

Note that there is a tab before each of the questions (ex: "`Plural noun:`"):

Hints

- Your program will not need to be able to handle files of unlimited length. The file should have the following properties (though you will need to do error-checking to make sure):
- There are no more than 1024 characters total in the file.
- There are no more than 32 lines in the file.
- Each line has no more than 80 characters in it.
- There are no more than 256 words in the file.
- Each word is no more than 32 characters in length.

Hint: To see how to declare and pass an array of strings, please see Chapter 3.0 of the text.

Assignment

Perhaps the easiest way to do this is in a four-step process:

1. Create the framework for the program using stub functions based on the structure chart from your design document.
2. Write each function. Test them individually before "hooking them up" to the rest of the program. You are not allowed to use the String Class for this problem; only c-strings!
3. Verify your solution with testBed:

```
testBed cs124/project09 project09.cpp
```

4. Submit it with "`Project 09, Mad Lib`" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj09.out
```

Project 10

The final part of the Mad Lib project is to write the code necessary to make the Mad Lib appear on the screen:

```
Please enter the filename of the Mad Lib: madlibZoo.txt
        Plural noun: boys
        Plural noun: girls
        Type of liquid: lemonade
        Adjective: fuzzy
        Funny noise: squeak
        Another funny noise: snort
        Adjective: hungry
        Animal: mouse
        Another animal: blue-fin tuna

Zoos are places where wild boys are kept in pens or cages
so that girls can come and look at them. There is a zoo
in the park beside the lemonade fountain. When it is feeding time,
all the animals make fuzzy noises. The elephant goes "squeak"
and the turtledoves go "snort." My favorite animal is the
hungry mouse, so fast it can outrun a/an blue-fin tuna.
You never know what you will find at the zoo.

Do you want to play again (y/n)? n
Thank you for playing.
```

Hints

A few hints to make the code writing a bit easier:

- The best way to store the story is in an array of words. Thus the process of reading the story from the file removes all spaces from the story.
- By default, you insert a space before each word when you display the story. The only conditions when a space is not inserted is when the preceding character is a open quote or a newline, or when the following character is a closed quote, period, or comma. In other words, do not think about removing spaces, but rather about adding them when conditions are right.
- Your program will need to be able to play any number of Mad Lib games. The easiest way to handle this is to have a while loop in `main()`

Assignment

Perhaps the easiest way to do this is in a five-step process:

1. Start with the code from Project 09.
2. Fix any necessary bugs.
3. Write the code to display the Mad Lib on the screen.
4. Verify your solution with testBed:

```
testBed cs124/project10 project10.cpp
```

5. Submit it with "`Project 10, Mad Lib`" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj10.out
```

Unit 4. Advanced Topics

4.0 Multi-Dimensional Arrays

Sam had so much fun dabbling with ASCII-art that he thought he would try his hand at computer graphics. The easiest way to get started is to load an image from memory and display it on the screen. This seems challenging, however; memory (including the type of data stored in an array) is one-dimensional but images are two-dimensional. How can he store two-dimensional data in an array? How can he convert the one-dimensional data in a file into this array? While trying to figure this out, Sue introduces him to multi-dimensional arrays.

Objectives

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

- Declare a multi-dimensional array.
- Pass a multi-dimensional array to a function as a parameter.
- Read multi-dimensional data from a file and put it in an array.

Prerequisites

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

- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Pass an array to a function (Chapter 3.0)
- Write the code to read data from a file (Chapter 2.6).
- Write the code to write data to a file (Chapter 2.6).

Overview

Often we work with data that is inherently multi-dimensional. A few common examples include pictures (row and column), coordinates (latitude, longitude, and altitude), and position on a grid (x and y). The challenge arises when we need to store the multi-dimensional data in a memory store that is inherently one dimensional.

Consider the following code to put the numbers 0-15 on the screen:

```
for (int index = 0; index < 16; index++)
   cout << index << '\t';
cout << endl;
```

Observe how the numbers are one-dimensional (just an index). However, we would like to put the numbers in a nice two-dimensional grid that is 4 × 4. How do we do this? The first step is we need some way to detect when we are on the 4th column. When we are on this column, we display a newline character rather than a white space to properly align the columns.

column 0	column 1	column 2	column 3	
0	1	2	3	row 0
4	5	6	7	row 1
8	9	10	11	row 2
12	13	14	15	row 3

Are there any patterns in the numbers? Can we find any way to derive the row or column based on the index? The first thing to realize is that the column numbers seem to increase by one as the index increases by one. This occurs until we get to the end of the row. When that happens, the column number seems to reset.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
column	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

This pattern should be familiar. As we divide the index (`index`) by the number of columns (`numCol`), the remainder appears to be the column (`column`) value.

```
column = index % numCol;
```

The row value appears to be an entirely different equation. We increment the row value only after we increment four index values:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
row	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

This pattern is also familiar. We can derive `row` by performing integer division on `index` by `numCol`:

```
row = index / numCol;
```

Based on these observations, we can re-write our loop to display the first 16 whole numbers:

```
for (int index = 0; index < 16; index++)
   cout << index << (index % 4 == 3 ? '\n' : '\t');
```

In other words, when the column value (`index % 4`) is equal to the fourth column (`== 3`) then display a newline character (`'\n'`) rather than the tab character (`'\t'`). We can re-write this more generally as:

```
/**********************************************
 * DISPLAY NUMBERS
 * Display the first 'number' whole numbers
 * neatly divided into a grid of numCol columns
 **********************************************/
void displayNumbers(int number, int numCol)
{
   for (int index = 0; index < number; index++)               // one dimensional index
      cout << index                                           // display the index
           << (index % numCol == numCol - 1 ? '\n' : '\t');   // break into rows
}
```

After converting an inherently one-dimensional value (`index`) into a two-dimensional pair (`row` & `column`), how do we convert two-dimensional values back into an index? To accomplish this, we need to recall the things we learned when going the other way:

- An increase in the `index` value yields an increase in the `column` value. To turn this around, we could also say that an increase in the `column` value yields an increase in the `index` value.
- The `row` value changes one fourth (`1 / numCol`) as often as the `index` value. To turn this around, a change in the `row` value yields a jump in the `index` value by four (`numCol`).

We can combine both these principles in a single equation:

```
index = row * 4 + column;
```

Check this equation for correctness:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
row	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
column	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

If we add one to the `row`, then the `index` jumps by four. If we add one to the `column`, the `index` jumps by one. Thus we have the ability to convert two-dimensional coordinates (`row` & `column`) into a one dimensional value (`index`). The general form of this equation (worth memorizing) is:

```
index = row * numCol + column;
```

Why would we ever want to do this? Consider the scenario when we want to put the multiplication tables for the values 0 through 3 in an array. This can be accomplished with:

```
{
   int grid[4 * 4];                         // the area of the array is the width
                                            //     times the height.
   for (int row = 0; row < 4; row++)        // rows first, 0...3
      for (int col = 0; col < 4; col++)     // columns next, also 0...3
         grid[row * 4 + col] = row * col;   // convert to row,col to index for the []
}                                           //     the right-side is the product
```

In memory, the resulting `grid` array appears as the following:

	row 0				row 1				row 2				row 3			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

In both of these cases (converting 2-dimensional to 1 and converting 1-dimensional to 2), the same piece of information is needed: the number of columns (`numCol`) in the data. This should make sense. If you have 32 items in a data-set, is the grid 1×32 or 2×16 or 4×8 or 8×4 or 16×2 or 32×1? Each of these possibilities is equally likely. One must know either the number of columns or the number of rows to make the conversion.

Syntax

As you may have noticed, multi-dimensional arrays are quite commonly needed to solve programming problems. Similarly, the conversion from index to coordinates and back is tedious and overly complicated. Fortunately, there is an easier way:

Declaring an array	Referencing an array	Passing as a parameter
Syntax: `<Type> <name>[size][size]`	Syntax: `<name>[index][index]`	Syntax: `(<Type> <name>[][size])`
Example: `int data[200][15];`	Example: `cout << data[i][j];`	Example: `void func(int data[][15])`
A few details: • Any data-type can be used. • The size must be a natural number {1, 2, etc.} and not a variable.	A few details: • The index starts with 0 and must be within the valid range.	A few details: • You must specify the base-type. • No size is passed in the square brackets `[]`.

Declaring an array

Multi-dimensional arrays are declared by specifying the base-type and the size of each dimension. The basic syntax is:

```
<base-type> <variable>[<number of rows>][<number of columns>];
```

A grid of integers that is 3 × 4 can be declared as:

```
int grid[4][3];
```

We can also initialize a multi-dimensional array at declaration time. The best way to think of the initialization syntax is "an array of arrays." Consider the following example:

```
{
   int grid[4][3] =
   {// col 0    1    2
      {    8,  12,  -5 },  // row 0
      {  421,   4, 153 },  // row 1
      {  -15,  20,  91 },  // row 2
      {    4, -15, 182 },  // row 3
   };
}
```

Sue's Tips

Notice how the horizontal dimension comes *second* in multi-dimensional arrays. In Geometry, we learned to specify coordinates as (X, Y) where the horizontal dimension comes first. Multi-dimensional arrays are the opposite! Rather than trying to re-learn (Y, X) (which just doesn't feel right, does it?), it is more convenient to use (Row, Column) as our array dimensions.

Storing a digital image is a slightly more complex example. Each pixel consists of three values (red, green, and blue) with 256 possible values in each (`char`). The pixels themselves are arrayed in a two-dimensional image (4,000 × 3,000). The resulting declaration is:

```
char image[3][3000][4000];
```

In this example, each element is a `char` (eight bits in a byte so there are 2^8 possible values). The first dimension (`[3]`) is for the three channels (red, green, and blue). The next is the horizontal size of the image (4,000). The final dimension is the vertical dimension (3,000). The total size of the image is:

```
int size = sizeof(char) * sizeof(3) * sizeof(3000) * sizeof(4000);
```

This is 36,000,000 bytes of data (34.33 megabytes). A twelve mega-pixel image is rather large!

Referencing an array

When referencing an array, it is important to specify each of the dimensions. Again, we use the vertical dimension first so we use (Row, Column) variables rather than (X, Y). Back to our 3 × 4 grid example:

```
{
   int grid[4][3] =
   {// col 0    1    2
      {    8,  12,  -5 },  // row 0
      {  421,   4, 153 },  // row 1
      {  -15,  20,  91 },  // row 2
      {    4, -15, 182 },  // row 3
   };

   int row;    // vertical dimension
   int col;    // horizontal dimension

   cout << "Specify the coordinates (X, Y) ";  // people think in terms of X,Y
   cin  >> col >> row;

   assert(row >= 0 && row < 4);                // a loop would be a better tool here
   assert(col >= 0 && col < 3);                // always check before indexing into
                                               //     an array
   cout << grid[row][col] << endl;
}
```

Working with more than two-dimensions is the same. Back to our image example consisting of a two-dimensional grid of pixels (4,000 × 3,000) where each pixel has three values. If the user wishes to find the value of the top-left pixel, then the following code would be required:

```
cout << "red:   " << image[0][0][0] << endl
     << "green: " << image[1][0][0] << endl
     << "blue:  " << image[2][0][0] << endl;
```

Passing as a parameter

Passing arrays as parameters works much the same for multi-dimensional arrays as they do for their single-dimensional brethren. There is one important exception, however. Recall from earlier in the semester (Chapter 3.0) that arrays are just pointers to the first item in the list. Only having this pointer, the callee does not know the length of the buffer. For this reason, it is important to pass the size of the array as a parameter.

There is another important component to understanding multi-dimensional array parameters. Recall that, for a given 32 slots in memory, there may be many possible ways to convert it into a two-dimensional grid { (32 × 1), (16 ×2), (8 × 4), (4 × 8), (2 × 16), or (1 × 32) }. The only way to know which conversion is correct is to know the number of columns (typically called the `numCol` variable). This information is essential to performing the conversion.

When using the double-square-bracket notation for multi-dimensional arrays (`array[3][4]` instead of `array[3 * numCol + 4]`), the compiler needs to know the `numCol` value. The same is true when passing multi-dimensional arrays as parameters. In this case, we specify the size of all the dimensions except the left-most dimension. Back to our 3 × 4 example, a prototype might be:

```
void displayGrid(int array[][3]);     // column size must be present
```

Back to our image example, the following code will fill the image with data from a file.

```
/**************************************
 * READ IMAGE
 * Read the image data from a file
 **************************************/
bool readImage(unsigned char image[][3000][4000],   // specify all dimensions but first
               const char fileName[])               // also need the filename as const
{
   // open stream
   ifstream fin(fileName);
   if (fin.fail())                                  // never forget error checking
      return false;                                 // return and report

   bool success = true;                             // our return value

   // read the grid of data
   for (int row = 0; row < 3000; row++)             // rows are always first
      for (int col = 0; col < 4000; col++)          // then columns
         for (int color = 0; color < 3; color++)    // three color dimensions: r, g, b
         {
            int input;                              // data in the file is a number so
            fin >> input;                           //     we read it as an integer
            if (  input < 0 || input >= 256 ||      //     before storing it as a
                  fin.fail())                       //     char (a small integer). Make
               success = false;                     //     sure it is valid!
            image[color][row][col] = input;
         }

   // paranoia!
   if (fin.fail())                                  // report if anything bad happened
      success = false;

   // make like a tree
   fin.close();                                     // never forget to close the file
   return success;
}

/***********************************
 * MAIN
 * Simple driver for readImage
 ***********************************/
int main()
{
   unsigned char image[3][3000][4000];              // 12 megapixel image
   if (!readImage(image, "image.bmp"))              // .bmp images are just arrays
      return 1;                                     //    of pixels!  Note that they
   else                                             //    are binary files, not text
      return 0;                                     //    files so this will not quite
}                                                   //    work the way you expect...
```

One quick disclaimer about the above example… Images are stored not as text files (which can be opened and read in emacs) but as binary files (such as `a.out`. Try opening it in emacs to see what I mean). To make this above example work, we will need to create 36 million integers (3 x 3,000 x 4,000), each of which with a value between 0 and 255. That might take a bit of patience.

Example 4.0 – Array of Strings

Demo

This example will demonstrate how to create, pass to a function, and manipulate an array of strings. This is a multi-dimensional array of characters.

Solution

Since strings are arrays, to have an array of strings we will need a two dimensional array. Note that the first dimension must be the number of strings and the second the size of each.

```
/********************************************
 * PROMPT NAMES
 * Prompt the user for his or her name
 ********************************************/
void promptNames(char names[][256])          // the column dimension must be the
{                                            //     buffer size
   // prompt for name (first, middle, last)
   cout << "What is your first name? ";
   cin  >> names[0];                         // passing one instance of the array
   cout << "What is your middle name? ";     //     of names to the function CIN
   cin  >> names[1];                         // Note that the data type is
   cout << "What is your last name? ";       //     a pointer to a character,
   cin  >> names[2];                         //     what CIN expects
}

/********************************************
 * MAIN
 * Just a silly demo program
 ********************************************/
int main()
{
   char names[3][256];                       // arrays of strings are multi-
                                             //      dimensional arrays of chars
   // fill the array
   promptNames(names);                       // pass the entire array of strings

   // first name:
   cout << names[0] << endl;                 // this is an array of characters

   // middle initial
   cout << names[1][0] << endl;              // first letter of second string

   // loop through the names for output
   for (int i = 0; i < 3; i++)
      cout << names[i] << endl;

   return 0;
}
```

Challenge

As a challenge, extend the `names` array to include an individual's title. Thus the `promptNames()` function will consider the fourth row in the `names` array to be the title. You will also need to modify `main()` so the output can be displayed.

See Also

The complete solution is available at 4-0-arrayOfStrings.cpp or:

```
/home/cs124/examples/4-0-arrayOfStrings.cpp
```

Unit 4

Example 4.0 – Array of Integers

Demo

This example will create a 4 × 4 array of integers. This will be done both the old-fashion way of using a single-dimensional array as well as the new double-bracket notation. In both cases, the arrays will be filled with multiplication tables (`row * col`).

Solution

The 16 items in a 4 × 4 multiplication table represented as a single-dimensional array are:

	row 0				row 1				row 2				row 3			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

To write a function to fill this array, two parameters are needed: the number of rows (`numRow`) and the number of columns (`numCol`).

```
void fillArray1D(int grid[], int numCol, int numRow)
{
   for (int row = 0; row < numRow; row++)
      for (int col = 0; col < numCol; col++)
         grid[row * numCol + col] = row * col;
}
```

To do the same thing as a multi-dimensional array, the data representation is:

	grid[0][]				grid[1][]				grid[2][]				grid[3][]			
	0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

To work with multi-dimensional arrays, the compiler has to know the number of rows in the array. This means that, unlike with the single-dimensional version, we can only pass the `numRow` parameter. The `numCol` must be an integer literal specified in the parameter.

```
void fillArray2D(int grid[][4], int numRow)
{
   for (int row = 0; row < numRow; row++)
      for (int col = 0; col < 4; col++)
         grid[row][col] = row * col;
}
```

Challenge

As a challenge, change the program to display a 5 × 6 table. What needs to change in the calling function? What needs to change in the two fill functions?

See Also

The complete solution is available at 4-0-arrayOfIntegers.cpp or:

```
/home/cs124/examples/4-0-arrayOfIntegers.cpp
```

Unit 4

Example 4.0 – Convert Place To Points

Demo

Recall that there are two main uses for arrays: either they are a "bucket of variables" useful for storing lists of items, or they are tables useful for table-lookup scenarios. This example will demonstrate the table-lookup use for arrays.

Problem

Sam can make varsity on the track team if he gets 12 points. There are 10 races and points are awarded according to his placing:

Place	Points
1	5
2	3
3	2
4	1

Solution

We can create a data-driven program to compute how many points Sam will get during the season. If he gets 12 points and his varsity jacket, possibly Sue will want to go on another date with him!

```
{
   int points = 0;                              // initial points for the season
   int breakdown[4][2] =
   {
      {1, 5},                                   // 1st place gets 5 points
      {2, 3},                                   // 2nd place gets 3…
      {3, 2},
      {4, 1}
   };

   // Loop through the 10 races in the season
   for (int cRace = 0; cRace < 10; cRace++)      // "cRace" for "count Race"
   {
      // get the place for a given race
      int place;
      cout << "what was your place? ";
      cin  >> place;

      // add the points to the total
      for (int cPlace = 0; cPlace < 4; cPlace++) // Loop through all the places
         if (breakdown[cPlace][0] == place)      // if place in the table matches
            points += breakdown[cPlace][1];      // assign the points
   }

   cout << points << endl;
}
```

Observe how the first column is directly related to the row (`breakdown[row][0] == row + 1`). This means we technically do not need to have a multi-dimensional array for this problem.

Challenge

As a challenge, adapt this solution to the points awarded to the finishers at the Tour de France:

20	17	15	13	12	10	9	8	7	6	5	4	3	2	1

In other words, the first finisher wins 20 points, the second 17, and so on.

See Also

The complete solution is available at 4-0-convertPlaceToPointers.cpp or:

```
/home/cs124/examples/4-0-convertPlaceToPoints.cpp
```

Example 4.0 – Read Scores

Demo

This example will demonstrate how to fill a multi-dimensional array of numbers from a file, how to display the contents of the array, and how to process data from the array.

Problem

Write a program to read assignment scores from 10 students, each student completing 5 assignments. The program will then display the average score for each student and for each assignment. If there were three students, the file containing the scores might be:

```
92  87 100  84  95
71  79  85  62  81
95 100 100  92  99
```

Solution

The function to read five scores for `numStudents` individuals is the following:

```
bool readData(int grades[][5], int numStudents, const char * fileName)
{
   ifstream fin(fileName);
   if (fin.fail())
      return false;

   // read the data from the file, one row (student) at a time
   for (int iStudent = 0; iStudent < numStudents; iStudent++)
   {
      // read all the data for a given student: 5 assignments
      for (int iAssign = 0; iAssign < 5; iAssign++)
         fin >> grades[iStudent][iAssign];

      if (fin.fail())
      {
         fin.close();
         return false;
      }
   }

   fin.close();
   return true;
}
```

Observe how two loops are required: the outter loop `iStudent` to go through all the students in the list. The inner loop `iAssign` reads all the scores for a given student.

Challenge

As a challenge, modify the above program and the associated data file to contain 6 scores for each student. What needs to change? Can you create a `#define` to make changes like this easier?

See Also

The complete solution is available at 4-0-readScores.cpp or:

```
/home/cs124/examples/4-0-readScores.cpp
```

Example 4.0 – Pascal's Triangle

Problem

Pascal's triangle is a triangular array of numbers where each value is the sum of the two numbers "above" it:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Consider the number 6 in the second from bottom row. It is the sum of the 3 and the 3 from the preceding row. For a graphical representation of this relationship, please see this animation.

We will implement Pascal's triangle by turning the triangle on its side:

```
  1     1     1     1     1     1     1     1     1     1      1
  1     2     3     4     5     6     7     8     9    10     11
  1     3     6    10    15    21    28    36    45    55     66
  1     4    10    20    35    56    84   120   165   220    286
  1     5    15    35    70   126   210   330   495   715   1001
  1     6    21    56   126   252   462   792  1287  2002   3003
  1     7    28    84   210   462   924  1716  3003  5005   8008
  1     8    36   120   330   792  1716  3432  6435 11440  19448
  1     9    45   165   495  1287  3003  6435 12870 24310  43758
  1    10    55   220   715  2002  5005 11440 24310 48620  92378
  1    11    66   286  1001  3003  8008 19448 43758 92378 184756
```

Solution

One the first row, the values are 1, 1, 1, etc. From here, the first item on each new row is also the value 1. Every other item is the sum of the previous row and the previous column.

```
void fill(int grid[][SIZE])
{
   // 1. fill the first row
   for (int column = 0; column < SIZE; column++)
      grid[0][column] = 1;

   for (int row = 1; row < SIZE; row++)
   {
      // 2. The first item on a new row is 1
      grid[row][0] = 1;

      // 3. Every other item is the sum of the item above and to the left
      for (int column = 1; column < SIZE; column++)
         grid[row][column] = grid[row - 1][column] + grid[row][column - 1];
   }
}
```

See Also

The complete solution is available at 4-0-pascalsTriangle.cpp or:

```
/home/cs124/examples/4-0-pascalsTriangle.cpp
```

Problem 1

What is returned if the input is 82?

```
char convert(int input)
{
   char letters[] = "ABCDF";
   int  minRange[] =
      {90, 80, 70, 60, 0};

   for (int i = 0; i < 5; i++)
      if (minRange[i] <= input)
         return letters[i];

   return 'F';
}
```

Answer:

____

Please see page 218 for a hint.

Problem 2

What is the output of the following code?

```
int num(int n, float * a)
{
   int s = 0;

   for (int i = 0; i < n; i++)
      s += (a[i] >= 80.0);

   return s;
}

int main()
{
   cout << num(5, {71.3, 84.7, 63.9, 99.8, 70})
        << endl;

   return 0;
}
```

Answer:

____

Please see page 223 for a hint.

Problem 3

What is the output of the following code?

```
{
   char a[8] = "Rexburg";
   bool b[8] =
      {true,  false, true,  true,
       false, true,  true,  false};

   for (int i = 0; i < 8; i++)
      if (b[i])
         cout << a[i];

   cout << endl;
}
```

Answer:

____

Please see page 218 for a hint.

Problem 4

What is the syntax error?

```
{
   char letter = 'a';

   switch (letter)
   {
      case 'a':
         cout << "A\n";
      case true:
         cout << "B!\n";
         break;
         break;
      case 1:
         cout << "C!\n";
         break;
   }
}
```

Answer:

____

Please see page 287 for a hint.

Problem 5

Declare a variable to represent a Sudoku board:

____

Please see page 49 for a hint.

Problem 6

What is wrong with each of the following array declarations?

`int x = 6;` `int array[x][x];`	
`const float x = 1;` `int array[x]`	
`int array[][]`	
`int array[6 * 5 + 2][4 / 2];`	

Please see page 215 for a hint.

Problem 7

What is the output of the following code fragment?

```
{
   int array[2][2] =
      { {3, 4}, {1, 2} };

   cout << array[1][0];
}
```

Answer:

____

Please see page 313 for a hint.

Problem 8

Consider an array that is declared with the following code:

```
int array[7][21];
```

Write a prototype of a function that will accept this array as a parameter.

Answer:

____

Please see page 313 for a hint.

Assignment 4.0

Write a function to read a Tic-Tac-Toe board into an array. The file format is:

```
X O .
. . .
. X .
```

The character 'X' means that the 'X' player has taken that square. The character '.' means that the square is currently unclaimed. There is one space between each symbol.

Note: You will need to store the results in a 2D array. The function should take the filename as a parameter.

Write a function to display a Tic-Tac-Toe board on the screen. Given the above board, the output is:

```
 X | O |
---+---+---
   |   |
---+---+---
   | X |
```

Write a function to write the board to a file. The file format is the same as with the read function.

Example

The user input is **underlined**.

```
Enter source filename: board.txt
 X | O |
---+---+---
   |   |
---+---+---
   | X |
Enter destination filename: board2.txt
File written
```

Assignment

The test bed is available at:

```
testBed cs124/assign40 assignment40.cpp
```

Don't forget to submit your assignment with the name "Assignment 40" in the header.

Please see page 234 for a hint.

4.1 Allocating Memory

After Sam finished his image program involving a multi-dimensional array, something was bothering him. While the program worked with the current 12 megapixel camera he owns, it will not work with images of any other size. This struck him as short-sighted; an image program should be able to work with any size image, even image sizes not known at compile time. In an effort to work around this glaring shortcoming, he discovers memory allocation.

Objectives

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

- Allocate memory with the `new` operator.
- Free allocated memory with the `delete` operator.
- Allocate and free single and multi-dimensional arrays.

Prerequisites

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

- Declare a pointer variable (Chapter 3.3).
- Get the data out of a pointer (Chapter 3.3).
- Pass a pointer to a function (Chapter 3.3).

Overview

Dynamic memory allocation is the process of a program reserving an amount of memory that is known at runtime rather than at compile time. In other words, the program is able to reserve as much memory as the user requires, even when the programmer has no idea how much memory that will be.

This is best explained by an analogy. Imagine a developer wishing to purchase an acre of land. He goes to City Hall to acquire two things: a deed and the address of the land. The deed is a guarantee the land will not be developed by anyone else, and the address is a pointer to the land so he knows where to find it. If there is no land available, then he will have to deal with the setback and make other plans. If land is available, the developer will walk out with a valid address. With this address and deed, the developer goes off to do something useful and productive with the newly acquired acre. Of course, the acre is not clean. The previous inhabitant of the land left some landscaping and structures on the land which will need to be removed and leveled before any building occurs. The developer retains ownership of the land until his business is completed. At this time, he returns to City Hall and returns the deed and forgets the address of the land.

This process is exactly what happens when working with memory allocation. The program (developer) asks the operating system (City Hall) for a range of memory (acre of land). If the request is greater than the amount of available memory, the request returns a failure condition which the program will need to handle. Otherwise, a pointer to the memory (address) will be given to the program as well as a guarantee that no other program will be given the same memory (deed). This memory is filled with random 1's and 0's from the previous occupant (landscaping and structures on the land) which will need to be initialized (leveled). When the program is finished with the memory, it should be returned to the operating system (returns the deed to City Hall) and the pointer should be set to `NULL` (forgets the address).

There are three parts to this process:

- `NULL`: The empty address indicating a pointer is invalid
- `new`: The operator used to request memory from the operating system
- `delete`: The operator used to tell the operating system the memory is no longer needed

NULL Pointer

Up until this point, all the pointers we have used in our programs pointed to an existing location in memory. This location was always a local variable, meaning we could always assume that the pointer is referring to a valid location in memory. However, there often arises the occasion when the pointer refers to nothing. This situation requires us to mark the pointer so we can tell by inspection whether the address is valid.

To illustrate this point, remember our assignment (Assignment 3.2) where we computed the student's grade. The important thing about this assignment is that we are not to factor in the assignments the student has yet to complete. We marked these assignments with a `-1` score. In essence, the `-1` is a special token indicating the score is invalid or not yet completed. Thus, by inspection, we can tell if a score is invalid:

```
if (scores[i] == -1)
   cout << "No score for assignment " << i << endl;
```

The `NULL` pointer is essentially the same thing: an indication that a given pointer refers to no location in memory. We can check the validity of a pointer with a "`NULL`-check:"

```
if (p == NULL)
    cout << "The pointer does not refer to a valid location in memory\n";
```

Definition of NULL

The first thing to realize about `NULL` is that it is an address. While we have created pointers to characters and pointers to integers, `NULL` is a pointer to `void`. This means we can assign `NULL` to any pointer without error:

```
{
   int   * pGrade   = NULL;           // we can assign NULL to any
   float * pAccount = NULL;           //    type of pointer without
   char  * name     = NULL;           //    casting
}
```

The second thing to realize about `NULL` is that the numeric address is zero. Thus, the definition of `NULL` is:

```
#define NULL (void *)0x00000000
```

As you can well imagine, choosing zero as the `NULL` address was done on purpose. All valid memory locations are guaranteed to be not zero (the operating system owns that location: the first instruction to be executed when a computer boots). Also, zero is the only `false` value so `NULL` is the only `false` address. This makes doing a "`NULL`-check" easy:

```
if (p)                                  // same as "if (p != NULL)"
   cout << "The address of p is valid!\n";
```

Using NULL

One use of the `NULL` address is to indicate that a pointer is not valid. This can be done when there is nothing to point to. Consider the following function displaying the highest 'A' in a list of numeric grades. While commonly there is at least one 'A' in a list of student grades, it is not always the case.

```
/**************************************
 * DISPLAY HIGHEST A
 * Given a list of numeric grades
 * for a class, display the highest
 * A if one exists
 **************************************/
void displayHighestA(const int grades[],      // we won't be changing this so
                     int num)                 //    the array is a const
{
   const int * pHighestA = NULL;              // Initially no 'A's were found

   // find the highest A
   for (int count = 0; count < num; count++)  // loop through all the grades
      if (grades[count] >= 90)                // only A's please
      {
         if (pHighestA == NULL)               // if none were found, then any
            pHighestA = &(grades[count]);     //     'A' is the highest
         else if (*pHighestA < grades[count]) // otherwise, only the highest if it
            pHighestA = &(grades[count]);     //     is better than any other
      }

   // output the highest A
   if (pHighestA)                             // classic NULL check: only display the
      cout << *pHighestA << endl;             //     'A' if one was found
   else                                       // otherwise (pHighestA == NULL),
      cout << "There was not an A\n";         //     none was found
}
```

NULL check

Probably the most common use of `NULL` is to do a "`NULL`-check." Because we expect our program to be set up correctly and pointers to always have valid addresses, it is common to add an assert just before dereferencing a pointer to make sure we won't crash.

```
{
   char * pLetter = NULL;                    // first set the pointer to NULL
                                             //     to indicate it is uninitialized
   if (isalpha(value[0]))
      pLetter = value + 0;                   // observe how pLetter is set in both
   else                                      //     conditions of the IF statement
      pLetter = value + 1;

   assert(pLetter != NULL);                  // like any assert, this should never
   cout << "Letter: "                        //     fire unless the programmer made
        << *pLetter                          //     a mistake. It is better to fire
        << endl;                             //     than to crash, of course!
   pLetter = NULL;                           // indicate we are done by setting the
}                                            //     pointer back to NULL
```

If we habitually set our pointers to `NULL` and then assert just before they are dereferenced, we can catch a ton of bugs. These bugs are also easy to fix, of course; much easier than a random crash!

Allocation with New

When we declare a local variable (also known as a stack variable), the compiler takes care of memory management. The compiler makes sure that there is memory reserved for the variable as soon as the variable falls into scope. The compiler also makes sure the memory is freed as soon as the variable falls out of scope. While this is very convenient, it can also be very limiting: the compiler needs to know the size of the block of memory to be reserved and how long it will be needed. Memory allocation relaxes both of these constraints.

We request new memory from the operating system with the `new` operator. The syntax is:

```
<pointer variable> = new <data-type>;
```

If, for example, a `double` is to be allocated, it is accomplished with:

```
{
   double * p;                  // "p" is a pointer to a double
   p = new double;              // allocating a double returns a pointer to a double
}
```

We can also initialize a block of memory at allocation time. The syntax is very similar:

```
<pointer variable> = new <data-type> ( <initialization value> );
```

If, for example, you wish to allocate a character and initialize it with the letter 'A', then:

```
{
   char * p = new char('A');
}
```

This does three things: it reserves a byte of memory (`sizeof(char) == 1`), it initializes that value to 65 (`'A' == 65`), and it sends that address to the variable `p`.

Memory allocation failure

We cannot generally assume that a memory allocation is successful. In other words, it might be the case that there is no more memory to be had. Our code needs to be able to detect this condition and gracefully handle the error.

When a `new` request fails, the resulting pointer is `NULL`. However, we need to tell `new` that we wish to be notified of a failure in terms of the `NULL` pointer. This is done with the `nothrow` parameter:

```
{
   int * p = new (nothrow) int;                  // notice the nothrow parameter
   if (p == NULL)                                // failure comes in the form of a
      cout << "Memory allocation failure!\n";    //     NULL pointer
}
```

Every memory allocation should be accompanied by a `NULL` check; never assume an allocation succeeded.

Sam's Corner

There are two ways the new operator reports errors: returning a `NULL` pointer or throwing an exception. Since exception handling is a CS 165 topic, we will use the `NULL` check this semester.

Allocating arrays

We allocate arrays much the same way we allocate individual data-types. The difference, of course, is that we need to tell new how many instances of the data-type are needed. The syntax is:

```
<array variable> = new <data-type> [ <size> ];
```

Note that, unlike with array local variables, the size parameter does not need to be a constant or a literal. In other words, since `new` is essentially a function call, the compiler does not need to know how much data will be allocated; it can be determined at run-time. Consider the following code:

```
{
   // get the size of the text
   int size;                                   // memory size variables are integers
   do
   {
      cout << "How long is your name? ";
      cin  >> size;                            // continue prompting until the
   }                                           //    user gives us a positive
   while (size <= 0);                          //    size

   // allocate the memory
   char * text = new(nothrow) char [size + 1]; // allocate one more for \0
   if (!text)                                  // same as “if (text != NULL)”
      cout << "No memory! This is bad!\n";     // should return because we will
                                               //    crash in a minute…
   // prompt for the name
   cout << "What is your name? ";
   cin.getline(text, size + 1);                // treat “text” like any other string
}
```

Freeing with Delete

Once we are finished with a given block of memory, it is important to return it to the operating system so another program (or part of our own program!) can use it. This is accomplished with the `delete` operator. Note that we don't need to do this with traditional local variables because, once the variable falls out of scope, the compiler frees it for us. However, with memory allocation, the programmer (not the compiler!) indicates when the memory is no longer needed.

The syntax for the `delete` operator is:

```
delete <pointer variable>;
delete [] <array pointer variable>;
```

Consider the following example to allocate an integer and a string:

```
{
   int  * p = new int;           // allocate 4 bytes
   char * text = new char[256];  // allocate 256 bytes

   delete p;                     // no []s to free a single slot in memory
   delete [] text;               // the []s indicate an array is freed.
}
```

Sue's Tips

To make sure we don't try to use newly freed memory, always assign the pointer to `NULL` after `delete`.

Example 4.1 – AllocateValue

Demo

In the past, we used pointers to refer to data that was declared elsewhere. In the following example, the pointer is referring to memory we newly allocated: a `float`. We will allocate space for a variable, fill the variable with a value, and free the memory when completed.

Solution

There are four parts to this process: creating a pointer variable so we can remember the memory location that was allocated, allocate the memory with `new`, use the memory location using the dereference operator `*`, and freeing the memory with `delete` when finished.

```
/************************************
 * EXAMPLE
 * This is a bit contrived so I can't think
 * of a better name
 ************************************/
void example()
{
   // At first, the pointer refers to nothing. We use the NULL pointer
   // to signify the address is invalid or uninitialized
   float * pNumber = NULL;

   // now we will allocate 4 bytes for a float.
   pNumber = new(nothrow) float;
   if (!pNumber)
      return;

   // at this point (no pun intended), we can use it like any other pointer
   assert(pNumber);
   *pNumber = 3.14159;

   // Regular variables get recycled after they fall out of scopes. Not true
   // with allocated data. We need to free it with delete
   delete pNumber;
   pNumber = NULL;
}
```

Challenge

As a challenge, try to break this example into three functions: one function to allocate the memory returning a pointer to a float, one to change the value taking a pointer to a float as a parameter, and a final function to free the data.

See Also

The complete solution is available at 4-1-allocateValue.cpp or:

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

Example 4.1 – Allocate Array

Demo

This example will demonstrate how to allocate an array of integers. Unlike with traditional arrays, we will be able to prompt the user for the number of items in the array.

Problem

Write a program to prompt the user for the number of items in a list and the values for the list.

```
How many items? 3
Please enter 3 values
        # 1: 100
        # 2: 200
        # 3: 400
A display of the list:
        100
        200
        400
```

Solution

First, we will write a function to allocate the list given, as a parameter, the number of items.

```
int * allocate(int numItems)
{
   assert(numItems > 0);                       // better be a positive number!
   // Allocate the necessary memory
   int *p = new(nothrow) int[numItems];        // all the work is done here.

   // if p == NULL, we failed to allocate
   if (!p)
      cout << "Unable to allocate " << numItems * sizeof(int) << " bytes\n";
   return p;
}
```

This function is called by `main()`, which also calls a function to fill and display the list.

```
int main()
{
   int numItems = getNumItems();
   assert(numItems > 0);

   // allocate the memory
   int * list = allocate(numItems); // allocated arrays go in pointer variables
   if (list == NULL)
      return 1;

   // do something with it
   fillList(list, numItems);        // always pass the size with the array
   displayList(list, numItems);

   // make like a tree
   delete [] list;                  // never forget to release the memory
   list = NULL;                     // you can say I am a bit paranoid
   return 0;
}
```

See Also

The complete solution is available at 4-1-allocateArray.cpp or:

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

Example 4.1 – Expanding Array

Demo

This example will demonstrate how to grow an array to accomidate an unlimited amount of data. This will be accomplished by detecting when the array is full, allocating a new buffer of twice the size as the first, copying the original data to the new buffer, and freeing the original buffer.

Problem

Write a program to read all the data in a file into a single string, then report how much data was read.

```
Filename: 4-1-expandingArray.cpp
reallocating from 4 to 8
reallocating from 8 to 16
reallocating from 16 to 32
reallocating from 32 to 64
reallocating from 64 to 128
reallocating from 128 to 256
reallocating from 256 to 512
reallocating from 512 to 1024
reallocating from 1024 to 2048
reallocating from 2048 to 4096
Total size: 2965
```

Solution

Most of the work is done in the reallocate function. It will double the size of the current buffer.

```
char * reallocate(char * bufferOld, int &size)
{
   cout << "reallocating from " << size << " to " << size * 2 << endl;

   // allocate the new buffer
   char *bufferNew = new(nothrow) char[size *= 2];
   if (NULL == bufferNew)
   {
      cout << "Unable to allocate a buffer of size " << size << endl;
      size /= 2;                              // reset the size
      return bufferOld;
   }

   // copy the data into the new buffer
   int i;
   for (i = 0; bufferOld[i]; i++)             // use index because it is easier
      bufferNew[i] = bufferOld[i];            //   than two pointers
   bufferNew[i] = '\0';                       // don't forget the NULL

   // delete the old buffer
   delete [] bufferOld;

   // return the new buffer
   return bufferNew;
}
```

Challenge

It may seem a bit wasteful to double the size of the buffer with every reallocation. Would it be better to increase the size by 50%, by 200%, or by a fixed amount (say 100 characters)? Modify the above code to accommodate these different strategies and find out which has the smallest amount of wasted space and the smallest number of reallocations.

See Also

The complete solution is available at 4-1-expandingArray.cpp or:

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

Example 4.1 – Allocate Images

Demo

Our final example will demonstrate how to allocate the space necessary to display a digital picture. In this example, the user will provide the size of the image; it is not known at compile time. There is one important side-effect from this: we cannot use the multi-dimensional array notation with allocated memory because the compiler must know the size of the array to use that notation. Since the size is not known until run-time, this approach is impossible.

Solution

The code to allocate the image is the following:

```
/**********************************
 * ALLOCATE
 * Grab the memory, returning NULL if
 * anything went wrong
 **********************************/
char * allocate(int numRow, int numCol)
{
   assert(numRow > 0 && numCol > 0);

   // we allocate a 1-dimensional array and do the
   // two dimensional math ourselves
   char * image = new(nothrow) char[numRow * numCol];
   if (!image)
   {
      cout << "Unable to allocate "
           << numRow * numCol * sizeof(char)
           << " bytes for a "
           << numCol << " x " << numRow
           << " image\n";
      return NULL;
   }
   return image;
}
```

Observe how we must work with 1-dimensional arrays even though the image is 2-dimensional. Therefore we must do the transformations ourselves:

```
/**********************************
 * DISPLAY
 * Display the image. This is ASCII-art
 * so it is not exactly "High resolution"
 **********************************/
void display(const char * image, int numRow, int numCol)
{
   // paranoia
   assert(image);
   assert(numRow > 0 && numCol > 0);

   // display the grid
   for (int row = 0; row < numRow; row++)       // two dimensional loop, first
   {                                            //    the rows, then
      for (int col = 0; col < numCol; col++)    //    the columns
         cout << image[row * numCol + col];     // do the [] math ourselves
      cout << endl;
   }
}
```

See Also

The complete solution is available at:

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

Problem 1

What is the output of the following code?

```
{
   float * p = NULL;

   cout << sizeof(p) << endl;
}
```

Answer:

____

Please see page 258 for a hint.

Problem 2

What is the output of the following code?

```
{
   int a[40];

   cout << sizeof(a[42]) << endl
}
```

Answer:

____

Please see page 215 for a hint.

Problem 3

How much memory does each of the following variables require?

`char text[2]`	
`char text[] = "Software";`	
`int nums[2];`	
`bool values[8];`	

Please see page 215 for a hint.

Unit 4

Problem 4

Write the code to declare a pointer to an integer variable and allocate it.

Answer:

Please see page 327 for a hint.

Problem 5

How do you indicate that you no longer need memory that was previously allocated? Write the code to free the memory pointed to by the variable `p`.

Answer:

Please see page 328 for a hint.

Problem 6

What statement is missing in the following code?

```
{
   float * pNum = new float;

   delete pNum;

   <statement belongs here>
}
```

Answer:

Please see page 325 for a hint.

Unit 4

Problem 7

How much memory is allocated with each of the following?

`p = new double;`	
`p = new char[8];`	
`p = new int[6];`	
`p = new char(65);`	

Please see page 328 for a hint.

Problem 8

Write the code to prompt the user for a number of `float` grades, then allocate an array just big enough to store the array.

Please see page 330 for a hint.

Unit 4

Assignment 4.1

Write a program to:

- Prompt the user for the number of characters in a string
- Allocate a string of sufficient length (one more than # of characters!)
- Prompt the user for the string using `getline`
- Display the string back to the user
- Don't forget to release the memory and check for allocation failures!

Note that since the first `cin` will leave the stream pointer on the newline character, you will need to use `cin.ignore()` before `getline()` to properly fetch the section input.

Examples

Three examples… The user input is **underlined**.

Example 1

```
Number of characters: 13
Enter Text: NoSpacesHere!
Text: NoSpacesHere!
```

Example 2

```
Number of characters: 45
Enter Text: This is a ton of characters. How long is it?
Text: This is a ton of characters. How long is it?
```

Example 3 (allocation failure)

```
Number of characters: -10
Allocation failure!
```

Assignment

The test bed is available at:

```
testBed cs124/assign41 assignment41.cpp
```

Don't forget to submit your assignment with the name "Assignment 41" in the header.

Please see page 330 for a hint.

4.2 String Class

Sue has just finished the Mad Lib® assignment and found working with strings to be tedious and problematic. There were so many ways to forget the `NULL` character! She decides to spend a few minutes and create her own string type so she never has to make that mistake again. As she sits in the lab drafting a solution, Sam walks in and peers over her shoulder. "You know," he says "there is already a tool that does all those things…"

Objectives

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

- Use the String Class to solve a host of text manipulation problems.
- Understand which string operations are expensive and which are not.

Prerequisites

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

- Understand the role the `NULL` character plays in string definitions (Chapter 3.2).
- Write a loop to traverse a string (Chapter 3.2).

Overview

One of the principles of Object Oriented programming (the topic of CS 165) is encapsulation, the process of hiding unimportant implementation details from the user of a tool so he can focus on how the tool can be used to solve his problem. To date, we have used two tools exemplifying this property: input file streams (`cin` and `fin`) and output file streams (`cout` and `fout`). We learned how to use these tools to solve programming problems, but never got into the details of how they work. Another similarly powerful tool is the String Class.

The String Class is a collection of tools defined in a library allowing us to easily manipulate text. With the String Class, the programmer does not need to worry about buffer sizes (we were using 256 up to this point), `NULL` characters, or using loops to copy text. In fact, most of the most common operations work as though they are operating on a simple data type (such as an integer), allowing the programmer to forget he is even working with arrays. Consider the following simple example:

```
#include <string>                               // use the string library

/**************************************
 * DEMO: simple string-class demo
 **************************************/
void demo()
{
   string lastName;                             // the data-type is "string," no []s
   cout << "What is your last name? ";          //      as were needed with c-strings
   cin  >> lastName;                            // cin works the way you expect

   string fullName = "Mr. " + lastName;         // the + operator appends

   cout << "Hello " << fullName << endl;        // cout works the way you expect
}
```

Syntax

The syntax of the String Class is designed to be as streamlined and intuitive as possible. There are several components: the string library, declaring a string, interfacing with streams, and performing common text manipulation operations.

String library

Unlike c-strings (otherwise known as "the array type for strings"), the String Class is not built into the C++ language. These tools are provided in the string library as part of the standard namespace. Therefore, it is necessary to include the following at the top of your programs:

```
#include <string>
```

If this line is omitted, the following compiler message will appear:

```
example.cpp: In function "int main()":
example.cpp:4: error: "string" was not declared in this scope
```

Note that our compiler actually includes the `string` library as part of `iostream`. This, technically, is not part of the `iostream` library design. We should never rely on this quirk of our current compiler library and always include the string library when using the String Class.

Declaring a string

With a c-string and all other built-in data-types in C++, variables are not initialized when they are declared. This is not true with the String Class. The act of declaring a `string` variable also initializes it to the empty string.

```
{
   string text1;         // initialized to the empty string
   cout << text1;        // displays the empty string: nothing

   char text2[256];      // not initialized
   cout << text2;        // random data will be displayed
}
```

Observe how we do not specify the size of a string when we declare it. This is because the authors of the String Class did not want you to have to worry about such trivial details. Initially, the size is zero. However, as more data is added to the string, it will grow. We can always ask our string variable for its current capacity:

```
{
   string textEmpty;
   string textFull = "Introduction to Software Development";

   cout << textEmpty.capacity() << endl;       // 0:  the buffer is currently empty
   cout << textFull.capacity()  << endl;       // 64: the first power of 2 greater
}
```

Sam's Corner

The String Class buffer is dynamically allocated. This allows the buffer to grow as the need arises. It also means that we need to use the `capacity()` function to find the size rather than `sizeof()`. When the string variable falls out of scope, its storage capacity is automatically freed unless it has been allocated with new.

Stream interfaces

We were able to use `cin` and `cout` with all the built-in data-types (`int`, `float`, `bool`, `char`, `double`, etc.), but not with arrays (but we are able to use them with individual elements of arrays). The one exception to this is c-strings; `cin` and `cout` treat pointers-to-characters as c-strings. When we include the string library, `cin` and `cout` are also able to work with `string` variables:

```
{
   string text;

   cin  >> text;                        // works the same as a c-string
   cout << text << endl;                // both cin and cout work
}
```

Sue's Tips

When working with c-strings, we had to be careful to not put more data in the buffer than there was room. This was problematic with c-strings, unfortunately, because there was no way to tell `cin` how big the string is:

```
char text[10];
cin >> text;     // BUG!  The user can enter more than 9 characters
```

With the String Class, the buffer size grows to accommodate the user input. This means that it is impossible for the user to specify more input than there is space in the buffer; the buffer will simply grow until it is big enough.

```
string text;
cin >> text;       // Safe!  text can accommodate any amount
                   //        of user input
```

With c-strings, we can use `getline` to fetch an entire line of text. We can also use `getline` with the String Class, but the syntax is quite odd:

```
{
   // first the c-string syntax
   char text1[256];                      // don't forget the buffer
   cin.getline(text1, 256);              // the buffer size is a required parameter

   // now the String Class
   string text2;                         // no buffer needed here
   getline(cin, text2);                  // note how cin is the parameter!
}
```

This syntax is, unfortunately, something we will just have to remember.

Sam's Corner

The reason for the String Class' strange `getline` syntax is a bit subtle. All the functionality associated with `cin` and `cout` are in the `iostream` library. If the `getline` method associated with `cin` took a string as a parameter, then the `iostream` library would need to know about the String Class. The `iostream` library must be completely ignorant of the String Class; otherwise everyone would be required to include the string library when they do any console I/O. This would make the coupling between the libraries tighter than necessary.

The string library extends the definition of `cin` and `cout`. We will learn more about how this is done in CS 165 where we learn to make our own custom data-types work with `cin` and `cout`. .

Text manipulation

Great pains was taken to make text manipulation with `string` objects as easy and convenient as possible. We can append with the plus operator:

```
{
   string prefix =  "Mr. ";
   string postfix = "Smith";

   string name = prefix + postfix;   // concatenation with the + operator. There is a
}                                    //    FOR loop hidden here. See Sue's tip below
```

We can copy strings with the equals operator:

```
{
   string text = "CS 124";
   string copy;

   copy = text;                      // copy with the = operator. There is a FOR loop
}                                    //    hidden here. See Sue's tip below.
```

We can compare with the double-equals operator: Note that `>`, `>=`, `<`, `<=` and `!=` work as you would expect.

```
{
   string text1;
   string text2;

   cin >> text1 >> text2;

   if (text1 == text2)               // same with the other comparison operators
      cout << "Same!\n";             //    such as == != > >= < <=
   else                              //    There is a FOR loop hidden here!
      cout << "Different!\n";        //    See Sue's tip below for a hint.
}
```

Finally, we can retrieve individual members of a string with the square-brackets operator:

```
{
   string text = "CS 124";

   for (int i = 0; i < 6; i++)
      cout << text[i] << endl;       // access data with the [] operator;
}                                    //     this is very fast! No FOR loops here
```

For more functionality associated with the String Class, please see:

http://en.cppreference.com/w/cpp/string/basic_string

Sue's Tips

Be careful of hidden performance costs when working with the String Class. Seemingly innocent operations (like `=` or `==`) must have a FOR loop in them. Ask yourself "if this were done with c-strings, would a LOOP be required?"

Please see the following example for a test of the performance of the append operator in the String Class at 4-2-stringPerformance.cpp or:

```
/home/cs124/examples/4-2-stringPerformance.cpp
```

Example 4.2 – String Demo

Demo

This example will demonstrate how to declare a string, accept user input into a string, append text onto the end of a string, copy a string, and display the output on the screen.

Solution

All these common string operations will be demonstrated in a single function:

```
#include <iostream>
#include <string>           // don't forget this library
using namespace std;

/**********************************************************************
 * MAIN: Simple demo of the string class.
 **********************************************************************/
int main()
{
   string firstName;   // no []s required. The string takes care of the buffer
   string lastName;

   // cin and cout work as you would expect with the string class
   cout << "What is your name (first last): ";
   cin  >> firstName >> lastName;

   // Append ", " after last name so "Young" becomes "Young, ".
   // To do this, we will use the += operator.
   lastName += ", ";

   // Create a new string containing the first and last name so
   //          "Brigham" "Young"
   // becomes
   //          "Young, Brigham".
   // To do this we will use the + operator to combine
   // two strings and the = operator to assign the results to a new string.
   string fullName = lastName + firstName;

   // display the results of our nifty creation
   cout << fullName << endl;

   return 0;
}
```

Challenge

As a challenge, change the above program to prompt the user for his middle name. Append this new string into `fullName` so we get the expected output.

See Also

The complete solution is available at 4-2-stringDemo.cpp or:

```
/home/cs124/examples/4-2-stringDemo.cpp
```

Using string objects as file name variables

One might be tempted to use the string class for a file name. This makes the `getFilename()` function more intuitive and elegant:

```
/************************************************
 * GET FILENAME
 * Prompt the user for a filename and return it
 ************************************************/
string getFilename()                             // no pass-by-pointer parameter!
{
   string fileName;                              // local variable
   cout << "Please provide a filename: ";
   cin  >> fileName;
   return fileName;                              // we can return a local variable
}                                                //    without fear it will be destroyed
```

From this point, how do we use the string `fileName`? Consider the following code:

```
{
   string fileName = getFilename();              // this works as you might expect
   ifstream fin(fileName);                       // ERROR! Wrong parameter!
}
```

The error is:

```
example.cpp:215: error: no matching function for call to "std::basic_ofstream<char,
std::char_traits<char> >::open(std::string&)" /usr/lib/gcc/x86_64-redhat-
linux/4.4.5/../../../../include/c++/4.4.5/fstream
```

This means that the file name parameter needs to be a c-string, not a string class. To do this, we need to convert our string variable in to a c-string variable. This is done with the `c_str()` function:

```
{
   string fileName = getFilename();
   ifstream fin(fileName.c_str());               // c_str() returns a pointer to a char
}
```

Example 4.2 – String Properties

Demo

This example will demonstrate how to manipulate a string class object in a similar way to how we did this with c-strings. This will include using pointers as well as using the array notation to traverse a string.

Problem

Write a program to prompt the user for some text, display the number of characters, the number of spaces, and the contents of the string backwards.

```
Please enter some text: Software Development
        Number of characters: 20
        Number of spaces: 1
        Text backwards: tnempoleveD erawtfoS
```

Solution

We can get the length of a string with:

```
   // find the number of characters
   cout << "\tNumber of characters: "
        << text.length()
        << endl;
```

We can traverse the string using a pointer notation by getting a pointer to the start of the string with the `c_str()` method. This will return a constant pointer to a character.

```
   // find the number of spaces
   int numSpaces = 0;
   for (const char *p = text.c_str(); *p; p++)
      if (*p == ' ')
         numSpaces++;
```

Finally, we can traverse the string using the array index notation:

```
   // display the string backwards.
   cout << "\tText backwards: ";
   for (int i = text.length() - 1; i >= 0; i--)
      cout << text[i];
   cout << endl;
```

Challenge

As a challenge, try to change the case of all the characters in the string. This means converting uppercase characters to lowercase, and vice-versa. As you may recall, we did this earlier (please see p. 249).

As another challenge, try to count the number of digits in the string. You may need to use `isdigit()` form the `cctype` library to accomplish this.

See Also

The complete solution is available at 4-2-stringProperties.cpp or:

```
/home/cs124/examples/4-2-stringProperties.cpp
```

Problem 1

Declare three string class variables. Prompt the user for the values and display them:

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The names are: "Sam" "Sue," and "Sid"
```

Answer:

Please see page 338 for a hint.

Problem 2

From the code written for Problem 1, create a new string called `allNames`. This string will be created in the following way: first name, `", "`, second name, `", and "`, and third name. Display the new string.

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The names are: "Sam, Sue, and Sid"
```

Answer:

Please see page 341 for a hint.

Problem 3

From the code written for Problem 2, append the strings in alphabetical order to `allName`. You will need to use the > operator to compare `strings`, which works much like it does with integers.

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The sorted names are: "Sam, Sid, and Sue"
```

Answer:

Please see page 340 for a hint.

Assignment 4.2

Consider the child's song "Dem Bones" found at (http://en.wikipedia.org/wiki/Dem_Bones or http://www.youtube.com/). Sue would like to write a program to display the first eight verses of the song. However, realizing that the song is highly repetitive, she would like to write a function to help her with the task.

Please write a function to generate the Dem Bones song. This function takes an array of strings as input and returns a single string that constitutes the entire song as output:

```
string generateSong(string list[], int num);
```

Consider the case where `num == 4` and `list` has the following items: toe, foot, knee, and hip. This will generate the following string:

```
toe bone connected to the foot bone
foot bone connected to the knee bone
knee bone connected to the hip bone
```

Directions

For this problem, the stub function `generateSong()` as well as `main()` is written for you. Your job is to implement the function `generateSong()`. The file is located at:

```
/home/cs124/assignments/assign42.cpp
```

Please start with the above file because **most of the program is written for you!**

Assignment

The test bed is available at:

```
testBed cs124/assign42 assignment42.cpp
```

Don't forget to submit your assignment with the name "Assignment 42" in the header.

Please see page 341 for a hint.

4.3 Command Line

Sue has just finished writing a program for her mother to convert columns of Euros to Dollars. In order to make this program as convenient and user-friendly as possible, she chooses to allow the input to be specified by command line parameters.

Objectives

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

- Write a program to accept passed parameters.
- Understand jagged arrays.

Prerequisites

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

- Create a function in C++ (Chapter 1.4).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).
- Declare a pointer variable (Chapter 3.3).
- Pass a pointer to a function (Chapter 3.3).
- Declare a multi-dimensional array (Chapter 4.0).

Overview

Most operating systems (Windows, Unix variants such as Linux and OS X, and others) enable you to pass parameters to your program when the program starts. In other words, it is possible for the user to send data to the program before the program is run. Most of the Linux commands you have been using all semester take command line parameters:

Command	Example	Parameters
List the contents of a directory	`ls *.cpp`	`*.cpp`
Submit a homework assignment	`submit project4.cpp`	`project4.cpp`
Change to a new directory	`cd submittedHomework`	`submittedHomework`
Copy a file from one location to another	`cp /home/cs124/template.cpp assignment43.cpp`	`/home/cs124/template.cpp assignment43.cpp`

In each of these cases, the programmers configured their programs to accept command line parameters. The purpose of this chapter is to learn how to do this for our programs.

Syntax

To configure your program to accept command line parameters, it is necessary to define `main()` a little differently than we have done this semester up to this point.

```
int main(int argc, char ** argv)
{
   …
}
```

Observe the two cryptic parameter names. In the past we left the parameter list of `main()` empty. When this list is empty, we are ignoring any command line parameters the user may send to us. However, when we specify the `argc` and `argv` parameters, we can access the user-sent data from within our program.

argc

The first parameter is the number of arguments or parameters the user typed. For example, if the program name is `a.out` and the user typed...

```
a.out one two three
```

... then `argc` would equal 4 because four arguments were typed: `a.out, one, two, three`. Just like with any passed parameter, the user can name this anything he wants. The convention is to use the name `argc`, but you may want to name it something more meaningful like `countArgument`.

argv

The second parameter is the list of arguments. In all cases, we get an array of c-strings. Back to our example above where the user typed...

```
a.out one two three
```

... then `argv[0]` equals "`a.out`" and `argv[1]` equals "`one`". Again, note that the first parameter is always the name of the program. As with `argc`, `argv` is not the best name. It means "argument vector" or "list of unknown arguments." Possibly a better name would be `listArguments`.

Note the unusual `**` notation for the syntax of `argv`. This means that we have a pointer to a pointer. Possibly a better definition would be:

```
int main(int argc, char * argv[])
{
   ...
}
```

This might better illustrate what is going on. Here, we have an array of c-strings. How many items are in the c-string? This is where `argc` comes into play.

Example 4.3 – Reflect the Command Line

Demo

This example will demonstrate how to accept data passed from the user through the command line.

Problem

Write a program to display on the screen the name of the program, how many parameters were passed through the command line, and the contents of each parameter.

```
/home/cs124/examples> a.out one two three
The name of the program is: a.out
There are 3 parameters
        Parameter 1: one
        Parameter 2: two
        Parameter 3: three
```

Solution

The first thing to look for is how `main()` has two parameters. Of course we can call these anything we want, but `argc` and `argv` are the standard names. Next, observe how `argv[0]` is the name of the program. Finally, we access earch parameter with `argv[iArg]`.

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

/*******************************************
 * MAIN: Reflect back to the user what he
 * typed on the command prompt
 *******************************************/
int main(int argc, char ** argv)       // again, MAIN really takes two parameters
{
   // name of the program
   cout << "The name of the program is: "
        << argv[0]                     // the first item in the list is always
        << endl;                       //    the command the user typed

   // number of parameters
   cout << "There are "
        << argc - 1                    // don’t forget to subtract one due to
        << " parameters\n";            //    the first being the program name

   // show each parameter
   for (int iArg = 1; iArg < argc; iArg++)   // standard command line loop
      cout << "\tParameter" << iArg
           << ": " << argv[iArg] << endl;    // each argv[i] is a c-string

   return 0;
}
```

Challenge

As a challenge, try to rename `a.out` (using the `mv` command) and run it again. How do you suppose the program knows that it was renamed after compilation?

See Also

The complete solution is available at 4-3-commandLine.cpp or:

```
/home/cs124/examples/4-3-commandLine.cpp
```

Example 4.3 – Get Filename

Demo

This example will demonstrate one of the most common uses for command line parameters: fetching the filename from the user. This will be done using the string class.

Problem

Write a program to prompt the user for a filename. This can be provided either through a command line parameter or, if none was specified, through a traditional prompt.

```
/home/cs124/examples> a.out fileName.txt
The filename is "fileName.txt"
```

```
/home/cs124/examples> a.out
Please enter the filename: fileName.txt
The filename is "fileName.txt"
```

Solution

There are three parts to this program. First, the program will determine if an erroneous number of parameters were specified. If this proves to be the case, the program exists with a suitable error message. Next, the program prompts the user for a filename if there are no parameters specified on the command line. Finally, if there is a parameter specified, it is copied into the `fileName` variable.

```
int main(int argc, char ** argv)
{
   // ensure the correct number of parameters was specified
   if (argc > 2)   // one for the name of the program, one for the filename
   {
      cout << "Unexpected number of parameters.\nUsage:\n";
      cout << "\t" << argv[0] << " [filename]\n";
      return 1;
   }

   // parse the command line
   string fileName;
   if (argc == 1) // only the program name was specified
      fileName = getFilename();
   else
      fileName = argv[1];

   // display the results
   cout << "The filename is \"" << fileName << "\"\n";

   return 0;
}
```

Challenge

As a challenge, can you do this without the `string` class? Hint: use `strcpy()` from the `cstring` library.

Another challenge is to modify Project 3 to accept a filename as a command line parameter. Can you do this for Project 4 as well?

See Also

The complete solution is available at 4-3-getFilename.cpp or:

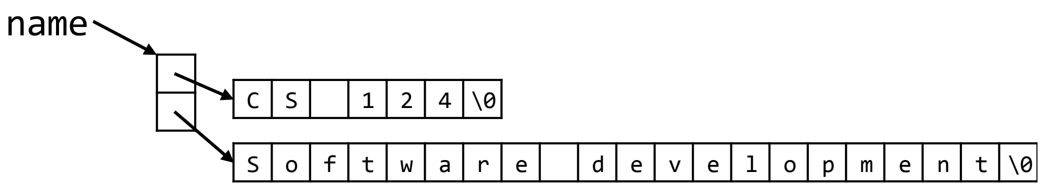
```
/home/cs124/examples/4-3-getFilename.cpp
```

Jagged Arrays

Multi-dimensional arrays can be thought of as arrays of arrays where each row is the same length and each column is the same length. In other words, multi-dimensional arrays are square. There is another form of "arrays of arrays" where each row is not the same length. We call these "jagged arrays." Consider the following code:

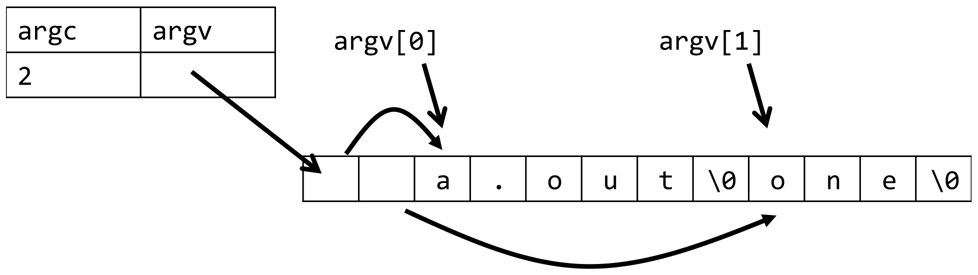
```
{
   char * name[] =
   {
      "CS 124",
      "Software development"
   };
}
```

The first string is a different size than the second. Thus we have:



In this example, we have declared an array of pointers to characters (`char *`). Each one of these pointers is pointing to a string literal. Observe how each string is of a different size. Also, the two strings do not need to be next to each other in memory as multi-dimensional arrays do.

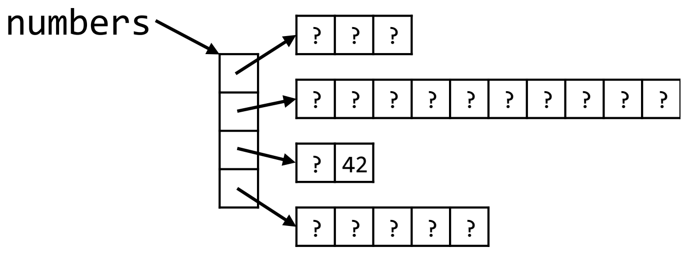
It turns out that `argv` works much like a pointer to dynamically allocated memory. The only difference is that the operating system typically puts each command line parameter immediately after the array of pointers in memory:



Example 4.3 – Jagged Array of Numbers

Demo

This example will demonstrate how to create a jagged array of integers. In this case, each row will be allocated dynamically. There will be four rows, each containing a different number of items.



Solution

There are five steps to setting up an jagged array of numbers:

1. Allocate the array of pointers. This must happen before the rows are allocated.
2. Allocate each row individually. This will require separate `new` statements.
3. Use the array. In this case, the value '42' will be assigned to one cell.
4. Free the individual rows.
5. Free the array of pointers. This must happen after the individual rows are freed.

Note that if the number of rows is known at compile time then step 1 and 5 can be skipped.

```
{
   int ** numbers;                    // pointer to a pointer to an integer

   // 1. allocate the array of arrays.
   numbers = new int *[4];            // an array of pointers to integers

   // 2. allocate each individual array
   numbers[0] = new int[3];           // an array of integers
   numbers[1] = new int[10];
   numbers[2] = new int[2];
   numbers[3] = new int[5];

   // 3. assign a value
   numbers[2][1] = 42;                // access is the same as with standard
                                      //   multi-dimensional array
   // 4. free each array
   for (int i = 0; i < 4; i++)        // we must free each individual array or we
      delete [] numbers[i];           //   will forget about them and have a leak

   // 5. free the array of arrays
   delete [] numbers;                 // finally free the original array
}
```

Challenge

As a challenge, change the code to work with arrays of floats. Make it work with five rows instead of four (making sure to free each row when finished). Finally, try to fill each item in the array with a number.

See Also

The complete solution is available at 4-3-jaggedArrayNumber.cpp or:

```
/home/cs124/examples/4-3-jaggedArrayNumber.cpp
```

Example 4.3 – Jagged Array of Strings

Demo

This example will demonstrate how to create a jagged array of strings. This allows us to use exactly the amount of memory required to represent the user's data.

Problem

Write a program to prompt the user for his name and store his name in a jagged array.

```
How man letters are there in your first, middle, and last name?
4 5 20
Please enter your first name: John
Please enter your middle name: Jacob
Please enter your last name: Jingleheimer Schmidt
Your name is "John Jacob Jingleheimer Schmidt"
```

Solution

To allocate the jagged array, we need to know the size of each row (`sizeFirst`, `sizeMiddle`, `sizeLast`).

```
char ** names;
names = new char *[3];  // three names
names[0] = new char[sizeFirst  + 1];
names[1] = new char[sizeMiddle + 1];
names[2] = new char[sizeLast   + 1];
```

From here, it behaves like any other multi-dimensional array.

```
// fill the jagged array
 cout << "Please enter your first name: ";
 cin.getline(names[0], sizeFirst + 1);
 cout << "Please enter you middle name: ";
 cin.getline(names[1], sizeMiddle + 1);
 cout << "Please enter your last name: ";
 cin.getline(names[2], sizeLast + 1);

 // display the results
 cout << "Your name is \""
      << names[0] << ' '
      << names[1] << ' '
      << names[2] << "\"\n";
```

Finally, it is important to free the rows and the pointers in the correct order.

```
delete [] names[0];
delete [] names[1];
delete [] names[2];
delete [] names;
```

Challenge

As a challenge, can you add another row to correspond to the user's title ("Dr.")? Again, don't forget to allocate the row and free the row.

See Also

The complete solution is available at 4-3-jaggedArrayString.cpp or:

```
/home/cs124/examples/4-3-jaggedArrayString.cpp
```

Problem 1

Given the following program:

```
int main(int argc, char ** argv)
{
   cout << argv[0] << endl;
   return 0;
}
```

What is the output if the user runs the following command:

```
%a.out one two three
```

Answer:

____

Please see page 348 for a hint.

Problem 2

Given the following program:

```
int main(int argc, char ** argv)
{
   <code goes here>
}
```

Write the code that will display the value "three":

```
%a.out one two three
three
```

Answer:

____

Please see page 348 for a hint.

Problem 3

Given the following program:

```
int main(int argc, char ** argv)
{
   cout << argc << endl;

   return 0;
}
```

What is the output if the user runs the following command:

```
%a.out one two
```

Answer:

____

Please see page 348 for a hint.

Problem 4

Given the following program:

```
int main(int argc, char ** argv)
{
   cout << atoi(argv[1])
         + atoi(argv[2]);
   return 0;
}
```

What is the output if the user types in:

```
a.out 4 5 6
```

Answer:

____

Please see page 348 for a hint.

Problem 5

Describe, in English, the functionality of the following program:

```
int main(int argc, char ** argv)
{
   while (--argc)
      cout << argv[argc] << endl;
   return 0;
}
```

Answer:

____

Please see page 347 for a hint.

Unit 4

Problem 6

Write a program to compute the absolute value given numbers specified on the command line:

```
%a.out -4.2 96.2 -3.90
The absolute value of -4.2 is 4.2
The absolute value of 96.2 is 96.2
The absolute value of -3.90 is 3.9
```

Answer:

Please see page 347 for a hint.

Assignment 4.3

Write a program to convert feet to meters. The conversion from feet to meters is:

1 foot = 0.3048 meters

The input will be numbers passed on the command line. You will want to use the library function `atof` to convert a string into a float. For example, consider the following code:

```
#include <cstdlib>      // the library for atof()
#include <iostream>
using namespace std;

int main()
{
   char text[] = "3.14159";      // a c-string
   float pi;                     // the float where the answer will go
   pi = atof(text);              // atof() translates the c-string into a float
   cout << pi << endl;           // this better be 3.14159

   return 0;
}
```

Pay special attention to the `#include <cstdlib>` code; you will need that for this assignment.

Example

Consider the output of a program called a.out:

```
a.out 1 2 3
1.0 feet is 0.3 meters
2.0 feet is 0.6 meters
3.0 feet is 0.9 meters
```

Assignment

The test bed is available at:

```
testBed cs124/assign43 assignment43.cpp
```

Don't forget to submit your assignment with the name "Assignment 43" in the header.

Please see page 348 for a hint.

4.4 Instrumentation

Sue has just finished her first draft of a function to solve a Sudoku puzzle. It seems fast, but she is unsure how fast it really is. How do you measure performance on a machine that can do billions of instructions per second? To get to the bottom of this, she introduces counters in key parts of her program to measure how many times certain operations are performed.

Objectives

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

- Instrument a function to determine its performance characteristics.

Prerequisites

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

- Search for a value in an array (Chapter 3.1).
- Look up a value in an array (Chapter 3.1).
- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Predict the output of a code fragment containing an array (Chapter 3.0).

Overview

One of the most important characteristics of a sorting or search algorithm is how fast it accomplishes its task. There are several ways we can measure speed: the number of steps it takes, the number of times an expensive operation is performed, or the elapsed time.

To measure the number of steps the operation takes, we need to add a counter to the code. This counter will increment with every step, resulting in one metric of the speed of the algorithm. We call the process of adding a counter "**instrumenting**." Instrumenting is the process of adding code to an existing function which does not change the functionality of the task being measured. Instead, the instrumenting code measures how the task was performed. For example, if consider the following code determining if two strings are equal:

```
/**************************************************
 * IS EQUAL
 * Simple function to tell if two strings are equal
 **************************************************/
bool isEqual(const char * text1, const char * text2)
{
   while (*text1 == *text2 && *text2)
   {
      text1++;
      text2++;
   }

   return *text1 == *text2;
}
```

This same code can be instrumented by adding a counter keeping track of the number of characters compared:

```
/**************************************************
 * IS EQUAL
 * Same function as above except it will keep track
 * of how many characters are looked at to determine
 * if the two strings are equal. This function is
 * instrumented
 **************************************************/
bool isEqual(const char * text1, const char * text2)
{
   int numCompares = 1; // keeps track of the number of compares

   while (*text1 == *text2 && *text2)
   {
      text1++;
      text2++;
      numCompares++;  // with every compare (in the while loop), add one!
   }

   cout << "It took " << numCompares << " compares\n";
   return *text1 == *text2;
}
```

Write a program to manage your personal finances for a month. This program will ask you for your budget income and expenditures, then for how much you actually made and spent. The program will then display a report of whether you are on target to meet your financial goals

Sue's Tips

It is a good idea to put your instrumentation code in #ifdefs so you can easily remove it before sending the code to the customer. The most convenient way to do that is with the following construct:

```
#ifdef DEBUG
#define Debug(x) x
#else
#define Debug(x)
#endif
```

Observe how everything in the parentheses is included in the code if `DEBUG` is defined, while it is removed when it is not. With this code, we would say something like:

```
Debug(numCompare++);
```

Again, if `DEBUG` were defined, the `numCompare++;` would appear in the code. If it were not defined, then an empty semi-colon would appear instead.

Example 4.4 – Instrument the Bubble Sort

Demo

This example will demonstrate how to instrument a complex function: a sort algorithm. This algorithm will order the items in an array according to a given criterion. In this case, the criterion is to reorder the numbers in `array` to ascending order. The question is: how efficient is this algorithm?

Solution

Possibly the simplest algorithm to order a list of values is the **Bubble Sort**. This algorithm will first find the largest item and put it at the end of the list. Then it will find the second largest and put it in the second-to-last spot (`iSpot`) and so on.

We will instrument this function with the `numCompare` variable. Initially set to zero, we will increment `numCompare` each time a pair of numbers (`array[iCheck] > array[iCheck + 1]`)is compared. For convenience, we will return this value to the caller.

```
/**********************************************************************
 * BUBBLE SORT
 * Instrumented version of the Bubble Sort. We will return the number
 *  of times elements in the array were compared.
 **********************************************************************/
int bubbleSort(int array[], int numElements)
{
   // number of comparisons is initially zero
   int numCompare = 0;

   // did we switch two values on the last time through the outer loop?
   bool switched = true;

   // for each spot in the array, find the item that goes there with iCheck
   for (int iSpot = numElements - 1; iSpot >= 1 && switched; iSpot--)
      for (int iCheck = 0, switched = false; iCheck <= iSpot - 1; iCheck++)
      {
         numCompare++;       // each time we are going to compare, add one

         if (array[iCheck] > array[iCheck + 1])
         {
            int temp = array[iCheck];     // swap 2 items if out of order
            array[iCheck] = array[iCheck + 1];
            array[iCheck  + 1] = temp;
            switched = true;              // a swap happened, do outer loop again
         }
      }

   return numCompare;
}
```

It is important to note that instrumenting should not alter the functionality of the program. Removing the instrumentation code should leave the algorithm unchanged.

See Also

The complete solution is available at 4-4-bubbleSort.cpp or:

```
/home/cs124/examples/4-4-bubbleSort.cpp
```

Example 4.4 – Instrument Fibonacci

Demo

This example will demonstrate how to instrument two functions computing the Fibonacci sequence. It will not only demonstrate how to add counters at performance-critical locations in the code, but will demonstrate how to surround the instrumentation code with `#ifdefs` enabling the programmer to conveniently remove the instrumentation code or quickly add it back.

Problem

As you may recall, the Fibonacci sequence is defined as the following:

$$F(n) \coloneqq \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Write a program to compute the nth Fibonacci number.

```
How many Fibonacci numbers shall we identify? 10
Array method:     55
Recursive method: 55
```

In debug mode (compiled with the `-DDEBUG` switch) , the program will display the cost:

```
How many Fibonacci numbers shall we identify? 10
Number of iterations for the array method: 9
Number of iterations for the recursive method: 177
```

Solution

In order to not influence the normal operation of the Fibonacci functions, the following code was added to the top of the program:

```
#ifdef DEBUG          // all the code that will only execute if DEBUG is defined
int countIterations = 0;
#define increment()     countIterations++
#define reset()         countIterations = 0
#define getIterations() countIterations

#else // !DEBUG
#define increment()
#define reset()
#define getIterations()  0
#endif
```

Thus in ship mode (when `DEBUG` is not defined), the "functions" `increment()`, `reset()`, and `getIterations()` are defined as nothing. In debug mode, these manipulate the global variable `countIterations`. Normally global variables are dangerous. However, since this variable is only visible when `DEBUG` is defined, its damage potential is contained. The last thing to do is to carefully put `reset()` before we start counting and `getIterations()` when we are done counting.

```
   reset();
   int valueArray = computeFibonacciArray(num);
   int costArray  = getIterations();
```

This, coupled with `increment()` when counting constitutes all the instrumentation code in the program.

See Also

The complete solution is available at 4-4-fibonacci.cpp or:

```
/home/cs124/examples/4-4-fibonacci.cpp
```

Assignment 4.4

Our assignment this week is to determine the relative speed of a linear search versus a binary search using instrumentation. To do this, start with the code at:

```
/home/cs124/assignments/assign44.cpp
```

Here, you will need to modify the functions `binary()` and `linear()` to count the number of comparisons that are performed to find the target number.

Next, you will need to modify `computeAverageLinear()` and `computeAverageBinary()` to determine the number of compares on average it takes to find each element in the array.

Example

Consider the file `numbers.txt` that has the following values:

```
1 4 10 36 47 92 100 110 125 136 142 143 150 160 167
```

For the above example, the list is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	10	36	47	92	100	110	125	136	142	143	150	160	167

When we run the program on the above list, the program will compute how long it takes to find an element in the list using the linear search method and using the binary search method. If I call `linear()` (a function performing a linear search from left to right) with the search parameter set to `4`, then I will find it with two comparisons. This means `linear(list, num, 4) == 2` because the function `linear()` will return the number of comparisions, `list` contains the list of numbers, and `num` is the number of items in `list`. Thus `linear(list, num, 47) == 5`. To find the average cost of the linear search, the equation will be:

```
        (linear(list, num, 1) + linear(list, num, 4) + … + linear(list, num, 167)) / num ==
                                 (1 + 2 + ... + 15) / 15 ==
                                        120 / 15 ==
                                           8.0
```

To compute the average cost of the binary search, the equation will be:

```
      (binary(list, num, 1) + binary(list, num, 4) + … + binary(list, num, 167)) / num == 3.3
```

The user input is **underlined**.

```
Enter filename of list: numbers.txt
Linear search:         8.0
Binary search:         3.3
```

Assignment

The test bed is available at:

```
testBed cs124/assign44 assignment44.cpp
```

Don't forget to submit your assignment with the name "Assignment 44" in the header.

Please see page 358 for a hint.

Unit 4 Practice Test

Practice 4.2

Sam has read that many programmers get paid by the number of lines of code they generate. Wanting to maximize the size of his paycheck, he has decided to write a program to count the number of lines in a given file.

Problem

Write a program to perform the following tasks:

1. Attempt to gather the filename from the command line
2. If no filename is present, prompt the user for the filename
3. Loop through the file, counting the number of lines
4. Display the results using the correct tense for empty, 1, and more than one.

Example

The user may specify the filename from the command line. Assume the file "`example1.txt`" has 2 lines:

```
a.out example1.txt
        example1.txt has 2 lines.
```

If no file is specified on the command line, then the program will prompt the user for a file. Assume the file "`example2.txt`" has 1 line of text:

```
a.out
Please enter the file name: example2.txt
        example2.txt has 1 line.
```

Note how "`line`" is singular. Finally, if the file is empty or non-existent, then the following message will appear:

```
a.out missing.txt
        missing.txt is empty.
```

Grading

The test bed is:

```
testBed cs124/practice42 practice42.cpp
```

The sample solution is:

```
/home/cs124/tests/practice42.cpp
```

Continued on the next page

Continued from previous page

Grading for Test4

Sample grading criteria:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
Modularization 10%	Perfect cohesion and coupling	No bugs exist in calling functions, though modularization is not perfect	Redundant data passed between functions	A bug exists passing parameters or calling a function	Only one function used
Command line 10%	Input logic "elegant"	Able to access the filename from the command line and from a prompt	One bug	An attempt was made to accept input from both the command line and from prompts.	No attempt was made to access the command line parameters
File 20%	"Good" and perfect error detection	Able to fetch all the text from a file	One bug	Elements of the solution exist	No attempt was made to read from the file
Problem Solving 30%	The most elegant and efficient solution was found	Zero test bed errors	Correct on at least one test case	Elements of the solution are present	No attempt was made to count all the words <u>or</u> program failed to compile
Handle tense on output 10%	The most elegant and efficient solution was found	All three cases handled	One bug	Logic exists to attempt to detect need for different tenses	No attempt was made to handle tense
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 4 Project : Sudoku

Write a program to allow the user to play Sudoku. For details on the rules of the game, see:

http://en.wikipedia.org/wiki/Sudoku

The program will prompt the user for the filename of the game he or she is currently working on and display the board on the screen. The user will then be allowed to interact with the game by selecting which square he or she wishes to change. While the program will not solve the game for the user, it will ensure that the user has not selected an invalid number. If the user types 'S' in the prompt, then the program will show the user the possible valid numbers for a given square. When the user is finished, then the program will save the board to a given filename and exit.

This project will be done in three phases:

- Project 11 : Design the Sudoku program
- Project 12 : Allow the user to interact with the Sudoku game
- Project 13 : Enforce the rules of the Sudoku game

Interface Design

Consider a game saved as `myGame.txt`:

```
7 2 3 0 0 0 1 5 9
6 0 0 3 0 2 0 0 8
8 0 0 0 1 0 0 0 2
0 7 0 6 5 4 0 2 0
0 0 4 2 0 7 3 0 0
0 5 0 9 3 1 0 4 0
5 0 0 0 7 0 0 0 3
4 0 0 1 0 3 0 0 6
9 3 2 0 0 0 7 1 4
```

Note that '`0`' corresponds to an unknown value. The following is an example run of the program. Please see the following program for an example of how this works:

```
/home/cs124/projects/prj13.out
```

Program starts

An example of input is **underlined**.

```
Where is your board located? myGame.txt
```

With the filename specified, the program will display a menu of options:

```
Options:
   ?  Show these instructions
   D  Display the board
   E  Edit one square
   S  Show the possible values for a square
   Q  Save and quit
```

After this, the board as read from the file will be displayed:

```
   A B C D E F G H I
1  7 2 3|     |1 5 9
2  6    |3   2|    8
3  8    |  1  |    2
   -----+-----+-----
4    7  |6 5 4|  2
5      4|2   7|3
6    5  |9 3 1|  4
   -----+-----+-----
7  5    |  7  |    3
8  4    |1   3|    6
9  9 3 2|     |7 1 4
```

Here, the user will be prompted for a command (the main prompt).

```
> Z
```

Please note that you will need a newline, a carat (`'>'`), and a space before the prompt.

The next action of the program will depend on the user's command. If the user types an invalid command, then the following message will be displayed:

```
ERROR: Invalid command
```

Show Instructions

If the user types '`?`', then the menu of options will be displayed again. These are the same instructions that are displayed when the program is first run.

Display the Board

If the user types '`D`', then the board will be redrawn. This is the same as the drawing of the board when the program is first run.

Save and Quit

If the user types '`Q`', then he or she will be prompted for the filename:

```
What file would you like to write your board to: newGame.txt
```

The program will display the following message:

```
Board written successfully
```

Then the program will terminate.

Edit One Square

If the user types '`E`', then the program will prompt him for the coordinates and the value for the square to be edited:

```
What are the coordinates of the square: A1
```

If the value is invalid or if the square is already filled, then the program will display one of the following message and return to the main prompt:

```
ERROR: Square 'zz' is invalid
ERROR: Square 'A1' is filled
```

With a valid coordinate, then the program next prompts the user for the value:

```
What is the value at 'A1': 9
```

If the user types a value that is outside the accepted range ($1 \le$ value ≤ 9) or does not follow the rules of Sudoku, then a message appears and the program returns to the main prompt:

```
ERROR: Value '9' in square 'A1' is invalid
```

Show Possible Values

If the user types 's', then the program will prompt him for the coordinates and display the possible values:

```
What are the coordinates of the square: A1
```

The same parsing logic applies here as for the Edit One Square case. Once the user has selected a valid coordinate, then the program will display the possible values:

```
The possible values for 'A1' are: 1, 5, 8, 9
```

After the message appears, the program returns to the main prompt.

Project 11

The first part of the project is the design document. This consists of three parts:

1. Create a structure chart describing the entire Sudoku program.

2. Write the pseudocode for the function `computeValues()`. This function will take as parameters coordinates (row and column) for a given square on the board and calculate the possible values for the square. To do this, `computeValues()` must be aware of all the rules of Sudoku. Make sure to include both the logic for the rules of the game (only one of each number on a row, column, and inside square), but also to display the values.

3. Write the pseudocode for the function `interact()`. This function will prompt the user for his option (ex: '`D`' for "`Display the board`" or '`S`' for "`Show the possible values for a square`") and call the appropriate function to carry out the user's request. Note that the program will continue to play the game until the user has indicated he is finished (with the '`Q`' option).

On campus students are required to attach this rubric to your design document. Please self-grade.

Project 12

The second part of the Sudoku Program project (the first part being the design document due earlier) is to write the code necessary to make the Sudoku appear on the screen:

```
Where is your board located? prj4.txt
Options:
   ?  Show these instructions
   D  Display the board
   E  Edit one square
   S  Show the possible values for a square
   Q  Save and Quit

   A B C D E F G H I
1  7 2 3|     |1 5 9
2  6    |3   2|    8
3  8    |  1  |    2
   -----+-----+-----
4    7  |6 5 4|  2
5      4|2   7|3
6    5  |9 3 1|  4
   -----+-----+-----
7  5    |  7  |    3
8  4    |1   3|    6
9  9 3 2|     |7 1 4

> E
What are the coordinates of the square: e5
What is the value at 'E5': 8

> q
What file would you like to write your board to: deleteMe.txt
```

Perhaps the easiest way to do this is in a four-step process:

1. Create the framework for the program using stub functions based on the structure chart from your design document.
2. Write each function. Test them individually before "hooking them up" to the rest of the program.
3. Verify your solution with testBed:

```
testBed cs124/project12 project12.cpp
```

4. Submit it with "`Project 12, Sudoku`" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj12.out
```

Project 13

The final part of the Sudoku Program project is to enforce the rules of Sudoku. This means that there can be only one instance of a given number on a row, column, or inside square.

```
Where is your board located? prj4.txt
Options:
   ?  Show these instructions
   D  Display the board
   E  Edit one square
   S  Show the possible values for a square
   Q  Save and Quit

   A B C D E F G H I
1  7 2 3|     |1 5 9
2  6    |3   2|    8
3  8    |  1  |    2
   -----+-----+-----
4    7  |6 5 4|  2
5      4|2   7|3
6    5  |9 3 1|  4
   -----+-----+-----
7  5    |  7  |    3
8  4    |1   3|    6
9  9 3 2|     |7 1 4

> s
What are the coordinates of the square: b2
The possible values for 'B2' are: 1, 4, 9

> e
What are the coordinates of the square: b2
What is the value at 'B2': 2
ERROR: Value '2' in square 'B2' is invalid

> q
What file would you like to write your board to: deleteMe.txt
```

Perhaps the easiest way to do this is in a four-step process:

1. Start with the code written in Project 12.
2. Fix any necessary bugs.
3. Verify your solution with testBed:

```
testBed cs124/project13 project13.cpp
```

4. Submit it with "`Project 13, Sudoku`" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj13.out
```

A. Elements of Style

While the ultimate test of a program is how well it performs for the user, the value of the program is greatly limited if it is difficult to understand or update. For this reason, it is very important for programmers to write their code in the most clear and understandable way possible. We call this "programming style."

Elements of Style

Perhaps the easiest way to explain coding style is this: give the bugs no place to hide. When our variable names are clearly and precisely named, we are leaving little room for confusion or misinterpretation. When things are always used the same way, then readers of the code have less difficulty understanding what they mean.

There are four components to our style guidelines: variable and function names, spacing, function and program headers, general comments, and other standards.

Variable and function names

The definitions of terms and acronyms of a software program typically consist of variable declarations. While variables are declared in more than one location, the format should be the same. Using descriptive identifiers reduces or eliminates the need for explanatory comments. For example, `sum` tells us we are adding something; `sumOfSquares` tells us specifically what we are adding. Use of descriptive identifiers also reduces the need for comments in the body of the source code. For example, `sum += x * x;` requires explanation. On the other hand, `sumOfSquares += userInput * userInput;` not only tells us where the item we are squaring came from, but also that we are creating a sum of the squares of those items. If identifiers are chosen carefully, it is possible to write understandable code with very few, if any, comments. The following are the University conventions for variable and function names:

Identifier	Example	Explanation
Variable	`sumOfSquares`	Variables are nouns so it follows that variable names should be nouns also. All the words are TitleCased except the first word. We call this style camelCase.
Function	`displayDate()`	Functions are verbs so it follows that function names should also be verbs. Like variables, we camelCase functions.
Constant	`PI`	Constants, include `#defines`, are ALL_CAPS with an underscore between the words.
Data Types	`Date`	Classes, enumeration types, type-defs, and structures are TitleCased with the first letter of each word capitalized. These are CS 165 constructs.

Function and program headers

It takes quite a bit of work to figure out what a program or function is trying to do when all the reader has is the source code. We can simplify this process immensely by providing brief summaries. The two most common places to do this are in function and program headers.

A function header appears immediately before every function. The purpose is to describe what the program does, any assumptions made about the input parameters, and describe the output. Ideally, a programmer should need no more information than is provided in the header before using a function. An example of a function header is the following:

```
/*****************************************************
 * GET YEAR
 * Prompt the user for the current year. Error checking
 * will be performed to ensure the year is valid
 *    INPUT:  None (provided by the user)
 *    OUTPUT: year
 *****************************************************/
```

A program header appears at the beginning of every file. This identifies what the program does, who wrote it, and a brief description of what it was written for. Our submission program reads this program header to determine how it is to be turned in. For this reason, it is important to start every program with the template provided at `/home/cs124/template.cpp`. The header for Assignment 1.0 is:

```
/************************************************************************
* Program:
*    Assignment 10, Hello World
*    Brother Helfrich, CS124
* Author:
*    Sam Student
* Summary:
*    Display the text “Hello world” on the screen.
*    Estimated:  0.7 hrs
*    Actual:     0.5 hrs
*      I had a hard time using emacs.
************************************************************************/
```

General Comments

We put comments in our code for several reasons:

- To describe what the next few lines of code do
- To explain to the reader of the code why code was written a certain way
- To write a note to ourselves reminding us of things that still need to be done
- To make the code easier to read and understand

Since a comment can be easily read by a programmer and source code, in many cases, must be decoded, one purpose of comments is to clarify complicated code. Comments can be used to convey information to those who will maintain the code. For example, a comment might provide warning that a certain value cannot be changed without impacting other portions of the program. Comments can provide documentation of the logic used in a program. Above all else, comments should add *value* to the code and should not simply restate what is obvious from the source code.

The following are meaningless comments and add no value to the source code:

```
int i; // declare i to be an integer
i = 2; // set i to 2
```

On the other hand, the following comments add value:

```
int i; // indexing variable for loops
i = 2; // skip cases 0 and 1 in the loop since they were processed earlier
```

With few exceptions, we use line comments (`//`) rather than block comments (`/* … */`) inside functions. Please add just enough comments to make your code more readable, but not so many that it is overly verbose. There is no hard-and-fast rule here.

"Commenting out" portions of the source code can be an effective debugging technique. However, these sections can be confusing to those who read the source code. The final version of the program should not contain segments of code that have been commented out.

Spacing

During the lexing process, the compiler removes all the spaces between keywords (such as `int`, `for`, or `if`) and operators (such as `+` or `>=`). To make the code human-readable, it is necessary to introduce spaces in a consistent way. The following are the University conventions for spaces:

Rule	Example	Explanation
Operators	`tempC = 5.0 / 9.5 (tempF - 32.0)`	There needs to be one space between all operators, including arithmetic (`+` and `%`), assignment (`=` and `+=`) and comparison (`>=` and `!=`)
Indention	`{` `   int answer = 42;` `   if (answer > 100)` `      cout << "Wrong answer!";` `}`	With every level of indention, add three white spaces. Do not use the tab character to indent.
Functions		Put one blank line between functions. More than one results in unnecessary scrolling, less feels cramped
Related code	`   // get the data` `   float income;` `   cout << "Enter income: ";` `   cin  >> income;`	Much like an essay is sub-divided into paragraphs, a function can be sub-divided into related statements. Each statement should have a blank line separating them.

Other Standards

Because of the way printers and video displays handle text, readability is improved by keeping each line of code less than 80 characters long.

Subroutines and classes should be ordered in a program such that they are easy to locate by the reader of the source code. This usually means grouping functions that perform similar operations. For example, all input functions should be next to each other in a file, as should output functions.

Each curly brace should be on its own line; this makes them easier to match up.

Please make sure there are no spelling or grammatical errors in your source code.

Style Checklist

Comments

- program introductory comment block
- identify program
- identify instructor and class
- identify author
- brief explanation of the program
- brief explanation of each class
- brief explanation of each subroutine

Variable declarations

- declared on separate lines
- comments (if necessary)

Identifiers

- descriptive
- correct use of case
- correct use of underscores

White space

- white space around operators
- white space between subroutines
- white space after key words
- each curly brace on its own line

Indentation

- statements consistently indented
- block of code within another block of code further indented

General

- code appropriately commented
- each line less than 80 characters long
- correct spelling
- no unused (e.g. commented out) code

Examples

The following are two examples of programs with excellent programming style.

```
/***********************************************************************
* Program:
*    Homework 00, Add Integers
*    Brother Twitchell, CS 124
* Author:
*    Brother Twitchell
* Summary:
*    Demonstrates the amazing ability to add a positive integer and a
*    negative integer and to display the resulting sum.
***********************************************************************/

#include <iostream>
using namespace std;

/***********************************************************************
* Prompts the user for a positive and a negative integer.
* If required input is supplied, the two integers are added and the
* sum is displayed.
***********************************************************************/
int main()
{
   int positiveIntegerFromUser;
   int negativeIntegerFromUser;
   int sumOfIntegersFromUser;

   // Prompt the user for a number
   cout << "Enter a positive integer" << endl;
   cin  >> positiveIntegerFromUser;
   if (positiveIntegerFromUser > 0)
   {
      cout << "Enter a negative integer" << endl;
      cin  >> negativeIntegerFromUser;

      if (negativeIntegerFromUser < 0)
      {
         // amazing! we have both a positive and a negative integer
         // add them and output the results
         sumOfIntegersFromUser = positiveIntegerFromUser +
                                 negativeIntegerFromUser;
         cout << "The sum of " << positiveIntegerFromUser;
         cout << " and " << negativeIntegerFromUser;
         cout << " is " << sumOfIntegersFromUser << endl;
      }
      else
      {
         // while the user has demonstrated his/her ability to enter a
         //    positive integer, he/she failed to supply a negative
         //    integer; give up!
         cout << negativeIntegerFromUser << " is not negative" << endl;
         cout << "Next time please enter a number less than zero (0)." << endl;
         cout << "Program terminating." << endl;
      }
   }
   else
   {
      // the user has not supplied a positive integer; give up!
      cout << positiveIntegerFromUser << " is not positive" << endl;
      cout << "Next time please enter a number greater than zero (0)." << endl;
      cout << "Program terminating." << endl;
   }
   return 0;
}
```

Another example:

```
/***********************************************************************
* Program:
*    Homework 00, Cube a Number
*    Brother Twitchell, CS 124
* Author:
*    Brother Twitchell
* Summary:
*    This program reads a number from a text file, cubes the number,
*    and outputs the result.
***********************************************************************/

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

/***********************************************************************
* Returns the cube of the supplied integer value.
* Receives a pointer to the value to be cubed.
***********************************************************************/
int cubedInteger(int number)
{
   // return the cube the supplied value
   return (number * number * number);
}

/***********************************************************************
* Opens the input file and reads the number to be cubed. Outputs the
* original and cubed values. Closes the input file.
***********************************************************************/
int main()
{
   int numberFromFile = 0;

   // open the input file, read a single integer from it, and close it
   ifstream inputFile("number.txt" /*filename containing the number*/);

   // yes, yes, I know we are not testing to see if we succeeded!
   //      This is only a short demonstration program.
   inputFile >> numberFromFile;
   inputFile.close();

   // output the original value and its cube
   cout << "Impress your date!\n";
   cout << "The cube of "
        << numberFromFile
        << " is "
        << cubedInteger(numberFromFile)
        << "."
        << endl;
   return 0;
}
```

B. Order of Operations

The order of operations is the evaluation order for an expression. When parentheses are not included, the following table describes which order the compiler assumes you meant. Of course, it is always better to be explicit by including parentheses. Operators in rows of the same color have the same precedence.

Name	Operator	Example
Array indexing	`[]`	`array[4]`
Function call	`()`	`function()`
Postfix increment and decrement	`++ --`	`count++ count--`
Prefix increment and decrement	`++ --`	`++count --count`
Not	`!`	`!married`
Negative	`-`	`-4`
Dereference	`*`	`*pValue`
Address-of	`&`	`&value`
Allocate with new	`new`	`new int`
Free with delete	`delete`	`delete pValue`
Casting	`()`	`(int)4.2`
Get size of	`sizeof`	`sizeof(int)`
Multiplication	`*`	`3 * 4`
Division	`/`	`3 / 4`
Modulus	`%`	`3 % 4`
Addition	`+`	`3 + 4`
Subtraction	`-`	`3 - 4`
Insertion	`<<`	`cout << value`
Extraction	`>>`	`cin >> value`
Greater than, etc.	`>= <= > <`	`3 >= 4`
Equal to, not equal to	`== !=`	`3 != 4`
Logical And	`&&`	`passed && juniorStatus`
Logical OR	`||`	`passed || juniorStatus`
Assignment, etc.	`= += *= -= /= %=`	`value += 4`
Conditional expression	`? :`	`passed ? "happy" : "sad"`

C. Lab Help

Behold, you have not understood; you have supposed that I would give it unto you, when you took no thought save it was to ask me.

But, behold, I say unto you, that you must study it out in your mind ..."

D&C 9:7-8

The Linux lab is staffed with lab assistants to provide support for students using the lab for computer science courses. Their major role is to provide assistance, support, advice, and recommendations to students. They are not to be considered a help desk where you bring a problem to them and expect them to solve it! As the semester progresses, lab assistants become increasingly busy and are not able to provide as much tutoring. Those struggling with a class or those that have been away from programming for a significant period (e.g. mission) are encouraged to sign up for class tutors from the tutoring center.

Lab assistants can be very useful when you have encountered a brick wall and just don't know how to proceed. They can help get you going again! You must put forth effort to complete your homework assignment. Do not expect lab assistants to give you answers or source code for your specific assignments. Instead, expect questions to guide you to a solution. For example they might say, "Have you tried using a while loop?", or "Are you sure you haven't exceeded your array bounds?", or "Check the syntax of your switch statement," or "Try putting some debug statements in your code *here* to see what is happening." With this kind of help you will understand what you have done wrong, you will be less likely to make this mistake in the future, and if this mistake is made in the future, you will be in a better situation to solve the problem without assistance. As a general rule, lab assistants are *not* allowed to touch the keyboard.

Lab assistants have been hired to help all students with general questions regarding use of the computers in the Linux lab. This includes, but is not limited to, difficulties with text editors, `submit`, `styleChecker`, `testBed`, `svn`, `PuTTY`, `winscp`, and simple operating system issues. CS 124 and CS 165 students should expect appropriate assistance from lab assistants. CS 235 students, particularly those who have been away for a couple of years, should also expect appropriate assistance from lab assistants during the first few weeks of the semester. Generally speaking, however, students should have learned how to help themselves and resolve their own problems by the time they have completed CS 235. CS 213 students may expect some help from the lab assistants.

Lab assistants are expected to give first priority to CS 124 and CS 165 students. Students in upper-division classes should not expect assistance from lab assistants.

To more effectively respond to questions, a "Now Serving" system has been setup in the lab which allows a student to click on an icon and it automatically places a request for help in a queue. The lab assistants monitor the queue and help the next student in the queue. It's like taking a number and waiting for your number to be called. However, instead of calling your number they just come to your machine. To use the "Now Serving" system you will need to ask the lab assistants to set it up for you the first time. Once initially setup an icon will be visible each time you login then you simply click on the Icon to make a request for help. You won't need to keep holding your hand up waiting for a lab assistant to see you, and it makes sure that you get help in the proper order. Please ask the lab assistants to show you how to use the "Now Serving" software so they can serve you better. Students in upper-division classes should not expect assistance from lab assistants

D. Emacs & Linux Cheat-Sheet

The emacs editor and the Linux system are most effectively used if commonly-used commands are memorized. The following are the commands used most frequently for CS 124:

Common Emacs Commands

`C-a`	Beginning of line
`C-e`	End of line
`C-k`	Kill line. Also puts the line in the buffer
`C-_`	Undo
`C-x C-f`	Load a new file or an existing file. The name will be specified in the window below
`C-x 2`	Split the window into two
`C-x 1`	Go back to one window mode
`C-x ^`	Enlarge window
`C-x o`	Switch to another window
`Alt-x shell`	Run the shell in the current window
`Alt-x goto-line`	Goto a given line
`C-x C-c`	Save buffer and kill Emacs
`C-x C-s`	Save buffer
`C-space`	Set mark
`C-w`	Cut from the cursor to the mark
`Alt-w`	Copy from the cursor to the mark
`C-y`	Paste from the buffer

Common Linux Commands

`cd`	Change Directory, move from the current directory to another
`ls`	List information about file(s)
`ll`	More verbose version of ls
`mkdir`	Create new folder(s)
`mv`	Move or rename files or directories
`cp`	Copy a file from one location to another
`rm`	Remove files
`rmdir`	Remove folder(s)
`pwd`	Print Working Directory
`cat`	Display the contents of a file to the screen
`more`	Display output one screen at a time
`clear`	Clear terminal screen
`exit`	Exit the shell
`fgrep`	Search file for lines that match a string
`kill`	Stop a process from running
`man`	Review the help page of a given command
`yppasswd`	Modify a user password
`emacs`	Common code editor
`vi`	More primitive but ubiquitous editor
`nano`	Another editor
`g++`	Compile a C++ program
`styleChecker`	Run the style checker on a file
`testBed`	Run the test bed on a file
`submit`	Turn in a file

E. C++ Syntax Reference Guide

Name	Syntax	Example

Pre-processor directives	`#include <libraryName>`
	`#define <MACRO_NAME> <expansion>`
		`#include <iostream> // for CIN & COUT`
		`#include <iomanip>  // for setw()`
		`#include <fstream>  // for IFSTREAM`
		`#include <string>   // for STRING`
		`#include <cctype>   // for ISUPPER`
		`#include <cstring>  // for STRLEN`
		`#include <cstdlib>  // for ATOF`
		
		`#define PI        3.14159`
		`#define LANGUAGE  "C++"`

Function	`<ReturnType> <functionName>(<params>)`
	`{`
	`   <statements>`
	`   return <value>;`
	`}`
		`int main()`
		`{`
		`   cout << "Hello world\n";`
		`   return 0;`
		`}`

Function parameters	`<DataType> <passByValueVariable>,`
	`<DataType> & <passByReferenceVariable>,`
	`const <DataType> <CONSTANT_VARIABLE>,`
	`<BaseType> <arrayVariable>[]`
		`void function(int value,`
		`              int &reference,`
		`              const int CONSTANT,`
		`              int array[])`
		`{`
		`}`

COUT	`cout << <expression> << … ;`	`cout << "Text in quotes"`
		`     << 6 * 7`
		`     << getNumber()`
		`     << endl;`

Formatting output for money	`cout.setf(ios::fixed);`
	`cout.setf(ios::showpoint);`
	`cout.precision(<integerExpression>);`
		`cout.setf(ios::fixed);`
		`cout.setf(ios::showpoint);`
		`cout.precision(2);`

Declaring variables	`<DataType> <variableName>;`
	`<DataType> <variableName> = <init>;`
	`const <DataType> <VARIABLE_NAME>;`
		`int integerValue;`
		`float realNumber = 3.14159;`
		`const char LETTER_GRADE = 'A';`

CIN	`cin >> <variableName>;`	`cin >> variableName;`

IF statement	`if (<Boolean-expression>)`
	`{`
	`   <statements>`
	`}`
	`else`
	`{`
	`   <statements>`
	`}`
		`if (grade >= 70.0)`
		`   cout << "Great job!\n";`
		`else`
		`{`
		`   cout << "Try again.\n";`
		`   pass = false;`
		`}`

Asserts	`assert(<Boolean-expression>);`	`#include <cassert> // at top of file`
		`{`
		`   assert(gpa >= 0.0);`
		`}`

Name	Syntax	Example

FOR loop	for (<initialization statement>;
	     <Boolean-expression>;
	     <increment statement>)
	{
	   <statements>
	}
		for (int iList = 0;
		     iList < sizeList;
		     iList++)
		   cout << list[iList];

WHILE loop	while (<Boolean-expression>)
	{
	  <statements>
	}
		while (input <= 0)
		   cin >> input;

DO-WHILE Loop	do
	{
	   <statements>
	}
	while (<Boolean-expression>);
		do
		   cin >> input;
		while (input <= 0);

Read from File	ifstream <streamVar>(<fileName>);
	if (<streamVar>.fail())
	{
	   <statements>
	}
	
	<streamVar> >> <variableName>;
	
	<streamVar>.close();
		#include <fstream> // at top of file
		{
		   ifstream fin("data.txt");
		   if (fin.fail())
		      return false;
		
		   fin >> value;
		
		   fin.close();
		}

Write to File	ofstream <streamVar>(<fileName>);
	if (<streamVar>.fail())
	{
	   <statements>;
	}
	
	<streamVar> << <expression>;
	
	<streamVar>.close();
		#include <fstream> // at top of file
		{
		   ofstream fout("data.txt");
		   if (fout.fail())
		      return false;
		
		   fout << value << endl;
		
		   fout.close();
		}

Fill an array	<BaseType> <arrayName>[<size>];
	<BaseType> <arrayName>[] =
	    { <CONST_1>, <CONST_2>, … };
	
	for (int i = 0; i < <size>; i++)
	   <arrayName>[i] = <expression>;
		int grades[10];
		for (int i = 0; i < 10; i++)
		{
		   cout << "Grade " << i + 1 << ": ";
		   cin >> grades[i];
		}

C-Strings	char <stringName>[<size>];
	cin >> <stringName>;
	for (char * <ptrName> = <stringName>;
	     *<ptrName>;
	     <ptrName>++)
	   cout << *<ptrName>;
		char firstName[256];
		cin >> firstName;
		for (char * p = firstName; *p; p++)
		   cout << *p;

String Class	string <stringName>;
	cin >> <stringName>;
	cout << <stringName>;
	getline(<streamName>, <stringName>);
	
	if (<stringName1> == <stringName2>)
	   <statemement>;
	
	<stringName1> += <stringName2>;
	<stringName1> = <stringName2>;
		string string1; // declare
		string string2 = "124"; // initialize
		
		cin >> string1; // input
		getline(cin, string2); // getline
		if (string1 == string2) // compare
		   string1 += string2; // append
		string2 = string1; // copy

Name	Syntax	Example

Switch	`switch (<integer-expression>)`
	`{`
	`   case <integer-constant>:`
	`      <statements>`
	`      break;               // optional`
	`   …`
	`   default:             // optional`
	`      <statements>`
	`}`
		`switch (value)`
		`{`
		`   case 3:`
		`      cout << "Three";`
		`      break;`
		`   case 2:`
		`      cout << "Two";`
		`      break;`
		`   case 1:`
		`      cout << "One";`
		`      break;`
		`   default:`
		`      cout << "None!";`
		`}`

Conditional Expression	`<Boolean-expression> ? <expression> :`
	`                       <expression>`
		`cout << "Hello, "`
		`     << (isMale ? "Mr. " : "Mrs. ")`
		`     << lastName;`

Multi-dimensional array	`<BaseType> <arrayName>[<SIZE>][<SIZE>];`
	`<BaseType> <arrayName>[][<SIZE>] =`
	`   {`
	`     { <CONST_0_0>, <CONST_0_1>, … },`
	`     { <CONST_1_0>, <CONST_1_1>, … },`
	`       …`
	`   };`
	
	`<arrayName>[<index>][<index>] =`
	`    <expression>;`
		`int board[3][3];`
		
		`for (int row = 0; row < 3; row++)`
		`   for (int col = 0; col < 3; col++)`
		`     board[row][col] = 10;`

Allocate memory	`<ptr> = new(nothrow) <DataType>;`
	`<ptr> = new(nothrow> <DataType>(<init>);`
	`<ptr> = new(nothrow) <BaseType>[<SIZE>];`
		`float * p1   = new(nothrow) float;`
		`int   * p2   = new(nothrow) int(42);`
		`char  * text = new(nothrow) char[256];`

Free memory	`delete <pointer>;          // one value`
	`delete [] <arrayPointer>;  // an array`
		`delete pNumber;`
		`delete [] text;`

Command line parameters	`int main(int <countVariable>,`
	`         char **<arrayVariable>)`
	`{`
	`}`
		`int main(int argc, char ** argv)`
		`{`
		`}`

Library	Function Prototype

`#include <cctype>`	`bool isalpha(char);      // is the character an alpha  ('a' - 'z' or 'A' - 'Z')?`
	`bool isdigit(char);      // is the character a number  ('0' - '9')?`
	`bool isspace(char);      // is the character a space   (' ' or '\t' or '\n')?`
	`bool ispunct(char);      // is the character a symbol such as %#$!-_*@.,?`
	`bool isupper(char);      // is the character uppercase ('A' - 'Z')?`
	`bool islower(char);      // is the character lowercase ('a' - 'z')?`
	`int  toupper(char);      // convert lowercase character to uppercase. Rest unchanged`
	`int  tolower(char);      // convert uppercase character to lowercase. Rest unchanged`

`#include <cstring>`	`int strlen(const char *);                        // find the length of a c-string`
	`int strcmp(const char *, const char *);          // 0 if the two strings are the same`
	`char * strcpy(char *<dest>, const char *<src>);  // copies src onto dest`

`#include <cstdlib>`	`double atof(const char *); // parses input for a floating point number and returns it`
	`int atoi(const char *);    // parses input for an integer number and returns it`

F. Glossary

`#define` — A `#define` (pronounced "pound define") is a mechanism to expand macros in a program. This macro expansion occurs before the program is compiled. The following example expands the macro `PI` into 3.1415 — Chapter 2.1

```
#define PI 3.1415
```

`#ifdef` — The `#ifdef` macro (pronounced "if-deaf") is a mechanism to conditionally include code in a program. If the condition is met (the referenced macro is defined), then the code is included. — Chapter 2.1

```
#ifdef DEBUG
   cout << "I was here!\n";
#endif
```

abstract — One of the three levels of understanding of an algorithm, abstract understanding is characterized by a grasp of how the parts or components of a program work together to produce a given output. — Chapter 2.4

address-of operator — The address-of operator (`&`) yields the address of a variable in memory. It is possible to use the address-of operator in front of any variable. — Chapter 3.3

```
{
   int variable;
   cout << "The address of 'variable' is "
        << &variable;
}
```

ALU — Arithmetic Logic Unit. This is the part of a CPU which performs simple mathematical operations (such as addition and division) and logical operations (such as `OR` and `NOT`) — Chapter 0.2

`argc` — When setting up a program to accept input parameters from the command line, argc is the traditional name for the number of items or parameters in the jagged array of passed data. The name "`argc`" refers to "count of arguments." — Chapter 4.3

```
int main(int argc,          // count of parameters
         char ** argv);
```

`argv` — When setting up a program to accept input parameters from the command line, `argv` is the traditional name for the jagged array containing the passed data. The name "`argv`" refers to "argument vector" or "list of unknown arguments." — Chapter 4.3

```
int main(int argc,
         char ** argv);    // array of parameters
```

array	An array is a data-structure containing multiple instances of the same item. In other words, it is a "bucket of variables." Arrays have two properties: all instances in the collection are the same data-type and each instance can be referenced by index (not by name).
	`{`
	`   int array[4];          // a list of four integers`
	`   array[2] = 42;         // the 3rd member of the list`
	`}`
		Chapter 3.0
		Chapter 3.1

assembly	Assembly is a computer language similar to machine language. It is a low-level language lacking any abstraction. The purpose of Assembly language is to make Machine language more readable. Examples of Assembly language include `LOAD M:3` and `ADD 1`.	Chapter 0.2
`assert`	An assert is a function that tests to see if a particular assumption is met. If the assumption is met, then no action is taken. If the assumption is not met, then an error message is thrown and the program is terminated. Asserts are designed to only throw in debug code. To turn off asserts for shipping code, compile with the `-DNDEBUG` switch.	Chapter 2.1
bitwise operator	A bitwise operator is an operator that works on the individual bits of a value or a variable.	Chapter 3.5
`bool`	A `bool` is a built-in datatype used to describe logical data. The name "`bool`" came from the father of logical data, George Boole. `bool isMarried = true;`	Chapter 1.2 Chapter 1.5
Boolean operator	A Boolean operator is an operator that evaluates to a `bool` (`true` or `false`). For example, consider the expression (`value1 == value2`). Regardless of the data-type or value of `value1` and `value2`, the expression will always evaluate to `true` or `false`.	Chapter 1.5
data-driven	Data-driven design is a programming design pattern where most of the elements of the design are encoded in a data structure (typically an array) rather than in the algorithm. This allows a program to be modified without changing any of the code; only the data structure needs to be adjusted.	Chapter 3.1
`case`	A `case` label is part of a `switch` statement enumerating one of the options the program must select between.	Chapter 3.5
casting	The process of converting one data type (such as a `float`) into another (such as an `int`). For example, `(float)3` equals `3.0`.	Chapter 1.3
`char`	A `char` is a built-in datatype used to describe a character or glyph. The name "Char" came from "Character," being the most common use. `char letterGrade = 'B';`	Chapter 1.2

comments	Comments are notes placed in a program not read by the compiler.	Chapter 0.2
compiler	A compiler is a program to translate code in a one language (say C++) into another (say machine language).	Chapter 1.0
compound statement	A compound statement is a collection of statements grouped with curly braces. The most common need for this is inside the body of an IF statement or in a loop.	Chapter 1.6

```
if (failed == true)
{                          // compound statement start
   cout << "Sorry!\n";     //   first statement
   return false;           //   second statement
}                          // compound statement end
```

cohesion	The measure of the internal strength of a module. In other words, how much a function does one thing and one thing only. The four levels of cohesion are: Strong, Extraneous, Partial, and Weak.	Chapter 2.0
coincidental	A measure of cohesion where items are in a module simply because they happen to fall together. There is no relationship.	Chapter 2.0
communicational	A measure of cohesion where all elements work on the same piece of data.	Chapter 2.0
conceptual	One of the three levels of understanding of an algorithm, conceptual understanding is characterized by a high level grasp of what the program is trying to accomplish. This does not imply an understanding of what the individual components do or even how the components work together to produce the solution.	Chapter 2.4
concrete	One of the three levels of understanding of an algorithm, concrete understanding is characterized by knowing what every step of an algorithm is doing. It does not imply an understanding of how the various steps contribute to the larger problem being solved. The desk check tool is designed to facilitate a concrete understanding of code.	Chapter 2.4
conditional expression	A conditional expression is a decision mechanism built into C++ allowing the programmer to choose between two expression, rather than two statements.	Chapter 3.5

```
cout << (grade >= 60.0 ? "pass" : "fail");
```

control	A measure of coupling where one module passes data to another that is interpreted as a command.	Chapter 2.0

counter-controlled	One of the three loop types, a counter-controlled loop keeps iterating a fixed number of types. Typically, this number is known when the loop begins. A counter-controlled loop has four components: the start, the end, the counter, and a loop body.	Chapter 2.5
coupling	Coupling is the measure of information interchange between functions. The seven levels of coupling are: Trivial, Encapsulated, Simple, Complex, Document, Interactive, and Superfluous.	Chapter 2.0
`cout`	COUT stands for Console OUTput. Technically speaking, `cout` a the destination or output stream. In other words, it in the following example, it is the destination where the insertion operator (<<) is sending data to. In this case, that destination is the screen. `cout << "Hello world!";`	Chapter 1.1
CPU	Central Processing Unit. This is the part of a computer that interprets machine-language instructions	Chapter 0.2
c-string	A c-string is how strings are stored in C++: an array of characters terminated with a null (`'\0'`) character.	Chapter 3.2
data	A measure of coupling where the data passed between functions is very simple. This occurs when a single atomic data item is passed, or when highly cohesive data items are passed	Chapter 2.0
decoder	The instruction decoder is the part of the CPU which identifies the components of an instruction from a single machine language instruction.	Chapter 0.2
`default`	A `default` label is a special `case` label in a `switch` statement corresponding to the "unknown" or "not specified" condition. If none of the `case` labels match the value of the controlling expression, then the `default` label is chosen.	Chapter 3.5

`delete`	The `delete` operator serves to free memory previously allocated with `new`. When a variable is declared on the stack such as a local variable, this is unnecessary; the operating system deletes the memory for the user. However, when data is allocated with new, it is the programmer's responsibility to delete his memory.
	`{`
	`   int * pValue = new int;`
	`   delete pValue;`
	`}`
		Chapter 4.1

DeMorgan	Just as there are ways to manipulate algebraic equations using the associative and distributed properties, it is also possible to manipulate Boolean equations. One of these transformations is DeMorgan. A few DeMorgan equivalence relationships are:
	`!(p || q) == !p && !q`
	`!(p && q) == !p || !q`
	`a || (b && c) == (a || b) && (a || c)`
	`a && (b || c) == (a && b) || (a && c)`
		Chapter 1.5

dereference operator	The dereference operator '*' will retrieve the data refered to by a pointer.
	`cout << "The data in the variable pValue is "`
	`     << *pValue;`
		Chapter 3.3

desk check	A desk check is a technique used to predict the output of a given program. It accomplished by creating a table representing the value of the variables in the program. The columns represent the variables and the rows represent the value of the variables at various points in the program execution.	Chapter 2.4

`do … while`	One of the three types of loops, a DO-WHILE loop continues to execute as long as the condition in the Boolean expression evaluates to true. This is the same as a WHILE loop except the body of the loop is guaranteed to be executed at least once.
	`do`
	`   cin >> gpa;`
	`while (gpa > 4.0 || gpa < 0.0);`
		Chapter 2.3

`double`	A `double` is a built-in datatype use to describe large read numbers. The word "Double" comes from "Double-precision floating point number," indicating it is just like a `float` except it can represent a larger number more precisely. `double pi = 3.14159265359;`	Chapter 1.2
driver	A driver is a program designed to test a given function. Usually a driver has a collection of simple prompts to feed input to the function being tested, and a simple output to display the return values of the function.	Chapter 2.1

dynamically-allocated array	A dynamically-allocated array is an array that is created at run-time rather than at compile time. Stack arrays have a size known at compile time. Dynamically-allocated arrays, otherwise known as heap arrays, can be specified at run-time.
	`{`
	`   int * array = new int[size];`
	`}`
		Chapter 4.1

`endl`	ENDL is short for "END of Line." It is one of the two ways to specify that the output stream (such as `cout`) will put a new line on the screen. The following example will put two blank lines on the screen: `cout << endl << endl;`	Chapter 1.1

`eof`	When reading data from a file, one can detect if the end of the file is reached with the `eof()` function. Note that this will only return `true` if the end of file marker was reached in the last read.
	`if (fin.eof())`
	`   cout << "The end of the file was reached\n";`
		Chapter 2.6

escape sequences	Escape sequences are simply indications to `cout` that the next character in the output stream is special. Some of the most common escape sequences are the newline (`\n`), the tab (`\t`), and the double-quote (`\"`)	Chapter 1.1
event-controlled	One of the three loop types, an event-controlled loop is a loop that keeps iterating until a given condition is met. This condition is called the event. There are two components to an event-controlled loop: the termination condition and the body of the loop.	Chapter 2.5
expression	A collections of values and operations that, when evaluated, result in a single value. For example, `3 * value` is an expression. If value is defined as `float value = 1.5;`, then the expression evaluates to `4.5`.	Chapter 1.3
external	A measure of coupling where two modules communicate through a global variable or another external communication avenue.	Chapter 2.0
extraction operator	The extraction operator (>>) is the operator that goes between `cin` and the variable receiving the user input. In the following example, the extraction operator is after the `cin`. `cin >> data;`	Chapter 1.2
fetcher	The instruction fetcher is the part of the CPU which remembers which machine instruction is to be retrieved next. When the CPU is ready for another instruction, the fetcher issues a request to the memory interface for the next instruction.	Chapter 0.2

`for`	One of the three types of loops, the FOR loop is designed for counting. It contains fields for the three components of most counting problems: where to start (the Initialization section), where the end (the Boolean expression), and what to change with every count (the Increment section).
	`for (int i = 0; i < num; i++)`
	`   cout << array[i] << endl;`
		Chapter 2.3

`fstream`	The `fstream` library contains tools enabling the programmer to read and write data to a file. The most important components of the `fstream` library are the `ifstream` and `ofstream` classes. `#include <fstream>`	Chapter 2.6
function	One division of a program. Other names are sub-routine, sub-program, procedure, module, and method.	Chapter 1.4
functional	A measure of cohesion where every item in the function is related to a single task.	Chapter 2.0

`getline`	The `getline()` method works with `cin` to get a whole line of user input.
	`char text[256];            // getline needs a string`
	`cin.getline(text, 256);    // the size is a parameter`
		Chapter 1.2

global variable	A global variable is a variable defined outside a function. The scope extends to the bottom of the file, including any function that may be defined below the global. It is universally agreed that global variables are evil and should be avoided.	Chapter 1.4

`ifstream`	The `ifstream` class is part of the `fstream` library, enabling the programmer to write data to a file. IFSTREAM is short for "Input File STREAM."
	`#include <fstream>`
	
	`{`
	`   ifstream fin("file.txt");`
	`   …`
	`}`
		Chapter 2.6

insertion operator	The insertion operator (`<<`) is the operator that goes between `cout` and the data to be displayed. As we will learn in CS 165, the insertion operator is actually the function and `cout` is the destination of data. In the following example, the insertion operator is after the `cout`. `cout << "Hello world!";`	Chapter 1.1
instrumentation	The process of adding counters or markers to code to determine the performance characteristics. The most common ways to instrument code is to track execution time (by noting start and completion time of a function), iterations (by noting how many times a given block of code has executed), and memory usage (by noting how much memory was allocated during execution).	Chapter 4.4
`int`	An `int` is a built-in datatype used to describe integral data. The word "Int" comes from "Integer" meaning "all whole numbers and their opposites." `int age = 19;`	Chapter 1.2
`iomanip`	The IOMANIP library contains the `setw()` method, enabling a C++ program to right-align numbers. The programmer can request the IOMANIP library by putting the following code in the program: `#include <iomanip>`	Chapter 1.1
`iostream`	The IOSTREAM library contains `cin` and `cout`, enabling a simple C++ program to display text on the screen and gather input from the keyboard. The programmer can request IOSTREAM by putting the following code in the program: `#include <iostream>`	Chapter 0.2

jagged array	A jagged array is a special type of multi-dimensional array where each row could be of a different size.	Chapter 4.3
lexer	The lexer is the part of the compiler to break a program into a list of tokens which will then be parsed.	Chapter 1.0
local variable	A local variable is a variable defined in a function. The scope of the variable is limited to the bounds of the function.	Chapter 1.4
logical	A level of cohesion where items are grouped in a module because they do the same kinds of things. What they operate on, however, is totally different.	Chapter 2.0
machine	Machine language is a computer language understandable by a CPU. It is language of the lowest abstraction. Machine language consists of noting but 1's and 0's.	Chapter 0.2
modularization	Modularization is the process of dividing a problem into separate tasks, each with a single purpose.	Chapter 2.0
modulus	The remainder from division. Consider 14 ÷ 3. The answer is 4 with a remainder of 2. Thus fourteen modulus 3 equals 2: 14 % 3 == 2	Chapter 1.3
multi-dimensional array	A multi-dimensional array is an array of arrays. Instead of accessing each member with a single index, more than one index is required. The following is a multi-dimensional array representing a tic-tac-toe board:	Chapter 4.0

```
{
   char board[3][3];
}
```

multi-way IF	Though an IF statement only allows the programmer to distinguish between at most two options, it is possible to specify more options through the use of more than one IF. This is called an multi-way IF.	Chapter 1.6

```
if (grade >= 90.0)
   cout << "A";               // first condition
else if (grade >= 90.0)
   cout << "B";               // second condition
else
   cout << "not so good!";    // third condition
```

nested statement	A nested statement is a statement inside the body of another statement. For example, an IF statement inside the body of another IF statement would be considered a nested IF.	Chapter 1.6

```
if (grade >= 80.0)      // outer IF statement
   if (grade >= 90)     // nested IF statement
      cout << 'A';      // body of nested IF statement
   else
      cout << 'B';
```

`new` — It is possible to allocate a block of memory with the `new` operator. This serves to issue a request to the operating system for more memory. It works with single items as well as arrays. — Chapter 4.1

```
{
   int * pValue = new int;       // one integer
   int * array = new int[10];    // ten integers
}
```

null — The null character, also known as a null terminator, is a special character marking the end of a c-string. The null character is represented as `'\0'`, which is always defined as zero. — Chapter 3.2

```
{
   char nullCharacter = '\0';  // 0x00
}
```

`NULL` — The `NULL` address corresponds to the zero address `0x00000000`. This address is guaranteed to be invalid, making it a convenient address to assign to a pointer when the pointer variable does not point to anything. — Chapter 4.1

```
{
   int * pValue = NULL;   // points to nothing
}
```

`ofstream` — The `ofstream` class is part of the `fstream` library, enabling the programmer to write data to a file. OFSTREAM is short for "<u>O</u>utput <u>F</u>ile <u>STREAM</u>." — Chapter 2.6

```
#include <fstream>

{
   ofstream fout("file.txt");
   …
}
```

online desk check — An online desk check is a technique to gain an understanding of how data flows through an existing program. This is accomplished by putting COUT statements at strategic places in a program to display the value of key variables. — Chapter 2.4

parser — The parser is the part of the compiler understanding the syntax or grammar of the language. Knowing this, it is able to take all the components from the input language and place it into the format of the target or output language. — Chapter 1.0

Pascal-string — One of the two main implementations of strings, a Pascal-string is an array of characters where the length is stored in the first slot. This is not how strings are implemented in C++. — Chapter 3.2

pass-by-reference — Pass-by-reference, also known as "call-by-reference," is the process of sending a parameter to a function where the caller and the callee share the same variable. This means that changes made to the parameter in the callee will be reflected in the caller. You specify a pass-by-reference parameter with the ampersand `&`. — Chapter 1.4, Chapter 3.3

```
void passByReference(int &parameter);
```

pass-by-pointer	Pass-by-pointer, more accurately called "passing a pointer by value," is the process of passing an address as a parameter to a function. This has much the same effect as pass-by-reference. `void passByPointer(int * pParameter);`	Chapter 3.3
pass-by-value	Pass-by-value, also known as "call-by-value," is the process of sending a parameter to a function where the caller and the callee have different versions of a variable. Data is sent one-way from the caller to the callee; no data is sent back to the caller through this mechanism. This is the default parameter passing convention in C++. `void passByValue(int parameter);`	Chapter 1.4 Chapter 3.3
pointer	A pointer is a variable holding and address rather than data. A data variable, for example, may hold the value 3.14159. A pointer variable, on the other hand, will contain the address of some place in memory.	Chapter 3.3
procedural	A measure of cohesion where all related items must be performed in a certain order.	Chapter 2.0
prototype	A prototype is the name, parameter list, and return value of a function to be defined later in a file. The purpose of the prototype is to give the compiler "heads-up" as to which functions will be defined later in the file.	Chapter 1.4

pseudocode	Pseudocode is a high-level programming language designed to help people design programs. Though it has most of the elements of a language like C++, pseudocode cannot be compiled. An example of pseudocode is:
	computeTithe(income)
	  RETURN income ÷ 10
	END
		Chapter 2.2

register	The part of a CPU which stores short-term data for quick recall. A CPU typically has many registers.	Chapter 0.2
sentinel-controlled	One of the three loop types, a sentinel-controlled loop keeps iterating until a condition is met. This condition is controlled by a sentinel, a Boolean variable set by a potentially large number of divergent conditions.	Chapter 2.5
sequential	A measure of cohesion where operations in a module must occur in a certain order. Here operations depend on results generated from preceding operations	Chapter 2.0
scope	Scope is the context in which a given variable is available for use. This extends from the point where the variable is defined to the next closing braces `}`.	Chapter 1.4

`sizeof`	The `sizeof` function returns the number of bytes that a given datatype or variable requires in memory. This function is unique because it is evaluated at compile-time where all other functions are evaluated at run-time.	Chapter 1.2 Chapter 3.0

```
{
   int integerVariable;
   cout << sizeof(integerVariable) << endl;    // 4
   cout << sizeof(int)             << endl;    // 4
}
```

stack variable	A stack variable, otherwise known as a local variable, is a variable that is created by the compiler when it falls into scope and destroyed when it falls out of scope. The compiler manages the creation and destruction of stack variables wherease the programmer manages the createion and destruction of dynamically allocated (heap) variables.	Chapter 4.1
stamp	A measure of coupling where complex data or a collection of unrelated data items are passed between modules.	Chapter 2.0
string	A "string" is a computer representation of text. The term "string" is short for "an alpha-numeric string of characters." This implies one of the most important characteristics of a string: is a sequence of characters. In C++, a string is defined as an array of characters terminated with a null character.	Chapter 1.2 Chapter 3.2

```
{
   char text[256];   // a string of 255 characters
}
```

structure chart	A structure chart is a design tool representing the way functions call each other. It consists of three components: the name of the functions of a program, a line connecting functions indicating one function calls another, and the parameters that are passed between functions.	Chapter 2.0
`styleChecker`	`styleChecker` is a program that performs a first-pass check on a student's program to see if it conforms to the University style guide. The `styleChecker` should be run before every assignment submission.	Chapter 1.0 Appendix A
stub	A stub function is a placeholder for a function that is not written yet. The closest analogy is an outline in an essay: a placeholder for a chapter or paragraph to be written later. An example stub function is:	Chapter 2.1

```
void display(float value)
{
}
```

`submit`	`submit` is a program to send a student's file to the appropriate instructor. It works by reading the program header and, based on what is found, sending it to the instructor's class and assignment directory.	Chapter 1.0
`switch`	A `switch` statement is a mechanism built into most programming langauges allowing the programmer to specify between more than two options.	Chapter 3.5

tabs	The tab key on a traditional typewriter was invented to facilitate creating tabular data (hence the name). The tab character (`'\t'`) serves to move the cursor to the next tab stop. By default, that is the next 8 character increment. `cout << "\tTab";`	Chapter 1.1
temporal	A measure of cohesion where items are grouped in a module because the items need to occur at nearly the same time. What they do or how they do it is not important	Chapter 2.0
`testBed`	`testBed` is a tool to compare a student's solution with the instructor's key. It works by compiling the student's assignment and running the program against a pre-specified set of input and output.	Chapter 1.0
variable	A variable is a named location where you store data. The name must be a legal C++ identifier (comprising of digits, letters, and the underscore _ but not starting with a digit) and conform to the University style guide (camelCase, descriptive, and usually a noun). The location is determined by the compiler, residing somewhere in memory.	Chapter 1.2
`while`	One of the three types of loops, a WHILE-loop continues to execute as long as the condition inside the Boolean expression is true. `while (grade < 70)` `   grade = takeClassAgain();`	Chapter 2.3

G. Index

Appendix

Appendix