

Unit 3. Pointers & Arrays

3.0 Array Syntax	214
3.1 Array Design	233
3.2 Strings	242
3.3 Pointers	255
3.4 Pointer Arithmetic	269
3.5 Advanced Conditionals	284
Unit 3 Practice Test	301
Unit 3 Project : MadLib	303

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:

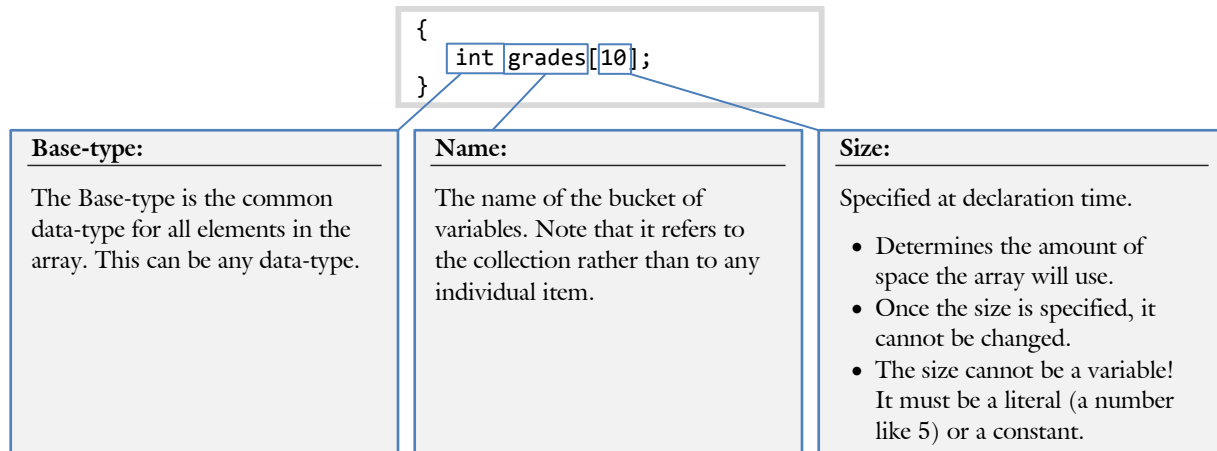
```
char text[256];
```

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: <pre><Type> <variable>[size]</pre>	Syntax: <pre><variable>[<index>]</pre>	Syntax: <pre>(<Type> <variable>[])</pre>
Example: <pre>int grades[200];</pre>	Example: <pre>cout << grades[i];</pre>	Example: <pre>void func(int grades[])</pre>
A few details: <ul style="list-style-type: none"> • Any data-type can be used. • The size must be a natural number {1, 2, etc.}, not a variable. 	A few details: <ul style="list-style-type: none"> • The index starts with 0 and must be within the valid range. 	A few details: <ul style="list-style-type: none"> • 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.



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'
    };
}
```

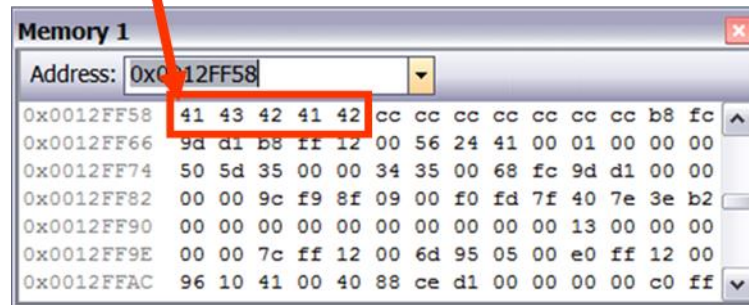
Observe how the list of items to initialize are delimited with curly braces ({}). Since the Elements of Style demands that each curly brace be on its own line, we align them with the base-type. Finally, the individual items in the array are presented in a comma-separated list.

Declaration	In memory	Description						
<pre>int array[6];</pre>	<table><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	?	?	?	?	?	?	Though six slots were set aside, they remain uninitialized. All slots are filled with unknown values.
?	?	?	?	?	?			
<pre>int array[6] = { 3, 6, 2, 9, 1, 8 };</pre>	<table><tr><td>3</td><td>6</td><td>2</td><td>9</td><td>1</td><td>8</td></tr></table>	3	6	2	9	1	8	The initialized size is the same as the declared size so every slot has a known value.
3	6	2	9	1	8			
<pre>int array[6] = { 3, 6 };</pre>	<table><tr><td>3</td><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	3	6	0	0	0	0	The first 2 slots are initialized, the balance are filled with 0. This is a partially filled array.
3	6	0	0	0	0			
<pre>int array[] = { 3, 6, 2, 9, 1, 8 };</pre>	<table><tr><td>3</td><td>6</td><td>2</td><td>9</td><td>1</td><td>8</td></tr></table>	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
3	6	2	9	1	8			
<pre>int array[6] = {};</pre>	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	This is the easiest way to initialize an array with zeros in all the slots
0	0	0	0	0	0			



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'};
```



In this example, the location 0x0012FF58.

memory starts at

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	<p>It is not possible to perform the array copy with a simple assignment statement:</p> <pre>arrayDestination = arraySource; // this will not work</pre> <p>Instead, it is necessary to write a loop and copy the items one at a time.</p> <pre>{ 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]; }</pre>
Challenge	<p>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.</p> <p>As an additional challenge, try to display the list backwards.</p>
See Also	<p>The complete solution is available at 3-0-arrayCopy.cpp or:</p> <pre>/home/cs124/examples/3-0-arrayCopy.cpp</pre> 

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 incremter (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

```
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
```

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
<code>char a;</code> <code>cout << sizeof(a) << endl;</code>	
<code>char b[10];</code> <code>cout << sizeof(b) << endl;</code>	
<code>cout << sizeof(b[0]) << endl;</code>	
<code>int c;</code> <code>cout << sizeof(c) << endl;</code>	
<code>int d[20];</code> <code>cout << sizeof(d) << endl;</code>	
<code>cout << sizeof(d[1]) << endl;</code>	

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:

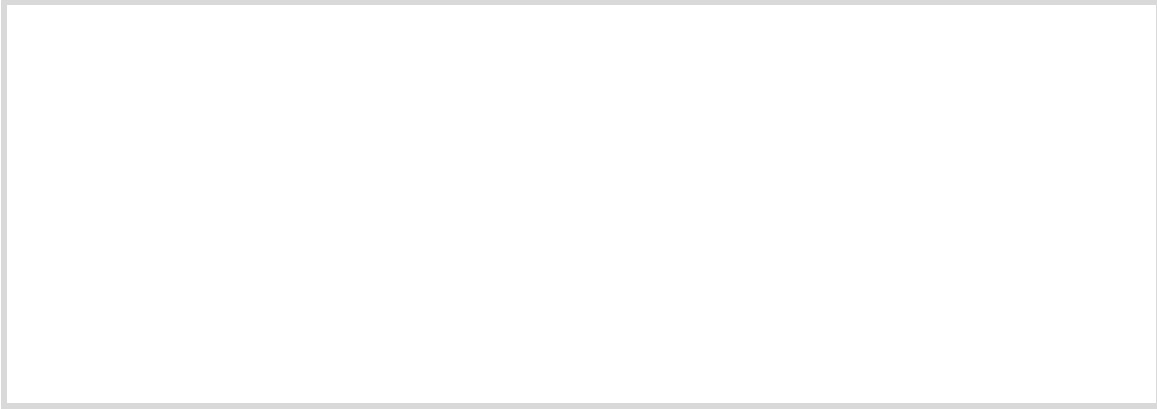
```
Average is 28.1
```

Answer:

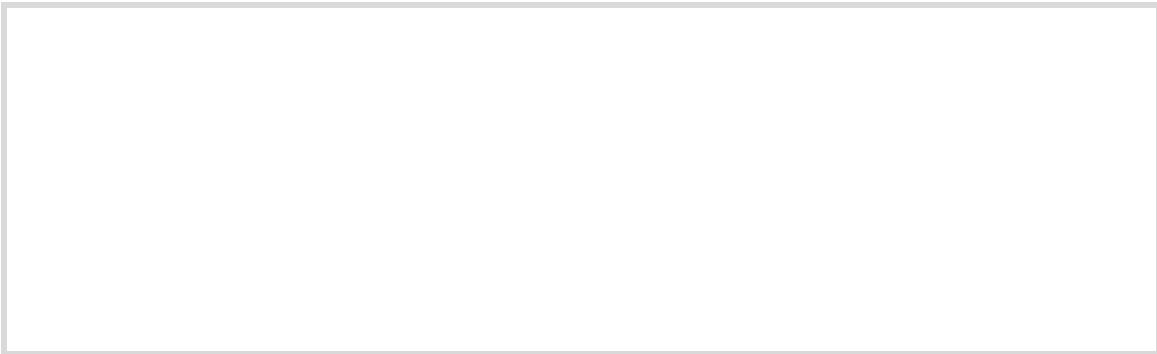
Please see page 220 for a hint.

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.

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	<p>Write a function to determine if a given search value is present in a list of integers:</p> <pre>bool linearSearch(const int numbers[], int size, int search);</pre> <p>If the value <code>search</code> is present in <code>numbers</code>, return <code>true</code>, otherwise, return <code>false</code>.</p>
Solution	<p>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.</p> <pre>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; }</pre> <p>Observe how the larger the list (<code>size</code>), 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.</p>
Challenge	<p>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:</p> <pre>int findLargestValue(const int numbers[], int size);</pre> <p>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.</p>
See Also	<p>The complete solution is available at 3-1-linearSearch.cpp or:</p> <pre>/home/cs124/examples/3-1-linearSearch.cpp</pre> 

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

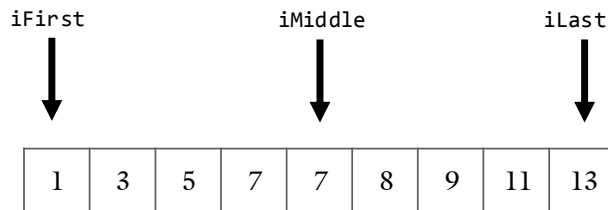
Write a function to determine if a given search value is present in a list of integers:

```
bool binarySearch(const int numbers[], int size, int search);
```

If the value `search` is present in `numbers`, return `true`, otherwise, return `false`.

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.

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 within 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];	<table><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	?	?	?	?	?	?	?	?	?	?	None of the slots are initialized
?	?	?	?	?	?	?	?	?	?			
char text[7] = "CS 124";	<table><tr><td>C</td><td>S</td><td></td><td>1</td><td>2</td><td>4</td><td>\0</td></tr></table>	C	S		1	2	4	\0	The initialized size is the same as the declared size			
C	S		1	2	4	\0						
char text[10] = "CS 124";	<table><tr><td>C</td><td>S</td><td></td><td>1</td><td>2</td><td>4</td><td>\0</td><td>\0</td><td>\0</td><td>\0</td></tr></table>	C	S		1	2	4	\0	\0	\0	\0	The first 7 are initialized, the balance are filled with �
C	S		1	2	4	\0	\0	\0	\0			
char text[] = "CS 124";	<table><tr><td>C</td><td>S</td><td></td><td>1</td><td>2</td><td>4</td><td>\0</td></tr></table>	C	S		1	2	4	\0	Declared to exactly the size necessary to fit the text			
C	S		1	2	4	\0						



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

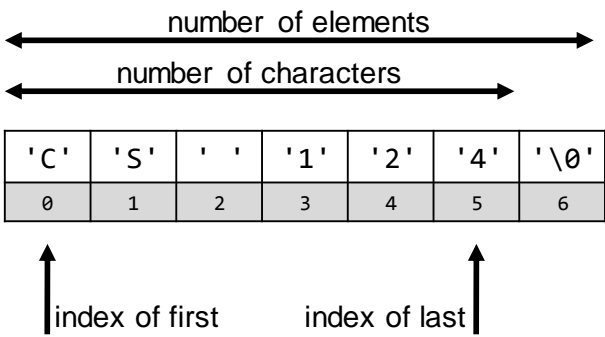
Strings are just arrays of characters where the end is marked with the null character ('\\0'). This means we can use a FOR loop to walk through a string much the same way we do with an integer array. The only difference is how we know we have reached the end of the array. With arrays of numbers, we typically 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	<code>sizeof(name)</code>
Number of elements in the array	<code>sizeof(name) / sizeof(name[0])</code>
Number of characters in the string (minus null)	<code>(sizeof(name) / sizeof(name[0]) - 1)</code>
Index of the last item	<code>(sizeof(name) / sizeof(name[0]) - 2)</code>

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 eENGINEERING**
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`



Example 3.2 – Sentence Case

Demo	This example will demonstrate how to traverse a string, processing each character individually.
Problem	<p>Write a program to to perform a sentence-case conversion (where only the first letter of a sentence is capitalized) on an input string.</p> <div> Please enter your input text: <u>this IS SOME tExT. soME More TeXt.</u> The sentence-case version of the input text is: This is some text. Some more text. </div>
Solution	<p>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, exclamation point, or question mark is encountered, the next letter needs to be uppercase. We capture this condition with the <code>isNewSentence</code> variable. If a sentence-ending character is found, we set <code>isNewSentence</code>. This variable remains set until an alphabetic character is found (<code>isalpha()</code>). In this case, we convert the letter to uppercase (<code>toupper()</code>) and set <code>isNewSentence</code> to false. Otherwise, we set the letter to lowercase (<code>tolower()</code>).</p> <pre> 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]); } } </pre>
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	<p>The complete solution is available at 3-2-sentenceCase.cpp or:</p> <div>/home/cs124/examples/3-2-sentenceCase.cpp</div> 

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	<p>Write a program to prompt the user for some text and display the text backwards:</p> <pre>Please enter some text: Software Engineering The text backwards is "gnireenignE erawtfoS"</pre>
Solution	<p>This function will consist of two parts: looping through the string to find the null-character, then progressing backwards to display the contents of the string.</p> <pre> /***** * 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; } </pre> <p>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).</p>
Challenge	<p>Notice how a WHILE loop is 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.</p> <p>Again, notice how the second loop uses a FOR loop. Can you re-write it to use a WHILE loop instead?</p>
See Also	<p>The complete solution is available at 3-2-backwards.cpp or:</p> <pre>/home/cs124/examples/3-2-backwards.cpp</pre> 

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 Booleans
    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
```



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.

account	pAccountNumber	receipt
120.12	←	100.00

Challenge

As a challenge, change the value `receipt` to reflect a modest “adjustment” to your account.

```
receipt = 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	<p>In this example, there are three variables: an integer representing the professor’s master copy of the grade (<code>gradeProf</code>), a pointer representing the student’s ability to access the grade (<code>pGradeStudent</code>), and a pointer representing the registrar’s copy of the grades (<code>pGradeRegistrar</code>).</p> <table><tr><th><code>gradeProf</code></th><th><code>pGradeStudent</code></th><th><code>pGradeRegistrar</code></th></tr><tr><td>86</td><td>←</td><td>←</td></tr></table> <p>The code for this examples is:</p> <pre>{ // 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; }</pre>	<code>gradeProf</code>	<code>pGradeStudent</code>	<code>pGradeRegistrar</code>	86	←	←
	<code>gradeProf</code>	<code>pGradeStudent</code>	<code>pGradeRegistrar</code>				
86	←	←					
	<p>Observe how neither <code>pGradeRegistrar</code> nor <code>pGradeStudent</code> actually contain a grade; they only contain the address of the professor’s grade. Thus, when <code>gradeProf</code> changes to 89, they will reflect the new value when <code>*pGradeRegistrar</code> and <code>*pGradeStudent</code> are dereferenced. This is one of the advantages of using pointers: sharing.</p>						
Challenge	<p>You may notice how <code>pGradeStudent</code> and <code>pGradeRegistrar</code> both have the <code>const</code> prefix. This guarentees that <code>*pGradeStudent</code> won’t inadvertently change <code>gradeProf</code>. As a challenge, try to change the value with the following code:</p> <pre>*pGradeStudent = 99; // Yahoo! Easy ‘A’!</pre> <p>Now, remove the <code>const</code> prefix from the <code>pGradeStudent</code> declaration. Does the compile error go away?</p>						
See Also	<p>The complete solution is available at 3-3-pointerToMasterCopy.cpp or:</p> <pre>/home/cs124/examples/3-3-pointerToMasterCopy.cpp</pre>						

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:

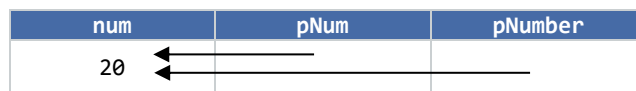
```
/******
 * SET TWENTY
 * Change the value to 20
 *****/
void setTwenty(int * pNumber)           // pass-by-pointer
{
    *pNumber = 20;                     // by changing the dereference value, we
}                                       // change who pNumber points to: num

/******
 * 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
    pointer = &reference;               // num1 in main().
    // 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 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.

num1	p1	pointer	num2/reference	p2
1.0	←	←	1.0 ←	←



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:

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

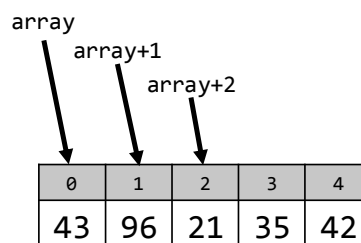
Before reading this section, please make sure you are able to:

- 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.

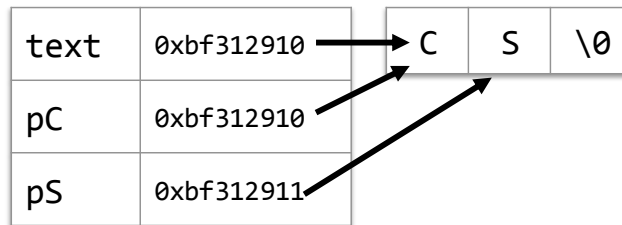
Recall from Chapter 3.3 that a pointer is a reference to a location in memory. We typically do not know where this memory is located until run-time; the operating system places the program in memory and often puts it in a different location every time. Recall from Chapter 3.0 that arrays are collections 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 leveraging 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
}
```

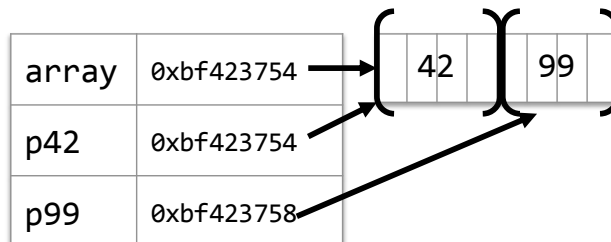
In this example, `text` points to the string "CS" or, more explicitly, to the first letter of the string. Next, the variable `pC` inherits its address from `text` which points to 'C' (hence the name `pC`). Finally, since the letter 'S' is one letter beyond 'C' in the string, it follows it has an address 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
    };
    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.



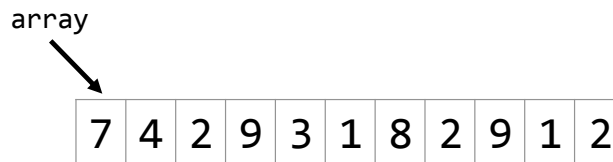
Because arrays (including strings) are just pointers, it is often most convenient to traverse an array with a pointer than with an index. This involves incrementing the pointer variable rather than an index.

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:

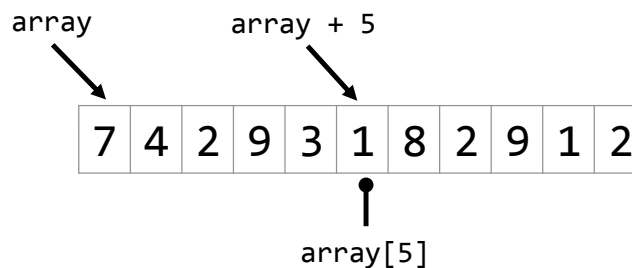
```
cout << "array[0] == " << array[0] << endl;  
cout << "*array == " << *array << endl;  
assert(array[0] == *array);
```

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.



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:

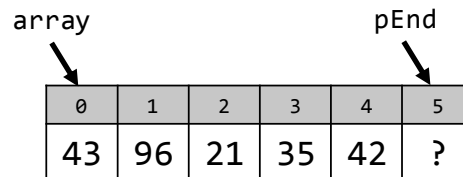
```
for (int i = 0; i < num; i++)
    cout << array[i];
```

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:

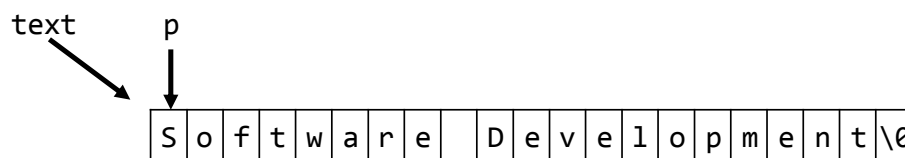
```
/home/cs124/examples/3-4-pointerArray.cpp
```

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	<p>Write a program to convert uppercase characters to lowercase and vice-versa.</p> <div>Please enter your text: <u>SOFTWARE eNGINEERING</u> Software Engineering</div>				
Solution	<p>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.</p> <table border="1"> <thead> <tr> <th>Array notation</th><th>Pointer notation</th></tr> </thead> <tbody> <tr> <td> <pre>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]); }</pre> </td><td> <pre>void convert(char * text) { for (char * p = text; *p; p++) if (isupper(*p)) *p = tolower(*p); else *p = toupper(*p); }</pre> </td></tr> </tbody> </table> <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.”</p> <p>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.</p>	Array notation	Pointer notation	<pre>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]); }</pre>	<pre>void convert(char * text) { for (char * p = text; *p; p++) if (isupper(*p)) *p = tolower(*p); else *p = toupper(*p); }</pre>
Array notation	Pointer notation				
<pre>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]); }</pre>	<pre>void convert(char * text) { for (char * p = text; *p; p++) if (isupper(*p)) *p = tolower(*p); else *p = toupper(*p); }</pre>				
Challenge	<p>As a challenge, try to modify the above program to count the number of uppercase letters in the input stream. The prototype is:</p> <div>int countUpper(const char *text);</div>				
See Also	<p>The complete solution is available at 3-4-toggleCase.cpp or:</p> <div>/home/cs124/examples/3-4-toggleCase.cpp</div> 				

Example 3.4 – String Length

Demo

This example will demonstrate how to walk through a string using the pointer notation.

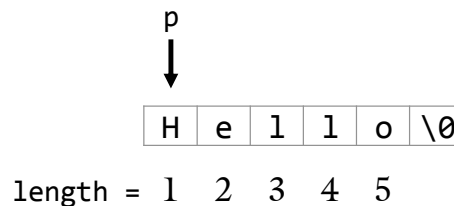
Problem

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.

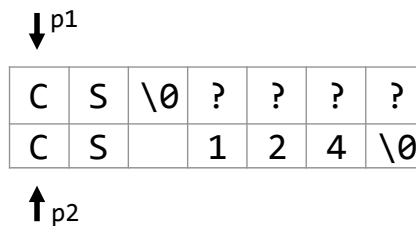
Problem

Write a program to determine if two strings are the same. Note that the equivalence 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.



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.

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.

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).

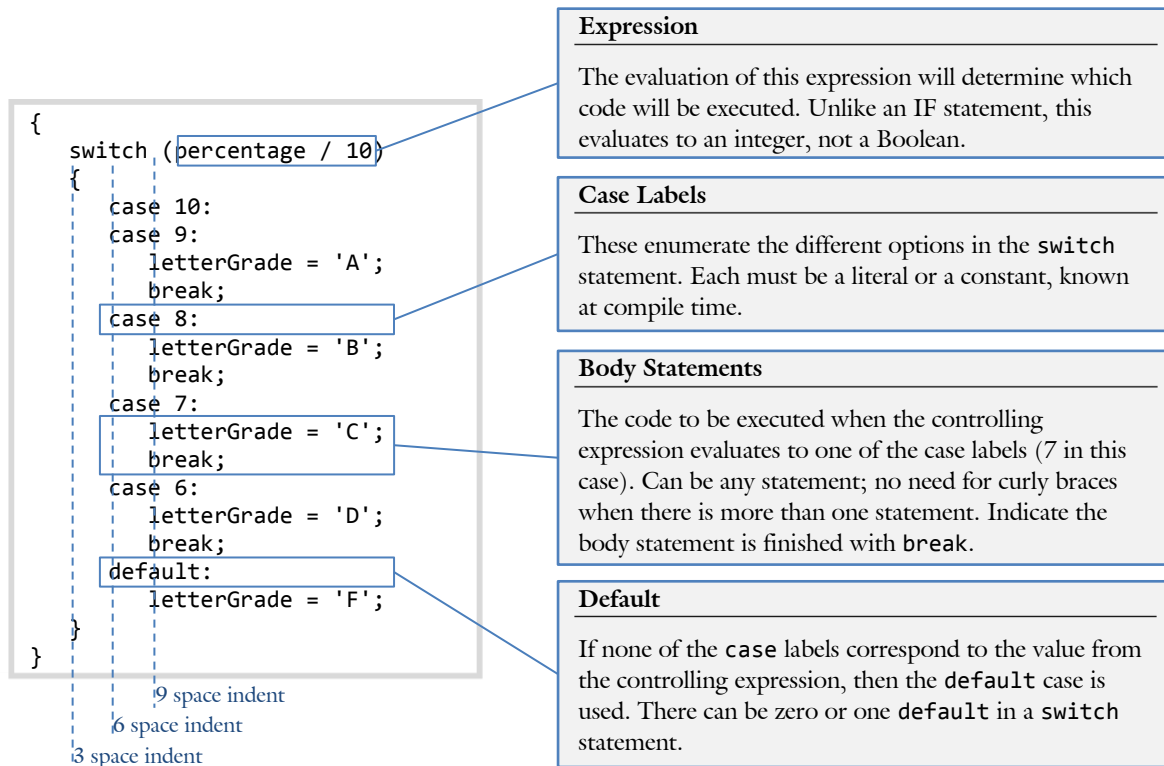
Overview

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
<p>Allows the programmer to choose between more than two options. Each option is specified with an integral value.</p> <pre>switch(option) { case 'Q': return true; case 'D': display(data); break; ... }</pre>	<p>Useful for selecting between two expressions or values (rather than two statements as an IF statement does).</p> <pre>cout << "Greetings " << (isMale ? "Mr." : "Mrs.") << lastName << endl;</pre>	<p>Enables a single integer to store 32 Boolean values worth of data for a more compact representation of an array of bools.</p> <pre>int value = 0x0001 // 1st bit 0x0004 // 3rd bit 0x0200; // 10th bit if (value & 0x001) cout << "1st bit";</pre>

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.



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 9³/₄. 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
```



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 convert 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
<code>~</code>	Bitwise NOT	<code>0101 == ~1010</code>
<code>&</code>	Bitwise AND	<code>1000 == 1100 & 1010</code>
<code> </code>	Bitwise OR	<code>1110 == 1100 1010</code>
<code>^</code>	Bitwise XOR	<code>0110 == 1100 ^ 1010</code>
<code><<</code>	Left shift	<code>0110 == 0011 << 1</code>
<code>>></code>	Right shift	<code>0110 == 1100 >> 1</code>

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
                                           // not use the && here... it
    // eat breakfast?                        // will always evaluate
    if (setting & eatBreakfast)               // to TRUE
        goEatBreakfast();

    // 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 (22)` and `ios::showpoint == 1024 (210)`, 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.

[illegible]

Solution

The first step to solving this problem is to create a number (called `mask`) with a single bit in the leftmost 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.

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

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`:

```
this 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.
<i>anything else</i>	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. Precede 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
```