Chapter: 4

Strings, Arrays & Pointers

Arrays : one-dimensional & multi-dimensional, Strings, Arrays of Strings, pointers.

4.1 One dimensional Arrays

In C, a one-dimensional array is a list of variables that are all of the same type and are accessed through a common name.

- An individual variable in the array is called an array element.
- Arrays form a convenient way to handle groups of related data.

To declare a one-dimensional array, use the ***general form***

```
type var_name[size];
```

where ***type*** is a valid C data type, ***var_name*** is the name of the array, and ***size*** specifies the number of elements in the array.

For example, to declare an integer array with 20 elements called ***myarray***, use this statement.

```
int myarray[20];
```

NOTES

[1] An array element is accessed by indexing the array using the number of the element.

[2] Remember, ***arrays start at zero***, so an index of ***1*** references the ***second element***. In C, all arrays begin at zero. This means that if you want to access the first element in an array, use zero for the index.

[3] To index an array, specify the index of the element you want inside square brackets. For example, **myarray[1]** refers to the second element of ***myarray***.

[4] TO assign an array element a value, put the array on the left side of an assignment statement. For example, **myarray[0] = 100;** gives the first element in ***myarray*** the value ***100***.

[5] C stores one-dimensional arrays in one contiguous memory location with the first element at the lowest addressJFor example, after this fragment executes,

```
int i[5] ; int j;
for(j=0; j<5; j++) i[j]=j;
```

array i will look like this:

	i[0]	i[1]	i[2]	i[3]	i[4]
i	0	1	2	3	4

[6] you may use the value of an array element anywhere you would use a simple variable or constant.

[7] When you want to use ***scanf( )*** to input a numeric value into an array element, simply put the **"&"** in front of the array name. For example, this call to ***scanf( )*** reads an integer into ***count[9].***

```
scanf("%d", &count[9]);
```

[8] C does not perform any bounds checking on array indexes. This means that **it is possible to overrun the end of an array**. For example, if an array called ***a*** is declared as having Five elements, the compiler will still let you access the (nonexistent) tenth element with a statement like ***a[9]*** Of course, attempting to access nonexistent elements will generally have disastrous results, often causing the program to crash. ***Make sure that the ends of arrays are never overrun*** .

```
Char a1[10], a2[10];
. . .
. . .
a1=a2;        /*This is wrong*/
```

If you wish to copy the values of all the elements of one array to another, you must do so by copying each element separately.

Example : One of the sorting algorithm is the ***bubble sort***. The bubble sort algorithm is not very efficient, but it is simple to understand and easy to code. The general concept behind the bubble sort, indeed how it got its name, is the repeated comparisons and, if necessary, exchanges of adjacent elements. This is a little like bubbles in a tank of water with each bubble, in turn, seeking its own level.

```
                    /*Sorting Numbers*/
#include<stdio.h>
int main(void){
    int s[100], count, i, j, x;
    printf("How many numbers ? : "); scanf("%d", &count);
    printf("\n Enter the numbers one by one. : "); for(i=0; i<count; i++) {scanf("%d", &s[i]);}
    /*Now sort them using bubble sort*/
    for(i=1; i<count; ++i)
       for(j=(count-1); j>=i; --j){
           /*Compare adjacent element*/
           if(s[j-1]>s[j]){ x=s[j];  s[j]=s[j-1];  s[j-1]=x; /*Exchange elements*/ }
                    /*Ascending : s[j-1]>s[j], Descending : s[j-1]<s[j]*/
             }
    for(i=0; i<count; i++) printf(" %d", s[i]);
    return 0;
}
```

4.2 USE STRINGS : gets(), 4-string functions, atoi(), STRING.H & STDLIB.H

The most common use of the one-dimensional array in C is the string. C has no built-in string data type. Instead, a string is defined as a ***null-terminated character array***. In C, a null is zero. So we must define the array that is going to hold a string to be one byte larger than the largest string it will be required to hold, to make room for the null. A string constant is null-terminated by the compiler automatically.

The gets() function : There are several ways to read a string from the keyboard. Here we use one of C's standard library functions: ***gets( )***. Like the other standard ***I/O*** functions, ***gets( )*** also uses the ***STDIO.H*** header file. To use ***gets( )***, call it using the name of a character array without any index. The ***gets( )*** function reads characters until you press ***ENTER***. The **ENTER key (i.e., carriage return)** is not stored, but is replaced by a null, which terminates the string. For example, this program reads a string entered at the keyboard. It then displays the contents of that string one character at a time.

```
char str[80] ;  int i;
printf("Enter a string (less than 80 chars): ");
gets (str);   /*Reads the string from the keyboard and store at "str" array*/
for(i=0; str[i]; i++) printf("%c", str[i]);
```

Notice how the program uses the fact that a null (means **str[i]** is zero/null or false) is false to control the loop that outputs the string (i.e. until the string reach to its end).

NOTE : *There is a potential problem with **gets( )**. The **gets( )** function performs no bounds checking, so it is possible for the user to enter more characters than the array receiving them can hold. For example, if you call **gets( )** with an array that is **20 characters** long, there is no mechanism to stop you from entering more than 20 characters. Which may cause program crash.* ***just be sure to call "gets( )" with an array that is more than large enough to hold the expected input.***

Output string directly using *printf() :* There is, of course, a much easier way to display a string using ***printf( )***, as shown in this segment:

```
printf("Enter a string (less than 80 chars): ");
gets (str);   /*Reads the string from the keyboard and store at "str" array*/
printf(str); /*Without any specifire*/
```

Recall that the first argument to ***printf( )*** is a string. Since ***str*** contains a string it can be used as the first argument to ***printf( )***. The contents of ***str*** will then be displayed.

- If you wanted to output other items in addition to ***str***, you could display ***str*** using the ***%s*** format code. For example, to output a ***newline*** after ***str***: `printf("%s\n", str);` This method uses the ***%s*** format ***specifier*** followed by the newline character and uses ***str*** as a ***second argument*** to be matched by the ***%s specifiers***.

4.2.1 Some string-related functions

The four most important string-related functions are ***strcpy( )***, ***strcat( )***, ***strcmp( )***, and ***strlen( )***. These functions require the header file ***STRING.H.***

[1] ***strcpy( ) :*** The ***strcpy( )*** function has this general form: `strcpy(to, from);`
It copies the contents of ***from*** to ***to***. The contents of ***from*** are unchanged. For example, this fragment copies the string "***hello***" into ***str*** and displays it on the screen:

```
char str[80];
strcpy (str, "hello" ) ; printf("%s", str);
```

The ***strcpy( )*** function performs **no bounds checking**, so you must make sure that the array on the receiving end is large enough to hold what is being copied, including the null terminator.

[2] ***strcat( ) & Concatenation :*** The ***strcat( )*** function adds the contents of one string to another. This is called ***Concatenation***. Its general form is: `strcat(to, from);`
It adds the contents of ***from*** to the contents of ***to***. It **performs no bounds checking** so you must make sure that ***to*** is large enough to hold its current contents plus what it will be receiving. This fragment displays **"hello there"**.

```
char str[80];
strcpy (str, "hello" ) ; strcat (str, "there" ) ; printf("%s", str);
```

[3] ***strcmp( ) :*** The ***strcmp( )*** function compares two strings. It takes this general form:

```
strcmp(s 1, s2);
```

This fragment prints 0, because the strings are the same:

```
printf("%d", strcmp("one", "one"));
```

It returns *zero if the strings are the same. It returns less than zero if s1 is less than s2 and greater than zero if s1 is greater than s2.* The strings are compared lexicographically; that is, in dictionary order. Therefore, a string is less than another when it would appear before the other in a dictionary. A string is greater than another when it would appear after the other. The comparison is not based upon the length of the string. Also, the comparison is case-sensitive, lowercase characters being greater than uppercase.

[4] ***strlen( ) :*** The ***strlen( )*** function returns the length, in characters, of a string. Its general form is: `strlen(str);`
The ***strlen( )*** function does not count the null terminator. This means that if ***strlen( )*** is called using the string ***'test'***, it will return ***4.***

NOTES

- One common use of strings is to support a command-based interface.
- The ***atoi( )*** function returns the integer equivalent of the number represented by its ***string argument***. For example, ***atoi("100")*** returns the ***value 100***. The reason that ***scanf( )*** is not used to read the numbers is because, in this context, ***scanf( )*** **is incompatible**

with *gets( )*. (Need to know more about C before one can understand the cause of this incompatibility.) The ***atoi( )*** function uses the header file ***STDLIB.H***.

- You can create a zero-length string using a ***strcpy( )*** statement like this:

```
strcpy(str, "");
```

Such a string is called a **null string**. It contains only one element: **the null terminator**.

4.3 Create multidimensional Arrays

we can create arrays of two or more dimensions. To add a dimension, we simply specify its size inside square brackets. For example, to create a 10x12 two-dimensional integer array called count :

```
int count[10] [12];
```

A two-dimensional array is essentially **an array of one-dimensional arrays** (***yep !! Array of arrays***) and is most easily thought of in a **row-column** format.

For ***example***, given a ***4x5*** integer array called two_d, looking like

```
int two_d[4][5], i, j;
for(i=0; i<4; i++){
    for(j=0; j<5; j++) {two_d[i][j]=i*j; /*printf(" %d", two_d[i][j]);*/}
    /*printf("\n");*/
    }

/*or print saperately */
for(i=0; i<4; i++){ for(j=0; j<5; j++) printf(" %d", two_d[i][j]);
                    printf("\n");}
```

	0	1	2	3	4
0					
1		1	2	3	4
2		2	4	6	8
3		3	6	9	12

A two-dimensional array is **accessed a row at a time, from left to right**. This means that the rightmost index will change most quickly(?) when the array is accessed sequentially from the lowest to highest memory address (`i.e for tst[i][j] : tst[0][0], tst[0][1], tst[0][2], tst[0][3], . . . etc. so j changes quickly.` **`In a row the column varies`**).

Three dimensions or greater : TO create arrays of three dimensions or greater, simply add the size of the additional dimension. For example, the following statement creates a **10x12x8** three-dimensional array.

```
float values[10][12][8];
```

A **three-dimensional array** is essentially an **array of two-dimensional arrays.**

NOTES

- You may create arrays of more than three dimensions, but this is seldom done because the amount of memory they consume increases exponentially with each additional dimension. For example, a **100**-character one-dimensional array requires **100** bytes of memory. A **100x100** character array requires **10,000** bytes, and a **100x100x100** array requires **1,000,000** bytes.

4.4 Initialize Arrays

Initialization of an array can be done by specifying a list of values the array elements will have. The general form of array initialization for one-dimensional arrays is:

```
type array_name[size] = { value-list }
```

The ***value-list*** is a comma-separated list of constants that are type compatible with the base type of the array. Moving from left to right, the first constant will be placed in the first position of the array, the second constant in the second position, and so on. In the following example, a **five-element integer array** is initialized with the squares of the numbers **1 through 5**.

```
int i[5] = {1, 4, 9, 16, 25};  /*This means i[0]=1 and  i[4]= 25*/
```

Initialize character arrays : There are two ways. ***First***, if the array is not holding a null-terminated string, we simply specify each character using a comma-separated list. For example, this initializes a with the letters 'A', 'B', and 'C'.

```
char a[3] = {'A', 'B', 'C'};
```

Second, If the character array is going to hold a string, you can initialize the array using a quoted string, as shown here:

```
char name[S] = "Herb";
```

Notice that no curly braces surround the string. **Braces are not used in this form of initialization : when a string constant is used, the compiler automatically supplies the null terminator**. Because strings in C must end with a null, you must make sure that the array you declare is long enough to include the null. This is why **name is 5 characters long, even though 'Herb' is only 4**.

Initialization of Multidimensional arrays : Multidimensional arrays are initialized in the same way as one-dimensional arrays. For example, here the array ***sqr*** is initialized with the values **1** through **9,** using row order:

```
int sqr[3] [3] = {  1, 2, 3,
                    4, 5, 6,
                    7, 8, 9  } ;
```

This initialization causes ***sqr[0][0]=1***, ***sqr[0][1]=2***, ***sqr[0][2]= 3*** and so forth.

Unsized arrays – Implicit way of initialization of arrays : To not specify the size of the array simply put nothing inside the square brackets. If you don't specify the size, the compiler counts the number of ***initializers*** and uses that value as the size of the array. For example, ***int*** *pwr[]* causes the compiler to create an initialized array eight elements long.

```
int pwr[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

Arrays that don't have their dimensions explicitly specified are called ***unsized arrays***. It helps avoid counting errors on long lists, which is especially important when initializing strings. For example, here an ***unsized*** array is used to hold a prompting message.

```
char prompt[]="Enter your name : ";
```

"Enter your name : " can be changed any time in future , no need to count the characters. The size of prompt will automatically adjust.

Multidimensional unsized array initializations : For multidimensional arrays you must specify all but the leftmost dimension to allow C to index the array properly (useful only for listed tables). In this way you may build tables of varying lengths with the compiler allocating enough storage for them automatically. For example, the declaration of ***sqr*** as an unsized array is shown here:

```
int sqr[] [3] = {   1, 2, 3,
                    4, 5, 6,
                    7, 8, 9  } ;
```

The advantage to this declaration over the sized version is that we can add more row into these tables.

4.5 ARRAYS OF STRINGS

Arrays of strings, often called ***string tables***, are very common in C programming. A string table is created like any other ***two-dimensional array***. For example, here is a small string table.

```
char names[10][40];
```

This statement specifies a table that can contain ***10*** strings, each up to ***40*** characters long (including the null terminator). To access a string within this table, ***specify only the left-most index***. For example, to read a string from the keyboard into the **third string** in ***names*** :

```
gets(names[2]);
```

By the same token, to output the **first string**, use this ***printf( )*** statement: `printf (names[0] ) ;`

Three-dimensional table of strings : `char animals[3][5][80];`

The declaration creates a three-dimensional table with three lists of strings. Each list is five strings long, and each string can hold 80 characters.

To access a specific string in this situation, you must ***specify the two left-most indexes***. For example, to access the **second string in the third list**, specify ***animals[2][1]***.

4.6 The POINTERS

Understanding pointer : Pointer is a kind of variable by which we can work with **other variables data** using **the location of the pointed variable**. It's a kind of indirect use of a variables data using its memory location.

POINTERs : A pointer is a variable that holds the ***memory address of another Object***. For example, if a variable called ***p*** contains the address of another variable called ***q***, then ***p*** is said to point to ***q*** . Therefore if ***q*** is at **location 100** in memory, then ***p*** would have the value ***100*** . To declare a pointer variable, use this general form:

```
type *var_name;
```

Here, ***type*** is the ***base type*** of the pointer. The base type specifies the type of the object that the pointer can point to. Notice that the variable name is preceded by an asterisk (also asterisk can be preceded by type : **`type*`** , The point is there is no space between **`"type & * "`** or **`" * & var_name "`**). This tells the computer that a pointer variable is being created.

For example : **`int *p;`** creates a pointer to an integer:.

The * and & operators for *point* and *return address* : C contains two ***special pointer operators***: ***** and ***&***. The ***&*** operator returns ***the address of the variable*** it precedes. The ***** operator ***returns the value*** stored at the address that it precedes. we can verbalize the ***"&"*** operator as "**address of**" and ***"*"*** operator as "**at address**". For example, examine this short program:

```
#include <stdio.h>
int main(void){     int *p, q;
                    q = 199;       /* assign q 199 */
                    p = &q;       /* assign p the address of q */
                    printf("%d\n", *p);  /* display q's value using pointer */
                    printf("q = %d, location = %x, *p = %d", q, p, *p);/* displays all */
                    return 0;}
```

First, the line: **`int *p, q;`** defines two variables: ***p***, which is declared as an ***integer pointer***, and ***q*** , which is ***an integer***. Next, ***q*** is assigned the value ***199***. In the next line, ***p*** is assigned the address of ***q***. This line can be read as "**assign p the address of q** ". Finally, the value is displayed using the ***** operator applied to ***p***. The ***printf( )*** statement can be read as "**print the value at address q**".

assigning values to a variables indirectly : It is possible to use the ***** operator on the left side of an assignment statement in order to assign a variable a new value given a pointer to it (i.e. ***assigning values to a variables indirectly using a pointer***). For example, this fragment assigns ***q*** a value indirectly using the pointer ***p*** :

```
int *p, q;
p = &q:       /* get q's address */
*p = 199;     /* assign q a value using a pointer */
```

The specifires %p and %x , Difference between %p and %x : Functions belonging to the ***printf*** function family have the type ***specifiers*** "***%p***" and "***%x***".

- "***x***" and "***X***" serve to output a ***hexadecimal*** number, "***x***" stands for ***lower case*** letters (***abcdef***) while "***X***" for ***capital letters*** (***ABCDEF***).
- "***p***" serves to output a ***pointer***. It may differ depending upon the compiler and platform (*32bit* or *64bit*).

One **specifier** is often used instead of another on 32-bit systems, but it is a mistake. Here is an example:

```
int a = 10;
int *b = &a;
printf("%p\n",b);
printf("%X\n",b);
```

On a Win32 system, the following result will be printed:

```
0018FF20
18FF20
```

As you may see, the output results for "***%p***" and "***%X***" are rather similar. This similarity leads to inaccuracy in the code and this, in turn, results in **errors occurring when you port a program to a 64-bit platform** (in 64 bit may get : ***000018FF20*** for X). Most often it is "***%X***" that is used instead of "***%p***" to output the value of a ***pointer***, and **this results in printing of an incorrect value** if the object is situated outside the four less significant Gbytes of the address space. So ***it is better to use " %p " to display the memory address***.

NOTES

[1] When a variable's value is referenced through a pointer, the process is called indirection.

[2] There is no reason to use a pointer in previous 2 example. Pointers are used to support linked lists and binary trees, for example.

[3] The ***base type*** of a pointer is very important. Although C allows any type of pointer to point anywhere in memory, it is the base type that determines **how the object pointed to will be treated**. To understand the importance of this, consider the following fragment:

```
int q; double *fp;
fp = &q;
*fp = 100.23; /*Assigning double type value to a int type variable using pointer*/
```

Here ***fp*** assigns the location(address) of an ***int*** type variable but ****fp's*** base type is ***duoble***. ****fp*** assigns ***double*** type data (***floating-point value***) to the ***int*** type veriable ***q***. Although not syntactically incorrect, this fragment is wrong.

We know ***ints*** are usually shorter than ***doubles***, and this assignment statement causes memory adjacent to q to be overwritten. Assume that ***integers*** are ***2 bytes*** and doubles are ***8 bytes***, the assignment statement uses the 2 bytes allocated to ***q*** as well as 6 adjacent bytes, thus causing an error.

[4] it is very important that you always use the **proper base type for a pointer**. Except in special cases, ***never use a pointer of one type to point to an object of a different type***.

[5] If you attempt to use a pointer before it has been assigned the address of a variable, your program will probably crash. In most case both ******* and ***&*** operators appear in programs, they don't appear alone. Remember, declaring a pointer variable simply creates a variable capable of holding a memory address. It does not give it any meaningful initial Value. This is why the following fragment is incorrect.

```
int *p;
*p=10;  /* incorrect - p is not pointing to anything "There is no location to assign 10"* /
```

Here the ***pointer p*** is not pointing to any known object. Hence, trying to indirectly assign a value using ***pointers*** is meaningless and dangerous.

[6] a pointer that contains a null value (zero) is assumed to be unused and pointing at nothing. Although compiler admits a ***null pointer***, usually with disastrous results.

[7] A pointer type declaration has two pats :

Part : 1	***Part : 2***
type *Pointer_ver_name	
type *	***Pointer_ver_name***
int *s_time	
int *	***s_time***

4.7 Restriction to Pointer Expression

In general, pointers may be used like other variables. However, there are few rules and restrictions.

- A In addition to the ******* and ***&*** operators, there are only four other operators that may be applied to pointer variables: the arithmetic operators ***+, ++, -,*** and ***- -***.
- We may ***add or subtract only integer*** quantities. But cannot add a floating-point number to a pointer.

[1] ***Pointer arithmetic differs from "normal" arithmetic in one very important way: it is performed relative to the "base type" of the pointer***. Each time a pointer is incremented, it will point to the next item, as defined by its base type, beyond the one currently pointed to. For example,

(a) Assume that an ***integer pointer*** called ***p*** contains the address **200**. After the statement ***p++;*** executes, ***p*** will have the value **202**, assuming **integers are two bytes long**.

(b) By the same token, if ***p*** had been a ***float pointer*** (assuming **4-byte floats**), then the resultant value contained in ***p*** would have been **204** .

(c) The only pointer arithmetic that appears as "***normal***" occurs when ***char pointers*** are used. Because characters are **one byte long**, an increment increases the pointer's value by one, and a decrement decreases its value by one.

[2] You may ***add or subtract any integer quantity to or from a pointer***. For example, the following is a valid fragment:

```
int *p;  . . . . . .   p = p + 200;
```

This statement causes p to point to the 200th integer past the one to which p was previously pointing.

[3] you may subtract one pointer from another in order to find the number of elements separating them.

[4] you may not ***multiply***, ***divide***, or take the ***modulus*** of a pointer.

[5] It is possible to ***apply the increment and decrement operators*** to either the ***pointer itself*** or ***the object to which it points***. However, you must **be careful when attempting to modify the object pointed to by a pointer**. For example, assume that p points to an integer that contains the value ***1***.

(a) What do you think the following statement will do?

```
*p++ ;
```

This statement first increments ***p*** and then obtains the value at the new location (***incremented the pointer location***).

(b) To ***increment what is pointed*** to by a pointer, you must use a form like this:

```
(*p) ++;
```

The parentheses cause the ***value pointed to*** by ***p*** (i.e. ***the pointed variable***) to be incremented.

[6] You may **compare two pointers using the relational operators**. However, pointer comparisons make sense only **if the pointers relate to each other-if they both point to the same object**, for example.

- You may also compare a **pointer to zero** to see if it is a ***null pointer***.

4.8 POINTERS WITH ARRAYS

In C, ***pointers*** and ***arrays*** are ***closely related***. In fact, they are often ***interchangeable***. It is this relationship between the two that makes ***their implementation both unique and powerful***. This fact is crucial to understanding the C language and it is the most important feature of C.

- When you use an ***array name*** without an index, you are generating a pointer to the start of the array. This is why no indexes are used when you read a string using ***gets( )***, for example. What is being passed to ***gets( )*** is not an array, but a pointer. In fact, ***you cannot pass an array to a function in C; you may only pass a pointer to the array***. The ***gets( )*** function uses the pointer to load the array it points to with the characters you enter at the keyboard. We'll see how this is done later.

- Since ***an array name without an index is a pointer to the start of the array***, it stands to reason that you can assign that value to another pointer and access the array using ***pointer arithmetic***. And, in fact, this is exactly what you can do. Consider this program:

```
#include <stdio.h>
int main(void){     int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
                    int *p;
                    p = a;        /* assign p the address of start of a */
                          /* this prints a's first, second and third elements */
                    printf("%d %d %d\n", *p, *(p+1), *(p+2));
                    /* this does the same thing using a */
                    printf ( "%d %d %d", a[0], a[1], a[2]);
                    return 0;}
```

Here, both ***printf( )*** statements display the same thing. The parentheses in expressions such as ****(p+2)*** are necessary because the ******* has a higher precedence than the ***+*** operator. That's why pointer arithmetic is done relative to the base type-it allows arrays and pointers to relate to each other.

To access multidimensional arrays : To use a pointer to access multidimensional arrays, you must manually do what the compiler does automatically. For example, in this array:

```
float balance[10][5];
```

each row is five elements long. Therefore, to access ***balance[3][1]*** using a pointer you must use a fragment like this:

```
float *p;
p = (float *) balance;     /*assigning the balance array to p with type cast */
printf("%d", *(p + (3*5) + 1);    /*accessing balance[3][1]*/
```

Here we converted the two dimensional array to an one dimensional array. If we look closely we see that the position of an element in an array is :

$$(position_{row} - 1) \times Totalcolumn + postion_{column}$$

for example position of ***a[i][j]*** element of $m \times n$ 2-D array is $(i-1)n + j$ or $n(i-1) + j$. Hence the position of ***balance[3][1]*** is $(4-1)5 + 2 = 3 * 5 + 2 = 16$ (we used ***4*** & ***2*** because array in C starts from ***0***). Again we know that to point k^{th} element of an one-dimensional array by a pointer p we use ****(p+(k-1))***, hence ****(p+(3*5)+1)*** since ***k=(3*5)+2***. So in short way to reach the desired element using pointer, you must ***multiply the row number by the number of elements in the row and then add the number of the element within the row*** **Eg:** $a[i][j] of\ a[m][n] : i * n + j;\ element\ in\ a\ row = n$. Generally, with multidimensional arrays it is ***easier to use array indexing rather than pointer arithmetic***.

The cast of ***balance*** to ***float**** was necessary. Since the array is being indexed manually, the pointer arithmetic must be relative to a ***float*** pointer (**to access the location of float type values**). However, the type of pointer generated by ***balance*** is to a two-dimensional array of ***floats***. Thus, there is need for the cast.

- We can ***index a pointer as if it were an array***. The following program, for example, is perfectly valid:

```
#include <stdio.h>
int main(void){     char str[] = "Pointers are fun";
                    char *p;
                    int i;
                    p = str; /*Assigning pointer to string*/
                    /* loop until null is found i.e. untill p[i]=0*/
                    for(i=0; p[i]; i++)
                    printf("%c", p[i]);  /*using pointer as string*/
                    return 0;}
```

NOTES

[1] ***You should index a pointer only when that pointer points to an array***. While the following fragment is syntactically correct, It is wrong; If you tried to execute it, you would probably crash your computer.

```
char *p, ch;  int i;
p = &ch;
for(i=0; i<10; i++) p[i] = 'A'+i; /* wrong */
```

Since ***ch*** is not an ***array***, it cannot be meaningfully indexed.

[2] Pointer arithmetic is usually more convenient. Also, in some cases a C compiler can generate faster executable code for an expression involving pointers than for a comparable expression using arrays.

[3] Because an array name without an index is a pointer to the start of the array, you can, if you choose, ***use pointer arithmetic rather than array indexing to access elements of the array***. For example, this program is perfectly valid and prints c on the screen:

```
#include <stdio.h>
int main(void) {char str[80];
                *(str+3) = 'c';
                printf("%c", *(str+3));
                return 0;}
```

- You cannot, however, modify the value of the pointer generated by using an array name. For example, assuming the previous program, this is an invalid statement:

```
str++;
```

The pointer that is generated by str must be thought of as a constant that always points to the start of the array. Therefore, it is invalid to modify it and the compiler will report an error.

[4] ***Lowercase-Uppercase transform function with CTYPE.H header :*** Two of C's library functions, ***toupper()*** and ***tolower(),*** are called using a character argument. These functions use the header file ***CTYPE.H***.

- In the case of ***toupper( )***, if the character is a lowercase letter, the uppercase equivalent is returned; otherwise the character is returned unchanged.
- For ***tolower( )***, if the character is an uppercase letter, the lowercase equivalent is returned; otherwise the character is returned unchanged.

4.9 Use pointers to string constants

C allows string constants enclosed between double quotes to be used in a program. When the compiler encounters such a string, it stores it in the program's string table and ***generates a pointer to the string***. For this reason, the following program is correct and prints ***one two three*** on the screen .

```
char *p;
p = "one two three"; printf(p);
```

How this program works : First, ***p*** is declared as a character pointer. This means that it may point to an array of characters. When the compiler compiles the line ***p = "one two three";*** it stores the string in the ***program's string table and assigns to p the address of the string in the table***. Therefore, when ***p*** is used in the ***printf( )*** statement, **one two three** is displayed on the screen.
This program can be written more efficiently, as shown here:

```
char *P = "one two three";
printf (p) ;
```

Here , ***p*** is initialized to point to the string.

NOTE

Using pointers to string constants can be very helpful when those constants are quite long. For example suppose that you had a program that at various times would prompt the user to insert a diskette into drive A. Using a pointer has following advantages

- To save yourself some typing, you might elect to initialize a pointer to the string and then simply use the pointer when the message needed to be displayed; for example:

```
char *InsDisk = "Insert disk into drive A then press ENTER ";
. . . .
printf(InsDisk);
. . . .
printf(InsDisk);
```

- Another advantage to this approach is that to change the prompt, you only need to change it once, and all references to it will reflect the change.

4.10 Arrays of Pointers

Pointers may be arrayed like any other data type. For example, the following statement declares an integer pointer army that has ***20*** elements:

```
int *pa[20];
```

- The ***address*** of an integer variable called ***myvar*** is ***assigned*** to the ninth element of the array as follows:

```
pa[8] = &myvar:; /*Assigning the address*/
```

Because pa is an array of pointers, the only values that the array elements may hold are the addresses of integer variables.

- To assign the integer pointed to by the third element of pa the value 100, use the statement:

```
*pa[2] = 100; /*Assigning values by pointer*/
```

NOTE

Common use of arrays of pointers is to create string tables in much the same wav that unsized arrays were used in the previous section.

4.11 Multiple INDIRECTION

It is possible in C to have a pointer point to another pointer. This is called ***multiple indirection***. When a pointer points to another pointer, the first pointer contains the address of the second pointer, which points to the location containing the object.

- To declare a pointer to a pointer, an additional asterisk is placed in front of the pointer's name For example, this declaration tells the compiler that nip is a pointer to a character pointer:

  ```
  char **mp;
  ```

  It is important to understand that ***mp*** is not a pointer to a character but rather a ***pointer to a character pointer***.

- Accessing the target value indirectly pointed to ***by a pointer to a pointer*** requires that the ***asterisk*** operator be applied ***twice***. For example,

  ```
  char **mp, *p, ch;
  p = &ch;      /* get address of ch */
  mp = &p;      /* get address of p */
  **mp = 'A';   /* assign ch the value A using multiple indirection */
  ```

  As the comments suggest, ch is assigned a value indirectly using two pointers.

NOTE

- ***Not Recommended :*** Multiple indirection is not limited to merely "a pointer to a pointer." You can apply the ******* as often as needed. However, ***multiple indirection*** beyond a pointer to a pointer is very difficult to follow and is ***not recommended***.
- As you learn more about C, you will see some examples in which it is very valuable.

4.12 Pointers as Parameters

Pointers may be passed to functions. For example, when you call a function like ***strlen()*** with the name of a string, you are actually passing a pointer to a function.

- When you pass a pointer to a function, the ***function must be declared as receiving a pointer of the same type***. In the case of ***strlen()***, this is a character pointer. EG: ***void my_func(char *p).*** `/* pointer of the same type */`
- When you pass a pointer to a function, the code inside that function has access to the variable pointed to by the parameter. This means that the function can change the variable used to call the function. This is why functions like strcpy( ), for example, can work. Because it is passed a pointer, the function is able to modify the array that receives the string.

NOTE

[1] ***& in scanf():*** why you need to precede a variable's name with an ***&*** when using ***scanf()*** ? In order for ***scanf()*** to modify the value of one of its arguments, ***it must be passed a pointer to that argument.***

[2] ***puts() :***Another of C's standard library' functions is called ***puts( );*** it writes its string argument to the screen followed by a newline.

[3] When the compiler encounters a string constant, it places it into the programs string table and generates a pointer to it.