UNIT- III
Arrays

Introduction

An array contains a collection of data elements of the same type. An array is referenced by a common name. Each element of an array is stored in successive locations of the memory.

Types of Arrays: Arrays can be used to represent not only simple lists of values but also tables of data in two or three or more dimensions.

a) One – dimensional arrays
b) Two – dimensional arrays
c) Multidimensional arrays

Array elements and Subscript

The elements of the array are known as members of the array. Each array element is identified by assigning a unique subscript or index to it. The dimension of an array is determined by the number of subscripts needed to identify each element. A subscript is enclosed in bracket [ ] placed after the array name. Space is not allowed between array name and subscript. A subscript is an integer value starting from zero.

Thus the array named mark with 5 elements will be represented as mark[0], mark [1], mark[2], mark [3], mark [4], and stored in successive locations of the memory as shown in the following figure.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]

Subscripts always start with 0. So, for an array of N elements, the last element has an index of N-1.

One dimensional array declarations

In one-dimensional array, single subscript is used. The one-dimensional array is also called as list.

Before using an array, it must be declared. An array is defined in the same way as a variable. Each array name must be accompanied by a size specification (i.e. number of elements). The general from for array declaration is

storage-type data-type array_name[SIZE]

Where storage-type is optional. It may be either static, extern, automatic. The data-type specifies the type of element that will be stored in the array. SIZE is an integer constant, indicating the maximum number of elements of the array. The rules for forming array names are the same as for variable names.

Examples

(i) int number[100]; (ii) char text[100]; (iii) float ave[10]; (iv) signed char name[50];

In the first example, int specifies the type of the variable and the word number specifies the name of the array. The number 100 indicates the maximum number of elements in the array. This number is called as ‘dimension’ of the array. The bracket ([]) indicates that the given variable is an array.

The size of array should be a constant value.

The main purpose of the declaration of the array is to reserve space in memory. For example when declaring the array

```
int marks[10];
```

reserves the 10x 2 = 20 bytes in memory. Each int occupies 2 bytes of memory.

Initialization of Arrays

After an array is declared, its elements must be initialized. Otherwise they will contain "garbage". An array can be initialized at either of the following stages. (i) At compile time (During declaration of the array) (ii) At run time. (using for loop and scanf() functions)

1. During the declaration of the array

The general form of declaring the array during the initialization is

```
data_type array_name[SIZE] = {list of values}
```

The values in the list must be separated by commas.

Examples : int mark[5] = {55,60,70,23,100}; float avg[3] = {25.6,75.5,23.3};

In the first example, the size of the array is 5 and the values 55, 60, 70, 23, 100 will be assigned to the variable mark[0], mark[1], mark[2], mark[3] and mark[4] respectively.

If the number of values in the list is less than the size of the array, the remaining elements will be set to zero automatically.

For example, the statement float height[4] = {0.25, 0.50}; will initialize the first two elements 0.25 and 0.50 to height[0] and height[1] respectively. The remaining two elements height[2] and height[3] will be set to 0.0

The size of the array may be omitted. In this situation, the compiler allocates enough space for all the elements given for initialization. For example, the statement int marks[ ] = {25, 50, 100, 75}; will declare the size of the array is 4.

If there are more elements than the declared size, the compiler will produce an error.

int number[3] = {10,20,30,40}; will not work. It is illegal in C.

2. Initialization of an array using for…loop.

for…loop is used to initialize values for array elements. for…loop can be used when initializing the constant value or values which are having some relation to the array which are having large number of elements.

Examples

(i) int array[100], i ;
for (i = 0; i < 100; i++)
array [i] = 0;
The above for…loop initialize all the elements with the value of 0.

(ii) int array[50], i, j = 0;
for (i = 0; i < 50; i++)
{
array [i] = j;
j = j + 2;
}
The above for…loop initializes the value 0 to array[0], 2 to array[1], 4 to array[2] and so on.

(iii) To initialize different values, scanf( ) function is used in the body of the for-loop statement.

```
int array[50], i ;
for ( i = 0 ; i < = 50; i++ )
scanf("%d", &array[i])
```

Two dimensional Arrays

One-dimensional arrays have a single subscript. But two-dimensional arrays have two subscripts. The two dimensional array is called a matrix or table. Two dimensional arrays are used to represent the values which are in matrix form. A two dimensional array can be represented with two pairs of square bracket.

Two-dimensional array can be represented in the following form

```
data-type array_name[subscript1][subscript2]
```

where subscript1, subscript2 are positive value integer constants or expression. This two subscripts indicates the number of array elements associated with each subscript.

Two square brackets are used to indicate the value of two subscripts. Assume the values of two subscripts as 'm' and 'n'. Then two dimensional array can be arranged as a table with m rows and n columns.

Examples : float table [3] [3]; int marks [10] [5];

The first example defines table as a floating-point array having 3 rows and 3 columns. An array element starts with an index of 0 so that the individual elements of the array are

table	[0][0]	[0][1]	[0][2]
	[1][0]	[1][1]	[1][2]
	[2][0]	[2][1]	[2][2]

The total number of elements of the two dimensional array can be calculated by multiplying the number of rows and number of columns.

Example for a two - dimensional array

A typical example for a two – dimensional array is the marks obtained by 50 students in 5 tests. To represent the above values 50*5 matrix is used. This can be declared by

int marks [50] [5];

The first index ([50]) is the number of the students and the second index ([5]) is the number of tests. This declaration allocates 500 memory locations. The test number is the column number and the student number is the row number. In each row, the marks obtained by the student in all the 5 tests are represented.

	Test1	Test2	Test3	Test4	Test5
student1	90	96	98	100	99

---------					
student50	79	87	88	67	55

INITIALIZATION OF A TWO DIMENSIONAL ARRAY

Care must be given to the order in which the initial values are assigned to the array elements. The second subscript increases most rapidly and the first subscript increases least rapidly. The elementsof a two dimensional array will be assigned row wise. That is, all the elements of the first row will be assigned, then all the elements of the second row and so on. For example, consider the following two dimensional array mark [4] [3] (three test marks of four students)

```
                          75   25   30
mark [4] [3]    =         45   50   22
                          40   72   45
                          41   55   78
```

The above example array can be initialized as: int mark [4] [3] = {75,25,30,45,50,22,40,72,45,41,55,78};

The first subscript ranges from 0 to 3 and the second subscript ranges from 0 to 2. Array elements will be stored in continuous locations in memory. Two dimensional arrays are also can be initialized in the following way

```
int mark [4] [3] = {
{75, 25, 30},
{45, 50, 22},
{40, 72, 45},
{41, 55, 78}, };
```

The three values in the first inner pair of braces are assigned to the array elements in the first row, the values in the second inner pair of braces are assigned to the array elements in the second row and so on. An outer pair of braces is required, containing the inner pairs. Each line contains the three marks of one student separated by comma and enclosed in braces and separated from the next students' marks by comma. The whole array is enclosed in a pair of braces.

While initializing an array, the second dimension is a must. The first dimension (row) is optional. Thus the following two declarations

```
int mark [2] [3] = {12, 24, 36, 48, 60, 70};
int mark [ ] [3] = {12, 24, 36, 48, 60, 70};
```

are equal.

If the values are missing in an initialization, they are automatically set to zero. For example

```
int mark [4] [3] = {
{12, 75},
{55, 75, 23},
 {45, 98, 57},
{48}
};
```

will assign the following values

mark[0] [0] = 12 mark[0] [1] = 75 mark[0] [2] = 0
mark[1] [0] = 55 mark[1] [1] = 75 mark[1] [2] = 23
mark[2] [0] = 45 mark[2] [1] = 98 mark[2] [2] = 57
mark[3] [0] = 48 mark[3] [1] = 0 mark[3] [2] = 0

Initialization by using for statement

The initialization can be done by using two for-loop statements. Here all the values are initialized to zero.

```
for ( i = 0; i < 4; i++ )
{
for ( j = 0; j < 3; j++ )
{
mark[i] [j] = 0;
}
}
```

The for statement is used when initializing the same value or values which are having some relationship to each other to array elements.

Two dimensional arrays are used to iitailize a set of string values. For example to store the name of 10 students (each name with a maximum of 30 characters) in a class, the following code is used

```
for ( i = 0; i < 10; i++ )
{
for ( j = 0; j < 30; j++ )
{
scanf("%s", name[i]);
}
}
```

Multi Dimensional Arrays

An array of three or more dimension is called as multidimensional array. The general form of a multidimensional array is

```
data-type array_name[subscript1][subscript2][subscript3]
```

where subscript I is size of the i-th dimension.

Examples

float sales[4][5][12]; int table[2][4][10][5];

where sales is a three-dimensional array, contains 240 float type elements. Similarly table is a four dimensional array containing 400 elements of int data type. The array sales may represent a data of sales during the last four years from January to December in five cities of a particular company.

Multi dimensional array can contain as many indices as needed. However the amount of memory needed for an array rapidly increases with each dimension.

For example

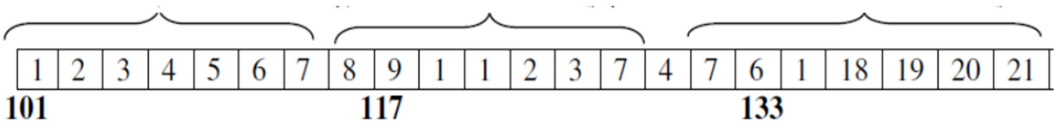
char arr [100][365][24][60][60];declaration would consume more than 3 gigabytes of memory.

Memory does not contain rows and columns, so whether it is a one dimensional array or two dimensional arrays, the array elements are stored linearly in one continuous chain. For **example,**

the multidimensional array

```
int arr[3][4][2]= {
{ {1,2},{3,4},{5,6},{7,8} },
{ {9,1},{1,2},{3,7},{4,7} },
{ {6,1},{18,19},{20,21} },
};
```

is stored in memory just like an one-dimensional array shown below:



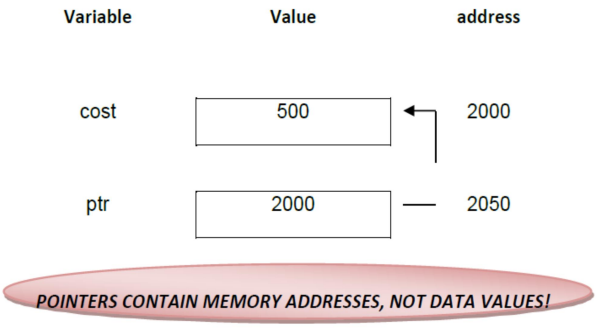
POINTERS

DEFINITION

A pointer is a variable that is used to store the address of another variable. Since a pointer is also another variable, its value is also stored in the memory in another location. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Each memory cell in the computer has an address that can be used to access that location. So a pointer variable points to a memory location. The contents of this memory location can be accessed and changed by using pointer variable.

Assume a variable "cost". Then the address of this cost is assigned to pointer variable ptr. The link between the variables ptr and cost is shown in figure. Assume the address of ptr is 2050. The link between the two variables are shown in the following figure.



ADVANTAGES OF POINTERS

1. Pointers are used to increase the speed of the execution.
2. Pointers are used to reduce the length and complexity of program.
3. A pointer is used to access a variable that is defined outside the function.
4. Storage space is saved when using pointer array to character strings.
5. Pointers can also be used in efficient manner to access the individual array elements.

DEFINING A POINTER VARIABLE

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration is somewhat different than the interpretation of other variable declarations. For defining a pointer variable '*' symbol is used. A pointer variable must be declared with preceding the variable name. This signifies that it is not an ordinary variable but a pointer variable. The type of data it is pointing must be specified. The general structure for declaring a pointer variable is

```
data_type *ptr_name;
```

For example, the statement int *p declares p as a pointer variable that points to an integer data type.

{

For example,

```
float a,b;
float *pb;
```

The first line declares a and b to be floating-point variables. The second line declares pb to be a pointer variable whose object is a floating-point quantity i.e. 'pb' point to a floating point quantity. 'pb' represents an address, not a floating-point quantity. Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable.

ADDRESS OPERATOR – ACCESSING THE ADDRESS OF A VARIABLE

The address of a variable is obtained with the help of "&" operator at any time. & operator is called as "address operator". The operator & immediately preceding a variable returns the address of the variable.For example, the statement

```
ptr=&num;
```

places the address of num into the variable ptr. If num is stored in memory 21260 address, then the variable ptr has the value 21260.

```
/* A program to illustrate pointer declaration*/
main()
{
int *ptr;
int sum;
sum=45;
ptr= &sum;
printf ("\n Sum is %d\n", sum);
printf ("\n The sum pointer is %d", ptr);
}
```

Accessing A Variable Through Its Pointer

The content of any pointer variable can be accessed with the help of "*" operator. The operator "*" is known as indirection operator. If "p" is an pointer variable, then *p is used to access the content of the pointer variable "p". For example the statement t=*p is used to assign the contents of the pointer variable p to t.

```
/* Program to display the contents of the variable using pointer variable*/
 include<stdio.h>
{
 int num, *intptr;
float x, *floptr;
char ch, *cptr;
num=123; x=12.34;
ch='a';
intptr=&x;
cptr=&ch;
floptr=&x;
printf("Num %d stored at address %u\n",*intptr,intptr);
printf("Value %f stored at address %u\n",*floptr,floptr);
printf("Character %c stored at address %u\n",*cptr,cptr);
}
```

Initialization of Pointers

Before using the pointer, it should be initialized. Static local pointer variables and global pointer variables are initialized with NULL by default. "NULL" is a pointer constant whose numerical value is zero.

Automatic pointer variable can be either initialized with NULL or with address of some other variable that is already defined.

For example in the statement,

```
p = & cost;
```

p contains the address of cost. This is called pointer initialization.

A pointer variable can also be initialized in the declaration statement itself. For example.

int a; *b = &a; is also valid.

Example :

```
# include <stdio.h>
main ( )
{
int a=3;
int b;
int *pa; /* to an integer */
int *pb;/*pointer to an integer */
pa=&a;/* assign address of a to pa */
b=*pa;/* assign value of a to b*/
pb=&b;/* assign address of b to pb */
printf ("\n% d %d % d %d", a, &a,pa, *pa);
printf ("\n% d %d % d %d", b, &b, pb, *pb);
}
```

The output is : 3 F8E F8E 3
3 F8C F8C 3

The unary operators & and * are members of the same precedence group.The address operator (&) must act upon operands associated with unique addresses, such as ordinary variables or single array elements. Thus, the address operator cannot act upon arithmetic expression, such as 2*(a+b). The indirection operator (*) can only act upon operands that are pointers.

Example:

```
# include <stdio.h>
main() {
int i=3,* j;k; /*j is a pointer to integer */
j=&i; /*j points to the location of i*/
k=*j ; /*assign to k the value pointed to by j*/
*j=4 /assign 4 to the location pointed to by j*/
printf("i=%d, *j=%d, k=%d\n," i, *j, k);
}
```

The output is as follows: i=4,*j=4, k=3

Null Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main () {
int *ptr = NULL;
printf("The value of ptr is : %x\n", ptr );
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

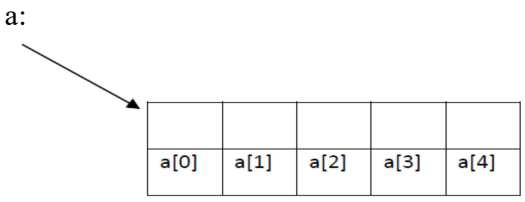
The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */

Pointers and Arrays

Pointers and arrays have got close relationship. In fact array name itself is a pointer. Some times pointers and array names can be interchangeably used.
The array name always points to the first element of the array. For example, an array int a[5]; is shown in the following figure.



Here *(a+2) actually refers to a [2]. At some places pointers and arrays can be interchangeably used.

A program to display the contents of array using pointer main() { int a[100]; int *ptr; int i,j,n; printf("\nEnter the elements of the array\n"); scanf("%d",&n); printf("Enter the array elements"); for(i=0;i< n;i++) scanf("%d",&a[i]); printf("Array element are"); for(ptr=a,ptr< (a+n);ptr++) printf("Value of a[%d]=%d stored at address %u",j+=,*ptr,ptr); }

A Program to compute the sum of all the elements in an array.

```
main()
{
int *p, total = 0; i=0;
int a[5]= {2,3,4,6,7};
p = a;
for (i=0; i<5; i++)
{
total = total + *p;
p++;
}
printf("\nTotal%d", total)
 }
```

Pointers and Functions

A function also has an address location in memory. It is therefore possible to declare a pointer to a function which can be used as an argument in another function.
A pointer to a function is described as follows
type (*pointer-name) ( );
For example int (*sqrt) ( ) tells the compiler that sqrt is a pointer to a function which returns an int value.
For example, in the statements
int *sqr ( ), square ( );
sqr = square;

sqr is pointer to a function and sqr points to the function square. To call the function square, the pointer sqr is used with the list of parenthesis. That is,
(*sqr) (a,b)
Is equivalent to
square (a,b)

Pointers as functions arguments (Call by Reference)

A pointer plays an important role when used with functions. Pointer variables can be passed as arguments to some other function. That is, the address of a variable is passed to another function. This type of function call is known as call by reference. The function is allowed to access and change the variable of a calling function. When passing the address of a variable as an argument to a function, the parameters receiving the address should be pointers.

When the function invocation uses the method call by value, the value of actual parameter is copied to dummy parameters.

In the case of call by reference method, instead of passing the values, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Hence it is possible to change the actual argument within the function body. This means, using these addresses it is possible to have an access to the actual arguments and hence manipulate them.

Since the actual argument variable and the corresponding dummy pointer refer to the same memory location, changing the contents of the dummy pointer will- by necessity- change the contents of the actual argument variable. The following program illustrates this fact.

```
#include <stdio.h>
main ()
{
int i,j;
i=2;
j=5;
printf("i=%d and j=%d\n", i,j);
/* Now exchange them */
swap(&i,&j);
printf("\nnow i=%d and j=%d\n", i,j);
}
swap(i,j)
int *i,*j;
{
int temp=*i; /* create temp and store into it the value pointed to by "i"*/
*i=*j; /*The value pointed to by j is stored in the location pointed to by i*/
*j=temp;/* Assign temp to the location pointed to by j*/
}
```

The output is as follows:
i=2 and j=5
Now i=5 and j=2

If the formal parameters i and j of the swap function were declared merely as integers, and the main function passed only i and j rather than their addresses, the exchange made in swap would have no effect on the variables i and j in main. The variable temp in swap function is of type int, not int*. The values being exchanged are not the pointers, but the integers being pointed to by them. Also temp is initialized immediately upon declaration to the value pointed to by i.

C program to find the highest of three numbers using pointer to function is listed below:

```
#include<conio.h>
void main()
{
int x,y,z;
clrscr();
printf("\n Enter three numbers : ");
scanf("%d %d %d",&x,&y,&z);
printf("\n\n The highest of the three numbers is : ");
highest(&x,&y,&z);
getch();
}
highest(a,b,c)
int *a,*b,*c;
{
if(*a > *b && *a > *c)
printf("%d",*a);
else if(*b > *a && *b > *c)
printf("%d",*b);
else
printf("%d",*c);
}
```

Pointers and Character Strings

Character array is called as string variable. Pointer is used to access the individual character in a string.

```
Example:
main()
{
char name[30];
char *p;
p = name;
strcpy(p,””salem””);
printf(“\n%s”, name);
}
```

Output : salem

Here ‘p’ is an alias name of character array ‘name’. Character pointers can be used effectively for various purposes.

Example:

```
main()
{
char name[30] ={“”salem””};
char name1[30];
void mycopy ( char *, char *);
mycopy(name1,name);
printf(“\n%s”, name1);
}
void mycopy ( char *d, char *s)
{
While ( *s !=’’\0’’)
```

```
{
(*d++ = *s++);
}
}
```

Output salem

The above program copies one string to another string. In the above function pointer arithmetic is used. After copying the content of one pointer to another, they are incremented to point the next element. Hence next element is copied. This is repeated until '\0' is encountered. The while loop will be continued as long as the condition is true i.e., non-zero.

Array of Pointers To Strings

Sometimes it is required to store more than one string in an array. One way is to use two dimensional character array.

Example

```
main()
{
int i;
char name[3][12]={"Ramu", "Govindan", "Rajasekaran"};
for (i=1; i<3; i++)
{
printf("\n%s ", name[i]);
}}
```

In this declaration name[ ] is an array of pointers. It contains base addresses of respective names. That is, base address of "Ramu" is stored in name[0], base address of "Govindan" is stored in name[1] and so on.

Here only required amount of memory is allocated. i.e.,5 +9 + 12 = 26 bytes. Hence we have saved 10 bytes.

Advantages of storing strings in an array of pointers:

1. One reason to store strings in an array of pointers is to make a more efficient use of available memory.
2. Another reason to use an array of pointers to store strings is to obtain greater ease in manipulation of the strings.

Limitation of array of pointers to strings

When using a two-dimensional array of characters, it is possible to initialize the strings when declaring the array or the strings can be received through scanf( ) function.

However, when using an array of pointers to strings, it is possible to initialize the strings at the place where we are declaring the array, but it is not possible to receive the strings from keyboard using scanf( ). Thus, the following program would never work out.

```
main( )
{
char *names[6] ;
int i ;
for ( i = 0 ; i <= 5 ; i++ )
{
printf ( "\nEnter name " ) ;
scanf ( "%s", names[i] ) ;
} }
```

Pointers and Structures

In an array, the name of an array stands for the address of its zeroth element. Similarly struct variable represents the address of its first element.

```
Example
typedef struct Bio
{
char name[30];
int age;
};
main()
{ Bio Record;
Bio *p;
p = & Record;
printf( "\n Enter Name;");
scanf( "%s", (*p).name);
printf( "\n Enter Age;");
scanf( "%d", &(*p).age);
printf( "\n Name;", (*p).name);
printf( "\n Age%d", (*p).age);
}
```

Output:

```
Enter Name : Ramu
Enter Age : 28
Name : Ramu
Age : 28
```

Here in this program first 'Bio' struct type is declared. Then in main( ), Record of type 'Bio' and a pointer 'p' of type 'Bio' is defined.

Then 'p' is made to point 'Record'. Hence '*p' is an alias name of 'Record'. There after whenever '*p' is referred, it should be interpreted as 'Record'. Then the fields of Record name and age are read and they are printed.

Note for referring the fields, use both indirection operation '*' and member operator ".".

i.e., (*p). name

Parenthesis is required since priority of '.' operator is higher than '*' operator. Apply '*' operator first. Hence change the order of priority by using parenthesis.

This leads to confusion. Hence instead of using '*', '.' and parenthesis,

the above is simplified by using the - > operator. The symbol -> is called as arrow operator.

Instead of writing (*p). name, we write

```
p-> name ; /* no space
```

which is more simple.

A pointer pointing to a structure just the same way a pointer pointing to an int, such pointers are known as structure pointers. For example consider the following example:

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[20];
int roll_no;
};
```

```
void main()
{
struct student stu[3],*ptr;
clrscr();
printf("\n Enter data\n");
for(ptr=stu;ptr<stu+3;ptr++)
{ printf("Name");
scanf("%s",ptr->name);
printf("roll_no");
scanf("%d",&ptr->roll_no);
}
printf("\nStudent Data\n\n");
ptr=stu;
while(ptr<stu+3)
{
printf("%s %5d\n",ptr->name,ptr->roll_no); ptr++;
}
getch();}
```

Here ptr is a structure pointer not a structure variable and dot operator requires a structure variable on its left. C provides arrow operator "->" to refer to structure elements. "ptr=stu" would assign the address of the zeroth element of stu to ptr. Its members can be accessed by statement like "ptr->name". When the pointer ptr is incremented by one, it is made to point to the next record, that is stu[1] and so on.

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include <stdio.h>
int main ()
{
int var;
int *ptr;
int **pptr;
var = 3000; /* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
```

```
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

When the above code is compiled and executed, it produces following result:
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000

Dynamic Memory Allocation

The process of allocating memory at run time is known as dynamic memory allocation. Dynamic memory management technique is used to optimize the use of storage space. These techniques are used to allocate additional memory space or to release the unwanted space at run time. Using dynamic memory management techniques, the programmer can allocate memory whenever he decides and releases it after using the memory.

The functions used in the dynamic memory management are
(i) malloc ( ) (ii) calloc( ) (iii) realloc( ) and (iv) free ( ).

malloc()

The name malloc stands for "memory allocation". This function is used to allocate a block of memory. The required amount of byte should be specified as argument to the function. After allocating memory in the heap (free memory) the function returns the starting address of the block of memory allotted.

Syntax

```
ptr = (cast_type *) malloc (size);
```

where ptr is a pointer of type cast_type.

For example the statement p = (int*) malloc(100); reserves 100 bytes of the memory and the address of the first byte of the memory allocated is assigned to the pointer p of type int.

Example

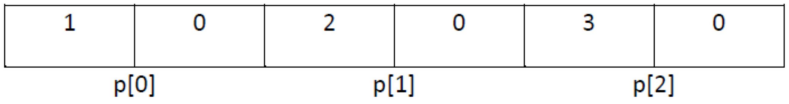
```
main()
{
int *p;
p = (int*) malloc(6);
p[0] = 10;
p[1] = 20;
p[2] = 30;
printf("\n%d %d %d", p[0] , p[1] , p[2]) ;
free (p);
}
```

Output: 10 20 30

Here 'p' is an integer pointer. A pointer variable stores the memory address.

The malloc( ) allocates 6 bytes in heap and the address is converted to integer address by means of (int *) cast operator. This converted address is assigned to the pointer 'p'.
Now 'p' points to 6 byte of memory block each 2 byte pair representing an integer. In short 'p' is an array having 3 elements namely p[0], p[1] and p[2].

There after 'p' can be used as if it is an array. But the difference is that whenever required, the elements can be created and after having used it may be released using free( ). The following figure shows this setup.



free()

This function is used to deallocate the memory. The release of storage space is important when the storage is limited. When data stored in a block of memory is not needed, then release that portion of the memory.

Syntax

```
free(pointer variable);
```

where pointer variable is a pointer to a memory block which has already been created by malloc or calloc.

Example

```
main()
    {
    int *p;
    p = ( int*) malloc(6);
    ------
    free (p);
    }
```

calloc()

The name calloc stands for "contiguous allocation" malloc( ) is used to allocate single block of memory. Whereas calloc( ) will allocate multiple blocks of storage, each of the same size and initialize all the bytes of the memory to zero. calloc function is used when requesting memory space at run time for storing derived data types such as arrays and structure.

Syntax

```
p = (cast_type) *) calloc (number of blocks, block size);
```

This allocates continuously memory blocks each with the specified size. The returned address will be the address of the starting byte. If the required amount of memory is not available, then NULL value is returned.

Example

```
main()
{
int *p;
p = (int*) calloc(10, sizeof(int));
------
}
```

This allocates 10 continuous blocks of memory and each block having 2 bytes.

Differences between malloc() and calloc()

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 malloc allocates a single block of storage space, calloc allocates multiple block of storage, each of the same size, and then sets all bytes to zero.

relloc()

This function is used to change the memory size previously allocated. The space may be increased or decreased.

Syntax

ptr= realloc (pointer variable name, new size);

This function allocates a new memory space of size equal to the new size to the pointer variable ptr and returns a pointer to the first byte of the new memory block.

The new size may be large or smaller then the original size. Consider the following **example.**

```
main()
{
int *p;
p = (int*) malloc(10);
------
------
P= (int*) realloc(p,20);
}
```

Here initially 'p' is allotted with 10 bytes. But later it is felt that amount of memory is to be increased. That is done with the help of realloc( ) function. This time 20 bytes are allotted.

Like that, it is also possible to decrease the memory size.

Here also if it is not possible to allocate memory space, realloc( ) returns NULL value, and the original block is lost. Here, ptr is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int *ptr,i,n1,n2;
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
printf("%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
printf("%u\t",ptr+i);
return 0;
}
```

Command Line Arguments

It is possible to pass some values from the command line to C programs when they are executed.These values are called command line arguments and many times they are important to control the program. Command line arguments are parameters. These parameters are supplied to a program when the program is executed. The first parameters may represent a file name the program should process.

main( ) function can also take arguments like other functions. main( ) can take two arguments. So when main is called, it is called with two arguments. They are argc and argv. The information contained in the command lines is passed onto the program through these arguments.

The argc is called as an "argument counter" and it represents the number of command line arguments.

The argv is an argument vector, is a pointer to an arrays of character strings that containing the arguments, one per string. The size of this array will be equal to the value of argc.

The c in argc stands for counter and v in argv stands for vector.

In order to access the command line arguments, main function is declared as follows:

```
main (int argc, char *argv[])
{
------
}
```

The first parameter in the command line is always the program name. Therefore argv[0] represents the program name.