7.Pointers and Memory Management

One of the powerful characteristics of C++ language is the ability the C++ programmer gets to manipulate memory locations. The **pointer feature of the language allows the programmer to access memory cells directly through their addresses instead of through variable names**. Using pointers increases efficiency of the programs through faster operation, and facilitates direct access to computer hardware and peripherals. This chapter introduces the pointers and describes its uses in programming.

We shall begin our discussion of pointers with a thorough description of how memory is allocated by the compiler to variable names and how the addresses of allocated memory can be found.

7.1. Anatomy of Memory and "Address of" Operator &

Recall that a byte of memory is the amount of memory needed to store one character data. One byte is equal to 8 bits or 8 circuits. The entire memory is made of millions of bytes. For the purpose of conveniently managing the memory, each byte of the memory is given an address. These addresses are contiguous numbers. Though the actual memory addresses are hexa-decimal coded numbers, for simplicity, we can use the decimal equivalent of them to identify the individual memory cells. So, **each byte of memory has a number that identifies it**. This is analogous to each mail box on a street having an address written on it for the mail-man to identify and put each mail in the right mail box. Figure 7.1.1 shows a set of memory bytes that begin with the address 1000. Here we use 1000 as the beginning address for illustrative purposes. Actual addresses may are different and they are coded in hexa-decimal coding system. The numbered boxes shown in the figure represent bytes of memory. For the purposes of our discussion, we shall assume this to be the memory of the computer system that we are working with.

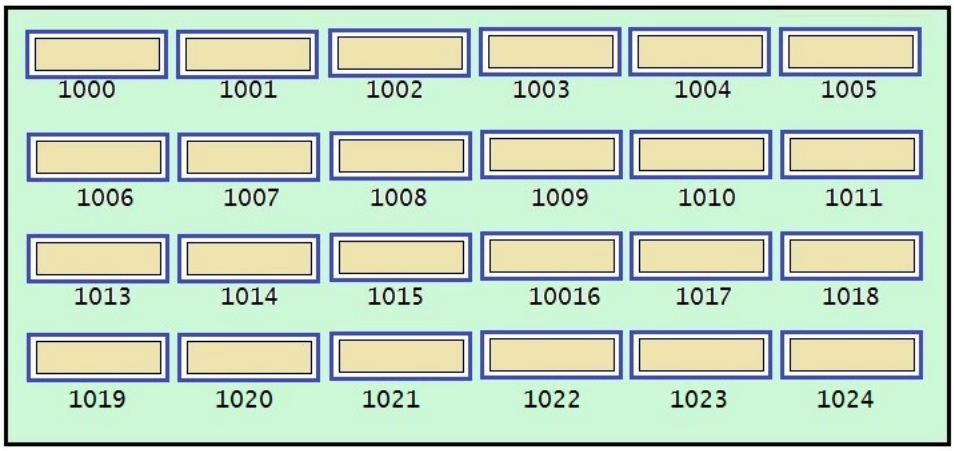


Figure 7.1.1. Memory bytes and their addresses

In the programs we have been writing, we never accessed memory bytes using their addresses. We instead used variable names to identify specific memory cells and wrote statements involving these variables to manipulate the memory cells in the program. When the compiler compiles such statements, however, it replaces the variable names with addresses of memory bytes that they identify. For example, suppose that we write the statement

int num1 = 21;

in a program. By writing this, we are requesting the system to store 21 in a memory cell and to name it as *num1*. When this statement is compiled, the compiler finds the next available set of four consecutive memory bytes (assuming 4 bytes are allocated for *int* type), **associates the name *num1* to the address of the first of these four**, and stores the (binary equivalent of the) value 21 in these four bytes. We use the term "memory cell" to refer to this block of four bytes. After the above statement is executed, the memory will look like what is shown in Figure 7.1.2. Note that when storing 21 in the memory, the binary code of it, namely 10101 gets padded with 0s at the front to make it 32 bits (4 bytes) long and then stored.

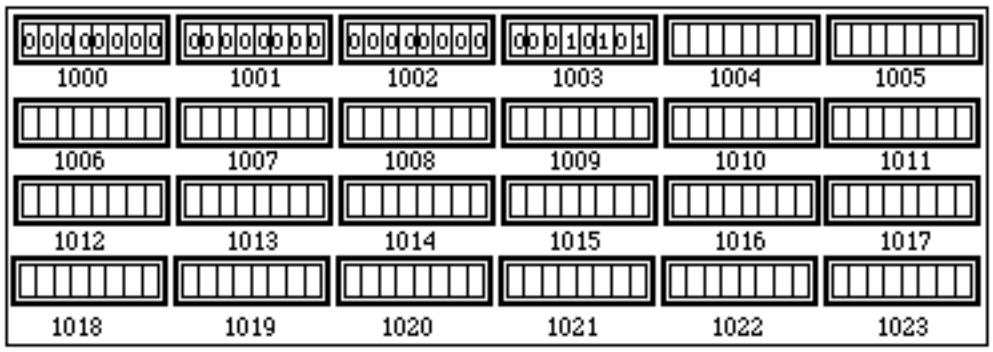


Figure 7.1.2. Memory with the value 21 stored in it.

Suppose that in a program after the above discussed statement

int num1 = 21;

we have another assignment statement

int num2 = 635;.

When this statement is executed, the next four bytes that begin at the address 1004 get allocated to the variable num2. Then the binary code of the number 635, namely 1001111011, gets padded with zeros at the front to make it 32 bits long and stored in these four bytes. The memory will then look like what is shown in Figure 7.1.3.

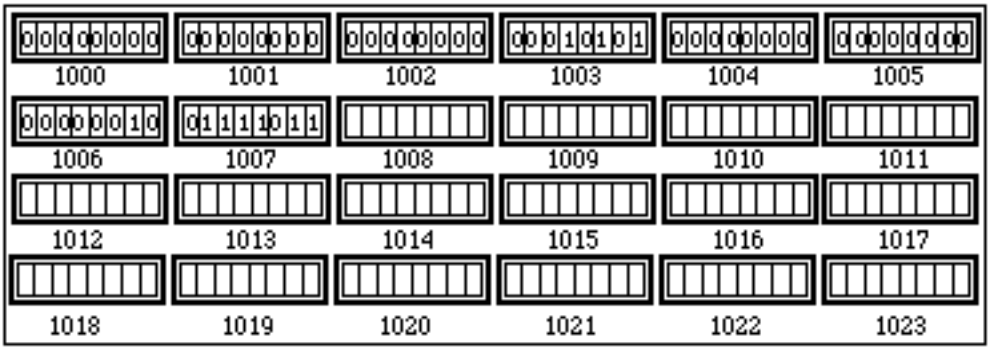


Figure 7.1.3. Memory with the values 21 and 635 stored in it.

Other assignment statements written in the program use up the available memory in the same manner.

The "address of" Operator &

There are times when a programmer needs to find the address of the memory cell allocated to a specific variable used in the program. For this purpose, the C++ language offers an operator called "address of" operator, identified by the symbol &.

> The **& symbol**, when **adjoined to the front of a variable name, becomes a name that has the address** of the memory cell allocated to the specific variable.

For example, in a program, after the statement *int num1 = 21, num2 = 635;* is executed the memory will look like in Figure 7.3. In the same program if we use the name

&num1

it **will have the value 1000** which is the address of the memory cell allocated to num1. Similarly, if we use the name

&num2

in the same program, it **will have the value 1004**. The programmer can have these memory addresses printed on the screen, if necessary, by using the *cout* statement as described in the following example.

Example 7.1.1. Assuming a computer memory as shown in Figure 7.1.1, the following statements are valid, and they produce the indicated output.

```
int  num1 = 21, num2 = 635;
cout << "The value of num2 is: " << num2 << endl;
cout << "The address of memory allocated to num2 is: " <<&num2;
```

Output:

```
The value of num2 is: 635
The address of the memory allocated to num2 is: 1004
```

The *-data type

Even though the memory addresses are actually integers, they are considered to be special integers because they identify memory cells. Therefore, **memory addresses are treated as another data type** in C++. This data type is **identified by the name ***. We read it as "star-type". That is,

all memory addresses are *-type values.

Exercises 7.1

Assuming a standard computer system (*float* and *int* types get 4 bytes of memory, *short* type gets 2 bytes of memory, and *long* type gets 8 byte memory) with the memory structure as shown in Figure 7.1.1 determine the output of the following statements when they are executed as part of complete programs.

1.
```
int  n1=32, n2=7;
cout << &n2;
```

2.
```
char val1= 'm';
int val2 = 13;
cout << &val2;
```

3.
```
long m1 = 278, m2 = 356;
char c1 = 's';
cout << &c1;
```

4.
```
short  x1=21, x2=14, x2=89;
float y1 = 3.452;
cout << &y1;
```

5.
```
float  num1=7.83;
short val = 41;
cout << &val;
```

6.
```
float  s1=3.2;
short x1=3, x2=5;
int x3=2456;
cout << &x1 << endl
     << &x3;
```

7. Write C++ statements to store the values 3.4 and 2.3 in two double type memory cells and to get the addresses of these values printed.

8. Write C++ statements to store the value 487 in a short type memory cell, and to store the value 7.34 in a float type memory cell and to print the addresses of these values.

7.2. Introduction to Pointers

Recall from the previous section that the addresses of memory bytes are *-type values. Also recall that the address of the memory cell allocated to a variable can be found by adding the & ("address of") operator at the front of the variable name.

We can have variables that hold *-type values (that is, the memory addresses). Such variables are called pointers. More precisely,

> **A pointer is a variable that can hold a memory address as its value**. By holding the address of a memory the **pointer is said to be pointing to the value** in that memory.

Declaring a Pointer

To declare a pointer in a program, we first need to determine its type. ***The type of a pointer is the type of value it will be pointing to***. For example, if a pointer to be declared will be later set to point to a float type value then the type of the pointer is float. Once the type of the pointer has been determined, the **pointer can be declared by writing the type identifier followed by a * followed by the name of the pointer** as shown below.

Pointer declaration:

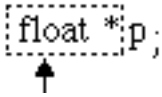
type-identifier **pointer-name;*

The following two examples illustrate pointer declaration.

Example 7.2.1. A pointer named p of float type is declared by writing

float *p;

Note that in the pointer declaration statement, **there is a space between the type identifier and the ***, but there is **no space between the * and the pointer-name**. Though separated by a space in the declaration statement, the *type identifier and the * together identify the type of the pointer*. For example, the statement float *p has the meaning as described below.



type identifier that identifies
p as a float * - type name.
That is, a float type pointer.

Example 7.2.2. (a) To declare the name ***ester*** **as a pointer that can point to an *int* type** value, we write

```
int  *ester;
```

(b) To declare the names ***p*** **and *s* as a pointers that can point *char* type** values, we write

```
char  *p, *s;
```

When a pointer variable is declared, it gets a memory cell allocated (just as any other variable) to hold the address of another memory.

> The amount of **memory allocated to a pointer is the same as that allocated to a long int type**. Number of byte allocated to long int depends on the computer system. We shall assume, for simplicity, that **4 bytes are allocated for pointers**.

The memory allocated to a pointer does not get initialized with anything just the same way any variable does not get initialized automatically. That is, the **pointer is not automatically set to point to anything**. **It can be set to point to any value (of matching type) by storing the address of that value**. However, **assigning an integer such as 1000 to a pointer, as shown in the statements below, is invalid**, even if there is a memory cell with that address.

```
int *ptr;
ptr = 1000;  //Invalid. An integer cannot be assigned to a pointer.
```

> To initialize a pointer, you must **first declare a variable of type that matches the type of the pointer** (which allocates a memory for that variable name). **Then find the address of its memory and assign it to the pointer**.

To find the address of a variable, we have a tool, namely the & ("address of") operator. The following example declares and sets a pointers to point to a specific values.

Example 7.2.3. The following statements store the integers 21 and 635 in the memory and set two pointers to point to them.

```
int  num1 = 21, num2 = 635;
int  *s1, *s2;
s1 = &num1;     //Setting s1 to point to 21.
s2 = &num2;     //Setting s2 to point to 635.
```

The declaration of pointers and the statements that set them to point to values found in the above example can be combined and written as follows.

```
int  num1 = 21, num2 = 635;
int *s1 = &num1, *s2 = &num2;
```

Note that each byte of the memory has its own address. A memory cell (that is, the memory allocated to a variable) consists of, in general, more than one byte. For example, the memory cell allocated to the variable num1 in the above example consists of 4 bytes (in the standard computer system). Then address of which byte of the cell is used as the address of the cell? It is always the address of the first one of all the allocated cells. That is,

address of a memory cell is the address of the first byte of it.

Assume a computer system with memory as shown in Figure 7.1.1, and assume that 4 bytes are allocated to pointer variables. After executing the statements of Example 3 in this computer system, the memory will be as shown in Figure 7.2.1. Note that the memory addresses of num1 (which is 1000) and num2 (which is 1004) are stored in binary form in s1 and s2 respectively. Figure 7.2.2 shows the same memory, but uses decimal forms of the numbers for easier reading. This figure also shows how the cells are divided and what their names are.

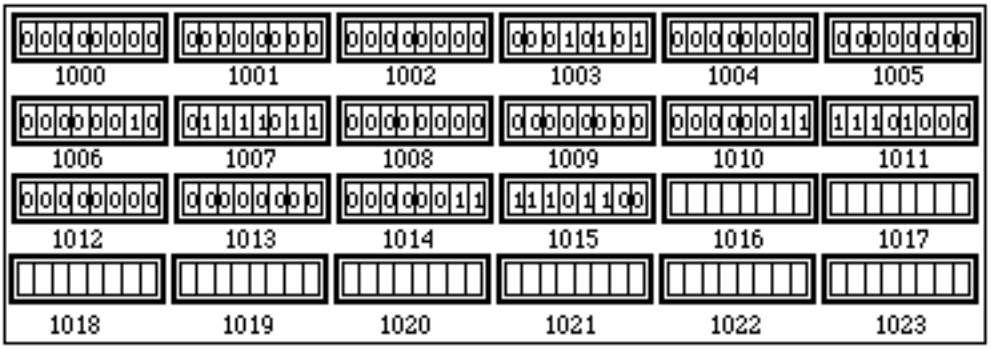


Figure 7.2.1.

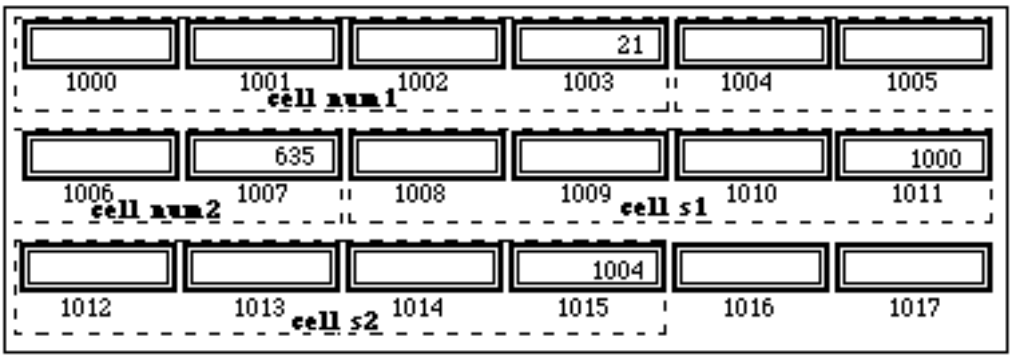


Figure 7.2.2

Example 7.2.4. C++ statements to store the character 'k' in a memory cell and to have a pointer point to it are:

```
char  val = 'k';
char *ptr = &val;
```

The **& ("address of") operator can be used with a pointer** to determine the address of the memory cell allocated to the pointer. The following example illustrates this.

Example 7.2.5. The following statements are the same as in Example 3, except for the last two statements which find the addresses of the pointers *s1* and *s2*. As we noted in Figure 7.2.2, the address of *s1* would be 1008 and the address of *s2* would be 1012, and therefore, these are the numbers that we get as output.

```
int  num1 = 21, num2 = 635;
int  *s1, *s2;
s1 = &num1;     //Setting s1 to point to 21.
s2 = &num2;     //Setting s2 to point to 635.
cout << s1 << ", " << s2 << endl;
cout << &s1 << ", " << &s2;
```

Output:

```
1000, 1004
1008, 1012
```

Exercises 7.2.

Assuming a standard computer system, and assuming that the memory of the computer system is as shown in Figure 7.1.1, **determine the outputs of the following statements** when they are executed as part of complete programs.

1.
```
float  k = 5.32, m=3.5, *p;
p = &k;
cout << p << endl << &p;
```

2.
```
long k=345;
short m = 5, n=12, *s;
s = &n;
cout << s << endl << &s;
```

3. Write C++ statements to store the value 324 in the memory and to have a pointer point to it.

4. Write C++ statements to store the characters 'r' and '$' in two memory cells and to have two pointers pointing to them.

5. Write C++ statements to store the numbers 5.4 and 13 in the memory, to set two pointers pointing to them and to determine the addresses of the pointers.

6. Write C++ statements to store the two characters M and P in the memory, to set two pointers pointing to them and then to print the addresses of these pointers.

7.3 De referencing a Pointer

De referencing a pointer means retrieving the value that a pointer is pointing to. We shall see in this section how to de-reference a pointer.

Recall that "*" (star) is the name of the data type of memory addresses. Therefore, * is used in the pointer declaration statements to identify pointers as variables that will hold * type values. The **same * is also an operator that de-references pointers**.

> If p is a pointer that is pointing to a value, **then *p refers to the value being pointed to by p**.

The expression ***p is interpreted as "find the address stored in p, and get the value stored at that address"**. The following example illustrates de referencing a pointer.

Example 7.3.1. The following statements are valid, and they produce the indicated output.

```
float  x = 3.5, *p;
p = &x;              // p is set to point to 3.5
cout << p << endl;  // prints the address where 3.5 is
cout << *p;          // prints 3.5  (de-references p)
```

The expression *p can appear on the left side of an assignment. That is, **a value or an expression can be assigned to *p**.

> **Assigning a value to *p means storing the value in the memory that p is pointing to**. Pointer p must be already pointing to a memory for such an assignment to be valid.

For example, a statement such as ***p = 3.5;** is valid **when p is a pointer pointing to some memory**, and it stores the value 3.5 in the memory that p is pointing to. The programmer must remember to set the pointer p to point to a memory cell before assigning a value to *p.

Example 7.3.2. The following statements illustrate how the value of a variable can be changed through a pointer.

```
float val = 7.2, *p;
p = &val;              //p is now pointing to 7.2.
cout << p << endl;     //Printing the address of 7.2.
cout << *p << endl;    //Printing 7.2.
*p = 5.6;              //Changing the value of val from 7.2 to 5.6.
cout << *p << endl;    //Printing 5.6.
```

Example 7.3.3. The following statements illustrate how arithmetic expressions can involve de-referenced pointers.

```
short num1, num2, ans, *s1, *s2;
s1 = &num1;            //s1 is now pointing to the memory of num1.
s2 = &num2;            //s2 is now pointing to the memory of num2.
*s1 = 23;              //Same as writing num1 = 23;
*s2 = 12;              //Same as writing num2 = 12;
ans = *s1**s2;   //Same as writing ans = num1*num2;
cout << ans;           //Prints 276.
```

Exercises 7.3

1. Write the exact output that you expect to see on the screen when the following statements are executed as part of complete programs.

(**a**)
```
float b1 = 39.6;
float  *b2;
b2 = &b1;
*b2 =12.5;
cout << b1;
```

(**b**)
```
int num = 10;
int *nptr = &num;
num = num + 2;
cout << *nptr;
```

(**c**)
```
double val1 = 5.3;
double *ptr ;
ptr = &val1;
*ptr = *ptr + 3.1;
cout << val1;
```

2. Assume a computer system that has a memory as shown in Figure 7.1.1, that allocates 4 bytes for float and int types, and 2 bytes for short type. Consider the statements

```
float  a = 6.5, b = 7.2, *ptr1, *ptr2;
short  c = 5, *ptr3;
ptr1 = &a;
ptr2 = &b;
ptr3 = &c;
```

If the following statements, along with the ones above, are executed as part of complete programs, what will be printed? In any of the problems, if you think an error message will be produced, explain why.

(a)
```
cout << *ptr1
```

(b)
```
float val = *ptr1 + *ptr2;
cout << val;
```

(c)
```
*ptr3 = a + b;
cout << c;
```

(d) *ptr1 += 3;
*ptr2 += *ptr1;
cout << a << endl << b;

(e) c = c + *ptr1;
cout << *ptr3;

3. Write statements to store the numbers 12, 4.3, and 48345 in the memory indirectly through pointers instead of assigning them to variables directly.

4. Write statements to store the values 'G' and 13.7 in the memory indirectly through pointers instead of assigning them to variables directly.

5. Two float type pointers named p1 and p2 are pointing to two float type memory cells that already have some values stored. Write C++ statements to print these values in increasing order on the screen.

6. Three float type pointers named ptr1 and ptr2 are pointing to three float type memory cells that already have some values stored. Write C++ statements to compute and print the average of the values found in those memory cells.

7.4. Pointer Arithmetic

Recall that pointers hold addresses of memory cells. Even though these addresses are integers, they are considered to be values of a new data type called *-type. **Limited arithmetic can be performed with the memory addresses, and therefore, with pointers as well**.

Since the result of adding, multiplying, dividing or performing other similar operations with two memory addresses do not result in anything meaningful, such operations are not permitted.

Example 7.4.1. When p1, p2, and p3 are pointers, the statements such as the following are **invalid**.

```
p3 = p1+p2;
p3 = p1*p2;
p1 = 2*p1;
```

There are **only two arithmetic operations that we are allowed to perform with pointers**:

1. **Subtracting** a pointer from another
2. **Adding an integer** to a pointer or **subtracting** one from it.

We shall explain next, what the above two operations result in.

1. Pointer Subtraction

If p1 and p2 are pointers of the same type, and

> if p2 has larger value than p1, the expression **p2-p1 produces how many memory cells** (not how many bytes) **apart the cells being pointed to by p2 and p1 are**. The expression p1- p2 is the negative of p2 - p1.

The expression *p1 - p2 or p2 - p1 means subtracting a *-type value from another,* and, as indicated above, such subtraction produces the count of the memory cells in between. Subtracting the values of two pointers as integers, on the other hand, produces the count of the bytes in between the two addresses. In other words, if we subtract the values of two pointers after converting them to integers by using the cast operator (int), we will get the number of bytes in between the two memory addresses.

> **Number of bytes between the memories** being pointed to by pointers p1 and p2 **is given by**
>
> (int) p2 - (int) p1

Example 7.4.2. Consider the following program-segment.

```
short  n1 = 5, n2 = 12, n3 = 23,  *p1, *p2, *p3, j, k, m;
p1 = &n1;
p2 = &n2;
p3 = &n3;
j = p2 - p1;   //j gets a 1 because the memory cell of n2 is 1 cell (of 4 byte
               // long) away from the memory cell of n1.
k = p3 - p1;  //k gets 2.
m = (int)p3 - (int)p1;    //m gets 4.
```

Assuming that *short* type variables get 2 bytes of memory allocated in our computer system whose memory is as shown in Figure 7.1.1, the following Figure 7.4.1 shows how the memory will look after the statements in the above example are executed. Here, we have shown the values stored in the memory cells in their decimal form rather than in their binary form.

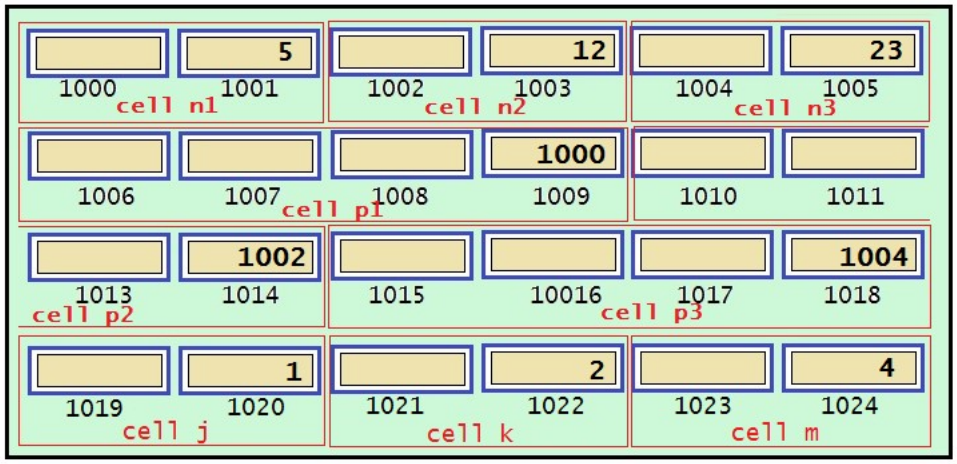


Figure 7.4.1

Note that, in the above example, the values of j and k will be 1 and 2 respectively even if we change the keyword *short* in the variable declaration to *int, float*, *long int* or anything else. This is because the pointer subtraction produces the number of cells in between them, not the number of bytes in between them, and this number does not depend on how big the cells are.

Pointer subtraction is valid only when the two pointers are of matching type. The following example illustrates this.

Example 7.4.3. The pointer subtraction in the following statements in invalid.

```
int  j, num, *p;
float  val, *s;
p = &num;
s = &val;
j = p - s;          //Invalid. Pointer type mis-match.
```

2. Adding or subtracting an integer from a pointer

If **p is a pointer** of certain type pointing to some memory of the computer system and if **n is an integer**, then

> **p + n is the address of the memory cell that is n cells away** from what p is pointing to.

Example 7.4.4. Assume that a *short* type variable gets two bytes of memory in our computer system. Suppose that **p is a pointer** of *short* type, and suppose that it **has 1000 in it** (that is, it is pointing to the memory cell whose address is 1000). **Then *p+1 will be 1002***, the address of the next short type memory cell. Similarly, ***p+2 will be 1004***, the

address of the short type cell that is 2 cells away from where p is pointing to. In other words, p+1 is pointing to the short type cell next to the one p is pointing to, and p+2 is pointing to the short type cell next to the one p+1 is pointing to and so on.

Note that the ***above described pointer arithmetic offers a way of accessing the entire memory using a single pointer***. That is, we can set a pointer *p* to point to one of the memory cells in the computer system and then add 1 to p to get to the next two byte memory cell, de-reference p+1 to retrieve the value stored there. Add 2 to p to get to the next two byte cell, and so on. The following example illustrates accessing many memory cells through one pointer.

Example 7.4.5. Assuming two byte memory allocation for *short* type, the following statements access three two byte memory cells using a pointer and use them to compute the sum of 23 and 45.

```
short  val, *p;
p = &val;                 //Setting the pointer p to point to a memory.
*p = 23;                  //Storing 23 in a two byte cell.
*(p+1) = 45;              //Storing 45 in the next two byte cell.
*(p+2) = *p+*(p+1);       //Storing the sum of 23 and 45 in the next cell.
cout << *(p+2);
```

When a pointer p is pointing to a memory cell of the computer system, in order to access contiguous memory cells that follow the one being pointed to by p, we can either add 1, 2, 3, … to p and de-reference the result as shown in the above example, or simply **increase the p by 1** repeatedly **by writing p++**, and de-reference it each time. Increasing p by 1 changes the address stored in it to make it point to the next memory cell of the pointer's type (not the next byte). The following example illustrates this.

Example 7.4.6. The following program stores 20 integers obtained from the user in 20 consecutive memory cells of the computer system without using variable names. It then finds the largest of these integers and prints it.

```
main()
{
int   i, max,  *p, val;  //Fourth 4-byte cell is allocated to val.
p = &val;
for (i = 0; i < 20; i++)
    {
    cout <<"Enter the number:";
    cin >> *p             //Storing the number in the cell that p
    p++;                  //is pointing to.
    }
p = &val;
max = *p;
```

```
for (i = 0; i <20; i++)
    {
    p++;
    if (*p > max)
          max = *p;
    }
cout << "Largest number is:" << max;
return 0;
}
```

Note that the program in Example 6 accesses 20 cells of 4 byte size starting at the fourth available cell and uses them to achieve the task. The programmer must ensure that these memory cells have not been allocated to any other variable in the program. The following example illustrates a potential danger with the above approach of accessing memory without reserving them.

Example 7.4.7. In the following program, we describe, in the comments, how an unintended result may be produced by carelessly manipulating memory by using pointers.

```
main()
{
int  i, max,  *p, val;   //Fourth 4-byte cell is allocated to val.
p = &val;
int  pro = 35;                  //Fifth cell is allocated to pro.
for (i = 0; i < 20; i++)
    {
    cout <<"Enter the number:";
    cin >> *p              //These statements change the fifth memory cell
    p++;
    }
cout << pro;               //Doesn't print 35.  One of the values
                           // entered by the user has replaced it
```

.

In Section 7.5 we will see how to avoid this type of problem and still use pointers to manipulate the memory.

Exercises 7.4.

1. Assume a computer system that has a memory as shown in Figure 7.1.1 that allocates 4 bytes for *float* and *int* types, and 2 bytes for *short* type. Consider the statements

```
float  a = 9.8, b = 7.2, *ptr1, *ptr2;
short  c = 13, *ptr3;
ptr1 = &a;
ptr2 = &b;
ptr3 = &c;
```

If the following statements, along with the ones above, are executed as part of complete programs, what will be printed? In any of the problems, if you think an error message will be produced, explain why.

(a)
```
int  j;
j = ptr2 - ptr1;
cout << j;
```

(b)
```
ptr1++;
cout << *ptr1;
```

(c)
```
ptr3 = ptr1 + ptr2;
cout << ptr3;
```

(d) `cout << ptr1+1;`

(e) `cout <<ptr3 - ptr1;`

(f) `cout << (int) ptr2 - (int) ptr1;`

2. If the following statements, **along with the five statements given in Problem #1**, are executed as part of complete programs, what will be printed? In any of the problems, if you think an error message will be produced, explain why.

(a)
```
int  j;
j = ptr3 - ptr1;
cout << j;
```

(b)
```
short j;
j = ptr2 - ptr1;
cout << j;
```

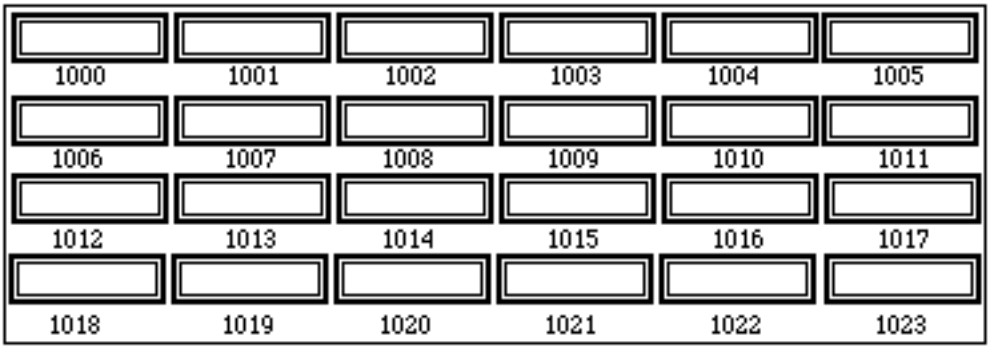
(c)
```
ptr2 = 3*ptr1;
cout << ptr2;
```

(d) `cout << ptr2 - 1;`

(e) `cout <<*(ptr2-1);`

(f) `cout <<(int)ptr3 - (int)ptr1;`

3. Assume that the memory of the computer system is as shown below.



Also consider the statements

```
short n1=12, n2 = 3, *s1, *s2, *s3;
```

```
s1 = &n1;
```

If the following statements are executed as part of complete programs, **show exactly what the memory will look like**. You may show the numbers in decimal form rather than in binary form.

(a)
```
*(s1+1) = 25;
```

(b)
```
s2 = &n2+12;
*s2 = 14;
```

4. In Problem #3, if the following statements are executed as part of complete programs, **show exactly what the memory will look like**. You may show the numbers in decimal form rather than in binary form.

(a)
```
s2 = &n2;
*(s2-1) = 19;
```

(c)
```
s3 = s1 + 12;
*s3 = *s1 + 5;
```

5. Write C++ statements to store the number 45 in a short type memory cell, and to store the number 23 five two byte cells after where 45 is stored.

6. Assume that the memory of the computer system is as shown in Problem #3. Write statements to store the numbers 9, 34, and 15 in first, fourth and seventh bytes.

7. Assume that the memory of the computer system is as shown in Problem #3, and assume that each byte of the memory has been used as a char type cell and stored some value starting from the very first memory cell whose address is 1000. The name of the very first of these cells is *cval1*. Write C++ statements to increase every other cell by 5.

8. Write a program ***without using more than one variable*** and a pointer variable, to repeatedly obtain float type numbers until -1 is entered, to store them in contiguous memory cells, and to compute their average and print it.

7.5. Arrays and Pointers

We have seen earlier that after declaring an array by using a declaration statement such as

```
int num[5];,
```

individual elements of the array can be accessed by using the array name followed by a subscript value in brackets. For example, the statement

```
num[3] = 9;
```

stores 9 in the third element of the array num.

An **alternative way of accessing individual entries of an array is through pointers**. We have seen in the last section that if a pointer is pointing to one of the memory cells, then by adding 1, 2, 3,… to that pointer we get the addresses of the

contiguous memory cells that follow the one that the pointer is pointing to. Since the elements of an array occupy contiguous memory cells, ***if we set a pointer p to point to the first element of the array, then by adding i to the pointer, and by de referencing the resulting pointer we can access the array element whose subscript value is i*** (that is, the i+1 st element). For example, the statements

```
int  num[5];
p = &num[0];
*p = 15;
*(p+2) = 9;
```

store 15 in the first element and 9 in the third element of the array num. That is, we have the following fact.

If p = &num[0], then **num[i] is the same as *(p+i)**.

This relationship is unique in C++ and is one of the most important features as well. In fact, we do not even need to declare a pointer and set it to point to the first entry of the array in order to be able to access array elements using pointers. C++ language provides one automatically. When an array declaration is compiled, the C++ compiler allocates enough memory for the array and initializes a pointer whose name is the same as the name of the array and sets it to point to the first of the allocated memory cells. In other words,

after an array is declared, **the name of the array automatically becomes a constant pointer pointing to the first element** of the

This means,

after num, for example, is declared as an array, **num[i] is the same as *(num+i)**.

In a program, after declaring num as an array, to access the i th element of num, we can use array notation num[i] or pointer notation *(num+i). They are equivalent. The compiler actually adds i to the address stored in num to get the address of num[i] whenever it compiles a statement involving num[i]. The pointer counter part of the above statement is also true. That is, **after declaring num as a pointer and setting it to point to a memory cell, we can access the i th cell from this cell by using either num[i] or *(num+i)**. The following example illustrates.

Example 7.5.1. The following program stores the numbers 93, 67, 81, 45, 78, 96, 87 in an array and uses the pointer approach to compute and print the sum of them.

```
short  ndata[7] = {93, 67, 81, 45, 78, 96, 87}, i, sum = 0;
for (i = 0; i < 7; i++)
        sum += *(ndata + i);
cout << sum;
```

Group assignment method that we use with arrays can also be used with pointers as in

```
short  *ndata = {93, 67, 81, 45, 78, 96, 87};.
```

When such a statement is executed, the numbers will be stored in memory cells and the pointer will be set to point to the first one of those cells.

The tasks of the Example 1 can be achieved by declaring ndata as a pointer, and by writing the program as shown in Example 2.

Example 7.5.2. The following program does the same as what the program in Example 1 does.

```
short  *ndata = {93, 67, 81, 45, 78, 96, 87}, i, sum = 0;
for (i = 0; i < 7; i++)
    sum += *(ndata + i);    //This can also be sum +=ndata[i];
cout << sum;
```

Since the *pointer used in the above Example 2 is a variable pointer*, to access all the memory cells where the numbers are stored, instead of adding different values of i to the pointer we could change the value of the pointer itself to make it point to the other memory cells. The following Example 3 illustrates this and it is the third version of the program in Example 1.

Example 7.5.3. The following program does the same as what the program in Example 1 does.

```
short  *ndata = {93, 67, 81, 45, 78, 96, 87}, i, sum = 0;
for (i = 0; i < 7; i++)
    sum += *ndata++;    //* operation is carried out before ++
cout << sum;
```

Note that there is one minor difference between declaring *ndata* as an array and declaring it as a pointer. **When the array approach is used, the name of the array is a constant pointer pointing to the beginning of the array. The address this pointer holds cannot change**. Therefore, we cannot increase this pointer by 1 to make it point to the next cell as we did in Example 3 above.

We can copy an array name pointer into another variable pointer and then change the new one. The following example illustrates this.

Example 7.5.4. The following program stores 20 integers obtained from the user in an array of size 20. It then finds the largest of these integers and prints it.

```
main()
{
int  val, i, max,  num[20], *p;  //num is an array, p is a pointer
p = num;          //Setting p to point to the beginning of num
for (i = 0; i < 20; i++)
          {
          cout <<"Enter the number:";
          cin >> *p++   //Storing the number in the cell that p is
          }                    //pointing to and then increasing p
 p = num;       //Resetting p to point to the beginning of array
 max = *p;
```

```
for (i = 1; i <20; i++)
          {
          if (*(++p) > max)
               max = *p;
          }
cout << "Largest number is:" << max;
return 0;
}
```

The program in the last example is the same as the one in Example 4 of Section 7.4 except that we are using an array here. **The only advantage in declaring an array instead of pointer is that we get the desired number of memory cells reserved**. An assignment statement such as *int pro = 35;* found in the Example 5 of Section 7.4 that follows the array declaration does not take up any of the memory cell allocated for the array.

Passing Arrays to Functions

Recall that in order to pass an array to a function we simply pass just the name of the array to the function. Passing the name is sufficient because the name is a pointer that has the address of the first element of the array. By receiving the value stored in the name of the array, the function is receiving the location of the first array element in the memory, and therefore by adding 1, 2, 3, … to it the function can access all other

elements of the array. Since when passing an array to a function what is really passed is the address of the beginning of the array,

> In the function header, the **parameter in which the array is received can be either an array or a pointer**.

The following example illustrates this by using two versions of a program that are equivalent. Note that the size of the array needs to be passed as a separate value since the function would otherwise have only the address of the beginning element and not the ending element.

Example 7.5.5. The following program uses a function called add-up that adds up the entries of any array it receives, to compute the average of the numbers 93, 67, 81, 45, 78, 96, and 87. We provide two versions of the function: (1) using an array as parameter, and (2) using a pointer.

```
short add_up (short[]);
main()
     {
      short  num[7] = {93, 67, 81, 45, 78, 96, 87};
      float  average;
      average = add_up(num, 7) / 7.0;
      cout << average;
     }

//Version 1 of the function: Using an array as parameter
short  add_up (short ndata[], short size)
     {
      short sum = 0;
      for (i = 0; i < size; i++)
              sum += ndata[i];
      return sum;
     }

//Version 2 of the function: Using pointer as parameter
short  add_up (short *ndata, short size)
     {
      short sum = 0;
      for (i = 0; i < size; i++)
              sum += *(ndata+i);
      return sum;
     }
```

In fact, whether we use the array notation *short ndata[]* or the pointer notation *short *data* as the parameter that receives the array, the two are equivalent for the compiler. The array name *ndata[]* used as the parameter in the Version 1 of the function makes the name *ndata* a pointer that points to the beginning of the array *ndata[]*. Therefore ***regardless of which notation we choose to use, the function is simply receiving the address of the beginning of the array num in the name ndata***. As we noted earlier, accessing the elements of the array *ndata* by using *ndata[i]* and by using**(ndata+i)* are equivalent and compile to be the same code. Therefore, the two versions of the function *add_up* written above are equivalent. One can mix the two notations as well. The following is a third version of the above function *add_up* that mixes the two notations, and yet equivalent to both of the above versions.

```
//Version 3 of the function: Mixing pointer and array notations
short  add_up (short *ndata, short size)     //Using pointer as parameter
     {
      short sum = 0;
      for (i = 0; i < size; i++)
              sum += ndata[i];   // Array notation. Equivalent to *(ndata+i)
      return sum;
     }
```

Exercises 7.5.

1. Assume a standard computer system with a memory as in Figure 7.1.1 that allocates four bytes to int and float type variables, and allocates 2 bytes for short type variables. Assume the following declaration

```
int  i, s, val[6] = {5, 12, 3, 7};
```

Write what the following statements will print.

(a) `cout << val[0];` (b) `cout << *val;` (c) `cout << *(val+2);`

(d)
```
val[4] = *(val+1)+10;
cout << *(val+4);
```

(e)
```
*(val+5) = *(val+3) + *(val+2);
cout << val[5];
```

(f)
```
for (i = 0; i < 4;  )
        s = s + *(val +i++);
  cout << s;
```

(g)
```
for (i = 0; i < 4; i++)
        cout << *(++val);
```

2. Assume a standard computer system with a memory as in Figure 7.1.1 that allocates four bytes to int and float type variables, and allocates 2 bytes for short type variables. Assume the following declaration

```
int  i, s, val[8] = {16, 7, 23, 9, 14};
```

Write what the following statements will print. If your conclusion is that there will be an error message, briefly explain why.

(a) cout << val[6]; (b) cout << *val; (c) cout << *(val+3);

(d) val[5] = *(val+4)+6;
cout << val[5];

(e) *(val+5) = val[0] + *(val+1);
cout << val[5];

(f) for (i = 0; i < 5; i++)
*(val + i) = *(val + i +1);
cout << val[3];

(g) for (i = 0; i < 4; i++)
cout << *(val++) << endl;;

3. Write a function using pointer notation to receive a float type array and its size, and to return the largest number found in that array. Then write a main function that uses this function to find and print the largest of the numbers 23, 41, 15, 34, 9, 31, 38, 27, 12, and 30.

4. Write a function using pointer notation that receives an integer array and another number and returns the count of how many numbers in the array are positive and smaller than the received number.

5. Write a program that uses pointer notation to obtain and store up to 200 integer numbers entered by a user in an array. The user may enter less than 200 numbers but will enter -1 to indicate end of entering data. Your program must then use the function written in Problem #3 to determine how many numbers in the array are positive and smaller than 60 and print this count.

7.6. Dynamic Memory Management

The *new* operator

The way in which we have been setting a pointer to point to a memory is declaring a pointer p of some type, declaring a variable of the same type such as *float val* in order to get a memory allocated, and then storing the address of the variable in the pointer by writing

p = &val;.

This approach works but awkward. The ***new* operator offers a way of allocating a memory and setting a pointer to point to it** without having to declare a variable. With pointers, it is used in the following manner.

```
float  *p;        //Declaring a pointer
p = new float;  //Allocating float type memory and setting p to point to it.
```

After the above statements, to store a value such as 12.6 in the memory we can use the de referencing operator * as in

```
*p = 12.6;
```

Example 7.6.1. The following program computes the average of two numbers 16.7 and 45.5 using only pointers.

```
float  *p, *q;
p = new float;
q = new float;
*p = 16.7;
*q = 45.5;
*p = (*p + *q)/2.0;
cout << *p;
```

An array can be created using the *new* operator and accessed through pointer by writing, for example,

```
float  *ptr;
ptr = new float[200];
```

The *delete* operator

One of the negative aspect of the classical approach of programming is inefficient management of memory. Suppose that in a program we declare variables, store numbers in the allocated memory, use these numbers to compute some desired answer, and proceed to use this answer in the rest of the program but not the original numbers. Those numbers stay in the memory cells allocated for them throughout the program execution, wasting the memory. In the mean time any variable declared later in the program uses up new memory cells and not the ones previously allocated for variables that are no longer in use. In this approach of programming a simple program some times requires unnecessarily large amount of memory to run. C++ offers an operator *delete* to de-allocate the memory allocated through the *new* operator. De-allocating memory means indicating to the system that the specific memory is available for future use in the same program.

If p is a pointer that is pointing to a memory cell that needs to be de-allocated, we write

```
delete p;
```

Example 7.6.2. Suppose that the program written in Example 1 is a part of a longer program that will use the average stored in *p but not the number stored in *q. By

adding the following statement after the statement *p = (*p + *q)/2.0; we de-allocate the memory that q is pointing to.

delete q;

If a pointer called *ptr* is pointing to an array, we can de-allocate the entire memory allocated for the array by writing

delete []ptr;

Minimizing memory usage by effectively using *new* and *delete* operators is called "dynamic memory allocation". The following example illustrates creating, manipulating, and deleting an array through the new operator and delete operator.

Example 7.6.3. The following program uses a pointer to obtain and store up to 100 integer numbers in an array and to find and print the maximum of those numbers.

```
int find_max (int *);
main()
{
 int *p, i = 0, count;
 p = new int[100];
 cout << "Enter the number:";
 cin >> num;
 while (num != -1 && i < 100)
       {
        *(p + i) = num;
        i++;
        cout << "Enter the number (-1 when finished):";
        cin >> num;
        }
if (i == 100)
       {
        cout << "Only up to 100 numbers can be entered"
        return 0;
       }
else
      {
       count = i;       //Count of the data entered.
       cout << "Maximum of the list = " << find_max (p, count);
       }
delete []p;
return 0;
}

//Function find_max begins
```

```
find_max (int *ptr, size)
{
 int  max = *ptr, j;

 for (j = 1; j < size; j++)
       if (*(ptr + j) > max)
              max = *(ptr + j);
 return max;
 }
```

In the last example, note that in place of *(p + i) we could have written p[i] while keeping everything else the same. Also within the function *find_max* in place of *(ptr + j) we could have written ptr[j].

Exercises 7.6.

In the following two problems, write the output when the statements are executed as part of complete programs.

1.
```
int  *peep, k;
peep = new int[5];
for (k=0; k < 5; k++)
    peep[k] = 2*k;
cout << *(peep + 4);
delete []peep;
```

2.
```
float  *din1, *din2, m;
    din1 = new float[4];
din2 = new float;
    for (m = 0; m < 4; m++)
          *(din1+m) = m*m;
    *din2 = 0;
    for (m = 0; m < 4; m++)
          *din2 += din1[m];
    cout << *din2;
    delete []din1, din2;
```

3. Write a function that receives an integer array and uses pointer notation to count how many numbers in the array are evenly divisible by 5. Write a complete program that obtains up to 100 positive integers from the user, uses the above written function to count how many of them are divisible by 5, and prints this count. User will enter a -1 to indicate end of entering data. Your program must use dynamic memory allocation.

4. Write a function that receives a character array and uses pointer notation to count how many times each of the vowels a, e, i, o, and u were found among the characters in the array and returns these counts as another array of size 5. Also write a main() function that obtains up to 200 characters from the user, uses the above written function to count the occurrence of each, and prints these counts. User will enter a -1 to indicate end of entering data. Your program must use dynamic memory allocation.

7.7. Strings

A character constant that we can assign to a char type variable can only be a single character such as 'c'. To store a single character data in a memory, we can write a simple assignment statement such as

```
char  cval = 'c';
```

Single characters need only one byte of memory, and therefore, char type variables get one byte of memory allocated. Often programs are needed to process a chain of many characters together.

> A chain of many characters together is called a **string**.

For example, names are strings, sentences found in books are strings etc.

String Constants

When using a string within a C++ program, ***to identify the string constant as such, we must use double quotes (") around it***. For example, the text

"brown fox"

is identified as a valid string constant, and it can be part of a program, but the texts such as

'brown fox' or brown fox (with no special characters around it)

found in a program will lead to an error message that they are invalid data.

> When the compiler compiles a string constant found in a program, it **appends a null character ('\0') at the end** of it.

The purpose of the null character is to enable the programmer to identify the end of the string when he or she retrieves the string without knowing the length of it. The technical definition of a string is, therefore, the following.

> A string is an **array** of **characters terminated by a null character ('\0')**.

C-Strings

C++ language evolved from C language. Features of C language are available in C++ as well. The C language provides no direct provision for storing a string in memory. In other words, **there is no built-in data type for handling strings** such as *char* for characters and *int* for integers provided by the C language. C++, on the other hand, has an externally defined data-type called "string", which means, in C++ **a string can be treated as a single data entity for manipulation purposes**. Since the way of handling strings in C language has advantages in certain applications, we shall discuss both ways of handling strings in this section. We will begin with the C-language strings, often referred to as C-strings. C-strings are stored in arrays of char type.

Declaring and Initializing C-strings

A string constant can be stored in the memory simply by assigning it to an array of char type. To write a statement that stores a string in the memory, the programmer needs to determine the number of characters the string has *including the null character*, and declare an array of that size. In other words,

> programmer must **use one more than the actual number of characters found in a string as the size of the string** in the statements that allocate memory for the string.

A compilation error occurs if there is not enough room for the string in the array. Let us see some examples.

Example 7.7.1. The storage space needed for the string constant "brown" is 6 bytes, and the storage space needed for "brown fox" is 10 bytes (blank space is also a character).

Example 7.7.2. 'b' is a character constant and the storage space needed for it is 1 byte.
"b" is a string constant, and the storage space needed for it is 2 bytes.

Example 7.7.3. To store the string constant "brown fox" in the memory, we write

```
char  s1[10] = "brown fox";
```

Here, s1 is the name of the array in which the string is stored. In our computer-model whose memory is as in Figure 7.1.1, the result of executing the statement in Example 3 is shown in Figure 7.7.1.

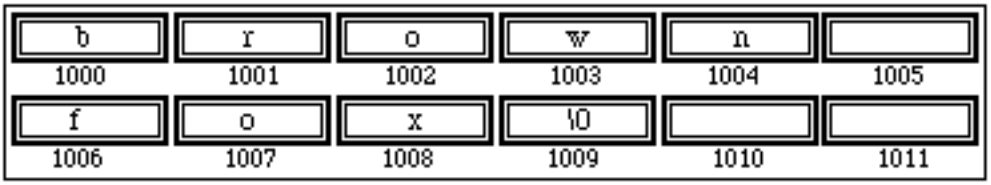


Figure 7.7.1

The statement in Example 3 is equivalent to the statement

```
char s1[10] = {'b', 'r', 'o', 'w', 'n', ' ', 'f', 'o', 'x'};
```

Just as for any array initialized at declaration, one can leave out the size of the array for the compiler to figure out, as in

```
char s1[] = "brown fox";
```

Recall that arrays are handled by the compiler by using pointers. The name of the array is a constant pointer pointing to the beginning of the array. The above statement stores the string in the memory and sets a pointer named s1 to point to the first element, which has the 'b' stored.

Storing a string can also be achieved by declaring a pointer and assigning the string to it.

Example 4 illustrates this.

Example 7.7.4. To store the string constant "brown fox" in the memory, we write

```
char *s1;
s1 = "brown fox";
```

or equivalently,

```
char  *s1 = "brown fox";
```

In other words,

a **string can be treated like a memory address, and assigned to a pointer**. When such a statement is executed, the string gets stored in the memory and the address of the beginning of the string gets stored in the pointer.

The only difference between the way the string is stored in the memory in Example 3 and the way it is done in Example 4 is that s1 is a constant pointer (that cannot be changed) in Example 3, and it is a variable pointer in Example 4.

Facts Regarding Statements that Involve C-Strings

1. If the array has more room than necessary, the remaining elements get 0 stored in them.

2. If array approach (instead of pointer) is used to store a string in the memory, then the string assignment must be done in the same statement that declares the array. For example the statements

```
char  p[10];     //p is a constant pointer.
p = "brown fox";  //An attempt to change the value of p.
```

are invalid. But if we combine them into one statement as

```
char p[10] = "brown fox";
```

then it is valid.

3. After declaring p as an array, initializing its elements one at a time as shown in Example 5 is valid.

Example 7.7.5. The following statements achieve the same results as the statement in Example 3 does.

```
char p[10];
p[0] = 'b';  p[1] = 'r';  p[2] = 'o';  p[3] = 'w';  p[4] = 'n' ;  p[5] = ' ';
p[6] = 'f';  p[7] = 'o';  p[8] = 'x';  p[9] = '\0';
```

4. After declaring p as a pointer or as an array, assigning a string to *p is invalid because *p represents the first element of the array, and it can only take a character, not a string.

```
char  *p;
char  q[10];
*p = "brown fox"      //Invalid. Attempt to store string in a
                      //char type memory.
*q = "brown fox"      //Invalid for the same reason.
*p = 'b';             //Valid.
```

C++ Strings

While C-language handled strings in the above manner, C++ language offers a better way because it allows users to define their own data types by using what is called "class" concept. This topic is reserved for later discussion. A class under the name "string" has already been defined and made available to us. This class-definition allows us to use the word ***string*** as a data type just like ***int***, and ***float***. Therefore, if we include this class-definition by writing

```
#include <string>
```

Then, **we can declare a variable** *s1* of **string type** for storing strings by writing

```
string s1;
```

We shall refer to strings created this way as C++-string. Note that

C-string and C++ string are two very different objects. One cannot treat a C++ string as C-string or vice versa.

There are ways to convert one to the other. But we choose not to discuss that here. Programmer has to determine which type of string is appropriate for the specific application and use that.

From here on we shall discuss the ways of handling strings as C-strings as well as C++ strings.

Reading and Writing Strings

The *cin* statement can be used to read a string from keyboard into a pointer or an array. To read a string from keyboard into an array, declare an array of *char* type large enough for the string, and use the name of the array in a *cin* statement as in

```
char s1[20];     //Array approach.
cin >> s1;
```

or as in

```
char *s1;     // Pointer approach
cin >> s1;.
```

Or (using C++-string) as in

```
string s1;
cin >> s1;
```

When reading from files the "cin" here would have to be replaced by the stream-name of the file. Here, the latter one requires having #include <string> at the top of the program. Note that in all of the three approaches above, ***only the name of the array is used in the cin statement***, and no information about the size is included. When array approach is used, it is the programmer's responsibility to make sure that the array is large enough for the string expected from the user. Using larger than necessary size for the array does not do any damage other than reserving some memory cells that are not used for anything. **When the pointer approach or string class is used, the string is stored in as many cells as it takes starting from the next available cell and the pointer is set to point to the beginning of the string**. It is the programmer's responsibility to make sure that sufficient number of memory cells that follow the currently available cell are available for use. The array approach is recommended since it avoids potential disasters.

When the above *cin* statement is executed, the processor will stop and wait for the user to enter a string. **User should enter the string (without double quotes) followed by a return**. The *cin* reads all the characters in the string until a white space is encountered, adds a null character, and stores the resulting string in the array or pointer s1.

Multiple strings can be read is the same way as long as there is one (or more) white space separating them.

Example 7.7.6. If the user input on the keyboard (or a file) has the following line of data, write statements to read them into C-string and C++ string. Here, the rectangles represent white spaces.

Kenneth□□Carlson□□□35

```
char *first, *last;  //to read as C-strings
int age;
cin >> first >> last >> age;
```

```
string first, last;  //to read as C++ strings
int age;
cin >> first >> last >> age;
```

Note that the **above method cannot be used to read a line of text that has blanks** in it **as one string**.

There are functions already defined in the libraries that are useful for reading the data character by character, for reading a line of text as a C-string etc. These functions are described in Section 10.3 under the title "Reading From Files". When reading from keyboard, we just have use cin for the name of the stream instead of a file-stream.

The ***cout* statement can be used to write a C-string stored in an array or pointer or C++ string type variable on the screen**. For example, the string stored in the character array s1 can be printed on the screen by writing

```
cout << s1 << endl;
```

Note that no information of the size of the string is included in the above statement. The end of the string is recognized by reading the null character. That is, **when the statement *cout << s1;* is executed, all the characters found in the array s1 are printed until the null character (\0) is encountered**. For writing into files, the "cout" would have to be replaced with the stream-name of the file.

The following examples illustrate how the above described string related features of C and C++ are used in working with strings.

Example 7.7.7. The following program obtains a string of up to 15 characters from the user and prints it in the reverse order.

```
main()
{
char text[15];
int  j = 0, count =0;
cout << "Enter the string:"
cin >> text;
//Count the characters in the string.
while (text[j] != '\0'  && j < 15)
      {
      count++;
      j++;
      }
//Print the string in reverse, excluding the null character.
for (j = count - 1; j >= 0; j--)
       cout << text[j];
}
```

The machine language code of the null character is the same as that of 0. In other words the computer system cannot distinguish the null character and the number 0. The number 0 is also the same as the logical value 'false' in C++. Non-null characters have non-zero numbers as machine codes. Therefore, the condition text[j] != 0 found in the above program can be replaced with simply text[j], to make the while statement read as

while (**text[j]** && j < 15)

String Comparison

Recall from Chapter 1, that there is a standard ordering of all the characters from smallest to largest, in which alphabets are ordered in the same way they appear in the usual listing of alphabets. As you go down this list, the decimal equivalent of the machine language codes of these characters increase. Therefore whether or not a character is above another in the ordered list can be determined by comparing their machine language codes. If we just compare the characters in a program, actually their machine language codes get compared. For example, 'b' < 'd' is true.

The ordering of strings is the same as how they are ordered in dictionaries and phone books. We shall, however, describe here how strings are compared. To compare two strings, we need to compare the characters that comprise the string in the order they are found in the strings. Suppose the s1 and s2 are two strings (that is, two character arrays). When you compare each character of s1 with the corresponding character of s2,

if the **first character of s1 that doesn't match the corresponding character of s2 is in fact smaller than the corresponding character of s2** then s1 is said to be smaller than s2.

If the first character of s1 that doesn't match the corresponding character of s2 is larger than the corresponding character of s2, then s1 is said to be a larger string than s2. If all the characters in s1 match those in s2, then the two strings are equal. Note that when a string s2 has all of the characters of s1 in the same order and have more characters, then s1 is smaller. Generally, **whether a string is shorter than another or longer does not contribute to determining whether it is smaller or larger than the other**.

Example 7.7.8. The following inequalities of strings are true

(a) "a" < "b" (b) "maple" < "pine" (c) "man" < "men" (d) "apple" < "apu"

(e) "mark" < "marker"

Example 7.7.9. The following function receives two strings and returns a 1 if the first one is smaller than the second, returns a zero if they are equal, and returns -1 otherwise.

```
compare_string (char *s1, char *s2)
{
 int  j = 0;
 while (s1[j] == s2[j] && s1[j] != '\0' && s2[j] != '0')
        j++;
 if (s1[j] == '\0' && s2[j] != '0')
        return 1;   //s2 has all the characters of s1 and more
 else if (s1[j] != '\0' && s2[j] == '0')
        return -1;    //s1 has all the characters of s2 and more
 else if (s1[j] == '\0' && s2[j] == '0')
        return 0;
 else if (s1[j] < s2[j])
        return 1;
 else if (s1[j] > s2[j])
        return -1;
}
```

C-String Handling Library

Strings created as arrays (that is, C-strings) cannot be handled in a computer program in the same manner numbers and single characters can be handled. For example, to copy the value of the C-string s1 into s2, we cannot write a simple assignment statement s2 = s1;, we instead have to write statements to perform character by character copying. For another example to compare and determine which one of s1 and s2 is smaller, we cannot write s1< s2.

In order to facilitate string operations such as the ones described above, C-language provides a rich library of functions available in the *string.h* header file. To gain access to this library of functions, we must include this library by writing

#include <string.h>

at the top of the program. Notice that **#include <string> is a C++ statement, which includes the class called string, a different file**. We shall describe some of the functions in the ***string.h*** library including information such as what their parameters are. We will also provide an example that illustrates how to use each function.

(1) The **String Length** Function: ***int strlen(char s[])***

Receives a string as parameter and *returns the number of characters* found in the string (excluding null character).

Example 7.7.10.
```
char name[] = "Jacob Kizer";
int  m = strlen (name);
cout << m;       //Prints the length of the string "Jacob Kizer", which is 11.
```

(2) The **String Copy** Function: ***void strcpy (char s1[], char s2[])***

Receives two character arrays and *copies the string found in the first into the second*.

Example 7.7.11.
```
char s1[] = "Gilmer", s2[];
strcpy (s1, s2);       //Now s2 also has "Gilmer".
```

(3) The **Pattern Matching** Function: ***int strstr (char s1[], char s2[])***

Receives two strings, searches the first string for an occurrence of the second string. Returns the byte position of occurrence if found. Returns -1 otherwise.

Example 7.7.12.
```
char q1[] = "The quick brown fox",  q2[] = "quick";
int  k = strstr (q1, q2);  //Prints where q2 begins within q1. Answer: 4
```

(4) The **String Concatenating** Function: **char *strcat (char s1[], char s2[])**

Receives two strings, concatenates them into one (that is, appends s2 at the end of s1) and returns a pointer to the beginning of the resulting string.

Example 7.7.13.
```
char q1[] = "The quick ",  q2[] = "brown fox";
cout << strcat (q1, q2);  //Prints "The quick brown fox"
```

(5) The **String Comparison** Function: **int strcmp (char s1[], char s2[])**

Receives two strings s1 and s2 and compares them. Returns positive integer if s1 is less than s2; returns zero if they are equal and returns negative integer otherwise. Note that this function is similar to the compare_string function written in Example 8.

Example 7.7.14. *//This program prints two given names in alphabetical order.*

```
char q1[] = "Kinney",  q2[] = "Kizer";
if (strcmp (q1,q2) > 0)
        cout << q1 << endl << q2;
else
        cout << q2 << endl << q1;
```

C++-Strings

As noted earlier, string-class in C++ that we include by writing **#include <string>** allows us to use the key word *string* to create a variable to hold a string. It also allows us to manipulate them conveniently just like the other data types. Here are a few illustrations.

1. We can store a string constant in a string-type variable by writing

```
string s1 = "brown fox";
```

2. We can copy a string into another by writing

```
string s2 = s1;
```

3. We can compare two strings by using the usual comparison operators <, <=, ==, != etc, and write statements such as

```
if (s1 <= s2)
    cout << s1.
```

String-class also offers convenient functions such as ***size()*** that determines size of a string. We shall see a simple example that illustrates the convenience of using C++ string.

Example 7.7.15. Write a program that obtains a list of names of students in a class (of size up to 30) and their grades in two tests, stores them in array, and writes the names of the students who pass by earning 60% or better average grade. Use a sentinel to identify end of data.

```
main()
{
float test1[30], test2[30], average[30];
int i = 0, size;
string name[30];     //names are strings
cout<< "Enter name and two test grades separated by space";
cin >> name[i] >> test1[i] >> test2[i];
average[i] = test1[i] + test2[i];
//Continued on the next page
```

```
while (name[i] != "$" && i < 30)
    {
    i++;
    cout<< "Enter name ($ when no more) and two test grades"
    cin >> name[i] >> test1[i] >> test2[i];
    average[i] = test1[i] + test2[i];
    }
size = i ;   //class-size
for (i = 0; i < size; i++)
    if (average[i] >= 60)
        cout << name[i] << endl;
}
```

Example 7.7.16. Write a program that obtains last names, phone numbers and salaries of 40 employees of a company, stores them in three arrays, and then allows the user to do two kinds of searches:

1. Search by the name and print the record on the screen.
2. Search and print all the names of employees who exceed a specific salary to be entered by the user.

```
main()
{
string name[40], phone[40], search_key, find_name;
float salary[40], find_sal;
int i;
for(i = 0; i < 40; i++)
    {
    cout<< "Enter name, phone number and salary separated by space";
    cin >> name[i] >> phone[i] >> salary[i];
    }
cout << "Enter the search key (name or salary) as a string";
cin >> search_key;
if(search_key == "name")
    {
    cout << "Enter the name to search for";
    cin >> find_name;
    i = 0;
    while (name[i] != find_name  &&  i < 40)
        i++;
//Continued on the next page
```

```
//If the name is not in the list, then loop ends with i = 40
      if(i == 40)
          {
          cout << "Name is not in the list";
          return 0;
          }
      else  //ith element of phone array has the phone number.
          cout << name[i] << endl <<phone[i] << endl << salary[i];
     }
else
     {
      cout << "Enter the salary, I will find everyone who makes more";
      cin >> find_sal;
     for ( i = 0; i < 40; i++)
          if (salary[i] > find_sal)
               cout << name[i];
}
```

Accessing Individual Characters of a String

The **subscript operator []** and the member function ***at()*** are useful for accessing a character at a specified position in a string (counting from 0)

s[i] **Returns a reference to the character at position i** in string s.
No exception is raised if i is out of range.

s.at(i) **Returns a reference to the character at position i** in string s.
An out-of-range exception is raised if i is out of range.

The following example illustrates a use of these tools.

Example 7.7.17. Write a program that reads a line of text from keyboard and counts the number of occurrences of the letter a in it and prints this count.

Since the line of text may have white space characters in it, we cannot use cin statement to read it. We will use the getline() function to read the entire line of text.

```
#include <string>
main()
{
 string  line;
 char letter;
 int  count = 0, i = 0, flag = 1;
```

```
cout << "Enter a word: ";
getline (cin, line);
letter = line.at(i);
while(letter != '\n')
      {
      if (line.at(i) == 'a')
            count++;
      i++;
      }
cout << "The number of occurrences of letter a =" << count;
return 0;
}
```

Exercises 7.7

1. Write a C++ statement to store the string "running up" in the memory.

2. Write a C++ statement to store the string "to save the world" in the memory.

In Problems 3 through 6, write the output when the given statements are executed as part of complete programs.

3.
```
char  str[12] = "sweeper one";
cout << str[1] << str[8];
```

4.
```
char str[15] = "medal woman";
cout << str[2] << str[7];
```

5.
```
char  s[15] = "quick biter";
int  j = 0;
while (s[j] != '\0')
      j++;
cout << j;
```

6.
```
char  str[12] = "sweeper one";
int  j = 0, t = 0;
while (str[j] != ' ')
     j++;
cout << j;
```

7. Write a program to obtain a name (user will enter first name and last name together, with first name first and with a single space in between; no middle name) from the user and to write only the last name on the screen.

8. Write a function that receives a string and returns the count of vowels found in the string. Then write a main() function that obtains a string from the user and uses the function to count and print the vowels found in the string.

9. Write a program that obtains a string of characters from the user that consists of lower case alphabets only, and writes it with all upper case alphabets.

10. Write a program that reads a string of characters (that consists of both upper and lower case characters mixed) from keyboard, converts all the lower case alphabets to uppercase, and writes the entire string out.

11. Write a program to obtain two names from the user, and to use the function *strcmp* to write these names in alphabetical order.

12. Write the definition of the *strcpy* function.

13. Write a program that receives a sentence and a word (that is, two strings) from the user, determines how many times the word is used in the sentence and prints this number on the screen. (Hint: the *strstr* function is useful)

14. Write a program that obtains names and test grades of students in a class (of size up to 40), stores them in arrays, find the class average, and then print the names of students who have above average grades.

15. Write a function *bubblesort()* that receives an array of strings and sorts it to alphabetical order. Then write a complete program that obtains up to (but not necessarily equal to) 50 names from the user, stores them in an array, and uses *bubblesort()* function to write the names in alphabetical order.

16. Write a function that receives a string of lower case and uppercase letters, returns a copy of the string in all lowercase letters.

17. Write a function that accepts two strings and determines whether one string is an anagram of the other, that is, whether one string is a permutation of the characters in the other string. For example, "dear" and "dare" are anagrams of "read".

Programming Projects 7.7

Project 7.7.1. An instructor needs a computer program that can compute course-grades for his classes. When the program runs, it should request and last name, two test grades, and a homework grade of one student at a time. It should then compute the course-grade by using the following formula:

Course-grade = 60% of average of tests + 40% of homework grade

and **round it to the nearest integer**. It should then **print the name** of the student, **course grade as an integer number** next to it, and **course grade in letter form** next to that. Letter grade is determined by the following formula:

90 and above	A
80 to 89	B
70 to 79	C
60 to 69	D
Below 60	F

Class-size is unknown. Therefore, you need to use a sentinel-controlled loop to repeat this process as long as there is more data. Use a sentinel of your own choice, but let the user know what it is. Sentinel needs to be a string followed by three numbers. For example

$$ -1 -1 -1 or &&& 0 0 0

can be a sentinel. When the instructor finished entering data, program should **print the name and course-grade of the student who earned the highest grade**.

Conform your program to the following guidelines:

1. Have your **full name, course-name and section number typed** as a comment at the very top.
2. Insert an explanatory comment at the top that explains what your program performs.
3. Insert comments into sections of the program that explains what the section does.
4. **Use dynamic memory management** to make the program memory efficient.
5. When the program works, run it and enter the following data:

   Lennon 72 65 67
   Skyler 72 67 87
   Rogers 84 92 96
   Lopez 91 84 85
   Kenny 57 62 69

   Here the first two numbers are the test grades and the last one is the homework grade.

6. Program must be written to work with any number of data.

7. Produce a **print-out that shows at least a portion of the program and the user-computer interaction** during the program execution. Produce **another print-out that shows the entire program**. Staple the two print-outs, **hand-write your name** on it and turn in.

   Sample Output: Lennon 68% D

7.8. Array of Pointers and Array of strings

We can declare an array with each element being a pointer. **Each element of such an array is good for storing a string**. Therefore, **such an array is good for storing a list of strings**. In fact, this is the only way to store a list of many strings, such as names of customers, in C-language. For example, if we need to store the names of a group of people in the memory of the computer system and process them, we need to use an array of pointers.

An array of pointers of size 5, named *list* can be declared by writing

```
char *list[5];.
```

This declaration is not treated as a pointer to an array because the [] operator has higher precedence than the * operator.

The declaration shown above allocates 5 contiguous four byte memory cells for the 5 pointers (but does not initialize them with any address). One can store memory addresses of other values in these pointers. Particularly, we can assign strings to these pointers.

Array of pointers is necessary to store a list of strings only if they are to be handled as C-strings. To store a list of strings in C++, we can declare an array of string type. For example, the array needed to store five strings can be declared as

```
string list[5]
```

The following examples illustrate how list of strings can be manipulated in C and C++.

Example 7.8.1. We shall write a program that stores five names in an array of pointers and finds and writes the smallest of them.

```
#include <string.h>
main()
{
  char  *list[5], *lowest;
  int j;
  list[0] = "Smith";
  list[1] = "Kinney";
  list[2] = "Brown";
  list[3] = "Roper";
  list[4] = "Kizer";
  lowest = list[0];
  for (j = 1; j < 5; j++)
        if (strcmp(list[j], lowest) > 0)
              strcpy (list[j], lowest);
  cout << lowest;
  return 0;
  }
```

To rewrite the above program using C++ strings, in addition to changing **#include <string.h>** to **#include <string>** we only need to make the following changes:

1. Change the very first line in the main() function to: string list[5], lowest;

2. Change the if statement in the loop to:

```
if (list[j] < lowest)
```

```
                lowest = list[j];
```

Example 7.8.2. We shall write a function called *search* that receives an array of character pointers (that is, array of strings), the size of it, and another character string, searches the array of strings for the other string and returns its position if found. It should return a -1 otherwise.

```
int search (char *list[], int size, char *str)
{
 int i = 0, match;
 match = strcmp (list[i], str);
 while (match != 0 && i < size)
        {
         i++;
         match = strcmp (list[i], str);
        }
 if (i == size)
        return -1;   //No matching string.
 else
        return i;     //Match was found at i th element.
}
```

To rewrite the last program to use C++ strings, in addition to changing **#include <string.h>** to **#include <string>** we only need to make the following changes:

1. Change the header to: int **search** (**string** list[], **int** size, **string** str)

2. Change the second line in the loop to: match = list[i] ==str;

Example 3. Write a program that obtains names and grades of up to 30 students, stores them in arrays, and then prints the names of the students whose grades are over 90. Assume that the user will enter $ when he is finished entering data.

```
#include <string.h>
main()
{
 char *name[30];
 float grade[30];
 int i = 0, j ;
 cout << "Enter the name and grade with space in between";
 cin >> name[i] >> grade[i];
 while (strcmp(name[i], "$") != 0  && i < 30)
        {
         i++;
         cout << "Enter the name and grade ($ and 0 when finished);
         cin >> name[i] >> grade[i];
        }
if (i == 30)
```

```
    {
    cout << "too many data";
    return 0;
    }
//Count of data is i.
//Write the names that have corresponding grade entry 90 or over.
for (j = 0; j < i; j++)
            if (grade[j] > 90)
                    cout << name[j];
return 0;
}
```

To rewrite the last program to use C++ strings, in addition to changing **#include <string.h>** to **#include <string>** we only need to make the following changes:

1. Change the very first line of main() function to: **string** name[30]

2. Change the loop condition to: name[i] != "$" && i < 30

Example 7.8.4. Write a program that stores the following list of names of employees of a company and their salaries in arrays, and then finds and prints the salary of any name obtained from the user.

```
Simons      43000
Rogers      32000
Trueman     67400
Roberts     79200
Myers       45600

#include <string.h>
main()
{
 char *name[5] = {"Simons", "Rogers", "Trueman", "Roberts", "Myers"};
 float salary[5] = {43000, 32000, 67400, 79200, 45600};
 char *given_name;
 int i = 0;
 cout << "Enter the name whose salary is needed";
 cin >> given_name;
 while (strcmp(name[i], given_name) != 0  && i < 5)
        i++;
 if (i == 5)
        cout << "Name was not found";
 else
        cout << salary[i];
 return 0;
}
```

Reader can now easily determine what changes have to be made to the above program to make it use C++ strings.

Exercises 7.8.

1. Write a function *scounter()* that receives an array of character pointers, and another character string, determines how many elements of the array are smaller than the received string, and returns this count.

2. Write a function *find_name()* that receives an array of character pointers, and another character string, and finds and returns the location of the string in the array. If the string is not in the array, it should return a -1.

3. Write a program that obtains and stores names and ages of up to 50 people in arrays, and then write the names of those who are 35 years or older.

4. Names and ages of 7 people are as follows.

Smith	37
Kinney	21
Brown	18
Roper	45
Kizer	52
Simmons	19
Kline	39

Write a program that stores this data in arrays, and then obtains a name from the user and finds and prints the age of that name.

5. Write a complete program that obtains and stores up to 40 names of people in an array, uses bubble sort technique to sort this array to have the names in alphabetical order, and writes the sorted list.

Programming Projects 7.8

Project 7.8.1. Write a program that stores a list of names of students in a class and their grades in arrays, and then allows the user to carry out the following functions.

1. Sort the data by name (alphabetical order)
2. sort the data by grade (increasing order)
3. Search for a grade (of a student whose name is entered by the user)
4. Find names of students who have certain grade or over.

When the program runs, it should display the above four items like a menu, and have the user select an item by typing a number. Then it should obtain any additional information needed from the user and carry out the task.

Use the list given below *along with your own name and a grade you choose* as the data. Program must have your name typed as part of the comment. Comments and indentations must be addequately used.

Simmons	93
Rogers	68
Trueman	87
Roberts	98
Myers	45
Kinney	82
Baar	88
Lennon	75
Cohen	90
Wallah	62
Vernon	78

When the program works, print an output window after carrying out any two of the four tasks, and print the program.

Chapter Summary

1. An array represents a chain of memory cells.

2. A pointer is a variable that can take a memory address as value.

3. &*variable* gives us the address of the *variable*.

4. **pointer* means "value of the cell being pointed to by *pointer*".

5. The name of an array is a pointer that has the address of the beginning of the array.

6. An array is passed to a function by passing its name (which has the address of the beginning of the array)

7. The values in the entries of the array named *arr* can be retrieved by using the array name with subscript: arr[i] or by using the pointer arr by writing *(arr+i).

8. Passing by value: when a variable name that has a value is passed to a function, only a copy of the original value get passed to the function, and therefore, the function cannot change the original value stored in the variable. The function receives the value in a variable.
9. Passing by reference: when the address of the variable that has the value is passed to a function, the function has access to the original location of the value, and therefore it can change the value.

10. A string is a chain of characters written in double quotes.

11. Every string has a hidden null character '\0' at the end.

12. A string can be treated as the memory address of the beginning of the string.

13. The string "Bronson" can be stored in the array p[8] by writing (array method)

p[8] = "Bronson";

or by writing (pointer method)

p = "Bronson";

14. A string can be written out on the screen by using cout << p; where p is a pointer pointing to the beginning of the string. A string can be read into a pointer p from keyboard by using cin >> p;

15. Prototypes of the most important string manipulation functions are:

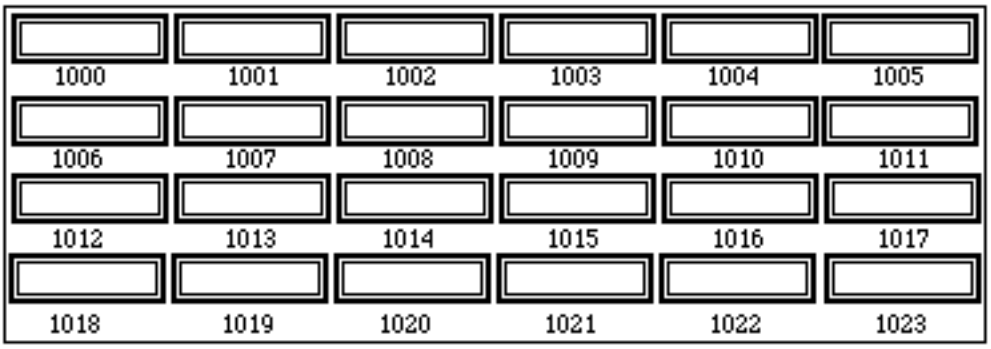
strlen(char *)	receives a string; returns its length
strstr(char *, char *)	receives two strings; returns the location of the second within the first.
strcpy(char *, char *)	copies the first string into the second.
strcmp(char *, char *)	returns positive integer if first string < second, zero if the strings are equal, negative otherwise.

16. An array of pointers is declared by writing, for example, char *ptr[size];

17. An array of pointers is useful to store a list of strings such as names of people.

Chapters 7 Review Exercises

In all the problems here, whenever a computer system is involved, assume working with a system that allocates **1 byte of memory for *char*** type, **2 bytes for *short*** type, **4 bytes for *float***, **4 bytes for *int*** type variables, **4 bytes for pointers** and **8 bytes for *long* and *double*** type variables. Also assume the following memory model:



Write the exact output (in the designated area) **you expect to see on the screen** when the following statements are executed as part of complete programs. Assume that all necessary libraries are included in each program. If your conclusion is that an error message will result, briefly explain why.

1.
```
short  a = 4, b = 2, *p1;
short  c = 13, d = 5, *p2;
 p1 = &a;
 p2 = &c;
 cout << (int)ptr2 – (int)ptr1 << endl
       << *(p2+1)
```
Output:_____

2.
```
int  ar[] = {2, 9, 14, 5, 3}, i = 2;
 cout << *(arr + i);
```
Output:_____

3.
```
int  x = 21, y = 9, *p;
p1 = &x;
cout << *p + *(p+1);
```
Output:_____

4.
```
int  a[] = {5, 8, 2}, *p;
 p = a;
 cout << *(p+1);
```
Output:_____

5.
```
// Assume that the next available
// memory cell has address 2000
int  val = 4, num = 12, *p;
p = &val;
cout << &p << endl;
cout << *(p+1);
```
Output:_____

6.
```
// Assume that the next available
// memory cell has address 2000
short int  x = 5, y = 3, *p, *q;
p = &x;
q = &y;
cout << *(q -1) << endl;
cout << q - p << endl;
cout << (int) q - (int) p;
```
Output:_____

7.
```
int *p, val = 2, num = 9;
 p = &val;
 *p = *p + 4 * num ;
 cout  << val;
```
Output:_____

8.
```
int *a = {13, 9, 11, 4, 7, 10}, s=0, i;
 s = *a;
 for(i = 1; i < 6; i++)
     if(*(a+i) < s)
         s = *(a+i);
cout << s;
```
Output:_____

9.
```
char *p = “shoemaker”
cout << *(p+4);
```
Output:_____

10.
```
char *p = “mania”, *q = “makers”;
 cout << strcmp(p, q);
```
Output:_____

11.
```
char *p = “socialist”, *q =”ali”;
 int m = strstr(p, q);
if (m > 0)
     cout << p[m-1];
else
     cout << “sorry”;
```
Output:_____

12.
```
char st[9] = “sansonit”;
int i=1;
while(*st != ‘|n’)
   {
    st++;
    i++;
    }
```
Output:_____

13.
```
void fun(int *, int *);
main()
{
 int  x = 12, y = 3, *p1, *p2;
 p1 = &x;
 p2 = &y;
 fun(p1, p2);
 cout << x << endl << y;
}
void fun(int *s, int *q)
{
 *q = *s;
 *s = *s + 8.;
}
```

Output:_____

14.
```
float jiggy(int *, int);
 main()
 {
 int  nu[] = {9, 12, 5, 7, 14};
 float m = jiggy(nu, 5);
 cout << m << endl;
 cout << *nu << endl << *(nu+2);
 }
// function begins.
float jiggy(int *p, int n)
 {
 int  x = 0, i = 0;
 for (i = 0; i < n; i++)
   {
    x += *(p+i);
    *(p+i) += 2;
     }
  return  x/ (float) n;
 }
```

Output:_____

In Problems 15 through 20, **write C++ statements** (not necessarily complete programs) **to achieve the indicated task**.

15. To store the number 14 in the memory and to print the address of the very next available memory cell.

16. To compute the product of the two numbers 23.9 and 31.6, and to print the answer with dynamic memory allocation.

17. To obtain a string from the user, and to print how many characters are there in it.

18. To obtain up to 20 strings (actual count is unknown) from the user and to print the largest one.

19. To read up to 100 strings from the file called *names.dat* and to count and print how many of them contain the string "coh" within them.

20. To count the number of words in a paragraph of text found in the file called words.txt, and to print it on the screen.

In Problems 21 through 23 **write complete programs**.

21. (a) Write a function *counter()* that receives a string an returns the number of non-alphabetic characters found in it.

(b) Write a program that obtains an English sentence from the user and uses the function *counter()* to count and print the non-alphabetic characters in it.

22. (a) Write a function bubble() that receives an array of strings and the size of the array, and sorts them to have alphabetical order. (The original array should be rearranged in increasing order.)

(b) Write a main program that reads up to 50 names from the file called ***list.dat***, and uses the function in (a) to write it in order on the screen.

23. (a) (a) Write a function ***finder()*** that receives a list (array) of strings, its size, and another string to be found in that list and returns the location of the string in the list. If the string is not found, the function should return a -1.

(b) Write a program that reads the names of up to 100 students and their test grades (integers) into arrays of pointers, and then repeatedly obtains a name from the user and finds and writes the grade of that student until the user enters a $ for name. When the student is not found in the list, the system should print an appropriate message and continue to obtain another name.