CHAPTER 7

Pointers and C-Strings

Objectives

- To describe what a pointer is (§7.1).
- To learn how to declare a pointer and assign a value to it (§7.2).
- To access elements via pointers (§7.2).
- To pass arguments by reference with pointers (§7.3).
- To understand the relationship between arrays and pointers (§7.4).
- To know how to access array elements using pointers (§7.5).
- To declare constant pointers and constant data (§7.5).
- To learn how to return pointers from functions (§7.6).
- To use the new operator to allocate persistent memory dynamically (§7.7).
- To test and convert characters using the character functions (§7.9.1).
- To store and access strings using arrays and pointers (§7.9.2).
- To read strings from the keyboard (§7.9.3).
- To process strings using the string functions (§7.9.4).
7.1 Introduction

The preceding chapter introduced arrays. You can pass an array to a function, but you cannot return an array from a function. Often it is desirable to return an array from a function. How can this be done? You can use pointers to accomplish that task. Pointer is a powerful feature in C++ that enables you to directly manipulate computer memory. A single pointer can be used to reference an array, a string, an integer, or any other variables.

7.2 Pointer Basics

Pointer variables, simply called pointers, are designed to hold memory addresses as their values. Normally, a variable contains a specific value, e.g., an integer, a floating-point value, and a character. However, a pointer contains the memory address of a variable that in turn contains a specific value.

Like any other variables, pointers must be declared before they can be used. To declare a pointer, use the following syntax:

```
dataType *pVarName;
```

Each variable being declared as a pointer must be preceded by an asterisk (*). For example, the following statement declares a pointer variable named `pCount` that can point to an `int` variable.

```
int *pCount;
```

You can now assign the address of a variable to `pCount`. For example, the following code assigns the address of variable `count` to `pCount`:

```
int count = 5;
pCount = &count;
```

The ampersand (&) symbol is called the address operator when placed in front of a variable. It is a unary operator that returns the address of the variable.

Listing 7.1 gives a complete example.

```
Listing 7.1 TestPointer.cpp (Using Pointers)

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

int main()
{
    int count = 5,
        *pCount = &count;

    cout << "The address of count is " << &count << endl;
    cout << "The address of count is " << pCount << endl;
    cout << "The value of count is " << count << endl;
    cout << "The value of count is " << *pCount << endl;

    return 0;
}

The address of count is 1245064
The address of count is 1245064
The value of count is 5
The value of count is 5

Line 6 declares a variable named count with an initial value 5. Line 7 declares a pointer variable named pCount and initialized with the address of variable count. A pointer can be initialized when it is declared or using an assignment statement. If you assign an address to a pointer, the syntax is

\[
pCount = &count; \quad // \text{Correct}
\]

rather than

\[
*pCount = &count; \quad // \text{Wrong}
\]

Line 9 displays the address of count using &count. Line 10 displays the value stored in pCount which is same as &count. The value stored in count is retrieved directly from count in line 11 and is retrieved indirectly through a pointer variable using *pCount in line 12. Figure 7.1 shows the relationship between count and pCount.

Figure 7.1
The pCount contains the address of variable count.
Referencing a value through a pointer is often called indirection. The syntax for referencing a value from a pointer is

<Side Remark: indirect referencing>

*pointer

For example, you can increase count using

    count++; // direct reference

Or

    (*pCount)++; // indirect reference

<Side Remark: indirection operator>
<Side Remark: dereferenced operator>
<Side Remark: dereferenced>

The asterisk (*) used in the preceding statement is known as the indirection operator or dereference operator (dereference means indirect reference). When a pointer is dereferenced, the value at the address stored in the pointer is retrieved.

NOTE

<Side Remark: * in three forms>

You have used the asterisk (*) in three different ways:

  o As multiplication operator, such as
      double area = radius * radius * 3.14159;

  o To declare a pointer variable, such as
      int *pCount = &count;

  o To use as the indirection operator, such as
      (*pCount)++;

***End NOTE

CAUTION

<side remark: pointer type>

A pointer variable is declared with a type such as int, double, etc. You have to assign the address of the variable of the same type. It is a syntax error if the type of the variable does not match the type of the pointer. For example, the following code is wrong.

    int area = 1;
    double *pArea = &area; // Wrong

***End CAUTION

TIP

<side remark: naming pointers>
Pointers are variables. So the naming conventions for variables are applied to pointers. This book names pointers with prefix p, such as pCount and pCity. There are some exceptions, however. Soon you will know that an array can also function as a pointer. In this case, the variable may be declared as an array or a pointer.

**TIP**
<side remark: initializing pointer>
Like a local variable, a local pointer is assigned an arbitrary value if you don’t initialize it. A pointer may be initialized to 0, which is a special value for a pointer to indicate that the pointer points to nothing. You should always initialize pointers to prevent errors. Dereferencing a pointer that is not initialized could cause fatal runtime error or it could accidentally modify important data.

**TIP**
<side remark: debugging in IDE>
If you use an IDE such as C++Builder and Visual C++.NET, please refer to Learning C++ Effectively with C++Builder/Visual C++.NET in the supplements. This supplement shows you how to use a debugger to inspect pointers.

7.3 Passing Arguments by References with Pointers

There are three ways to pass arguments to a function in C++: pass by value, pass by reference with reference arguments, and pass by reference with pointers. Listing 5.3, TestPassByValue.cpp, demonstrated the effect of pass by value. Listing 5.4, TestReferenceVariable, demonstrated the effect of pass by reference with reference variables. Both examples used the swap function to demonstrate the effect. Let us rewrite the swap function using pointers, as shown in Listing 7.2.

Listing 7.2 TestPointerArgument.cpp (Passing Pointer Arguments)
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***
<Side Remark line 5: pointers>
<Side Remark line 8: indirect accessing>
<Side Remark line 23: passing address>

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

// Swap two variables
void swap(int *pValue1, int *pValue2)
{
    // Swap n1 with n2
```
```cpp
int temp = *pValue1;
*pValue1 = *pValue2;
*pValue2 = temp;
}

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;
    cout << "Before invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;
    // Invoke the swap function to attempt to swap two variables
    swap(&num1, &num2);
    cout << "After invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;

    return 0;
}
```

Before invoking the swap function, num1 is 1 and num2 is 2
After invoking the swap function, num1 is 2 and num2 is 1

The `swap` function has two pointer parameters (line 5). You invoke this function by passing the address of `num1` to `pValue1` and `num2` to `pValue2` (line 23). The `swap` function swaps the values pointed by the two pointers. After invoking the function, the values in variables `num1` and `num2` are swapped.

### 7.4 Arrays and Pointers

Recall that an array variable without a bracket and a subscript actually represents the starting address of the array. In this sense, an array variable is essentially a pointer. Suppose you declare an array of `int` value as follows:

```cpp
int list[6] = {11, 12, 13, 14, 15, 16};
```

Figure 7.2 shows the array in the memory. C++ allows you to access the elements in the array using the indirection operator. To access the first element, use `*list`, and other elements can be accessed using `*(list + 1)`, `*(list + 2)`, `*(list + 3)`, `*(list + 4)`, and `*(list + 5)`.
Figure 7.2

list points to the first element in the array.

An integer may be added or subtracted from a pointer. The pointer is incremented or decremented by that integer times the size of the element to which the pointer points.

list points to the starting address of the array. Suppose this address is 1000. Will list + 1 be 1001? No. It is 1000 + sizeof(int). Why? Since list is declared as an array of int elements, C++ automatically calculates the address for the next element by adding sizeof(int). Recall that sizeof(type) is the size of a data type (see §2.9, “Numeric Data Types and Operations”). The size of each data type is machine-dependent. On Windows, the size of the int type is usually 4.

Listing 7.3 gives a complete program that uses pointers to access array elements.

Listing 7.3 ArrayPointer.cpp (Accessing Array Using Pointers)
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.

<Side Remark line 6: declare array>
<Side Remark line 9: incrementing address>
<Side Remark line 10: dereference operator>
<Side Remark line 11: array indexed variable>

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

int main()
{
    int list[6] = {11, 12, 13, 14, 15, 16};
    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) << " value: " << *(list + i) << " value: " << list[i] << endl;
    return 0;
}
```

<Output>
address: 1245040 value: 11 value: 11
address: 1245044 value: 12 value: 12
address: 1245048 value: 13 value: 13
address: 1245052 value: 14 value: 14
address: 1245056 value: 15 value: 15
address: 1245060 value: 16 value: 16
<End Output>
As shown in the sample output, the address of the array list is 1245040. So (list + 1) is actually 1245040 + 4, and (list + 2) is 1245040 + 2 * 4 (line 9). Line 11 accesses the elements using pointers. For example, *list returns 11 and *(list + 1) returns 12. Line 11 accesses the elements using indexed variable list[i], which is equivalent to *(list + i).

CAUTION:

<Side Remark: operator precedence>
*(list + 1) is different from *list + 1. The dereference operator (*) has precedence over +. So, *list + 1 adds 1 to the value of the first element in the array, while *(list + 1) dereference the element at address (list + 1) in the array.

NOTE:

<Side Remark: compare pointers>
Pointers can be compared using relational operators (==, !=, <, <=, >, >=) to determine their order.

Arrays and pointers form close relationship. An array is essentially a pointer. A pointer for an array can be used just like an array. You can even use indexed variables with pointers. Listing 7.4 gives such an example:

Listing 7.4 PointerWithIndex.cpp (Using Pointers with Index)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

<Side Remark line 6: declare array>
<Side Remark line 7: declare pointer>
<Side Remark line 10: incrementing address>
<Side Remark line 11: dereference operator>
<Side Remark line 12: array indexed variable>
<Side Remark line 13: dereference operator>
<Side Remark line 14: pointer indexed variable>

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

int main()
{
    int list[6] = {11, 12, 13, 14, 15, 16};
    int *pList = list;

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) << " value: " << *(list + i) << " value: " << list[i] << " value: " << *(pList + i) << " value: " << pList[i] << endl;

    return 0;
}
```
Line 7 declares an int pointer assigned with the address of the array. Note that the address operator (&) is not needed to assign the address of the array to the pointer, because the name of the array is already the starting address of the array. This line is equivalent to

```c
int *pList = &list[0];
```

Here, &list[0] represents the address of list[0].

The element can be accessed using *(list + i), list[i], *(pList + i), and pList[i] (lines 11-14). So, arrays are used as pointers, and pointers are used as arrays, vice versa. However, there is one difference. You cannot change the address of the array once an array is declared. For example, the following statement is illegal.

```c
int list1[10], list2[10];
list1 = list2; // Wrong
```

In this sense, an array is a constant pointer.

### 7.5 Using const with Pointers

**<side remark: constant pointer>**

You learned how to declare a constant using the const keyword. A constant cannot be changed once it is declared. You can declare a constant pointer. For example, see the following code:

```c
double radius = 5;
double * const pValue = &radius;
```

Here pValue is a constant pointer. It must be declared and initialized in the same statement. You cannot assign a new address to pValue later. Though pValue is a constant, the data pointed by pValue is not constant. You can change it. For example, the following statement changes radius to 10.

```
*pValue = 10;
```

**<side remark: constant data>**

Can you declare that dereferenced data be constant? Yes. You can add the const keyword in front of the data type, as follows:
In this case, the pointer is a constant and the data pointed by the pointer is also a constant.

If you declare the pointer as

```cpp
const double * const pValue = &radius;
```

then the pointer is not a constant, but the data pointed by the pointer is a constant.

For example, see the following code:

```cpp
double radius = 5;
double * const pValue = &radius;
double length = 5;
pValue = 6; // OK
pValue = &length; // Wrong because pValue is constant pointer

const double *pValue1 = &radius;
*pValue1 = 6; // Wrong because pValue1 points to a constant data
pValue1 = &length; // OK

const double * const pValue2 = &radius;
*pValue2 = 6; // Wrong because pValue2 points to a constant data
*pValue2 = &length; // Wrong because pValue2 is a constant pointer
```

The `const` keyword is particularly useful for declaring parameters in functions. If a value does not change, you should declare it `const` to prevent it from being modified accidentally. Listing 7.5 gives such an example:

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

void printArray(const int *, const int); // function prototype

int main()
```
```cpp
int list[6] = {11, 12, 13, 14, 15, 16};
printArray(list, 6);
return 0;
```

```cpp
void printArray(const int * list, const int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}
```

```
11 12 13 14 15 16
```

The printArray function declares an array parameter with constant data (line 4). This ensures that the contents of the array will not be changed. Note that the size parameter is also declared const. This is usually not necessary, since an int parameter is passed by value. Even though size is modified in the function, it does not affect the original size value outside this function.

### 7.6 Returning Pointers from Functions

You can use pointers as parameters in a function. Can you return a pointer from a function? The answer is yes.

Suppose you want to write a function that passes an array argument and returns a new array that is the reversal of the array argument. An algorithm for the function can be described as follows:

1. Let the original array be list.
2. Declare a new array named **result** that has the same size as the original array.
3. Write a loop to copy the first element, second, ..., and so on in the original array to the last element, second last, ..., into the new array, as shown in the following diagram.

```
|   |   |   |   |   |   |
```
```
|   |   |   |   |   |   |
```
```
|   |   |   |   |   |   |
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
The return value type is an int pointer. How do you declare a new array in Step 2? You may attempt to declare it as

```cpp
int result[size];
```

But C++ does not allow the size to be a variable. To avoid this limitation, let us assume that the array size is 6. So, you can declare it as

```cpp
int result[6];
```

You can now implement the code in Listing 7.6, but you will soon find out that it is not working correctly.

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

int * reverse(int const * list, const int size)
{
    int result[6];
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        result[j] = list[i];
    }
    return result;
}

void printArray(int const *list, const int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << list[i] << " ";
    }
}

int main()
{
    int list[] = {1, 2, 3, 4, 5, 6};
    int *pList = reverse(list, 6);
    printArray(pList, 6);
    return 0;
}
```

Listing 7.6 WrongReverse.cpp (Returning a Pointer)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 4: reverse function
.Side Remark line 6: declare result array
.Side Remark line 8: reverse to result
.Side Remark line 13: return result
.Side Remark line 16: print array
.Side Remark line 25: invoke reverse
.Side Remark line 26: print array
The sample output is incorrect. Why? The reason is that the array result is a local variable. Local variables don’t persist; when the function returns, the local variables are thrown away. Attempting to use the pointer will result in erroneous and unpredictable results. To fix this problem, you have to allocate persistent storage for the result array so that it can be accessed after the function returns.

### 7.7 Dynamic Memory Allocation

C++ supports dynamic memory allocation, which enables you to allocate persistent storage dynamically. The memory is created using the `new` operator. For example,

```cpp
int *pValue = new int;
```

Here, `new int` tells the computer to allocate memory space for an `int` variable at runtime, and the address of the variable is assigned to the pointer `pValue`. So you can access the memory through the pointer. Here is another example:

```cpp
int *list = new int[10];
```

Here, `new int[10]` tells the computer to allocate memory space for an `int` array of ten elements, and the address of the array is assigned to `list`.

Now you can fix the problem in the preceding example by creating a new array dynamically in the reverse function. Listing 7.7 gives the new program.

Listing 7.7 CorrectReverse.cpp (Returning a Pointer)

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

int *pValue = new int;
 int *list = new int[10];
```

```cpp
Listing 7.7 CorrectReverse.cpp (Returning a Pointer)
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.
<Side Remark line 4: reverse function>
<Side Remark line 6: create array>
<Side Remark line 8: reverse to result>
<Side Remark line 13: return result>
<Side Remark line 16: print array>
<Side Remark line 25: invoke reverse>
<Side Remark line 26: print array>
```
int * reverse(const int * list, int size) 
    int *result = new int[size];
    for (int i = 0, j = size - 1; i < size; i++, j--) 
        result[j] = list[i]; 
    return result;
}

void printArray(const int *list, int size) 
    for (int i = 0; i < size; i++) 
        cout << list[i] << " ";

int main() 
    int list[] = {1, 2, 3, 4, 5, 6};
    int *pList = reverse(list, 6);
    printArray(pList, 6);
    return 0;

<Output>
6 5 4 3 2 1
<End Output>

Listing 7.7 is almost identical to Listing 7.6 except that the new result array is created using the new operator dynamically. The size can be a variable when creating an array using the new operator.

.Side Remark: heap>
C++ allocates local variables in the stack, but the memory reserved by the new operator in an area of memory called the heap. The memory reserved remains available until you explicitly free it or the program terminates. If you reserve memory while in a function, the memory is still available after the function returns. The result array is created in the function (line 6). After the function returns in line 25, the result array is intact. So, you can access it in line 26 to print all the elements in the result array.

To explicitly free the memory created by the new operator, use the delete keyword before the pointer. For example,

delete pValue;
If the memory is allocated for an array, the [] symbol must be placed between the delete keyword and the pointer to the array. For example,

```cpp
delete [] list;
```

CAUTION:

<Side Remark: delete dynamic memory>
You should use the delete keyword only with the pointer that points to the memory created by the new operator. Otherwise, it may cause unexpected problems.

CAUTION:

<Side Remark: memory leak>
You might inadvertently reassign a pointer before deleting the memory to which it points. Consider the following code:

```cpp
***PD: Please add line numbers in the following code***
int *pValue = new int;
pValue = 45;
pValue = new int;
```

Line 1 declares a pointer assigned with a memory space for an int value. Line 2 assigns 45 to the value. Line 3 assigns a new memory space to pValue. The original memory space that holds value 45 is not available, because it is not pointed by any pointer. This memory cannot be accessed and cannot be deleted. This is a memory leak.

### 7.8 Case Studies: Counting the Occurrences of Each Letter

This section presents a program that counts the occurrences of each letter in an array of characters. The program does the following:

1. Generate one hundred lowercase letters randomly and assign them to an array of characters. You can obtain a random letter by using the getRandomLowerCaseLetter() function in the RandomCharacter.h header file in Listing 5.12.

2. Count the occurrences of each letter in the array. To count the occurrences of each letter in the array, create an array, say counts of twenty-six int values, each of which counts the occurrences of a letter, as shown in Figure 7.3. That is, `counts[0]` counts the
number of a’s, counts[1] counts the number of b’s, and so on.

| chars[0] | counts[0] |
| chars[1] | counts[1] |
| ...     | ...       |
| ...     | ...       |
| chars[98] | counts[25] |
| chars[99] | counts[26] |

**Figure 7.3**
The chars array stores 100 characters and the counts array stores 26 counts, each counts the occurrences of a letter.

Listing 7.8 gives the complete program.

```cpp
#include <iostream>
#include "RandomCharacter.h"
using namespace std;

const int NUMBER_OF_LETTERS = 100;
char * createArray();
void displayArray(char []);
int * countLetters(char []);
void displayCounts(int []);
```
int main()
{
    // Declare and create an array
    char * chars = createArray();

    // Display the array
    cout << "The lowercase letters are: " << endl;
    displayArray(chars);

    // Count the occurrences of each letter
    int * counts = countLetters(chars);

    // Display counts
    cout << endl;
    cout << "The occurrences of each letter are: " << endl;
    displayCounts(counts);

    return 0;
}

char * createArray()
{
    // Declare an array of characters and create it
    char *chars = new char[NUMBER_OF_LETTERS];

    // Create lowercase letters randomly and assign them to the array
    srand(time(0));
    for (int i = 0; i < NUMBER_OF_LETTERS; i++)
        chars[i] = getRandomLowerCaseLetter();

    // Return the array
    return chars;
}

void displayArray(char chars[])
{
    // Display the characters in the array 20 on each line
    for (int i = 0; i < NUMBER_OF_LETTERS; i++)
    {
        if ((i + 1) % 20 == 0)
        {
            cout << chars[i] << endl;
        }
        else
        {
            cout << chars[i] << " ";
        }
    }
}

int * countLetters(char chars[])
{
    // Declare and create an array of 26 int
    int *counts = new int[26];

    // Count the occurrences of each letter
    return counts;
}

void displayCounts(int counts[])
{
    // Display counts
    for (int i = 0; i < 26; i++)
    {
        cout << "The occurrence of " << i << " is: " << counts[i] << endl;
    }
}
// Initialize the array
for (int i = 0; i < NUMBER_OF_LETTERS; i++)
    counts[i] = 0;

// For each lowercase letter in the array, count it
for (int i = 0; i < 26; i++)
    counts[chars[i] - 'a'] ++;
return counts;

// Display counts
void displayCounts(int counts[])
{
    for (int i = 0; i < 26; i++)
    {
        if ((i + 1) % 10 == 0)
            cout << counts[i] << " " << static_cast<char>(i + 'a') << endl;
        else
            cout << counts[i] << " " << static_cast<char>(i + 'a') << " ";
    }
}

<Output>
The lowercase letters are:
pyaounsuibhgywqlbyoxbrgixwvcgraspysiznfjvcjca cvlajrxrdtqw
mayekdmemojvkmenvta rmovdhdfox dg iuwr iq h

The occurrences of each letter are:
6 a 3 b 4 c 4 d 3 e 2 f 4 g 4 h 6 i 4 j
2 k 2 1 6 m 2 n 6 o 2 p 3 q 6 r 2 s 3 t
4 u 8 v 4 w 4 x 5 y 1 z
<End Output>
The createArray function (lines 32-45) generates an array of
one hundred random lowercase letters. Line 14 invokes the
function and assigns the array to pointer chars. What would be
wrong if you rewrote the code as follows?

char * chars = new char[100];
chars = createArray();

You would be creating two arrays. The first line would create
an array by using new char[100]. The second line would create
an array by invoking createArray() and assign the reference of
the array to chars. The array created in the first line would
cause memory leak because it is no longer referenced and cannot
be deleted.

Invoking getRandomLowerCaseLetter() (line 41) returns a random
lowercase letter. This function is defined in the
RandomCharacter header file in Listing 5.12. This header file is included in line 2.

The countLetters function (lines 61-75) returns an array of twenty-six int values, each of which stores the number of occurrences of a letter. The function processes each letter in the array and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

```c
for (int i = 0; i < NUMBER_OF_LETTERS; i++)
    if (counts[chars[i] == 'a')
        count[0]++;
    else if (counts[chars[i] == 'b')
        count[1]++;
    ...
```

But a better solution is given in lines 51-52.

```c
for (int i = 0; i < NUMBER_OF_LETTERS; i++)
    counts[chars[i] - 'a']++;
```

If the letter (chars[i]) is 'a', the corresponding count is counts['a' - 'a'] (i.e., counts[0]). If the letter is 'b', the corresponding count is counts['b' - 'a'] (i.e., counts[1]) since the ASCII code of 'b' is one more than that of 'a'. If the letter is 'z', the corresponding count is counts['z' - 'a'] (i.e., counts[25]) since the ASCII code of 'z' is 25 more than that of 'a'.

Figure 7.4 shows the call stack and heap during and after executing createArray. See Review Question 7.14 to show the call stack and heap for other functions in the program.

![Figure 7.4](image)

(a) Executing createArray in Line 6

(b) After exiting createArray in Line 6

**Figure 7.4**

(a) An array of one hundred characters is created when executing createArray. (b) This array is returned and assigned to the variable chars in the main function.

### 7.9 Characters and Strings

*<side remark: pointer-based string>*

*<side remark: C-string>*
Strings are used often in programming. You have used string literals. A string is sequence of characters. There are two ways to process strings in C++. One way is to treat strings as arrays of characters. This is known as pointer-based strings or C-strings. The other way is to process strings using the string class. The string class will be introduced in §9.14. This section introduces pointer-based strings.

7.9.1 Character Functions

C++ provides several functions for testing a character, as shown in Table 7.1. These functions tests a single character and returns true or false. Note that these functions actually return an int value. A nonzero integer corresponds to true and zero corresponds to false. C++ also provides two functions for converting cases, as shown in Table 7.2.

Table 7.1
Character Test Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>isdigit(c)</td>
<td>Returns true if c is a digit.</td>
<td>isdigit('7') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isdigit('a') is false</td>
</tr>
<tr>
<td>isalpha(c)</td>
<td>Returns true if c is a letter.</td>
<td>isalpha('7') is false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isalpha('a') is true</td>
</tr>
<tr>
<td>isalnum(c)</td>
<td>Returns true if c is a letter or a digit.</td>
<td>isalnum('7') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isalnum('a') is true</td>
</tr>
<tr>
<td>islower(c)</td>
<td>Returns true if c is a lowercase letter.</td>
<td>islower('7') is false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>islower('a') is true</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>Returns true if c is an uppercase letter.</td>
<td>isupper('7') is false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isupper('A') is true</td>
</tr>
<tr>
<td>isspace(c)</td>
<td>Returns true if c is a whitespace character.</td>
<td>isspace('	') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isspace('A') is false</td>
</tr>
<tr>
<td>isprint(c)</td>
<td>Returns true if c is a printable character including space '. '</td>
<td>isprint(' ') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isprint('A') is false</td>
</tr>
<tr>
<td>isgraph(c)</td>
<td>Returns true if c is a printable character excluding space '. '</td>
<td>isgraph(' ') is false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isgraph('A') is true</td>
</tr>
<tr>
<td>ispunct(c)</td>
<td>Returns true if c is a printable character other than a digit, letter, or space.</td>
<td>ispunct('**') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ispunct('A') is false</td>
</tr>
<tr>
<td>iscntrl(c)</td>
<td>Returns true if c is a control character such as '\n', '\f', '\v', '\a', and '\b'.</td>
<td>iscntrl('**') is true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iscntrl('n') is true</td>
</tr>
</tbody>
</table>

Table 7.2
Case Conversion Functions
Function | Description | Example
---|---|---
tolower(c) | Returns the lowercase equivalent of c, if c is an uppercase letter. Otherwise, return c itself. | tolower('A') returns 'a'
 | | tolower('a') returns 'a'
toupper(c) | Returns the uppercase equivalent of c, if c is a lowercase letter. Otherwise, return c itself. | toupper('A') returns 'A'
 | | toupper('a') returns 'A'
toupper('\t') returns '\t'
toupper('\t') returns '\t'

Listing 7.9 gives a program to test character functions.

Listing 7.9 CharacterFunctions.cpp (Testing Character Functions)

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

int main()
{
    cout << "Enter a character: ";
    char ch;
    cin >> ch;
    cout << "You entered " << ch << endl;
    if (islower(ch)) {
        cout << "It is a lowercase letter " << endl;
        cout << "Its equivalent uppercase letter is " << static_cast<char>(toupper(ch)) << endl;
    } else if (isupper(ch)) {
        cout << "It is an uppercase letter " << endl;
        cout << "Its equivalent lowercase letter is " << static_cast<char>(tolower(ch)) << endl;
    } else if (isdigit(ch)) {
        cout << "It is a digit character " << endl;
    }
    return 0;
```

<Output>
Enter a character: a
You entered a
It is a lowercase letter
Its equivalent uppercase letter is A

Enter a character: T
You entered T
It is an uppercase letter
Its equivalent lowercase letter is t

Enter a character: 8
You entered 8
It is a digit character

7.9.2 Storing and Accessing Strings

A pointer-based string in C++ is an array of characters ending in the null terminator ('\0'), which indicates where a string terminates in memory. An array can be accessed via a pointer. So a string can also be accessed via a pointer, which points to the first character in the string. So you can declare a string variable using an array or a pointer. For example, the following two declarations are both fine:

```cpp
char city[7] = "Dallas"; // Option 1
char *pCity = "Dallas"; // Option 2
```

Each declaration creates a sequence that contain characters 'D', 'a', 'l', 'l', 'a', 's', and '\0'.

You can access city or pCity using the array syntax or pointer syntax. For example,

```cpp
cout << city[1] << endl;
cout << *(city + 1) << endl;
cout << pCity[1] << endl;
cout << *(pCity + 1) << endl;
```

each displays character a (the second element in the string).

NOTE:

There is a subtle difference between the two declarations. For the first array declaration, you cannot reassign the array city after it is declared. But you can reassign a new address to the pointer pCity for the second declaration. If you
add the `const` keyword in the second declaration as follows:

```c
char * const pCity = "Dallas";
```

`pCity` becomes a constant pointer and cannot be reassigned.

***END NOTE***

**TIP:**

You can display all characters in a string using

```c
cout << city;
```

or

```c
cout << pCity;
```

Note that

```c
cout << *pCity;
```

would print `D`, the first character in the string pointed by `pCity`.

***END TIP***

7.9.3 Reading Strings

You can read a string from the keyboard using the `cin` object. For example, see the following code:

```c
cout << "Enter a city: ";
cin >> city; // read to array city
cout << "You entered " << city << endl;
```

Line 2 reads a string to an array and line 6 reads a string to a pointer.

**CAUTION:**

When you read a string to array `city`, make sure to leave room for the null terminator character. Since
*city* has a size 7, your input should not exceed 6 characters.

***END NOTE***

This approach to read a string is simple, but there is a problem. The input ends with a whitespace character. Suppose you want to enter New York, you have to use an alternative approach. C++ provides the `cin.getline` function in the `<iostream>` header file, which reads a string into an array. The syntax of the function is:

```
cin.getline(char array[], int size, char delimitChar)
```

The function stops reading characters when the delimiter character is encountered or when the `size - 1` number of characters are read. The last character in the array is reserved for the null terminator (`'\0'`). If the delimiter is encountered, it is read, but not stored in the array. The third argument `delimitChar` has a default value (`'\n'`).

The following code uses the `cin.getline` function to read a string.

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***
***PD: Please add line numbers in the following code***

```
<Side Remark line 1: declare array>
char city[30];
<Side Remark line 3: string to array>
cout << "Enter a city: ";
cin.getline(city, 30, '\n'); // read to array city
cout << "You entered " << city << endl;
```

Since the default value for the third argument in the `cin.getline` function, line 3 can be replaced by

```
cin.getline(city, 30); // read to array city
```

NOTE:

<side remark: cin.getline>
The syntax for the `cin.getline` function is new. The `getline` is actually a function in the `cin` object. You will better understand the syntax in Chapter 9, “Classes and Objects.” For now, just accept that this is how to read input from the keyboard.

***END NOTE***

7.9.4 String Functions
C++ provides several useful and powerful functions for testing and processing strings, as shown in Table 7.3. To use these functions, your program needs to include the cstring header file.

Table 7.3
String Functions

Function       Description

int strlen(char *s1)
               Returns the length of the string, i.e., the number of the
               characters before the null terminator.

char *strcpy(char *s1, const char *s2)
               Copies the string s2 to string s1. The value in s1 is returned.

char *strncpy(char *s1, const char *s2, size_t n)
               Copies at most n characters from string s2 to string s1. The
               value in s1 is returned.

char *strcat(char *s1, const char *s2)
               Appends string s2 to s1. The first character of s2 overwrites the
               null terminator is s1. The value in s1 is returned.

char *strncat(char *s1, const char *s2, size_t n)
               Appends at most n characters from string s2 to s1. The first
               character of s2 overwrites the null terminator is s1 and appends
               a null terminator to the result. The value in s1 is returned.

int *strcmp(char *s1, const char *s2)
               Returns a value greater than 0, 0, or less than 0 if s1 is greater
               than, equal to, or less than s2 based on the numeric code of the
               characters.

int *strncmp(char *s1, const char *s2, size_t n)
               Returns a value greater than 0, 0, or less than 0 if the n
               characters in s1 is greater than, equal to, or less than the first
               n characters in s2 based on the numeric code of the characters.

int atoi(char *s1)
               Converts the string to an int value.

double atof(char *s1)
               Converts the string to a double value.

long atol(char *s1)
               Converts the string to a long value.

void itoa(int value, char *s1, int radix)
               Converts the value to a string based on specified radix.

NOTE

<size remark: copying strings>
size_t is a C++ type. For most compiler, it is same as unsigned int.
<size remark: copying strings>
<size remark: strcpy and strncpy>
Functions `strcpy` and `strncpy` can be used to copy a source string in the second argument to a target string in the first argument. The target string must have already been allocated with sufficient memory for the function to work. Function `strncpy` is equivalent to `strcpy` except that `strncpy` specifies the number of characters to be copied from the source. Listing 7.10 gives a program to demonstrate these two functions.

Listing 7.10 CopyString.cpp (Copying a String)

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

int main()
{
    char s1[20];
    char s2[20] = "Dallas, Texas";
    char s3[10] = "AAAAAAAAAA";

    strcpy(s1, s2);
    strncpy(s3, s2, 6);
    s3[6] = '\0'; // Insert null terminator

    cout << "The string in s1 is " << s1 << endl;
    cout << "The string in s2 is " << s2 << endl;
    cout << "The string in s3 is " << s3 << endl;
    cout << "The length of string s3 is " << strlen(s3) << endl;

    return 0;
}
```

<Output>
The string in s1 is Dallas, Texas
The string in s2 is Dallas, Texas
The string in s3 is Dallas
The length of string s3 is 6

<End Output>

<side remark: null terminator>
Three strings s1, s2, and s3 are declared in lines 7-9. Line 11 copies s2 to s1 using the `strcpy` function. Line 12 copies the
first 6 characters from s2 to s1 using the strncpy function. Note that the null terminator character is not copied in this case. To terminate the string properly, a null terminator is manually inserted at s3[6] (the end of the new string). What would happen if you run the program without line 13? String s3 would become DallasAAAA.

NOTE:

<side remark: ignoring return value>
Both strcpy and strncpy functions return a string value. But they are invoked as statements in lines 11-12. Recall that a function with nonvoid return value type may be invoked as a statement if you are not interested in the return value of the function. In this case, the return value is simply ignored.

***End of NOTE

<side remark: combining strings>
<side remark: strcat and strncat>
Functions strcat and strncat can be used to append the string in the second argument to the string in the first argument. The first string must have already been allocated with sufficient memory for the function to work. Function strncat is equivalent to strcat except that strncat specifies the number of characters to be appended from the second string. Listing 7.11 gives a program to demonstrate these two functions.

Listing 7.11 CombineString.cpp (Appending a String)
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***
***PD: Please add line numbers in the following code***
<Side Remark line 2: include cstring header>
<Side Remark line 7: declare three strings>
<Side Remark line 11: append strings to s1>
<Side Remark line 12: append strings to s3>
<Side Remark line 17: string s1 length>
<Side Remark line 18: string s3 length>

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

int main()
{
    char s1[20] = "Dallas";
    char s2[20] = "Texas, USA";
    char s3[20] = "Dallas";

    strcat(strcat(s1, ", ", s2);
```
strncat(strcat(s3, "," ), s2, 5);

cout << "The string in s1 is " << s1 << endl;
cout << "The string in s2 is " << s2 << endl;
cout << "The string in s3 is " << s3 << endl;
cout << "The length of string s1 is " << strlen(s1) << endl;
cout << "The length of string s3 is " << strlen(s3) << endl;

return 0;

<Output>
The string in s1 is Dallas, Texas, USA
The string in s2 is Texas, USA
The string in s3 is Dallas, Texas
The length of string s1 is 18
The length of string s3 is 13
<End Output>

<side remark: return string>
Three strings s1, s2, and s3 are declared in lines 7-9. Line 11 invokes strcat twice. First, strcat(s1, ",") appends "," to s1 and returns new s1. Second, strcat(strcat(s1, "," ), s2) appends s2 to the new s1. So, s1 is Dallas, Texas, USA. Similarly, line 12 appends ", " and the first five characters in s2 to s1. So, s1 is Dallas, Texas.

<side remark: comparing strings>
<side remark: strcmp and strncmp>
Functions strcmp and strncmp can be used to compare two strings. How do you compare two strings? You compare their corresponding characters according to their numeric code. Most compilers use the ASCII code for characters.

The function returns the value 0 if s1 is equal to s2, a value less than 0 if s1 is less than s2, and a value greater than 0 if s1 is greater than s2. For example, suppose s1 is "abc" and s2 is "abg", and strcmp(s1, s2) returns -4. The first two characters (a vs. a) from s1 and s2 are compared. Because they are equal, the second two characters (b vs. b) are compared. Because they are also equal, the third two characters (c vs. g) are compared. Since the character c is 4 less than g, the comparison returns -4.

Listing 7.12 gives a program to demonstrate these two functions.

Listing 7.12 CompareString.cpp (Comparing Strings)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.***
***PD: Please add line numbers in the following code***
<Side Remark line 2: include cstring header>
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char *s1 = "abcdefg";
    char *s2 = "abcdg";
    char *s3 = "abcdg";

    cout << "strcmp(s1, s2) is " << strcmp(s1, s2) << endl;
    cout << "strcmp(s2, s1) is " << strcmp(s2, s1) << endl;
    cout << "strcmp(s2, s3) is " << strcmp(s2, s3) << endl;
    cout << "strncmp(s1, s2, 3) is " << strncmp(s1, s2, 3) << endl;
    return 0;
}

<Output>
strcmp(s1, s2) is -2
strcmp(s2, s1) is 2
strcmp(s2, s3) is 0
strncmp(s1, s2, 3) is 0
<End Output>

<side remark: return string>
Three strings s1, s2, and s3 are declared in lines 7-9. Line 11 invokes 
\texttt{strcmp(s1, s2)}, which returns -2, because 'e' - 'g' is -2. Line 12 invokes 
\texttt{strcmp(s2, s1)}, which returns 2, because 'g' - 'e' is 2. Line 13 invokes 
\texttt{strcmp(s2, s3)}, which returns 0, because the two strings are identical. Line 14 invokes 
\texttt{strncmp(s1, s2, 3)}, which returns 0, because the first three 
character substrings in both s1 and s2 are identical.

NOTE:

<side remark: 1, 0, or -1?>
With some compilers, functions \texttt{strcmp} and \texttt{strncmp} always return 1, 0, or -1.

***End of NOTE

<side remark: conversion functions>
Functions \texttt{atoi}, \texttt{atof}, and \texttt{atol} convert a string to a numeric 
value. Function \texttt{itoa} converts an integer to a string.

Listing 7.13 gives a program to demonstrate these functions.
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    cout << atoi("4") + atoi("5") << endl;
    cout << atof("4.5") + atof("5.5") << endl;

    char s[10];
    itoa(42, s, 8);
    cout << s << endl;
    itoa(42, s, 10);
    cout << s << endl;
    itoa(42, s, 16);
    cout << s << endl;

    return 0;
}

<Output>
9
10
52
42
2a
<End Output>

<side remark: atoi>
<side remark: atof>
<side remark: itoa>
Invoking atoi("4") returns an int value 4 in line 7. Invoking atoi("4.5") returns a double value 4.5 in line 8. Invoking itoa (42, s, 8) converts an int value 42 to a string based radix 8 in line 11. Invoking itoa (42, s, 10) converts an int value 42 to a string based radix 10 in line 14. Invoking itoa (42, s, 16) converts an int value 42 to a string based radix 8 in line 17.
7.9.5 Example: Obtaining Substrings

Often it is useful to obtain a substring from a string. But there is no such function in C++. You can develop your own function for extracting substrings. The header of the function can be specified as:

```c
char * substring(const char * const s, int start, int end)
```

This function returns a string which is a substring in `s` from index `start` to index `end`. For example, invoking

```c
substring("Welcome to Java", 2, 5)
```

returns "lcom".

Listing 7.14 gives a header file for the function.

### Listing 7.14 Substring.h (Substring Function Header File)

```c
char * substring(const char * const s, int start, int end)
{
    char * pNewString = new char[end - start + 1 + 1];
    int j = 0;
    for (int i = start; i <= end; i++, j++)
    {
        pNewString[j] = s[i];
    }
    pNewString[j] = '\0'; // Set a null terminator
    return pNewString;
}
```

A new string with the right size `end - start + 1 + 1` is created in line 3. `end - start + 1` is the number of characters extracted from the original string to hold the substring. The extra +1 is for the null terminator. The substring from index `start` to end is copied to `pNewString` (lines 6-9). The null terminator is set in line 11.

Listing 7.15 gives a test program for this function.

### Listing 7.15 TestSubstring.h (Using Substring Function)

```c
char * substring(const char * const s, int start, int end)
```
#include <iostream>
#include "Substring.h"
using namespace std;

int main()
{
    char *s = "Atlanta, Georgia";
    cout << substring(s, 0, 6);
    return 0;
}

CAUTION
The substring returned from this function is created using the new operator. The user of the substring function should delete the string after it finishes using it to release the memory occupied by this substring. Otherwise, the string exists as long as the program is alive. It is better to redesign the function to pass the result substring as a parameter as follows:

void substring(const char * const s, int start, int end, char * substr)

***End CAUTION

7.10 Case Studies: Checking Palindromes

A string is a palindrome if it reads the same forward and backward. The words "mom," "dad," and "noon," for instance, are all palindromes.

How do you write a program to check whether a string is a palindrome? One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say low and high, to denote the position of two characters at the beginning and the end in a string s, as shown in Listing 7.16 (lines 24, 27). Initially, low is 0 and high is s.length() – 1. If the two characters at these positions match, increment low by 1 and
decrement high by 1 (lines 33-34). This process continues until (low >= high) or a mismatch is found.

Listing 7.16 CheckPalindrome.cpp (Is a String Palindrome?)

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

bool isPalindrome(const char * const s)
{
    // The index of the first character in the string
    int low = 0;

    // The index of the last character in the string
    int high = strlen(s) - 1;

    while (low < high)
    {
        if (s[low] != s[high])
            return false; // Not a palindrome

        low++;
        high--;
    }

    return true; // The string is a palindrome
}

int main()
{
    // Prompt the user to enter a string
    cout << "Enter a string: ";
    char s[80];
    cin.getline(s, 80);

    if (isPalindrome(s))
        cout << s << " is a palindrome" << endl;
    else
        cout << s << " is not a palindrome" << endl;

    return 0;
}
```

Enter a string: **abccba**
abccba is a palindrome
Enter a string: **abca**
abca is not a palindrome

Line 12 declares an array with a size 80. Line 13 reads a string to the array. Line 15 invokes the `isPalindrome` function to check whether the string is a palindrome.

The `isPalindrome` function is declared in lines 23-41 to return a Boolean value. The address of the string is passed to the pointer `s`. The length of the string is determined by invoking `strlen(s)` in line 29.

**Key Terms**

***PD: Please place terms in two columns same as in intro5e.***

- address operator 176
- C-string
- delete operator
- dereference operator
- heap
- indirection operator 177
- memory leak 177
- new operator
- null terminator 177
- pointer-based string

**Chapter Summary**

- Pointers are variables that store the memory address of other variables.
- The declaration
  
  ```
  int *pCount;
  ```
  
declares `pCount` to be a pointer that can point to an `int` variable.
- The ampersand (`&`) symbol is called the `address operator` when placed in front of a variable. It is a unary operator that returns the address of the variable.
- A pointer variable is declared with a type such as `int`, `double`, etc. You have to assign the address of the variable of the same type.
- Like a local variable, a local pointer is assigned an arbitrary value if you don’t initialize it. A pointer may be initialized to 0, which is a special value for a pointer to indicate that the pointer points to nothing.
• The asterisk (*) used in the preceding statement is known as the indirection operator or dereference operator (dereference means indirect reference). When a pointer is dereferenced, the value at the address stored in the pointer is retrieved.
• There are three ways to pass arguments to a function in C++: pass by value, pass by reference with reference arguments, and pass by reference with pointers.
• An array variable without a bracket and a subscript actually represents the starting address of the array.
• You can access array elements using pointers or index variables.
• An integer may be added or subtracted from a pointer. The pointer is incremented or decremented by that integer times the size of the element to which the pointer points.
• The const keyword can be used to declare constant pointer and constant data.
• A pointer may be returned from a function.
• You should not return the address of a local variable from a function.
• The new operator can be used to allocate persistent memory on the heap. You can use the delete operator to release this memory.
• C++ provides useful functions for testing characters and converting letters: isdigit(c), isalpha(c), isalnum(c), islower(c), isupper(c), isspace(c), isprint(c), isgraph(c), ispunct(c), and isdcntrl(c).
• A string can be initialized using a string literal. A null terminator character ('\0') is automatically inserted.
• The cin.getline function can be used to read a string from the keyboard.
• C++ provides useful functions for processing strings: strlen(s), strcpy(s1, s2), strncpy(s1, s2, n), strcat(s1, s2), strncat(s1, s2, n), strcmp(s1, s2), strncmp(s1, s2, n), isgraph(c), ispunct(c), and isdcntrl(c).

Review Questions
Section 7.2 Pointer Basics
7.1 How do you declare a pointer variable? Does a local pointer variable have a default value?
7.2 How do you assign a variable’s address to a pointer variable? What is wrong in the following code?

```cpp
int x = 30;
int *pX = x;
cout << "x is " << x << endl;
cout << "x is " << pX;
```
7.3
What is wrong in the following code?

```c
    double x = 3.0;
    int *pX = 6x;
```

7.4
Suppose you create a dynamic array and later you need to release it. Identify two errors in the following code.

```c
    double x[] = new double[30];
    ...
    delete x;
```

Section 7.3 Passing Arguments by References with Pointers

7.5
What is the output of the following code?

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

void f1(int x, int &y, int *z)
{
    x++;
    y++;
    (*z)++;
}

int main()
{
    int i = 1, j = 1, k = 1;
    f1(i, j, &k);

    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
    cout << "k is " << k << endl;

    return 0;
}
```

Section 7.4 Arrays and Pointers

7.6
Assume you declared `int *p` and p’s current value is 100. What is `p + 1`?

7.7
Assume you declared `int *p`. What are the differences among `p++`, `*p++`, and `(*p)++`?

7.8
Assume you declared `int p[4] = {1, 2, 3, 4}`. What is `*p`, `*(p+1)`, `p[0]` and `p[1]`?
Section 7.5 Using const with Pointers

7.9
What is wrong in the following code:

```cpp
int x;
int * const p = &x;
int y;
p = &y;
```

7.10
What is wrong in the following code:

```cpp
int x;
const int * p = &x;
int y;
p = &y;
*p = 5;
```

Section 7.6 Returning Pointers from Functions

7.11
Can you guarantee that p[0] displays 1 and p[1] displays 2 in the following main function?

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

int * f()
{
    int list[] = {1, 2, 3, 4};
    return list;
}

int main()
{
    int *p = f();
    cout << p[0] << endl;
    cout << p[1] << endl;
    return 0;
}
```

Section 7.7 Dynamic Memory Allocation

7.12
How do you create the memory space for a double value? How do you access this double value? How do you release this memory?

7.13
Is the dynamic memory destroyed when the program exits?

7.14
Explain memory leak.
Section 7.9 Characters and Strings

7.15 Which function do you use to test whether a character is a digit? a letter, a lowercase letter? a uppercase letter? or a digit or a letter?

7.16 Which function do you use to convert a letter to lowercase or to uppercase?

7.17 What is the last character in a C string?

7.18 How do you read a string from the keyboard?

7.19 How do you find the length of a string? How do you copy a string? How do you combine two strings? How do you compare two strings? How do you convert a string to an int value? How do you convert a string to a double value? How do you convert an int value to a string?

7.20 What is the printout of the following statements?

```c
char * const pCity = "Dallas">
    cout << pCity << endl;
    cout << *pCity << endl;
    cout << *(pCity + 1) << endl;
    cout << *(pCity + 2) << endl;
    cout << *(pCity + 3) << endl;
```

7.21 What is wrong in the following code?

```c
char *pCity;
    cout << "Enter a city: ";
    cin >> pCity; // read to pointer pCity
    cout << "You entered " << pCity << endl;
```

Programming Exercises
Sections 7.2-7.8
7.1 (Analyzing input) Use pointers to write a program that reads ten numbers, computes their average, and finds out how many numbers are above the average.

7.2**
(Printing distinct numbers) Use pointers to write a program that reads in ten numbers and displays distinct numbers (i.e., if a number appears multiple times, it is displayed only once). Hint: Read a number and store it to an array if it is new. If the number is already in the array, discard it. After the input, the array contains the distinct numbers.

7.3* (Increasing array size) Once an array is created, its size is fixed. Occasionally, you need to add more values to an array, but the array is full. In this case, you may create a new larger array to replace the existing array. Write a function with the following header:
*doubleCapacity(int *list)*

The function returns a new array that doubles the size of the parameter list.

7.4 (Averaging an array) Write two overloaded functions that return the average of an array with the following headers:
int average(int * array);
double average(double * array);

Use {1, 2, 3, 4, 5, 6} and {6.0, 4.4, 1.9, 2.9, 3.4, 3.5} to test the functions.

7.5 (Finding the smallest element) Use pointers to write a function that finds the smallest element in an array of integers. Use {1, 2, 4, 5, 10, 100, 2, -22} to test the function.

7.6* (Sorting) Implement a sort function that returns a new sorted array. The function header is
int * sort(const int * const array);

Sections 7.9 Characters and Strings

7.7* (Checking palindrome) Revise Listing 7.14 to ignore case.

7.8** (Checking substrings) Write a program that reads two strings and check whether the one string is a substring of the other. You need to write the following function to check whether string s1 is a substring of string s2.
bool isSubstring(const char *s1, const char *s2)
7.9*  
(Occurrences of a specified character) Write a function that finds the number of occurrences of a specified character in the string using the following header:

\[
\text{int count(const char * const str, char a)}
\]

For example, \( \text{count("Welcome", 'e')} \) returns 2.

7.10**  
(Occurrences of each digit in a string) Write a function that counts the occurrence of each digit in a string using the following header:

\[
\text{int * count(const char * const s)}
\]

The function counts how many times a digit appears in the string. The return value is an array of ten elements, each of which holds the count for a digit. For example, after executing \( \text{int counts[]} = \text{count("12203AB3")}, \) counts[0] is 1, counts[1] is 1, counts[2] is 2, counts[3] is 2.

Write a main function to display the count for "SSN is 343 32 4545 and ID is 434 34 4323".

Redesign the function to pass the count array in a parameter as follows:

\[
\text{void count(const char * const s, int * counts, int size)}
\]

where size is the size of the counts array. In this case, it is 10.

7.11*  
(Counting the letters in a string) Write a function that counts the number of letters in the string using the following header:

\[
\text{int countLetters(const char * const s)}
\]

Write a main function to invoke \( \text{countLetters("C++ and Java in 2008")} \) and display its return value.

7.12*  
(Counting occurrence of each letter in a string) Write a function that counts the occurrence of each letter in the string using the following header:

\[
\text{int * count(const char * const s)}
\]

This function returns the counts as an array of 26 elements. For example, after invoking

\[
\text{int counts[]} = \text{count("ABcaB")},
\]

counts[0] is 2, counts[1] is 2, and counts[2] is 1.
Write a main function to invoke

```c
count("ABcaBaddeekjdefdeg,TTeew44Tt")
```

and display the counts.

Redesign the function to pass the count array in a parameter as follows:

```c
void count(const char * const s, int * counts, int size)
```

where size is the size of the counts array. In this case, it is 26.

7.13*

*(Hex to decimal)* Write a function that parses a hex number as a string into a decimal integer. The function header is as follows:

```c
int parseHex(const char * const hexString)
```

For example, `hexString` A5 is 165 ($10 \times 16^1 + 5 = 165$) and FAA is 4129 ($15 \times 16^2 + 10 \times 16 + 10 = 4129$). So, `parseHex("A5")` returns 165 and `parseHex("FAA")` returns 4129. Use hex strings ABC and 10A to test the function.

7.14*

*(Binary to decimal)* Write a function that parses a binary number as a string into a decimal integer. The function header is as follows:

```c
public static int parseBinary(String binaryString)
```

For example, `binaryString` 10001 is 17 ($1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 = 17$). So, `parseBinary("10001")` returns 18. Use binary string 11111111 to test the function. Note that `Integer.parseInt("10001", 2)` parses a binary string to a decimal value. Do not use this function in this exercise.

7.15**

*(Decimal to hex)* Write two overloaded functions that parse a decimal number into a hex number as a string. The function headers are as follows:

```c
char * convertDecimalToHex(int value)
void convertDecimalToHex(int value, char * s, int size)
```

See §1.5, “Number Systems,” for converting a decimal into a hex. Use decimal 298 and 9123 to test the function.

7.16**

*(Decimal to binary)* Write two overloaded functions that parse a decimal number into a binary number as a string. The function headers are as follows:
char * convertDecimalToBinary(int value)
void convertDecimalToBinary(int value, char * s, int size)

See §1.5, “Number Systems,” for converting a decimal into a binary. Use decimal 298 and 9123 to test the function.

7.17**
(Sorting characters in a string) Write two overloaded functions that return a sorted string using the following header:

char * sort(const char * const s);
void sort(const char * const s, char * s1, int size);

For example, sort("acb") returns abc.

7.18**
(Anagrams) Write a function that checks whether two words are anagrams. Two words are anagrams if they contain the same letters in any order. For example, “silent” and “listen” are anagrams. The header of the function is as follows:

bool isAnagram(const char * const s1, const char * const s2);

Write a main function to invoke isAnagram("silent", "listen"), isAnagram("garden", "ranged"), and isAnagram("split", "lisp").