We can print this address through the following program:

```
/* Program 1 */
#include <stdio.h>
int main( )
{
      int i = 3 ;

      printf ( "Address of i = %u\n", &i ) ;
      printf ( "Value of i = %u\n", i ) ;
      return 0 ;
}
```

The output of the above program would be:

```
Address of i = 6485
Value of i = 3
```

```
/* Program 2 */
#include <stdio.h>
int main( )
{
    int i = 3 ;

    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of i = %d\n", *( &i ) ) ;
    return 0 ;
}
```

The output of the above program would be:

```
Address of i = 6485
Value of i = 3
Value of i = 3
```

Pointer Expressions

Let us now see what are pointers and how they can be used in various expressions. We have seen in the previous section that the expression **&i** returns the address of **i**. If we so desire, this address can be collected in a variable by saying,

```
j = &i ;
```

But remember that **j** is not an ordinary variable like any other integer variable. It is a variable, which contains the address of another variable (**i** in this case).

Since **j** is a variable the compiler must provide it space in memory. Once again, the following memory map would illustrate the contents of **i** and **j**.

Let us go by the meaning of *. It stands for 'value at addres
Thus, **int *j** would mean, the value at the address contained in j
an **int**.

Here is a program that demonstrates the relationships we hav
been discussing.

```
/* Program 3 */
#include <stdio.h>
int main( )
{
    int i = 3 ;
    int *j ;

    j = &i ;
    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n", j ) ;
    printf ( "Address of j = %u\n", &j ) ;
    printf ( "Value of j = %d\n", j ) ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of i = %d\n", *( &i ) ) ;
    printf ( "Value of i = %d\n", *j ) ;
    return 0 ;
}
```

The output of the above program would be:

```
Address of i = 6485
Address of i = 6485
```

```
/* Program 4 */
#include <stdio.h>
int main( )
{
    int i = 3 ;
    int *j ;
    int **k ;

    j = &i ;
    k = &j ;

    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n", j ) ;
    printf ( "Address of i = %u\n", *k ) ;
    printf ( "Address of j = %u\n", &j ) ;
    printf ( "Address of j = %u\n", k ) ;
    printf ( "Address of k = %u\n\n", &k ) ;

    printf ( "Value of j = %u\n", j ) ;
    printf ( "Value of k = %u\n", k ) ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of i = %d\n", *( &i ) ) ;
    printf ( "Value of i = %d\n", *j ) ;
    printf ( "Value of i = %d\n", **k ) ;
    return 0 ;
}
```

pointer will store address of variable

double pointer will store address of a pointer

The output of the above program would be:

```
Address of i = 6485
Address of i = 6485
Address of i = 6485
```

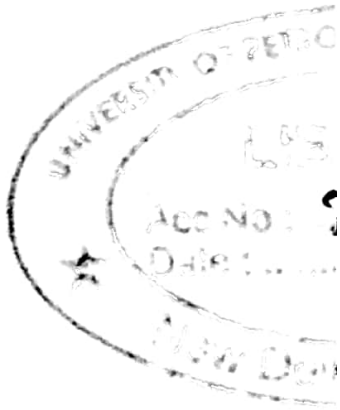
Address of j = 3276
Address of j = 3276
Address of k = 7234

Value of j = 6485
Value of k = 3276
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3

```
/* Program 5 */
#include <stdio.h>
int main( )
{
    char c, *cc ;
    int i, *ii ;
    float a, *aa ;

    c = 'A'  ; /* ascii value of A gets stored in c */
    i = 54 ;
    a = 3.14 ;
    cc = &c ;
    ii = &i ;
    aa = &a ;
    printf ( "Address contained in cc = %u\n", cc ) ;
    printf ( "Address contained in ii = %u\n", ii ) ;
    printf ( "Address contained in aa = %u\n", aa ) ;
    printf ( "Value of c = %c\n", *cc ) ;
    printf ( "Value of i = %d\n", *ii ) ;
    printf ( "Value of a = %f\n", *aa ) ;
    return 0 ;
}
```

And here is the output...



```
Address contained in cc = 1004
Address contained in ii = 2008
Address contained in aa = 7006
Value of c = A
Value of i = 54
Value of a = 3.140000
```

```
{
    int i = 54 ;
    float a = 3.14 ;
    char *ii, *aa  ;

    ii = ( char * ) &i ;
    aa = ( char * ) &a ;
    printf ( "Address contained in ii = %u\n", ii ) ;
    printf ( "Address contained in aa = %u\n", aa ) ;
    printf ( "Value at the address contained in ii = %d\n", *ii ) ;
    printf ( "Value at the address contained in aa = %d\n", *aa ) ;
    return 0 ;
}
```

The following program illustrates the 'Call by Value'.

```
/* Program 7 */
#include <stdio.h>
void swapv ( int, int ) ;
int main( )
{
    int a = 10 ;
    int b = 20 ;

    swapv ( a, b ) ;
    printf ( "a = %d\n", a ) ;
    printf ( "b = %d\n", b ) ;
    return 0 ;
}

void swapv ( int x, int y )
{
    int t ;

    t = x ;
    x = y ;
    y = t ;

    printf ( "x = %d\n", x ) ;
    printf ( "y = %d\n", y ) ;
}
```

```
/* Program 8 */
#include <stdio.h>
void swapr ( int *, int * ) ;

int main( )
{
        int a = 10 ;
        int b = 20 ;

        swapr ( &a, &b ) ;
        printf ( "a = %d\n", a ) ;
        printf ( "b = %d\n", b ) ;
        return 0 ;
}

void swapr ( int *x, int *y )
{
        int t ;
```

```
    t = *x ;
    *x = *y ;
    *y = t ;
}
```

The output of the above program would be:

```
a = 20
b = 10
```

Using 'call by reference' intelligently we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below.

```
/* Program 9 */
#include <stdio.h>
void areaperi ( int, float *, float * ) ;

int main( )
{
    int radius ;
    float area, perimeter ;

    printf ( "Enter radius of a circle\n" ) ;
    scanf ( "%d", &radius ) ;
    areaperi ( radius, &area, &perimeter ) ;
    printf ( "Area = %f\n", area ) ;
    printf ( "Perimeter = %f\n", perimeter ) ;
    return 0 ;
}

void areaperi ( int r, float *a, float *p )
{
    *a = 3.14 * r * r ;
    *p = 2 * 3.14 * r ;
}
```

Functions Returning Pointers

The way functions return an **int**, a **float**, a **double** or any other data type, it can even return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function definition. The following program illustrates this.

```
/* Program 10 */
#include <stdio.h>
int *fun( ) ;

int main( )
{
```

```
    int *p ;
    p = fun( ) ;
    printf ( "%u\n", p ) ;
    printf ( "%d\n", *p ) ;
    return 0 ;
}

int *fun( )  /* function definition */
{
    int i = 20 ;
    return ( &i ) ;
}
```

This program shows how a pointer can be returned from a function. Note that the prototype declaration tells the compiler that **fun( )** is a function which receives nothing but returns an integer pointer. The first **printf( )** would output the address contained in p (address of **i**). Can you guess what the second **printf( )** would output? No, it won't print 20. This is because, when the control comes back from **fun( )**, **i** dies. So even if we have its address in p we can't access **i** since it is already dead. If you want **i** to survive and ***p** to give 20 then make sure that you declare **i** as **static** as shown below:

```
static int i = 20 ;
```

[A] What will be the output of the following programs:

(1)
```
#include <stdio.h>
void fun ( int, int * ) ;

int main( )
{
    int i = -5, j = -2 ;
    fun ( i, &j ) ;
    printf ( "i = %d j = %d\n", i, j ) ;
    return 0 ;
}

void fun ( int i, int *j )
{
    i = i * i ;
    *j = *j * *j ;
}
```

Output

```
i = -5 j = 4
```

Explanation

One doubt immediately comes to the mind—can we use same variable names in different functions? Yes, by all means, without absolutely any conflict. Thus, the two sets of **i** and **j** are two totally different sets of variables. While calling the function **fun( )** the value of **i** and the address of **j** are passed to

it. Naturally, in **fun( ) i** is declared as an ordinary **int**, whereas **j** is declared as a pointer to an **int.**

Even though the value of **i** is changed to 25 in **fun( )**, this change will not be reflected back in **main( )**. As against this, since **j**'s address is being passed to **fun( )**, any change in **fun( )** gets reflected back in **main( ).** Hence ***j * *j**, which evaluates to 4 is reflected back in **main( ).**

(2)
```
#include <stdio.h>
int main( )
{
    int a, b = 5 ;
    a = b + NULL ;
    printf ( "%d\n", a ) ;
    return 0 ;
}
```

Output

5

Explanation

NULL has been defined in "stdio.h" as follows:

```
#define NULL 0
```

Hence, during preprocessing NULL will be replaced by 0, resulting into 5 getting stored in **a**.

(3)
```
#include <stdio.h>
int main( )
```

```
{
    printf ( "%d %d\n", sizeof ( NULL ), sizeof ( " " ) ) ;
    return 0 ;
}
```

Output

```
2 1
```

Explanation

While finding out size of **NULL**, we are truly speaking finding out size of 0. This is an integer, hence its size is reported as 2 bytes.

Even though the string "" is empty it still contains the character, '\0'. Hence its size turns out to be 1 byte.

(4)
```
#include <stdio.h>
int main( )
{
    float a = 7.999999 ;
    float *b,*c ;
    b = &a ;
    c = b ;
    printf ( "%u %u %u\n", &a, b, c ) ;
    printf ( "%d %d %d %d\n", a, *( &a ), *b, *c ) ;
    return 0 ;
}
```

Output

```
4200 4200 4200
```

(3) Are the expressions ***ptr++** and ***++ptr** same?

Explanation

No. ***ptr++** increments the pointer and not the value pointed by it, whereas, ***++ptr** increments the value being pointed to by **ptr**.

(9) What is a null pointer?

Explanation

For each pointer type (like say a **char** pointer) C defines a special pointer value that is guaranteed not to point to any object or function of that type. Usually, the null pointer constant used for representing a null pointer is the integer 0.

```
/* Program 14 */
#include <stdio.h>
void disp ( int * ) ;

int main( )
{
    int i ;
    int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

    for ( i = 0 ; i <= 6 ; i++ )
        disp ( &marks[i] ) ;
    return 0 ;
}

void disp ( int *n )
{
    show ( &n ) ;
}
```

Pointers and Arrays

To be able to see what pointers have got to do with arrays, let u first learn some pointer arithmetic. Consider the followin example:

```
/* Program 15 */
#include <stdio.h>
int main( )
{
    int i = 3, *x ;
    float j = 1.5, *y ;
```

```
    char k = 'c', *z ;

    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of j = %f\n", j ) ;
    printf ( "Value of k = %c\n\n", k ) ;

    x = &i ;
    y = &j ;
    z = &k ;

    printf ( "Original value in x = %u\n", x ) ;
    printf ( "Original value in y = %u\n", y ) ;
    printf ( "Original value in z = %u\n\n", z ) ;

    x++ ;
    y++ ;
    z++ ;

    printf ( "New value in x = %u\n", x ) ;
    printf ( "New value in y = %u\n", y ) ;
    printf ( "New value in z = %u\n\n", z ) ;
    return 0 ;
}
```

Suppose **i**, **j** and **k** are stored in memory at addresses 1002, 2004 and 5006, the output would be...

```
Value of i = 3
Value of j = 1.500000
Value of k = c

Original value in x = 1002
Original value in y = 2004
Original value in z = 5006

New value in x = 1004
New value in y = 2008
```

(a) Addition of two pointers
(b) Multiplying a pointer with a number
(c) Dividing a pointer with a number

Now we will try to correlate the following two facts, which we have already learnt:

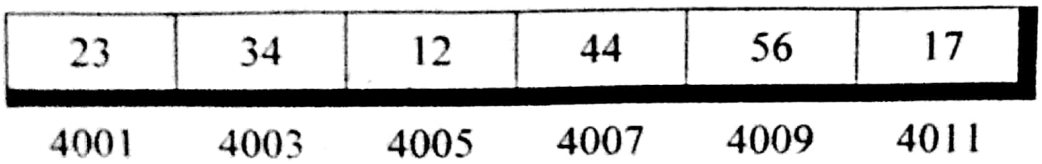
(a) Array elements are always stored in contiguous memory locations.

(b) A pointer when incremented always points to an immediately next location of its type.

Suppose we have an array,

```
int num[ ] = { 23, 34, 12, 44, 56, 17 } ;
```

The following figure shows how this array is located in memory.



```
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0 ;

    while ( i <= 5 )
    {
        printf ( "element no. %d\t", i ) ;
        printf ( "address = %u\n", &num[i] ) ;
        i++ ;
    }
    return 0 ;
}
```

The output of this program would be:

```
element no. 0     address = 4001
element no. 1     address = 4003
element no. 2     address = 4005
element no. 3     address = 4007
element no. 4     address = 4009
element no. 5     address = 4011
```

```
    int i = 0 ;

    while ( i <= 5 )
    {
        printf ( "address = %u\t", &num[i] ) ;
        printf ( "element = %d\n" , num[i] ) ;
        i++ ;
    }
    return 0 ;
}
```

The output of this program would be:

```
address = 4001    element = 24
address = 4003    element = 34
address = 4005    element = 12
address = 4007    element = 44
address = 4009    element = 56
address = 4011    element = 17
```

The next method accesses the array elements using pointers.

```
/* Program 18 */
#include <stdio.h>
int  main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0, *j ;

    j = &num[0] ;  /* assign address of zeroth element */

    while ( i <= 5 )
    {
        printf ( "address = %u\t" , &num[i] ) ;
        printf ( "element = %d\n", *j ) ;
```

```
        i++ ;
        j++ ; /* increment pointer to point to next location */
    }
    return 0 ;
}
```

The output of the program would look like this:

```
address = 4001   element = 24
address = 4003   element = 34
address = 4005   element = 12
address = 4007   element = 44
address = 4009   element = 56
address = 4011   element = 17
```

In this program, to begin with we have collected the base address of the array (address of 0^{th} element) in the variable **j** using the statement,

```
j = &num[0] ; /* assigns address 4001 to j */
```

When we are inside the loop for the first time **j** contains the address 4001, and the value at this address is 24. These are printed using the statements,

```
printf ( "address = %u\t", &num[i] ) ;
printf ( "element = %d\n", *j ) ;
```

Passing an Entire Array to a Function

Earlier we saw two programs one in which we passed individual elements of an array to a function, and another in which we passed addresses of individual elements to a function. Let us now see how to pass the entire array to the function rather than individual elements. Consider the following example:

```
/* Program 19 */
#include <stdio.h>
void display ( int *, int ) ;

int main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    display ( &num[0], 6 ) ;
    return 0 ;
}

void display ( int *j, int n )
```

```
{
    int i = 1 ;
    while ( i <= n )
    {
        printf ( "element = %d\n", *j ) ;
        i++ ;
        j++ ;  /* increment pointer to point to next location */
    }
}
```

```
/* Program 20 */
/* Accessing array elements in different ways */
#include <stdio.h>
int main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0 ;
```

```
    while ( i <= 5 )
    {
        printf ( "address = %u\t" , &num[i] ) ;
        printf ( "element = %d ", num[i] ) ;
        printf ( "%d ", *( num + i ) ) ;
        printf ( "%d ", *( i + num ) ) ;
        printf ( "%d\n", i[num] ) ;
        i++ ;
    }
    return 0 ;
}
```

The output of the program would look like this:

```
address = 4001   element = 24 24 24 24
address = 4003   element = 34 34 34 34
address = 4005   element = 12 12 12 12
address = 4007   element = 44 44 44 44
address = 4009   element = 56 56 56 56
address = 4011   element = 17 17 17 17
```

More Than One Dimension

So far we have looked at arrays with only one dimension. It possible for arrays to have two or more dimensions. The dimensional array is also called a matrix. Here is a program that initializes a 2-D array and prints out its elements

```
/* Program 21 */
#include <stdio.h>
int main( )
{
      int stud[5][2] = {
                              { 1234, 56 },
                              { 1212, 33 },
```

```
                    { 1434, 80 },
                    { 1312, 78 }
                };
    int i, j ;

    for ( i = 0 ; i <= 3 ; i++ )
    {
        printf ( "\n" ) ;
        for ( j = 0 ; j <= 1 ; j++ )
            printf ( "%d ", stud[i][j] ) ;
    }
    return 0 ;
}
```

Look at the **printf( )** statement...

```
printf ( "%d ", stud[i][j] ) ;
```

In **stud[i][j]** the first subscript is row number. The second subscript tells which of the two columns are we talking about... the zeroth column or the first column. Remember that counting of rows and columns begins with zero.

The complete array arrangement is shown below:

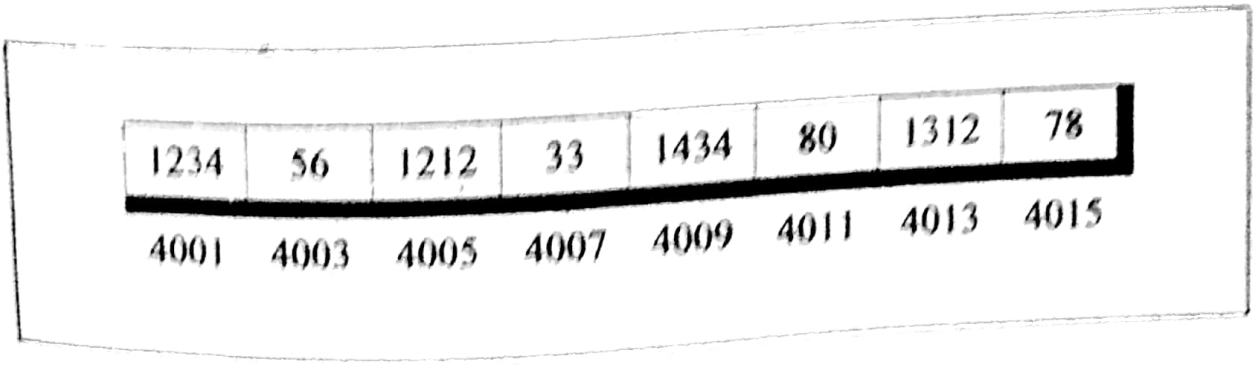


Figure 2.3

Thus, 1234 is stored in **stud[0][0]**, 56 is stored in **stud[0][1]** and so on. The above arrangement highlights the fact that a two-

This fact can be illustrated by the following program:

```
/* Program 22 */
/* Refer figure 2.3 given in the previous section */
#include <stdio.h>
int main( )
{
      int stud[5][2] = {
                              { 1234, 56 },
                              { 1212, 33 },
                              { 1434, 80 },
                              { 1312, 78 },
                              { 1203, 75 }
                            } ;
       int i, j ;

       for ( i = 0 ; i <= 4 ; i++ )
            printf ( "Address of %d th 1-D array = %u\n", i, stud[i] ) ;
       return 0 ;
  }
```

And here is the output...

```
Address of 0 th 1-D array = 4001
Address of 1 th 1-D array = 4005
Address of 2 th 1-D array = 4009
Address of 3 th 1-D array = 4013
Address of 4 th 1-D array = 4017
```

```
/* Program 23 */
#include <stdio.h>
int main( )
{
    int stud[5][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 },
                        { 1203, 75 }
                    } ;
    int i, j ;

    for ( i = 0 ; i <= 4 ; i++ )
    {
        printf ( "\n" ) ;
        for ( j = 0 ; j <= 1 ; j++ )
            printf ( "%d\t", *( *( stud+ i ) + j ) ) ;
    }
    return 0 ;
}
```

And here is the output...

```
1234    56
1212    33
1434    80
1312    78
1203    75
```

```
/* Program 24 */
#include <stdio.h>
int main( )
{
      int a[ ][4] =    {
                              5, 7, 5, 9,
                              4, 6, 3, 1,
                              2, 9, 0, 6
                       } ;
      int *p ;
      int ( *q )[4] ;

      p = ( int* ) a ;
      q = a ;

      printf ( "%u %u\n", p, q ) ;
      p++ ;
      q++ ;
      printf ( "%u %u\n", p, q ) ;
      return 0 ;
}
```

And here is the output...

```
65500 65500
65502 65508
```

Passing 2-D Array to a Function

There are three ways in which we can pass a 2-D array to a function. These are illustrated in the following program.

```
/* Program 25 */
/* Three ways of accessing a 2-D array */
#include <stdio.h>
#include <alloc.h>
void display ( int *, int, int ) ;
void show ( int ( *q )[4], int row, int col ) ;
void print ( int q[ ][4], int row, int col ) ;

int main( )
{
    int a[3][4] = {
                        1, 2, 3, 4,
                        5, 6, 7, 8,
                        9, 0, 1, 6
                  } ;

    clrscr( ) ;
    display ( a, 3, 4 ) ;
    show ( a, 3, 4 ) ;
    print ( a, 3, 4 ) ;
    return 0 ;
}

void display ( int *q, int row, int col )
{
    int i , j ;

    for ( i = 0 ; i < row ; i++ )
    {
```

```
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( q + i * col + j ) ) ;

        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}

void show ( int ( *q )[4], int row, int col )
{
    int i, j ;
    int *p ;

    for ( i = 0 ; i < row ; i++ )
    {
        p = q + i ;
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( p + j ) ) ;

        printf( "\n" ) ;
    }
    printf( "\n" ) ;
}

void print ( int q[ ][4], int row, int col )
{
    int i, j ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] ) ;

        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}
```

```
/* Program 26 */
#include <stdio.h>
int main( )
{
    int a[2][3][2] = {
                        {
                            { 2, 4 },
                            { 7, 8 },
                            { 3, 4 }
                        },
                        {
                            { 2, 2 },
                            { 2, 3 },
                            { 3, 4 }
                        }
                    };

    printf ( "%u\n", a ) ;
    printf ( "%u\n", *a ) ;
    printf ( "%u\n", **a ) ;
    printf ( "%d\n", ***a ) ;
    printf ( "%u\n", a + 1 ) ;l
    printf ( "%u\n", *a + 1 ) ;
    printf ( "%u\n, **a + 1 ) ;
    printf ( "%d\n", ***a + 1 ) ;
    return 0 ;
}
```

Passing 3-D Array to a Function

There are three ways in which we can pass a 3-D array to a function. These are illustrated below.

```
/* Program 27 */
/* Three ways of passing a 3-D array to function */
#include <stdio.h>
void display ( int *q, int ii, int jj, int kk ) ;
void show ( int ( *q )[3][4], int ii, int jj, int kk ) ;
void print ( int q[ ][3][4], int ii, int jj, int kk ) ;

int main( )
{
    int i, j, k ;
    int a[2][3][4] = {
                        {
                            1, 2, 3, 4,
                            5, 6, 7, 8,
                            9, 3, 2, 1
                        },
                        {
                            2, 3, 5, 7,
                            4, 3, 9, 2,
                            1, 6, 3, 6
                        }
                    } ;

    clrscr( ) ;
    display ( a, 2, 3, 4 ) ;
    show ( a, 2, 3, 4 ) ;
    print ( a, 2, 3, 4 ) ;
    getch ( ) ;
    return 0 ;
```

```
}

void display ( int *q, int ii, int jj, int kk )
{
    int i, j, k ;
    for ( i = 0 ; i < ii ; i++ )
    {
        for ( j = 0 ; j < jj ; j++ )
        {
            for ( k = 0 ; k < kk ; k++ )
                printf ( "%d ", *( q + i * jj * kk + j * kk + k ) ) ;
            printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
    }
}

void show ( int ( *q )[3][4], int ii, int jj, int kk )
{
    int i, j, k ;
    int *p ;

    for ( i = 0 ; i < ii ; i++ )
    {
        for ( j = 0 ; j < jj ; j++ )
        {
            p = q[i][j] ;
            for ( k = 0 ; k < kk ; k++ )
                printf ( "%d ", *( p + k ) ) ;
            printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
    }
}

void print ( int q[ ][3][4], int ii, int jj, int kk )
{
    int i, j, k ;
```

```
    for ( i = 0 ; i < ii ; i++ )
    {
        for( j = 0 ; j < jj ; j++ )
        {
            for ( k = 0 ; k < kk ; k++ )
                printf ( "%d ", q[i][j][k] ) ;
            printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
    }
}
```

And here is the output...

```
1 2 3 4
5 6 7 8
9 3 2 1

2 3 5 7
4 3 9 2
1 6 3 6

1 2 3 4
5 6 7 8
9 3 2 1

2 3 5 7
4 3 9 2
1 6 3 6

1 2 3 4
5 6 7 8
9 3 2 1

2 3 5 7
4 3 9 2
1 6 3 6
```

Expression	Meaning
int *q	q is an integer pointer
int (*q)[3][4]	q is a pointer to a 2-D array of 3 rows and 4 columns
int q[ ][3][4]	q is a pointer to a 2-D array of 3 rows and 4 columns

Table 2.1

Returning Array from Function

Now that we know how to pass a 2-D or a 3-D array to a function let us find out how to return an array from a function. There are again three methods to achieve this. Suppose we wish to return a 2-D array of integers from a function we can return the base address of the array as:

- A pointer to an integer
- A pointer to the zeroth 1-D array
- A pointer to the 2-D array

This is shown in the following program. The function **fun1( )** returns the base address as pointer to integer, the function **fun2( )** returns it as pointer to zeroth 1-D array, whereas **fun3( )** returns it as pointer to 2-D array of integers. Note the prototype declarations of the functions carefully.

```
/* Program 28 */
/* Three ways of returning a 2-D array from a function */
#include<stdio.h>
#define ROW 3
#define COL 4
```

```
int main( )
{
    int i, j ;

    int *a ;
    int *fun1( ) ;

    int ( *b )[COL] ;
    int ( *fun2( ) )[COL] ;
    int *p ;

    int ( *c )[ROW][COL] ;
    int ( *fun3( ) )[ROW][COL] ;

    clrscr ( ) ;
    a = fun1( ) ;

    printf ( "Array a[ ][ ] in main( ):\n" ) ;
    for ( i = 0 ; i < ROW ; i++ )
    {
        for ( j = 0 ; j < COL ; j++ )
            printf ( "%d ", * ( a + i * COL+ j ) ) ;

        printf ( "\n" ) ;
    }
    getch( ) ;

    b = fun2( ) ;

    printf ( "Array b[ ][ ] in main( ):\n" ) ;
    for ( i = 0 ; i < ROW ; i++ )
    {
        p = b + i ;
        for ( j = 0 ; j < COL ; j++ )
        {
            printf ( "%d ", *p ) ;
            p++ ;
        }
```

```
            printf ( "\n" ) ;
        }
        getch ( ) ;
        c = fun3( ) ;

        printf ( "Array c[ ][ ] in main( ):\n" ) ;
        for ( i = 0 ; i < ROW ; i++ )
        {
            for ( j = 0 ; j < COL ; j++ )
                printf ( "%d ", ( *c )[ i ][ j ] ) ;

            printf ( "\n" ) ;
        }
        getch( ) ;
        return 0 ;
}

int *fun1( )
{
        static int a[ROW][COL] = {
                                        1, 2, 3, 4,
                                        5, 6, 7, 8,
                                        9, 0, 1, 6
                                    } ;
        int i, j ;

        printf ( "Array a[ ][ ] in fun1( ):\n" ) ;
        for ( i = 0 ; i < ROW ; i++ )
        {
            for ( j = 0 ; j < COL ; j++ )
                printf ( "%d ", a[ i ][ j ] ) ;

            printf ( "\n" ) ;
        }
        return ( int * ) a ;
}
```

```
int ( *fun2( ) )[COL]
{
    static int b[ROW][COL ] =  {
                                    9, 4, 6, 4,
                                    1, 3, 2, 1,
                                    7, 5, 1, 6
                               } ;
    int i, j ;

    printf ( "Array b[ ][ ] in fun2( ):\n" ) ;
    for ( i = 0 ; i < ROW ; i++ )
    {
        for ( j = 0 ; j < COL ; j++ )
            printf ( "%d ", b[ i ][ j ] ) ;

        printf ( "\n" ) ;
    }
    return b ;
}

int ( *fun3( ) )[ROW][COL]
{
    static int c[ROW][COL] =  {
                                    6, 3, 9, 1,
                                    2, 1, 5, 7,
                                    4, 1, 1, 6
                               } ;
    int i, j ;

    printf ( "Array c[ ][ ] in fun3( ):\n" ) ;
    for ( i = 0 ; i < ROW ; i++ )
    {
        for ( j = 0 ; j < COL ; j++ )
            printf ( "%d ", c[ i ][ j ] ) ;

        printf ( "\n" ) ;
    }
    return ( int ( * )[ROW][COL] ) c ;
```

```
}
```

And here is the output...

```
Array a[ ][ ] in fun1( ):
1 2 3 4
5 6 7 8
9 0 1 6

Array a[ ][ ] in main( ):
1 2 3 4
5 6 7 8
9 0 1 6

Array b[ ][ ] in fun2( ):
9 4 6 4
1 3 2 1
7 5 1 6

Array b[ ][ ] in main( ):
9 4 6 4
1 3 2 1
7 5 1 6

Array c[ ][ ] in fun3( ):
6 3 9 1
2 1 5 7
4 1 1 6

Array c[ ][ ] in main( ):
6 3 9 1
2 1 5 7
4 1 1 6
```

Returning 3-D Array from a Function

If you have understood how to return a 2-D array from a function, on similar lines we can return a 3-D array from a function. The four possible ways to do so would be to return the base address as:

- A pointer to an integer
- A pointer to the zeroth 1-D array
- A pointer to the zeroth 2-D array
- A pointer to the 3-D array

Given below is the program, which implements these four ways of returning a 3-D array.

```
/* Program 29 */
/* Four ways of returning a 3-D array from a function */
#include<stdio.h>
#define SET 2
#define ROW 3
#define COL 4

int main( )
{
    int i, j, k ;

    int *a ;
    int *fun1( ) ;

    int ( *b )[COL] ;
    int ( *fun2( ) )[COL] ;

    int ( *c )[ROW][COL] ;
    int ( *fun3( ) )[ROW][COL] ;
    int *p ;

    int ( *d )[SET][ROW][COL] ;
    int ( *fun4( ) )[SET][ROWI[COL] ;
```

```
clrscr( ) ;
a = fun1( ) ;

printf ( "Array a[ ][ ][ ] in main( ):\n" ) ;
for ( i = 0 ; i < SET ; i++ )
{
    for ( j = 0 ; j < ROW ; j++ )
    {
        for ( k = 0 ; k < COL ; k++ )
            printf ( "%d " ,* ( a + i * ROW * COL + j * COL + k ) ) ;

        printf ( "\n" ) ;
    }

    printf ( "\n" ) ;
}
getch( ) ;

b = fun2( ) ;

printf ( "Array b[ ][ ][ ] in main( ):\n" ) ;
for ( i = 0 ; i < SET ; i++ )
{
    p = ( int * ) ( b + i * ROW ) ;
    for ( j = 0 ; j < ROW ; j++ )
    {
        for ( k = 0 ; k < COL ; k++ )
        {
            printf ( "%d ", *p ) ;
            p++ ;
        }

        printf ( "\n" ) ;
    }

    printf ( "\n" ) ;
}
```

```
getch( ) ;

c = fun3( ) ;

printf ( "Array c[ ][ ][ ] in main( ):\n" ) ;
for ( i = 0 ; i < SET ; i++ )
{
    p = ( int * ) ( c + i ) ;
    for ( j = 0 ; j < ROW ; j++ )
    {
        for ( k = 0 ; k < COL ; k++ )
        {
            printf ( "%d ", *p ) ;
            p++ ;
        }

        printf ( "\n" ) ;
    }

    printf( "\n" ) ;
}
getch( ) ;

d = fun4( ) ;

printf ( "Array d[ ][ ][ ] in main( ):\n" ) ;
for ( i = 0 ; i < SET ; i++ )
{
for ( j = 0 ; j < ROW ; j++ )
{
    for ( k = 0 ; k < COL; k++ )
        printf ( "%d ", ( *d )[ i ][ j ][ k ] ) ;

    printf ( "\n" ) ;
}

printf( "\n" ) ;
```

```
    }
    getch( ) ;
    return 0 ;
}

int *fun1( )
{
    int i, j, k ;
    static int a[SET][ROW][COL] = {
                                    {
                                        1, 2, 3, 4,
                                        5, 6, 7, 8,
                                        9, 3, 2, 1
                                    },
                                    {
                                        2, 3, 5, 7,
                                        4, 3, 9, 2,
                                        1, 6, 3, 6
                                    }
                                };

    printf ( "Array a[ ][ ][ ] in fun1( ):\n" ) ;
    for ( i = 0 ; i < SET ; i++ )
    {
        for ( j = 0 ; j < ROW ; j++ )
        {
            for ( k = 0 ; k < COL ; k++ )
                printf ( "%d ", a[ i ][ j ][ k ] ) ;

            printf ( "\n" ) ;
        }

        printf ( "\n" ) ;
    }
    return ( int * ) a ;
}

int ( *fun2( ) )[COL]
```

```
{
    int i, j, k ;
    static int b[SET][ROW][COL] =  {
                                        {
                                            9, 4, 6, 4,
                                            1, 3, 2, 1,
                                            7, 5, 1, 6
                                        },
                                        {
                                            6, 3, 9, 1,
                                            2, 1, 5, 7,
                                            4, 1, 1, 6
                                        }
                                    } ;

    printf ( "Array b[ ][ ][ ] in fun2( ):\n" ) ;
    for ( i = 0 ; i < SET ; i++ )
    {
        for ( j = 0 ; j < ROW ; j++ )
        {
            for ( k = 0 ; k < COL ; k++ )
                printf ( "%d ", b[ i ][ j ][ k ] ) ;
            printf ( "\n" ) ;
        }

        printf ( "\n" ) ;
    }

    return ( int ( * )[COL] ) b ;
}

int ( *fun3( ) )[ROW][COL]
{
    int i, j, k ;
    static int c[SET][ROW][COL] = {
                                        {
                                            9, 4, 6, 4,
                                            1, 3, 2, 1,
```

```
                                        7, 5, 1, 6
                                    },
                                    {
                                        6, 3, 9, 1,
                                        2, 1, 5, 7,
                                        4, 1, 1, 6
                                    }
                                };

    printf ( "Array c[ ][ ][ ] in fun3( ):\n" ) ;
    for ( i = 0 ; i < SET ; i++ )
    {
        for ( j = 0 ; j < ROW ; j++ )
        {
            for ( k = 0 ; k < COL ; k++ )
                printf ( "%d ", c[ i ][ j ][ k ] ) ;

            printf ( "\n") ;
        }

        printf( "\n" ) ;
    }

    return ( int ( * )[ROW][COL] ) c ;
}

int ( *fun4( ) )[SET][ROW][COL]
{
    int i, j, k ;
    static int d[SET][ROW][COL] = {
                                    {
                                        3, 1, 8, 5,
                                        9, 6, 5, 2,
                                        2, 0, 1, 6
                                    },
                                    {
                                        7, 3, 2, 7,
                                        1, 4, 2, 3,
```

```
                                                    9, 1, 0, 6
                                                }
                                            };

    printf ( "Array d[ ][ ][ ] in fun4( ):\n" ) ;
    for ( i = 0 ; i < SET ; i++ )
    {
        for ( j = 0 ; j < ROW ; j++ )
        {
            for ( k = 0 ; k < COL ; k++ )
                printf ( "%d ", d[ i ][ j ] [ k ] ) ;

            printf ( "\n" ) ;
        }

        printf ( "\n" ) ;
    }

    return ( int ( * )[SET][ROW][COL] ) d ;
}
```

And here is the output. . .

```
Array a[ ][ ][ ] in fun1( ):
1 2 3 4
5 6 7 8
9 3 2 1

2 3 5 7
4 3 9 2
1 6 3 6

Array a[ ][ ][ ] in main( ):
1 2 3 4
5 6 7 8
9 3 2 1
```

Array of Pointers

The way there can be an array of **int**s or an array of **float**s, similarly there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply in to the array of pointers as well. I think a program would clarify the concept.

```
/* Program 30 */
#include <stdio.h>
int main( )
{
    int *arr[4] ; /* array of integer pointers */
```

```
    int i = 31, j = 5, k = 19, l = 71, m ;

    arr[0] = &i ;
    arr[1] = &j ;
    arr[2] = &k ;
    arr[3] = &l ;
    for ( m = 0 ; m <= 3 ; m++ )
        printf ( "%d\n", *( arr[m] ) ) ;
    return 0 ;
}
```

And here is the output...

```
31
5
19
71
0
```

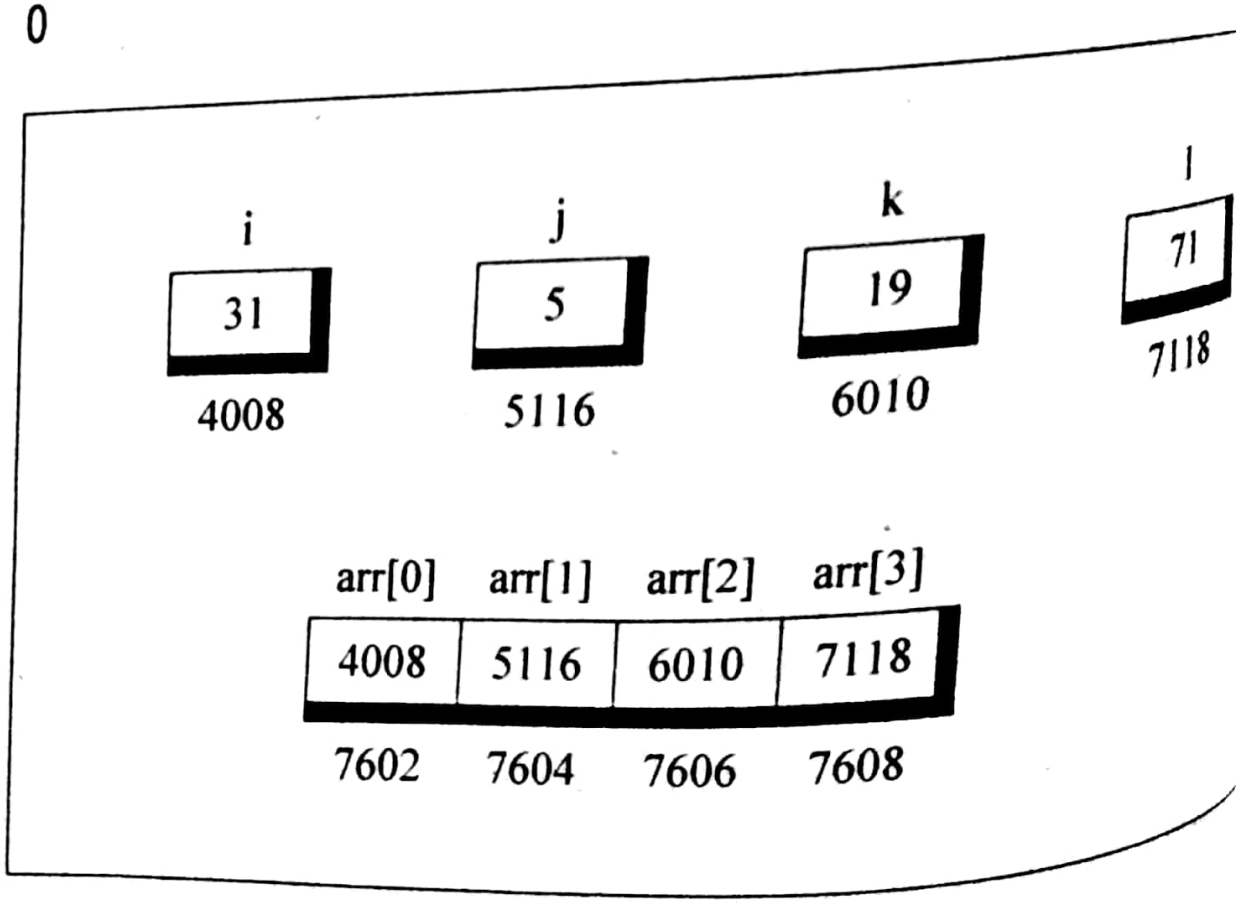


Figure 2.8

isolated **int** variables **i**, **j**, **k** and **l**. The **for** loop in the program picks up the addresses present in **arr** and prints the values present at these addresses.

An array of pointers can even contain the addresses of other arrays. The following program would justify this.

```
/* Program 31 */
#include <stdio.h>
int main( )
{
    static int a[ ] = { 0,1, 2, 3, 4 } ;
    static int *p[ ] = { a, a + 1, a + 2, a + 3, a + 4 } ;

    printf ( "%u %u %d\n", p, *p, *( *p ) ) ;
    return 0 ;
}
```

I would leave it for you to figure out the output of this program. An array of pointers is very popularly used for storing several strings in memory, as you would see in the next chapter.

Dynamic Memory Allocation

Consider the array declaration,

```
int marks[100] ;
```

Such a declaration would typically be used if 100 student's marks were to be stored in memory. The moment we make this declaration 200 bytes are reserved in memory for storing 100 integers in it. However, it may so happen that when we actually run the program, we might be interested in storing only 60 student's marks. Even in this case 200 bytes would get reserved in memory, which would result in wastage of memory.

Other way round there always exists a possibility that when you run the program you need to store more than 100 student's marks. In this case the array would fall short in size. Moreover, there is no way to increase or decrease the array size during execution of the program. In other words, when we use arrays static memory allocation takes place. What if we want to allocate memory only at the time of execution? This is done using standard library functions **malloc( )** and **calloc( )**. Since these functions allocate memory on the fly (during execution) they are often known as 'Dynamic memory allocation functions'. Let us now see a program, which uses the concept of dynamic memory allocation.

```
/* Program 32 */
#include "alloc.h"
#include<stdio.h>

int main( )
{
    int n, avg, i, *p, sum = 0 ;

    printf ( "Enter the number of students\n" ) ;
    scanf( "%d", &n ) ;

    p = ( int* ) malloc ( n * 2 ) ;
    if ( p == NULL )
    {
        printf ( "Memory allocation unsuccessful\n" ) ;
        exit( ) ;
    }

    for ( i = 0 ; i < n ; i++ )
        scanf ( "%d", ( p + i ) ) ;

    for ( i = 0 ; i < n ; i++ )
        sum = sum + *( p + i ) ;

    avg = sum / n ;
```

```
    printf ( "Average marks = %d\n", avg ) ;
    return 0 ;
}
```

Here, we have first asked for the number of students whose marks are to be entered and then allocated only as much memory as is really required to store these marks. Not a byte more, not a byte less. The allocation job is done using the standard library function **malloc( )**. **malloc( )** returns a NULL if memory allocation is unsuccessful. If successful it returns the address of the memory chunk that is allocated. We have collected this address in an integer pointer **p**. Since **malloc( )** returns a **void** pointer we have typecasted it into an integer pointer. In the first **for** loop using simple pointer arithmetic we have stored the marks entered from keyboard into the memory that has been allocated. In the second **for** loop we have accessed the same values to find the average marks.

The **calloc( )** functions works exactly similar to **malloc( )** except for the fact that it needs two arguments. For example,

```
int *p ;
p = ( int * ) calloc ( 10, 2 ) ;
```

Here 2 indicates that we wish to allocate memory for storing integers, (since an integer is a 2-byte entity) and 10 indicates that we want to reserve space for storing 10 integers. Another minor difference between **malloc( )** and **calloc( )** is that, by default, the memory allocated by **malloc( )** contains garbage values, whereas that allocated by **calloc( )** contains all zeros. While using these functions it is necessary to include the file "alloc.h" at the beginning of the program.

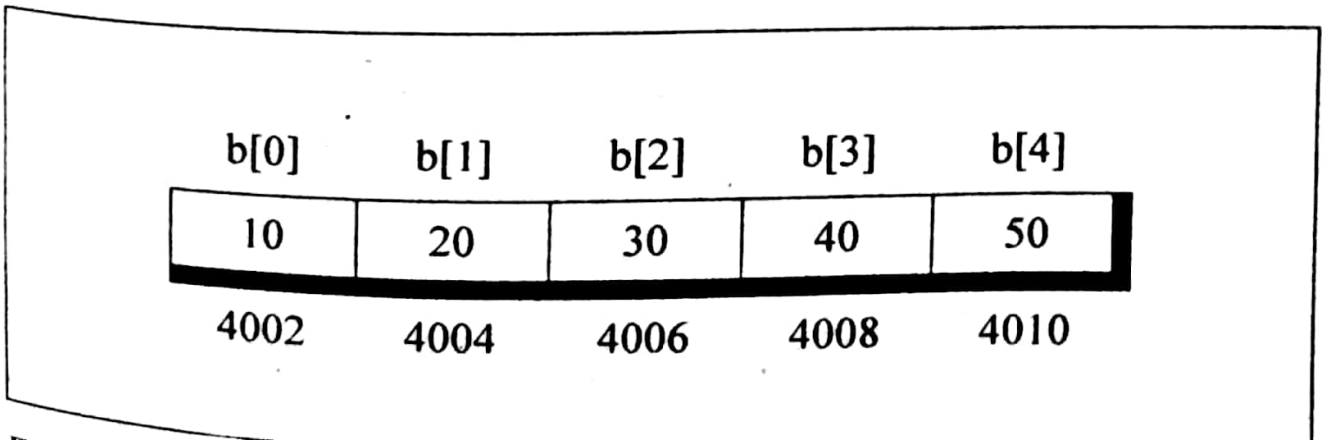
(4)
```
#include <stdio.h>
int main( )
{
    int b[ ] = { 10, 20, 30, 40, 50 } ;
    int i, *k ;
    k = &b[4] – 4 ;
    for ( i = 0 ; i <= 4 ; i++ )
    {
        printf ( "%d ", *k ) ;
        k++ ;
    }
    return 0 ;
}
```

Output

```
10 20 30 40 50
```

Explanation

First look at Figure 2.9. The array elements are stored in contiguous memory locations and each element is an integer, hence is occupying 2 locations.



(9)
```
#include <stdio.h>
int main( )
{
    /* Assume array begins at location 1002 */
    int a[3][4] =   {
                        1, 2, 3, 4,
                        5, 6, 7, 8,
                        9,10,11,12
                    } ;

    printf ( "%u %u %u\n", a[0] + 1, * ( a[0] + 1 ), * ( * ( a + 0 ) + 1 ) ) ;
    return 0 ;
}
```

Output

1004 2 2

(27)
```
#include <stdio.h>
int main( )
{
    static int a[3][3] = {
                                1, 2, 3,
                                4, 5, 6,
                                7, 8, 9
                            } ;
    static int *ptr[3] = { a[0], a[1], a[2] } ;
    int **ptr1 = ptr ;
    int i ;

    printf ( "\n" ) ;
    for ( i = 0 ; i <= 2 ; i++ )
        printf ( "%d ", *ptr[i] ) ;

    printf ( "\n" ) ;
    for ( i = 0 ; i <= 2 ; i++ )
        printf ( "%d ", *a[i] ) ;

    printf ( "\n" ) ;
    for ( i = 0 ; i <= 2 ; i++ )
    {
        printf ( "%d ", **ptr1 ) ;
        ptr1++ ;
    }
    return 0 ;
}
```

Output

```
1 4 7
1 4 7
1 4 7
```

Explanation

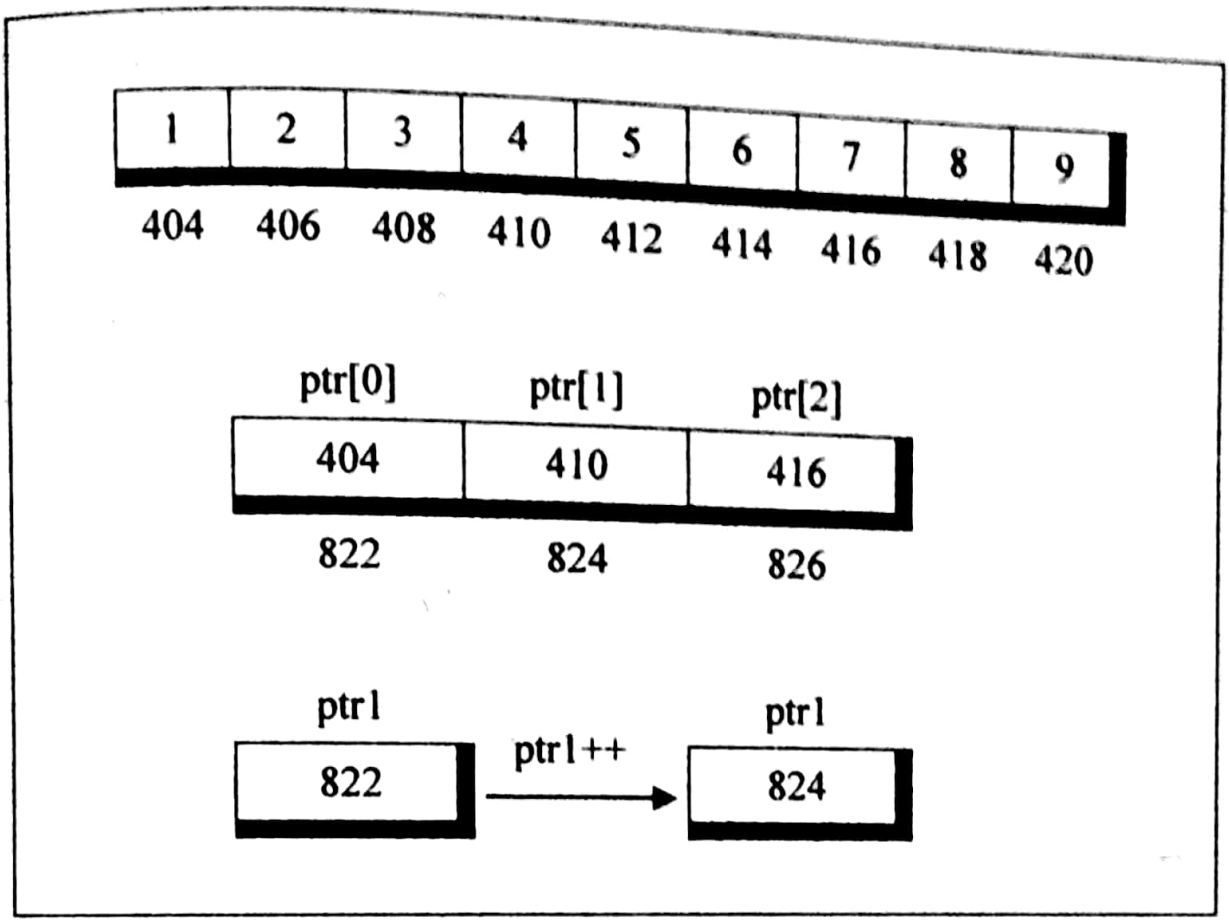


Figure 2.23

ptr[ ] has been declared as an array of pointers containing the base addresses of the three 1-D arrays as shown in Figure 2.23. Once past the declarations, the control reaches the first **for** loop. In this loop the **printf( )** prints the values at addresses stored in **ptr[0]**, **ptr[1]** and **ptr[2]**, which turn out to be 1, 4 and 7.

In the next **for** loop, the values at base addresses stored in the array **a[ ]** are printed, which once again turn out to be 1, 4 and 7. The third **for** loop is also simple.

Since **ptr1** has been initialized to the base address of the array **ptr[ ]**, it contains the address 822. Therefore ***ptr1** would give the value at address 822, i.e. 404, and ****ptr1** would give the value at address given by ***ptr1**, i.e. value at 404, which is 1. On incrementing **ptr1** it points to the next location after 822.

(2) For the statements in (1) does the compiler fetch the charact
arr[3] and **ptr[3]** in the same manner?

Explanation

No. For **arr[3]** the compiler generates code to start at location **arr**, move three past it, and fetch the character there. When it sees the expression **ptr[3]** it generates the code to start at location stored in **ptr**, add three to the pointer, and finally fetch the character pointed to.

In other words, **arr[3]** is three places past the start of the object named **arr**, whereas **ptr[3]** is three places past the object pointed to by **ptr**.

(3) Can you combine the following two statements into one?

```
char *p ;
p = malloc ( 100 ) ;
```

Explanation

```
char *p = malloc ( 100 )
```

(4) Does mentioning the array name gives the base address in all the contexts?

Standard Library String Functions

C has a large set of useful string handling library functions. Here, we would illustrate the usage of most commonly used functions (**strlen( ), strcpy( ), strcat( )** and **strcmp( )**) through a program.

```
/* Program 37 */
#include <string.h>
#include <stdio.h>
int main( )
{
    char str1[20] = "Bamboozled" ;
    char str2[ ] = "Chap" ;
    char str3[20] ;
    int l, k ;

    l = strlen ( str1) ;
    printf ( "length of string = %d\n", l ) ;

    strcpy ( str3, str1 ) ;
    printf ( "after copying, string str3 = %s\n", str3 ) ;

    k = strcmp( str1, str2) ;
    printf ( "on comparing str1 and str2, k = %d\n", k ) ;

    k = strcmp ( str3, str1 ) ;
    printf ( "on comparing str3 and str1 , k = %d\n", k ) ;

    strcat ( str1,str2 ) ;
    printf ( "on concatenation str1 = %s\n", str1 ) ;

    return 0 ;
}
```

The output would be...

```
length of string = 10
after copying, string str3 = Bamboozled
on comparing str1 and str2, k = -1
on comparing str3 and str1, k = 0
on concatenation str1 = BamboozledChap
```

Note that in the call to the function **strlen( )**, we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '\0'. Can we not write a function **xstrlen( )** which imitates the standard library function **strlen( )**? Let us give it a try...

```
/* Program 38 */
#include <stdio.h>
int main( )
{
    char arr[ ] = "Bamboozled" ;
    int len1, len2 ;

    len1 = xstrlen ( arr ) ;
    len2 = xstrlen ( "HumptyDumpty" ) ;
    printf ( "string = %s length = %d\n ", arr, len1 ) ;
    printf ( "string = %s length = %d\n ", "HumptyDumpty", len2 ) ;
    return 0 ;
}

xstrlen ( char *s )
{
    int length = 0 ;
    while ( *s != '\0' )
    {
        length++ ;
        s++ ;
    }
```

```
    return ( length ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = HumptyDumpty length =12
```

The function **xstrlen( )** is fairly simple. All that it does is it keeps counting the characters till the end of string is not met. Or in other words keeps counting characters till the pointer **s** doesn't point to '\0'.

Another function that we have used in Program 37 is **strcpy( )**. This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. On supplying the base addresses, **strcpy( )** goes on copying the source string into the target string till it doesn't encounter the end of source string. It is our responsibility to see to it that target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piecemeal, character by character. There is no shortcut for this. Let us now attempt to mimic **strcpy( )** via our own string copy function, which we would call **xstrcpy( ).**

```
xstrcpy ( char *t, char *s )
{
    while ( *s != '\0' )
    {
        *t = *s ;
        s++ ;
        t++ ;
    }
    *t = '\0' ;
}
```

Note that having copied the entire source string into the target string, it is necessary to place a '\0' into the target string to mark its end.

The **strcat( )** function concatenates the source string at the end of the target string. For example, "Bamboozled" and "Chap" on concatenation would result into a string "BamboozledChap". Note that the target string **str1[ ]** has been made big enough to hold the final string. I leave it to you to develop your own **xstrcat( )** on lines of **xstrlen( )** and **xstrcpy( )**.

Another useful string function is **strcmp( )** which compares two strings to find out whether, they are same or different. The two strings are compared letter by letter until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp( )** returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pair of characters.

The exact value of mismatch will rarely concern us. All we usually want to know is whether or not the first string is alphabetically above the second string. If it is, a negative value is returned; if it isn't, a positive value is returned. Any non-zero value means there is a mismatch. Let us try to implement this procedure into a function **xstrcmp( )**, which works similar to the **strcmp( )** function.

```
xstrcmp ( char *s1, char *s2 )
{
    while ( *s1 == *s2 )
    {
        if ( *s1 == '\0' )
            return ( 0 ) ;

        s1++ ;
```

```
        s2++ ;
    }
    return ( *s1 - *s2 ) ;
}
```

Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a **char** pointer. This is shown below:

```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer. This is shown in the following program.

```
/* Program 39 */
#include <stdio.h>
int main( )
{
    char str1[ ] = "Hello" ;
    char str2[10] ;

    char *s = "Good Morning" ;
    char *q ;

    str2 = str1 ; /* error */
    q = s ; /* works */

    return 0 ;
```

```
}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with **char** pointers.

```
/* Program 40 */
#include <stdio.h>
int main( )
{
    char str1[ ] = "Hello" ;
    char *p = "Hello" ;

    str1 = "Bye" ; /* error */
    p = "Bye" ; /* works */

    return 0 ;
}
```

The *const* Qualifier

The keyword **const** (for constant), if present, precedes the data type of a variable. It specifies that the value of the variable will not change throughout the program. Any attempt to alter the value of the variable defined with this qualifier will result into an error message from compiler, **const** is usually used to replace #defined constants.

const qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant. It also reminds anybody reading the program listing that the variable is not intended to change. Variables with this qualifier are often named in all uppercase, as a reminder that they are constants. The following program shows the usage of **const.**

```
/* Program 41 */
#include <stdio.h>
int main( )
{
    float r, a ;
    const float PI = 3.14 ;

    printf ( "Enter radius:\n" ) ;
    scanf ( "%f", &r ) ;

    a = PI * r * r ;
    printf ( "Area of circle = %f\n", a ) ;

    return 0 ;
}
```

const is a better idea as compared to **#define** because its scope of operation can be controlled by placing it appropriately either inside a function or outside all functions. If a **const** is placed inside a function its effect would be localized to that function, whereas, if it is placed outside all functions then its effect would be global. We cannot exercise such finer control while using a **#define**.

const Pointers

Look at the following program:

```
/* Program 42 */
#include <stdio.h>
int main( )
{
    char str1[ ] = "Nagpur" ;
    char str2[10] ;

    xstrcpy ( str2, str1 ) ;
    printf ( "%s\n" str2) ;
```

```
    return 0 ;
}

xstrcpy ( char *t, char *s )
{
    while ( *t  != '\0' )
    {
        *t = *s ;
        t++ ;
        s++ ;
    }
    *t = '\0' ;
}
```

This program simply copies the contents of **str1[ ]** into **str2[ ]** using the function **xstrcpy( )**. What would happen if we add the following lines beyond the last statement of **xstrcpy( )**?

```
s = s - 6 ;
*s = 'K' ;
```

This would change the source string to "Kagpur". Can we not ensure that the source string doesn't change even accidentally in **xstrcpy( )**? We can, by changing the prototype of the function to

```
void xstrcpy ( char*, const char* ) ;
```

Correspondingly the definition would change to:

```
void xstrcpy ( char *t, const char *s )
{
    /* code */
}
```

The following code fragment would help you to fix your ideas about **const** further.

```
char *p = "Hello" ; /* pointer is variable, so is string */
*p = 'M' ; /* works */
p = "Bye" ; /* works */

const char *q = "Hello" ; /* string is constant pointer is not */
*q = 'M' ; /* error */
q = "Bye" ; /* works */

char const *s = "Hello" ; /* string is constant pointer is not */
*s = 'M' ; /* error */
s = "Bye" ; /* works */

char * const t = "Hello" ; /* pointer is constant string is not */
*t = 'M' ; /* works */
t = "Bye" ; /* error */

const char * const u = "Hello" ; /* string is constant, so is pointer */
*u = 'M' ;  /* error */
u = "Bye" ; /* error */
```

Returning *const* Values

A function can return a pointer to a constant string as shown below.

```
/* Program 43 */
#include <stdio.h>
int main( )
{
    const char *fun( ) ;
    const char *p ;

    p = fun( ) ;
```

```
    *p = 'A' ; /* error */
    printf ( "%s\n", p ) ;

    return 0 ;
}

const char *fun( )
{
    return "Rain" ;
}
```

Here since the function **fun( )** is returning a constant string, we cannot use the pointer **p** to modify it. Not only this, the following operations too would be invalid:

(a) **main( )** cannot assign the return value to a pointer to a non-**const** string.

(b) **main( )** cannot pass the return value to a function that is expecting a pointer to a non-**const** string.

Two Dimensional Array of Characters

In the previous chapter we saw several examples of 2-D numeric arrays. Let's now look at a similar phenomenon, but one dealing with characters. The best way to understand this concept is through a program. Our example program asks you to type your name. When you do so, it checks your name against a master list to see if you are worthy of entry to the palace. Here's the program...

```
/* Program 44 */
#include <string.h>
#include <stdio.h>
#define FOUND 1
```

```
#define NOTFOUND 0
int main( )
{
    char masterlist[6][10] = {
                                "akshay",
                                "parag",
                                "raman",
                                "srinivas",
                                "gopal",
                                "rajesh"
                              } ;
    int i, flag, a ;
    char yourname[10] ;

    printf ( "Enter your name: " ) ;
    scanf ( "%s", yourname ) ;

    flag = NOTFOUND ;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        a = strcmp ( &masterlist[i][0], yourname ) ;
        if ( a == 0 )
        {
            printf ( "Welcome, you can enter the palace\n" ) ;
            flag = FOUND ;
            break ;
        }
    }

    if ( flag == NOTFOUND )
        printf ( "Sorry, you are a trespasser\n" ) ;

    return 0 ;
}
```

Array of Pointers to Strings

As we know, a pointer variable always contains an address. Therefore, if we construct an array of pointers it would contain a number of addresses. Let us see how the names in the earlier example can be stored in the array of pointers.

```
char *names[ ] = {
                    "akshay",
                    "parag",
                    "raman",
                    "srinivas",
                    "gopal",
                    "rajesh"
                  };
```

```
/* Program 45 */
#include <stdio.h>
int main( )
{
    char *names[ ] = {
                        "akshay",
                        "parag",
                        "raman",
                        "srinivas",
                        "gopal",
                        "rajesh"
                    } ;
    char *temp ;

    printf ( "Original: %s %s\n", names[2], names[3] ) ;
    temp = names[2] ;
    names[2] = names[3] ;
    names[3] = temp ;

    printf ( "New: %s %s\n", names[2], names[3] ) ;
    return 0 ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

cannot receive the strings from keyboard using **scanf( )**. Thus, the following program would never work out.

```
/* Program 46 */
#include <stdio.h>
int main( )
{
    char *names[6] ;
    int i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "Enter name: " ) ;
        scanf ( "%s", names[i] ) ;
    }
    return 0 ;
}
```

The program doesn't work because when we are declaring the array it is containing garbage values. And it would be definitely wrong to send these garbage values to **scanf( )** as the addresses where it should keep the strings received from the keyboard.

As a compromise solution we may first allocate space for each name using **malloc( )** and then store the address returned by **malloc( )** in the array of pointers to strings. This is shown in the following program.

```
/* Program 47 */
/* Program to overcome limitation of array of pointers to strings */
#include <alloc.h>
#include <string.h>
#include <stdio.h>
int main( )
{
    char *name[5] ;
```

```
    char str[20] ;
    int i ;

    for ( i = 0 ; i < 5 ; i++ )
    {
        printf ( "Enter a String: " ) ;
        gets ( str ) ;
        name[i] = ( char * ) malloc ( strlen ( str ) ) ;
        strcpy ( name[i], str ) ;
    }
    for ( i = 0 ; i < 5 ; i++ )
        printf ( "%s\n", name[i] ) ;
    return 0 ;
}
```

[A] What will be the output of the following programs:

(1)
```
#include <string.h>
#include <stdio.h>
int main( )
{
    char s[ ] = "Rendezvous !" ;
    printf ( "%d\n", * ( s + strlen ( s ) ) ) ;
    return 0 ;
}
```

Output

0

Explanation

No 'Rendezvous !', but a zero is printed out. Mentioning the name of the string gives the base address of the string. The function **strlen (s)** returns the length of the string **s[ ]**, which in this case is 12. In **printf( )**, using the 'value at address' operator (often called 'contents of' operator), we are trying to print out the contents of the 12^{th} address from the base address of the string. At this address there is a '\0', which is automatically stored to mark the end of the string. The ASCII value of '\0' is 0, which is what is being printed by the **printf( )**.

```
    printf ( 5 + "Fascimile" ) ;
    return 0 ;
}
```

Output

```
mile
```

Explanation

When we pass a string to a function, what gets passed is the base address of the string. In this case what is being passed to **printf( )** is the base address plus 5, i.e. address of 'm' in "Fascimile". **printf( )** prints a string starting from the address it receives, up to the end of the string. Hence, in this case 'mile' gets printed.

3)
```
#include <stdio.h>
int main( )
{
    char ch[20] ;
    int i ;
    for ( i = 0 ; i < 19 ; i ++ )
        *( ch + i ) = 67 ;

    *( ch + i ) = '\0' ;
    printf ( "%s\n", ch ) ;

    return 0 ;
}
```

cccccccccccccccccc

Explanation

Mentioning the name of the array always gives its base address. Therefore **(ch + i)** would give the address of the i^{th} element from the base address, and ***(ch + i)** would give the value at this address, i.e. the value of the i^{th} element. Through the **for** loop we store 67, which is the ASCII value of upper case 'C', in all the locations of the string. Once the control reaches outside the **for** loop the value of **i** would be 19, and in the 19^{th} location from the base address we store a '\0' to mark the end of the string. This is essential, as the compiler has no other way of knowing where the string is terminated. In the **printf( )** that follows, **%s** is the format specification for printing a string, and **ch** gives the base address of the string. Hence starting from the first element, the complete string is printed out.

(4)
```
#include <stdio.h>
int main( )
{
    char str[ ] = { 48, 48, 48, 48, 48, 48, 48, 48, 48, 48 } ;
    char *s ;
    int i ;
    s = str ;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        if ( *s )
            printf ( "%c ", *s ) ;

        s++ ;
```

(5)
```
#include <stdio.h>
int main( )
{
    char str1[ ] = "Hello" ;
    char str2[ ] = "Hello" ;
    if ( str1 == str2 )
        printf ( "Equal\n" ) ;
    else
        printf ( "Unequal\n" ) ;
```

```
    return 0 ;
}
```

Output

```
Unequal
```

Explanation

When we mention the name of the array we get its base address. Since **str1** and **str2** are two different arrays, their base addresses would always be different. Hence, the condition in **if** is never going to get satisfied. If we are to compare the contents of two **char** arrays, we should compare them on a character by character basis or use **strcmp( )**.

(6)
```
#include <stdio.h>
int main( )
{
    char str[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } ;
    char *s ;
    int i ;
    s = str ;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        if ( *s )
            printf ( "%c", *s) ;
        s++ ;
    }
    return 0 ;
}
```

While handling real world data, we usually deal with a collection of **int**s, **char**s and **float**s rather than isolated entities. For example, an entity we call a 'book' is a collection of things like a title, an author, a call number, a publisher, number of pages, date of publication, price, etc. As you can see, all this data is dissimilar; author is a string, price is a **float**, whereas number of pages is an **int**. For dealing with such collections, C provides a data type called 'structure'. A structure gathers together different atoms of information that form a given entity.

Look at the following program that combines dissimilar data types into an entity called structure.

```
/* Program 48 */
#include <stdio.h>
int main( )
{
    struct account
    {
        int no ;
        char acc_name[15] ;
        float bal ;
    } ;
    struct account a1, a2, a3 ;

    printf ( "Enter account nos., names, and balances\n" ) ;
    scanf ( "%d %s %f", &a1.no, a1.acc_name, &a1.bal ) ;
    scanf ( "%d %s %f", &a2.no, a2.acc_name, &a2.bal ) ;
    scanf ( "%d %s %f", &a3.no, a3.acc_name, &a3.bal ) ;

    printf ( "%d %s %f\n", a1.no, a1.acc_name, a1.bal ) ;
    printf ( "%d %s %f\n", a2.no, a2.acc_name, a2.bal ) ;
    printf ( "%d %s %f\n", a3.no, a3.acc_name, a3.bai ) ;

    return 0 ;
}
```

Now a few tips about the program:

(a) The declaration at the beginning of the program combin dissimilar data types into a single entity called **stru(t account**. Here **struct** is a keyword, **account** is the structu name, and the dissimilar data types are structure elements.

(b) **a1**, **a2** and **a3** are structure variables of the type **stru account**.

(c) The structure elements are accessed using a '.' operator. So refer **no** we use **a1.no** and to refer to **acc_name** we **a1.acc_name**. Before the dot there must always be a struct variable and after the dot there must always be a struct element.

(d) Since **a1.acc_name** is a string, its base address can obtained just by mentioning **a1.acc_name**. Hence the 'addı of operator **&** has been dropped while receiving the accc name in **scanf( )**.

(e) The structure elements are always arranged in contigu memory locations. This arrangement is shown in following figure.

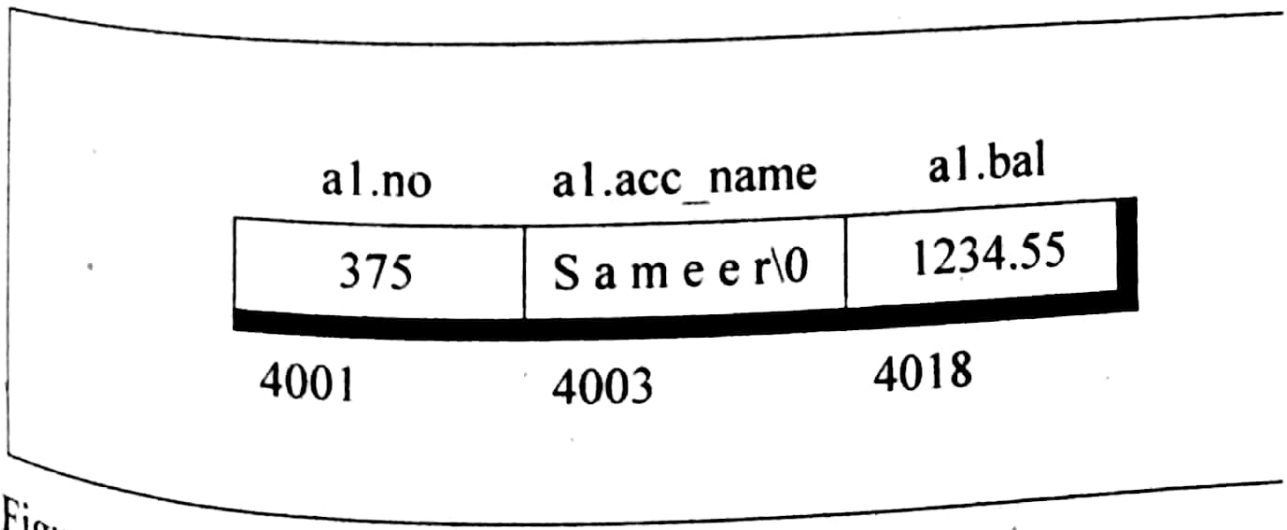


Figure 4.1 Structure elements in memory

An Array of Structures

In the above example if we were to store data of 100 accounts, we would be required to use 100 different structure variables from **a1** to **a100**, which is definitely not very convenient. A better approach would be to use an array of structures. The arrangement of the array of structures in memory is shown in the following figure.

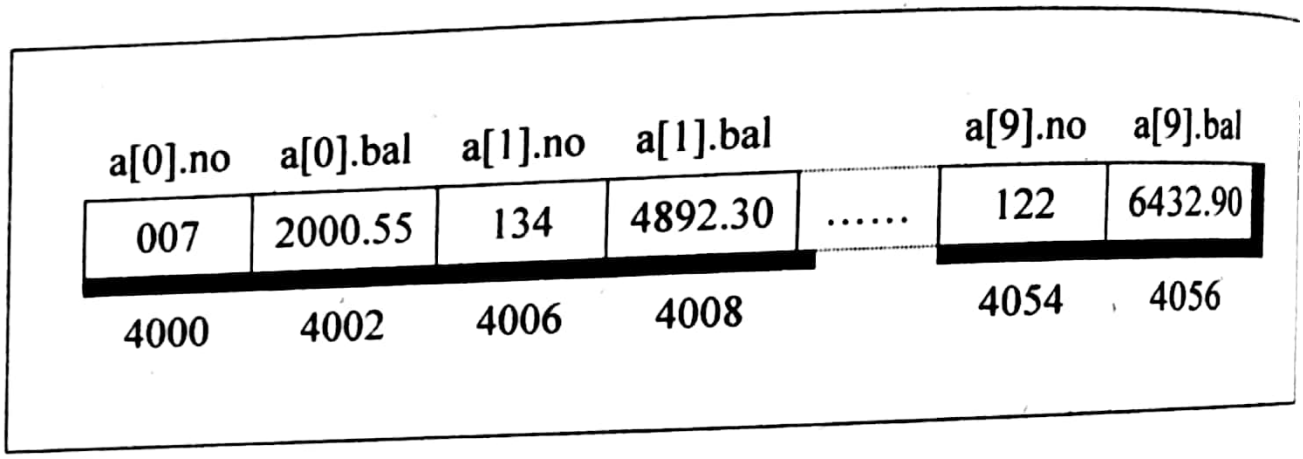


Figure 4.2 Array of structures in memory

Now let us write a program, which puts the array of structures to work.

```
/* Program 49 */
#include <stdio.h>
int main( )
{
    struct account
    {
        int no ;
        float bal ;
    } ;
    struct account a[10] ;
    int i, acc ;
    float balance ;

    for ( i = 0 ; i <= 9 ; i++ )
    {
```

```
        printf ( "Enter account no. and balance:\n" ) ;
        scanf ( "%d %f" , &acc, &balance ) ;
        a[i].no = acc ;
        a[i].bal = balance ;
        printf ( "%d %f\n", a[i].no, a[i].bal ) ;
    }
    return 0 ;
}
```

As you can see the structure elements are still accessed using the '.' operator, and the array elements using the usual subscript notation.

More about Structures

Let us now explore the intricacies of structures with a view of programming convenience.

(a) The declaration of structure type and the structure variable can be combined in one statement. For example,

```
struct player
{
    char name[20] ;
    int age ;
} ;
struct player p1 = { "Nick Yates", 30 } ;
```

is same as...

```
struct player
{
    char name[20] ;
    int age ;
} p1 = { "Nick Yates", 30 } ;
```

or even...

```
struct
{
    char name[20] ;
    int age ;
} p1 = { "Nick Yates", 30 } ;
```

(b) The value of one structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements piece-meal. For example,

```
struct player
{
    char name[20] ;
    int age ;
} ;
struct player p2, p1 = { "Nick Yates", 30 } ;
p2 = p1 ;
```

(c) One structure can be nested within another structure as shown below.

```
struct part
{
    char type ;
    int qty ;
} ;
struct vehicle
{
    char maruti[20] ;
    struct part bolt ;
} ;
struct vehicle v ;
v.bolt.qty = 300 ;
```

(d) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure at one go. If need be we can also pass addresses of structure elements or address of a structure variable as shown below.

```
struct player
{
    char nam[20] ;
    int age ;
} ;
struct player p1 = { "Nick Yates", 30 } ;
display ( p1.nam, p1.age ) ; /* passing individual elements */
show ( p1 ) ; /* passing structure variable */
d ( p1.nam, &p1.age ) ; /* passing addresses of structure elements */
print ( &p1 ) ; /* passing address of structure variable */
```

Structure Pointers

The way we can have a pointer pointing to an **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as 'structure pointers'. Let us look at a program, which demonstrates the usage of these pointers.

```
/* Program 50 */
#include <stdio.h>
int main( )
{
    struct book
    {
        char name[25] ;
        char author[25] ;
        int callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;
    struct book *ptr ;
```

```
    ptr = &b1 ;
    printf ( "%s %s %d\n", b1.name, b1.author, b1.callno ) ;
    printf ( "%s %s %d\n", ptr -> name, ptr -> author, ptr -> callno ) ;
    return 0 ;
}
```

The first **printf( )** is as usual. The second **printf( )** however is peculiar. We can't use **ptr.name** or **ptr.callno** because **ptr** is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator ->, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the '.' structure operator, there must always be a structure variable, whereas on the left hand side of the -> operator there must always be a pointer to a structure. The arrangement of the structure variable and pointer to structure in memory is shown in the figure given below.

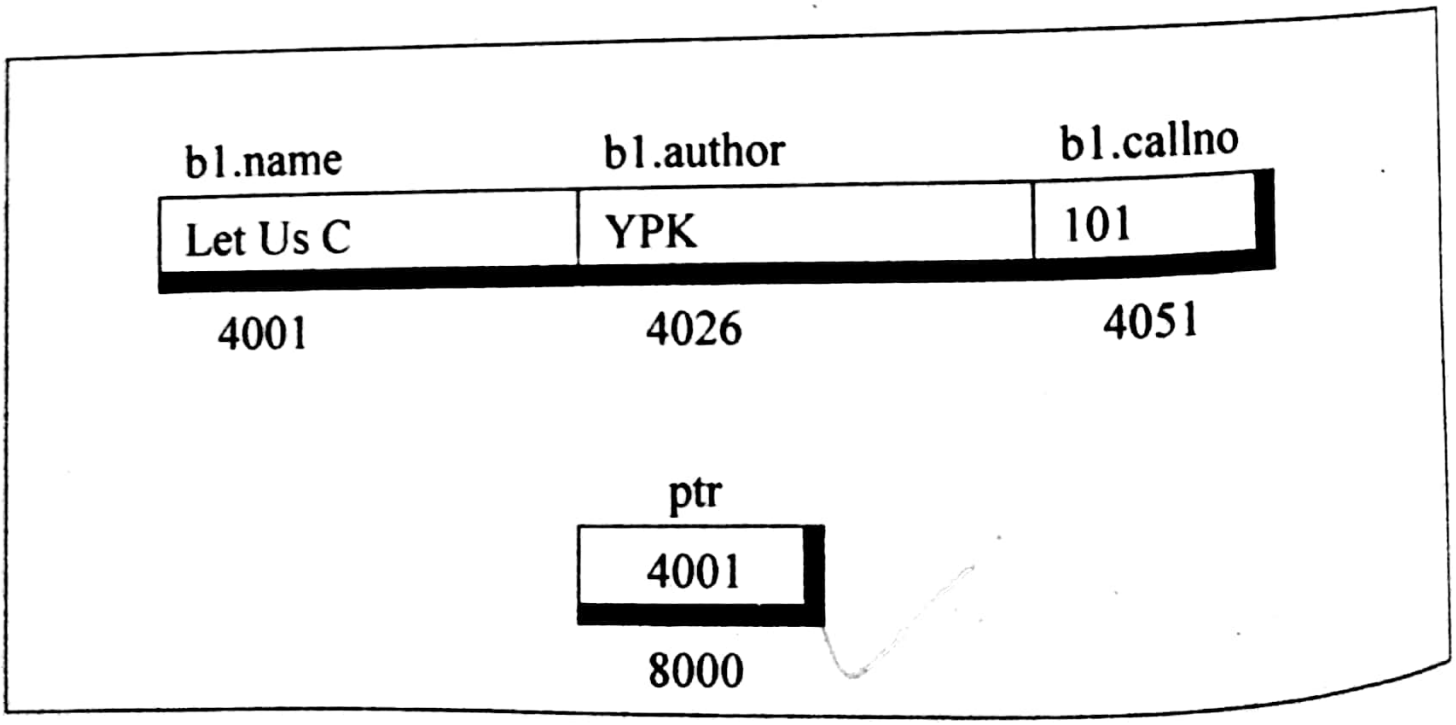


Figure 4.3

Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

```
/* Program 51 */
/* Passing address of a structure variable */
#include <stdio.h>
struct book
{
    char name[25] ;
    char author[25] ;
    int callno ;
};

int main( )
{
    void display ( struct book * ) ;
    struct book b1 = { "Let us C", "YPK", 101 } ;
    display ( &b1 ) ;
    return 0 ;
}

void display ( struct book *b ) /* b is a pointer to a structure */
{
    printf ( "%s\n%s\n%d\n", b -> name, b -> author, b -> callno ) ;
}
```

And here is the output...

```
Let us C
YPK
101
```

Again, note that, to access the structure elements using pointer to a structure we have to use the '->' operator. Also, the structure **struct book** should be declared outside **main( )** such that this data type is available to **display( )** while declaring the variable **b** as a pointer to the structure.

Offsets of Structure Elements

Consider the following structure:

```
struct a
{
    struct b
    {
        int i ;
        float f ;
        char ch ;
    } x ;

    struct c
    {
        int j ;
        float g ;
        char ch ;
    } y ;
} z ;
```

Suppose we make a call to a function as shown below:

```
fun ( &z.y ) ;
```

In the function **fun( )** can we access the elements of structure **b** through the address of **z.y**? We can. For this we need to first find out the offset of **j**. Using this offset we can find out address where **x** begins in memory. Once we get this address we can have an access to elements **i**, **f** and **ch**. This is shown in the following program.

```
/* Program 52 */
#include <stdio.h>
struct a
```

```
{
    struct b
    {
        int i ;
        float f ;
        char ch ;
    } x ;

    struct c
    {
        int j ;
        float g ;
        char ch ;
    } y ;
} z ;

int main( )
{
    int *p ;
    struct a z ;

    clrscr( ) ;
    fun ( &z.y ) ;
    printf ( "%d %f %c\n", z.x.i, z.x.f, z.x.ch ) ;
    getch ( ) ;
    return 0 ;
}

fun ( struct c * p )
{
    int offset ;
    struct b * address ;

    offset = ( char * ) & ( ( struct c * ) ( & ( ( struct a * ) 0 ) -> y ) -> j )
                                                - ( char * ) ( ( struct a * ) 0 ) ;
    address = ( struct b * ) ( ( char * ) & ( p -> j ) - offset ) ;
    address -> i = 400 ;
```

```
    address -> f = 3.14 ;
    address -> ch = 'c' ;
}
```

In the above program structures **b** and **c** having members **i, f, ch** and **j, g, ch** are nested within the structure **a** with structure variables **x** and **y** of structure **b** and **c** respectively. Next we have called the function **fun( )** with the base address of the structure variable **y**. Now from **fun( )**, we wish to access the members of structure variable **x**. But, since **fun( )** has been passed a pointer to structure **c**, we do not have direct access to elements of **x**. The solution is to calculate the offset of the first member of **y**, in our case, **j**. This has been achieved through the statement:

```
offset = ( char * ) & ( ( struct c * ) ( & ( ( struct a * ) 0 ) -> y ) -> j )
                      ( char * ) ( ( struct a * ) 0 ) ;
```

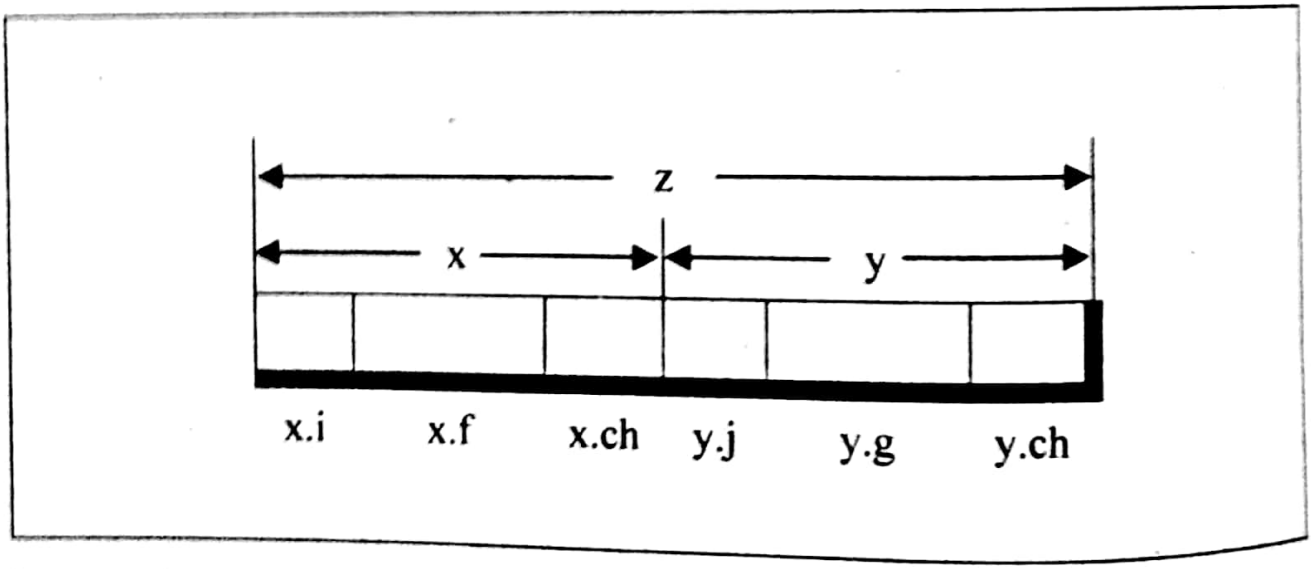


Figure 4.4

Let us understand this statement part by part.

In the expression **((struct a *) 0)**, 0 is being typecasted into pointer to **struct a**. This expression is pretending that there is a variable of type **struct a** at address 0. The expression **& (((struct a *) 0) -> y)** gives the address of structure variable **y**. But this is not the base address of the structure **c**. Hence we have typecasted it

using **struct c ***. Using this address we can access the member **j** of the structure variable **y**. Finally, after taking the address of member **j**, we have typecasted it using **char *** to make the subtraction possible. The statement to the right of the '-' operator is straightforward. On subtraction, we get the offset of member **j** of structure variable **y**. Now using offset the base address is calculated through the following statement.

```
address = ( struct b * ) ( ( char * ) & ( p -> j ) – offset ) ;
```

In the above statement **& (p -> j)** gives the address of member **j** of structure variable **y**. Subtracting offset from this yields the address of structure variable **x**. Using this address, we have stored the values in the member variables **i**, **f** and **ch**. Back into **main( )**, we have printed these values using **printf( )**.

Linked Lists

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent locations. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. For instance, the marks obtained by different students can be stored in a linked list as shown in Figure 4.5.

inters to Functions

Every type of variable, with the exception of register, has an address. We have seen how we can reference variables of type **char**, **int**, **float** etc. through their addresses—that is by using pointers. Pointers can also point to C functions. And why not? C functions have addresses. If we know the function's address we can point to it, which provides another way to invoke it. Let us see how this can be done.

```
/* Program 72 */
/* Demo to get address of a function */
#include <stdio.h>
int main( )
{
    int display( ) ;
    printf ("Address of function display is %u\n", display ) ;
    display( ) ;    /* usual way of invoking  a function */
    return 0 ;
}

int display( )
{
    printf ( "Long live viruses!!\n" ) ;
}
```

The output of the program would be:

```
Address of function display is 1125
Long live viruses!!
```

Note that to obtain the address of a function all that we have to do is to mention the name of the function, as has been done in the **printf( )** statement above. This is similar to mentioning the name of the array to get its base address.

Now let us see how using the address of a function we can manage to invoke it. This is shown in the program given below:

```
/* Program 73 */
/* Invoking function using pointer to a function */
#include <stdio.h>
int main( )
{
    int display( ) ;
    int ( *func_ptr )( ) ;

    func_ptr = display ;     /* assign address of function */
    printf ( "Address of function display is %u\n", func_ptr ) ;
    ( *func_ptr )( ) ;    /* invokes the function display( ) */
    return 0 ;
}

int display( )
{
    printf ( "Long live viruses!!\n" ) ;
}
```

The output of the program would be:

```
Address of function display is 1125
Long live viruses!!
```

In **main( )** we have declared the function **display( )** as a function returning an **int**. But what are we to make of the declaration,

```
int ( *func_ptr )( ) ;
```

What comes in the next line? We are obviously declaring something which, like **display( )**, will return an **int**. But what is it? And why is ***func_ptr** enclosed in parentheses?

If we glance down a few lines in our program, we see the statement,

```
func_ptr = display ;
```

So we know that **func_ptr** is being assigned the address of **display( )**. Therefore, **func_ptr** must be a pointer to the function **display( )**.

Thus, all that the declaration

```
int ( *func_ptr)( ) ;
```

means is, that **func_ptr** is a pointer to a function, which returns an int. And to invoke the function we are just required to write the statement,

```
( *fun_ptr )( ) ;
```

Pointers to functions are certainly awkward and off-putting. And why use them at all when we can invoke a function in a much simpler manner? What is the possible gain of using this esoteric feature of C? There are several possible uses:

(a) in writing memory resident programs
(b) in writing viruses, or vaccines to remove the viruses
(c) in developing COM / DCOM components
(d) in VC++ programming to connect events to function calls

All these topics form interesting and powerful applications and would call for separate chapters on each if full justice is to be given to them. Much as I would have liked to, for want of space I would have to exclude these topics.

typedef with Function Pointers

We know that **typedef** is used to give convenient names to complicated datatypes. It is immensely useful when using pointers to functions. Given below are a few examples of its usage.

(1)
```
typedef int ( *funcptr )( ) ;
funcptr fptr ;
```

fptr is a pointer to a function returning an **int**.

(2)
```
typedef int ( *fret_int )( char *, char * ) ;
fret_int fri1, fri2 ;
```

fri1 and **fri2** are pointers to function that accepts two **char** pointers and returns an **int.**

(3)
```
typedef void ( *complex )( ) ;
complex c ;
```

c is a pointer to a function that doesn't accept any parameter and doesn't return anything.

(4)
```
typedef char ( * ( * frpapfrc( ) ) [ ] ) ( ) ;
frpapfrc f ;
```

f is a function returning a pointer to an array of pointers to functions returning a **char**.

(5)
```
typedef int ( * ( *arr2d_ptr )( ) ) [3][4] ;
arr2d_ptr p ;
```

p is a pointer to a function returning a pointer to a 2-D **int** array.

(6)
```
typedef int ( * ( * ( * ptr2d_fptr )( ) ) [10] )( ) ;
ptr2d_fptr p ;
```

p is a pointer to a function returning a pointer to an array of 10 pointers to function returning an **int**.

(7)
```
typedef char ( * ( * arr_fptr[3] )( ) ) [10] ;
arr_fptr x ;
```

x is an array of 3 pointers to function returning a pointer to an array of 10 **chars**.

(8)
```
typedef float* ( * ( * ( * ptr_fptr )( ) ) [10] )( ) ;
ptr_fptr q ;
```

q is a pointer to function returning a pointer to an array of 10 pointers to functions returning a **float** pointer.

argc and *argv*—Arguments to *main( )*

Can we not pass arguments to **main( )** the way we pass them to other functions? We can. For this we specify the arguments at command prompt when we execute the program. For example, suppose 'PR1.EXE' is the name of the executable file, and we wish to pass the arguments 'Cat', 'Dog' and 'Parrot' to **main( )** in this lie.

We can do so through the following command at the command prompt:

```
C> PR1.EXE Cat Dog Parrot
```

Now if we are passing arguments to **main( )**, it must collect them in variables. Usually only two variables are used to collect these arguments. These are called **argc** and **argv**. Of these, **argc** contains the count (number) of arguments being passed to **main( )**, whereas **argv** contains addresses of strings passed to **main( )**. In the above example **argc** would contain 4, whereas **argv[0]**, **argv[1]**, **argv[2]**, and **argv[3]** would contain base addresses of PR1.EXE, Cat, Dog and Parrot respectively. If we so desire we can print these arguments from within **main( )** as shown in the following program.

```
/* Program 74 */
#include <stdio.h>
int main ( int argc, char *argv[ ] )
{
    int i ;

    for ( i = 0 ; i < argc ; i++ )
        printf ( "%s\n", argv[i] ) ;
    return 0 ;
}
```

Note the declaration of **argv[ ]**. It has been declared as an array of pointers to strings. Also observe the format specification used in **printf( )**. We are using **%s** because we wish to print out the various strings that are being passed to **main( )**.

It is not necessary that we should always use the variable names **argc** and **argv**. In place of them any other variable names can as well be used.

Pointers and Variable Number of Arguments

We use **printf( )** so often without realising how it works correctly irrespective of how many arguments we pass to it. How do we go about writing such routines, which can take variable number of arguments? And what have pointers got to do with it? There are three macros available in the file "stdarg.h" called **va_start**, **va_arg** and **va_list**, which allow us to handle this situation. These macros provide a method for accessing the arguments of the function when a function takes a fixed number of arguments followed by a variable number of arguments. The fixed number of arguments are accessed in the normal way, whereas the optional arguments are accessed using the macros **va_start** and **va_arg**. Out of these macros **va_start** is used to initialise a pointer to the beginning of the list of optional arguments. On the other hand the macro **va_arg** is used to advance the pointer to the next argument. Let us put these concepts into action using a program. Suppose we wish to write a function **findmax( )** which would find out the maximum value from a set of values, irrespective of the number of values passed to it.

```
/* Program 75 */
#include <stdio.h>
#Include <stdarg.h>
int main( )
{
    int max ;

    max = findmax ( 5, 23, 15, 1, 92, 50 ) ;
    printf ( "Max = %d\n", max ) ;

    max = findmax ( 3, 100, 300, 29 ) ;
    printf ( "Max = %d\n", max ) ;

    return 0 ;
}
```

```
findmax ( int tot_num )
{
    int max, count, num ;

    va_list ptr ;

    va_start ( ptr, tot_num ) ;
    max = va_arg ( ptr, int ) ;

    for ( count = 1 ; count < tot_num ; count++ )
    {
        num = va_arg ( ptr, int ) ;
        if ( num > max )
            max = num ;
    }

    return ( max ) ;
}
```

Here we are making two calls to **findmax( )** first time to find maximum out of 5 values and second time to find maximum out of 3 values. Note that for each call the first argument is the count of arguments that are being passed after the first argument. The value of the first argument passed to **findmax( )** is collected in the variable **tot_num**. **findmax( )** begins with a declaration of pointer ptr of the type **va_list**. Observe the next statement carefully:

```
va_start ( ptr, tot_num ) ;
```

This statement sets up **ptr** such that it points to the first variable argument in the list. If we are considering the first call to **findmax( )**, **ptr** would now point to 23. The next statement **max = va_arg (ptr, int)** would assign the integer being pointed to by **ptr** to **max**. Thus 23 would be assigned to **max**, and **ptr** would now start pointing to the next argument i.e., 15. The rest of the program is

fairly straightforward. We just keep picking up successive numbers in the list and keep comparing them with the latest value in **max**, till all the arguments in the list have been scanned. The final value in **max** is then returned to **main( )**.

How about another program to fix your ideas? This one calls a function **display( )** which is capable of printing any number of arguments of any type.

```
/* Program 76 */
#include <stdio.h>
#include <stdarg.h>
int main( )
{
    printf ( "\n" ) ;
    display ( 1, 2, 5, 6 ) ;
    printf ( "\n" ) ;
    display ( 2, 4, 'A', 'a', 'b', 'c' ) ;
    printf ( "\n" ) ;
    display ( 3, 3, 2.5, 299.3, -1.0 ) ;
    return 0 ;
}

display ( int type, int num )
{
    int i, j ;

    char c ;
    float f ;
    va_list ptr ;

    va_start ( ptr, num ) ;

    switch ( type )
    {
        case 1 :
            for ( j = 1 ; j <= num ; j++ )
```

```
            {
                i = va_arg ( ptr, int ) ;
                printf ( "%d ", i ) ;
            }
            break ;

        case 2 :
            for ( j = 1 ; j <= num ; j++ )
            {
                c = va_arg ( ptr, char ) ;
                printf ( "%c ", c ) ;
            }
            break ;

        case 3 :
            for ( j = 1 ; j <= num ; j++)
            {
                f = ( float ) va_arg ( ptr, double ) ;
                printf ( "%f ", f ) ;
            }
    }
}
```

Here we are passing two fixed arguments to the function **display( )**. The first one indicates the data type of the arguments to be printed and the second indicates the number of such arguments to be printed. Once again through the statement **va_start (ptr, num)** we have set up **ptr** such that it points to the first argument in the variable list of arguments. Then depending upon whether the value of **type** is 1, 2 or 3 we have printed out the arguments as **ints**, **chars** or **floats**.

near, far and *huge* Pointers

To understand **near**, **far** and **huge** pointers it's necessary to know the details about memory organization, memory addressing scheme and various memory models. Let us begin with the basics.

To enable the flow of data between the microprocessor and the memory there is a set of wires. This set of wires is called a 'data bus'. Each wire in this bus carries a bit of data at a time (either zero or one in the form of an electric pulse). If the data bus has 8 wires then 8 bits or 1 byte of data can flow in it at a time and this is called a 8-bit data bus. Similarly, we have 16-bit and 32-bit data buses that can respectively carry 16 and 32 bits at a time.

A microprocessor with a 32-bit data bus is faster than the one with a 16-bit data bus. This is because in the former, four bytes of data are brought to the microprocessor at a time; while in the latter only 2 bytes of data are brought at a time.

If a microprocessor with a 32-bit data bus is faster than the one with a 16-bit data bus, then why not have a 32-bit data bus instead of a 16-bit data bus? This is not possible because the microprocessors are designed that way. A microprocessor with a provision for connecting only a 16-bit data bus cannot be connected with a 32-bit data bus. Hence, each type of microprocessor can be distinguished by the width of the data bus that it is connected to. A microprocessor with a provision for 16-bit data bus is called a 16-bit microprocessor; the one with a provision for 32 bits is called a 32-bit microprocessor and so on.

The way the data bus width tells how many bits the bus can move at a time, there is another bus called address bus whose width tells how many addresses the microprocessor can access. For example, if the address bus width of a microprocessor is 20 bits then it can access 2^{20} locations (1 mb) in memory. The following figure

(1)
```
#include <stdio.h>
int main( )
{
    void ( *message )( ) ;
    void print( ) ;
    print( ) ;
    message = print ;
    ( *message )( ) ;
    return 0 ;
}
void print( )
{
    printf ( "Never trouble trouble till trouble troubles you\n" ) ;
}
```

(2)
```
#include <stdio.h>
int main( )
{
    long far *a ;

    printf ( "Love makes life lovely\n" ) ;
    printf ( "Press any key...\n" ) ;
    a = 36 ;
    *a = 0 ;
    getch( ) ;
    printf ( "... except when you hit a key\n" ) ;
    return 0 ;
}
```