UNIT V

Pointers: Understanding Computer Memory – Introduction to Pointers – declaring Pointer Variables – Pointer Expressions and Pointer Arithmetic – Null Pointers – Generic Pointers - Passing Arguments to Functions using Pointer – Pointer and Arrays – Passing Array to Function.

Structure, Union, and Enumerated Data Types: Introduction – Nested Structures – Arrays of Structures – Structures and Functions - Unions – Enumerated Data Types.

Files: Introduction to Files – Using Files in C – Reading Data from Files – Writing Data from Files – Detecting the End-of-file –Close a file – Random Access Files – Binary Files – Command line arguments.

5.1 Understanding Computer Memory

5.2 Introduction to Pointers

Pointer is a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers. C uses *pointers* a lot because:

- It is the only way to express some computations.
- It produces compact and efficient code.
- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

C uses pointers explicitly with arrays, structures and functions.

A **pointer** is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The ***unary*** operator **&** gives the "address of a variable''. The ***indirection*** or dereference operator ***** gives the "contents of an object ***pointed to*** by a pointer''.

5.3 Declaring Pointer Variables

The general syntax of pointer declaration is,

```
datatype *pointer_name;
```

The data type of the pointer and the variable to which the pointer variable is pointing must be the same.

Example

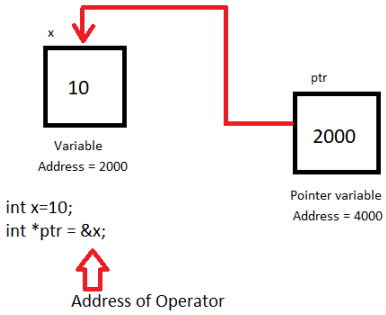
```
int *pointer;
float *p;
char *x;
```

We must associate a pointer to a particular type. We can't assign the address of a short int to a long int.

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a pointer variable. It contains the address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10;
int *ptr;       //pointer declaration
ptr = &a;       //pointer initialization
```



Pointer variable always points to variables of the same datatype. For example:

```
float a;
int *ptr = &a;       // ERROR, type mismatch
```

Consider the effect of the following code:

```
#include <stdio.h>
main()
{
        int x = 1, y = 2;
        int *ip;
        ip = &x;
        y = *ip;
        *ip = 3;
}
```

It is worth considering what is going on at the machine level in memory to fully understand how pointer works. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000 shown in figure 4.1.

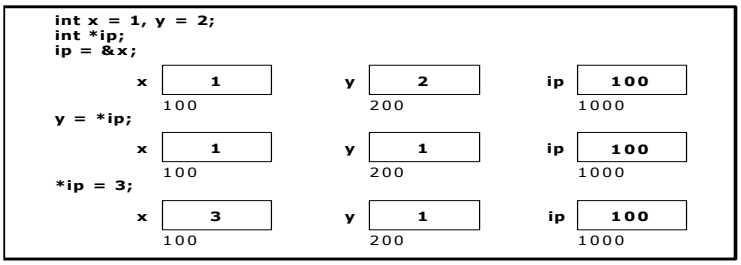


Fig. 4.1 Pointer, Variables and Memory

Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a pointer to an integer and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the contents of ip. In this example ip currently points to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1. Finally, we can assign a value 3 to the contents of a pointer (*ip).

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So,

```
int *ip;
```

```
*ip = 100;
```
will generate an error (program crash!!). The correct usage is:

```
int *ip;
int x;
ip = &x;
*ip = 100;
++ip;
```

5.4 Pointer Expressions and Pointer Arithmetic

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions, such as assignments, conversions, and arithmetic.

Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
int s = 56;
int *ptr1, *ptr2;
ptr1 = &s;
ptr2 = ptr1;
```

Program

```
#include <stdio.h>
int main(void)
{
        int s = 56;
        int *ptr1, *ptr2;
        ptr1 = &s;
        ptr2 = ptr1;
        /* print the value of s twice */
        printf("Values at ptr1 and ptr2: %d %d \n", *ptr1, *ptr2);
        /* print the address of s twice */
        printf("Addresses pointed to by ptr1 and ptr2: %p %p",ptr1,ptr2);
        return 0;
}
```

Output

```
Values at ptr1 and ptr2: 56 56
Addresses pointed to by ptr1 and ptr2: 0240FF20 0240FF20
```

Pointer Arithmetic

We can perform addition and subtraction of integer constant from pointer variable.

Addition

```
ptr1 = ptr1 + 2;
```

subtraction

```
ptr1 = ptr1 - 2;
```

We can not perform addition, multiplication and division operations on two pointer variables.

For Example:

```
ptr1 + ptr2 is not valid
```

However we can subtract one pointer variable from another pointer variable. We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.

For Example:

```
ptr1++;
ptr2--;
```

Multiplication

Example:

```
int x = 10, y = 20, z;
int *ptr1 = &x;
int *ptr2 = &y;
z = *ptr1 * *ptr2 ;
```

Will assign 200 to variable z.

Division

there is a blank space between '/' and * because the symbol /*is considered as beginning of the comment and therefore the statement fails.

```
Z=5*-*Ptr2/ *Ptr1;
```

If Ptr1 and Ptr2 are properly declared and initialized pointers, then the following statements are valid:

```
Y=*Ptr1**Ptr2;
Sum=sum+*Ptr1;
*Ptr2=*Ptr2+10;
*Ptr1=*Ptr1+*Ptr2;
*Ptr1=*Ptr2-*Ptr1;
```

```
Ptr1 = Ptr1 + 1 = 1000 + 2    = 1002;
Ptr1 = Ptr1 + 2 = 1000+ (2*2) = 1004;
Ptr1 = Ptr1 + 4 = 1000+ (2*4) = 1008;
Ptr2 = Ptr2 + 2 = 3000+ (2*2) = 3004;
Ptr2 = Ptr2 + 6 = 3000+ (2*6) = 3012;
```

Here addition means bytes that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

if Ptr1 and Ptr2 are properly declared and initialized pointers then, 'C' allows adding integers to a pointer variable.

EX:

```
int a=5, b=10;
int *Ptr1,*Ptr2;
Ptr1=&a;
Ptr2=&b
```

If Ptr1 & Ptr2 are properly declared and initialized, pointers then 'C' allows to subtract integers from pointers. From the above example,

```
Ptr1 = Ptr1 - 1 = 1000-2  = 998;
Ptr1 = Ptr1 - 2 = 1000-4  = 996;
Ptr1 = Ptr1 - 4 = 1000-8  = 992;
Ptr2 = Ptr2 - 2 = 3000-4  = 2996;
Ptr2 = Ptr2 - 6 = 3000-12 = 2988;
```

Here the subtraction means byte that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

If Ptr1 & Ptr2 are properly declared and initialize pointers, and both points to the elements of the same type. "Subtraction of one pointer from another pointer is also possible".

NOTE: this operation is done when both pointer variable points to the elements of the same array.

EX:

P2- P1 (It gives the number of elements between p1 and p2)

Pointer Increment and Scale Factor

We can use increment operator to increment the address of the pointer variable so that it points to next memory location.

The value by which the address of the pointer variable will increment is not fixed. It depends upon the data type of the pointer variable.

For Example:

```
int *ptr;
ptr++;
```

It will increment the address of pointer variable by 2. So if the address of pointer variable is 2000 then after increment it becomes 2002.

Thus the value by which address of the pointer variable increments is known as scale factor. The scale factor is different for different data types as shown below:

Char	1 Byte
Int	2 Byte
Short int	2 Byte
Long int	4 Byte
Float	4 Byte
Double	8 Byte
Long double	10 Byte

Write a C program to compute the sum of all elements stored in an array Using pointers.

Program

```
/*program to compute sum of all elements stored in an  array */
#include<stdio.h>
#include<conio.h>
main ()
{
        int a [10], i, sum=0,*p;
        printf ("enter 10 elements \n");
        for (i=0; i<10; i++)
        scanf ("%d", & a[i]);
        p = a;
        for (i = 0; i<10; i++)
        {
        sum = sum+(*p);
        p++;
        }
        printf ("the sum is % d", sum);
        getch ();
}
```

Output

```
enter 10 elements
1
2
3
4
5
```

6
7
8
9
10
the sum is 55

Write a C program using pointers to determine the length of a character String.

Program

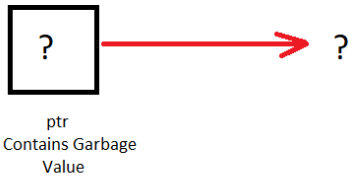
```
/*program to find the length of a char string */
#include<stdio.h>
#include<conio.h>
main ()
{
        char str[20], *ptr ;
        int l=0;
        printf("enter a string \n");
        scanf("%s", str);
        ptr=str;
        while(*ptr!='\0')
        {
        l++;
        ptr++;
        }
        printf("the length of the given string is %d \n", l);

 }
```

Output

```
enter a string
atnyla.com
the length of the given string is 10
Press any key to continue .
```

5.5 Null Pointers

While declaring a pointer variable, if it is not assigned to anything then it contains garbage value. Therefore, it is recommended to assign a NULL value to it,



A pointer that is assigned a NULL value is called a **NULL pointer in C**

- NULL Pointer is a pointer which is pointing to nothing.
- The NULL pointer points the base address of the segment.
- In case, if you don't have an address to be assigned to a pointer then you can simply use NULL
- The pointer which is initialized with the NULL value is considered as a NULL pointer.
- NULL is a macro constant defined in following header files –
  stdio.h

alloc.h
mem.h
stddef.h
stdlib.h

Defining NULL Value

#define NULL 0

Below are some of the variable representations of a NULL pointer.

```
float *ptr  = (float *)0;
char  *ptr  = (char *)0;
double *ptr = (double *)0;
char *ptr   = '\0';
int *ptr    = NULL;
```

Example of NULL Pointer

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

Output :

The value of ptr is 0

5.6 Generic Pointers

When a variable is declared as being a pointer to type **void,** it is known as a *generic pointer*. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term ***Generic pointer***.

This is very useful when you want a pointer to point to data of different types at different times.

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Why Void Pointers is important

1. Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
2. Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as an integer pointer, character pointer.

Declaration of Void Pointer

void * pointer_name;

Void Pointer Example :

```
void *ptr;    // ptr is declared as Void pointer
char c;
int i;
float f;
ptr = &c;  // ptr has address of character data
ptr = &i;  // ptr has address of integer data
ptr = &f;  // ptr has address of float data
```

Explanation :

void *ptr;

1. **Void pointer** declaration is shown above.

2. We have declared 3 variables of integer, character and float type.
3. When we assign **the address of the integer** to the void pointer, the pointer will become Integer Pointer.
4. When we assign **the address of Character** Data type to void pointer it will become Character Pointer.
5. Similarly, we can assign the address of any data type to the void pointer.
6. It is capable of storing the address of any data type

Example of Generic Pointer

Here is some code using a void pointer:

```
#include<stdlib.h>
 int main()
{
   int x = 4;
   float y = 5.5;
    //A void pointer
   void *ptr;
   ptr = &x;
    // (int*)ptr - does type casting of void
   // *((int*)ptr) dereferences the typecasted
   // void pointer variable.
   printf("Integer variable is = %d", *( (int*) ptr) );
    // void pointer is now float
   ptr = &y;
   printf("\nFloat variable is= %f", *( (float*) ptr) );
   return 0;
}
```

Another Example

```
#include"stdio.h"
int main()
{
  int i;
  char c;
  void *the_data;
  i = 6;
  c = 'a';
  the_data = &i;
  printf("the_data points to the integer value %d\n", *(int*) the_data);
  the_data = &c;
  printf("the_data now points to the character %c\n", *(char*) the_data);
  return 0;
}
```

5.7 Passing Arguments to Functions using Pointer

A function has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

Pointer as a function parameter list is used to hold the address of argument passed during the function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will affect the original variable.

Program

```
#include<stdio.h>
void swap(int *a, int *b); // function prototype
int main()
{
   int p=10, q=20;
   printf("Before Swapping:\n\n");
   printf("p = %d\n",p);
   printf("q = %d\n\n",q);
   swap(&p,&q); //passing address of p and q to the swap function
   printf("After Swapping:\n\n");
   printf("p = %d\n",p);
   printf("q = %d\n",q);
   return 0;
}

//pointer a and b holds and points to the address of p and q
void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Output

```
Before Swapping:
p = 10
q = 20

After Swapping:
p = 20
q = 10
Press any key to continue . . .
```

The address of memory locationmandnare passed to the function swap and the pointers*aand*baccept those values.

So, now the pointeraandbpoints to the address ofmandnrespectively.

When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly.

Hence, changes made to*aand*bare reflected inmandnin the main function.

This technique is known as Call by Reference in C programming.

Simple Example of Pointer to Function

Program

```
#include<stdio.h>
int add(int x, int y)
{
 return x+y;
}

int main( )
{
        int (*functionPtr)(int, int);
        int s;
        functionPtr = add;   //
```

```
        s = functionPtr(20, 45);
        printf("Sum is %d",s);
        getch();
        return 0;
}
```

Output

Sum is 65

Explanation

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name)(parameter);
```

Example :

```
        int (*add)();   //legal declaration of pointer to function
        int *add();   //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of the function.

```
        int add(int, int);
        int (*s)(int, int);
        sr = add;
```

sr is a pointer to a function sum. Now sum can be called using function pointer s with the list of parameter.

```
        sr(10, 20);
```

Function returning Pointer

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only till inside the function. Hence if you return a pointer connected to a local variable, that pointer be will pointing to nothing when function ends.

Program

This program will check who is larger among two number, it is not for quality checking

```
#include<stdio.h>
int* checklarger(int*, int*);
void main()
{
        int num1 ;
        int num2;
        int *ptr;
        printf("Enter Two number: \n");
        scanf("%d %d",&num1,&num2);
        ptr = checklarger(&num1, &num2);
        printf("%d is larger \n",*ptr);
}

int* checklarger(int *m, int *n)
{
        if(*m > *n)
                return m;
        else
                return n;
}
```

Output

```
Enter Two number:
        546    1213
1213 is larger
```

Address of the Function

We can fetch the address of an array by the array name, without indexes, Similarly We can fetch the address of a function by using the function's name without any parentheses or arguments. To see how this is done, read the following program, which compares two strings entered by the user. Pay close attention to the declarations of checkString( ) and the function pointer p, inside main( ).

```
#include<stdio.h>
void checkString(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
        char strng1[80], strng2[80];
        int (*ptr)(const char *, const char *); /* function pointer */
        ptr = strcmp; /* assign address of strcmp to ptr */
        printf("Enter two strings.\n");
        gets(strng1);
        gets(strng2);
        checkString(strng1,strng2,ptr); /* pass address of strcmp via ptr */
        return 0;
}

void checkString(char *m, char *n,
int (*cmp) (const char *, const char *))
{
        printf("Testing for equality.\n");
        if(!(*cmp)(m, n))
        {
                printf("Equal \n");
        }
        Else
        {
                 printf("Not Equal \n");
        }
}
Output 1:
Enter two strings.
        atnyla
        atnyla
        Testing for equality.
        Equal

Output 2:
Enter two strings.
atnyla
atnlla
Testing for equality.
Not Equal
Press any key to continue . . .
```

Explanation
First, examine the declaration for ptr in main( ). It is shown here:

int (*ptr)(const char *, const char *);
This declaration tells the compiler that ptr is a pointer to a function that has two const char * parameters, and returns an int result. The parentheses around ptr are necessary in order for the compiler to properly interpret this declaration. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ.

void checkString(char *m, char *n,
int (*cmp) (const char *, const char *))
Next, examine the checkString( ) function. It declares three parameters: two character pointers, m and n, and one function pointer, cmp. Notice that the function pointer is declared using the same format as was ptr inside main( ). Thus, cmp is able to receive a pointer to a function that takes two const char * arguments and returns an int result. Like the declaration for ptr, the parentheses around the *cmp are necessary for the compiler to interpret this statement correctly.

When the program begins, it assigns ptr the address of strcmp( ), the standard string comparison function. Next, it prompts the user for two strings, and then it passes pointers to those strings along with ptr to check( ), which compares the strings for equality. Inside checkString( ), the expression

(*cmp)(a, b)
calls strcmp( ), which is pointed to by cmp, with the arguments m and n. The parentheses around *cmp are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, can also be used.

cmp(a, b);
The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer (that is, that cmp is a function pointer, not the name of a function). Also, the first style was the form originally specified by C.

Note that you can call checkString( ) by using strcmp( ) directly, as shown here:

checkString(s1, s2, strcmp);
5.8 **Pointer and Arrays**
There is a close association between pointers and arrays. Let us consider the following statements:

```
int x[5] = {11, 22, 33, 44, 55};
int *p = x;
```

The array initialization statement is familiar to us. The second statement, array name x is the starting address of the array. Let we take a sample memory map as shown in figure 4.2.:
From the figure 4.2 we can see that the starting address of the array is 1000. When x is an array, it also represents an address and so there is no need to use the (&) symbol before x. We can write int *p = x in place of writing int *p = &x[0].

The content of p is 1000 (see the memory map given below). To access the value in x[0] by using pointers, the indirection operator * with its pointer variable p by the notation *p can be used.

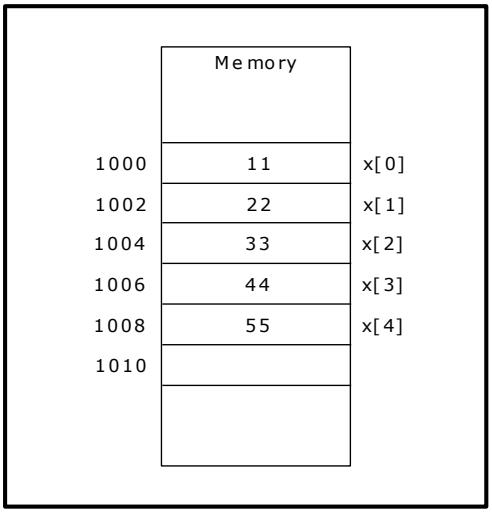


Figure 4.2. Memory map - Arrays

The increment operator ++ helps you to increment the value of the pointer variable by the size of the data type it points to. Therefore, the expression p++ will increment p by 2 bytes (as p points to an integer) and new value in p will be 1000 + 2 = 1002, now *p will get you 22 which is x[1].

Consider the following expressions:

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

How would they be evaluated when the integers 10 & 20 are stored at addresses 1000 and 1002 respectively with p set to 1000.

p++ : The increment ++ operator has a higher priority than the indirection operator * . Therefore p is increment first. The new value in p is then 1002 and the content at this address is 20.

*(p++): is same as *p++.

(*p)++: *p which is content at address 1000 (i.e. 10) is incremented. Therefore (*p)++ is 11.

Note that, *p++ = content at incremented address.

Example:

```
#include <stdio.h>
main()
{
        int x[5] = {11, 22, 33, 44, 55};
        int *p = x, i;                          /*  p=&x[0] = address of the first element */
        for (i = 0; i < 5; i++)
        {
                printf ("\n x[%d] = %d", i, *p);          /* increment the address*/
                p++;
        }
}
```

Output:

x [0] = 11
x [1] = 22
x [2] = 33
x [3] = 44
x [4] = 55

The meanings of the expressions p, p+1, p+2, p+3, p+4 and the expressions *p, *(p+1), *(p+2), *(p+3), *(p+4) are as follows:

P = 1000 P+1 = 1000 + 1 x 2 = 1002 P+2 = 1000 + 2 x 2 = 1004 P+3 = 1000 + 3 x 2 = 1006 P+4 = 1000 + 4 x 2 = 1008	*p = content at address 1000 = x[0] *(p+1) = content at address 1002 = x[1] *(p+2) = content at address 1004 = x[2] *(p+3) = content at address 1006 = x[3] *(p+4) = content at address 1008 = x[4]

Pointers and strings:

A string is an array of characters. Thus pointer notation can be applied to the characters in strings. Consider the statements:

```
char tv[20] = "ONIDA";
char *p = tv;
```

For the first statement, the compiler allocates 20 bytes of memory and stores in the first six bytes the char values as shown below:

Variable	tv[0]	tv[1]	tv[2]	tv[3]	tv[4]	tv[5]
Value	'O'	'N'	'I'	'D'	'A'	'\0'
Address	1000	1001	1002	1003	1004	1005

The statement:

```
char *p = tv;                    /* or  p = &tv[0] */
```

Assigns the address 1000 to p. Now, we will write a program to find the length of the string tv and print the string in reverse order using pointer notation.

Example:

```
#include <stdio.h>
main()
{
        int n, i;
        char tv[20] = "ONIDA";                  /* p = 1000 */
        char *p = tv, *q;                       /* p = &tv[0], q is a pointer */
        q = p;
        while (*p != '\0')              /* content at address of p is not null character */
                p++;
        n = p - q;                              /* length of the string */
        --p;                    /* make p point to the last character A in the string */
        printf ("\nLength of the string is %d", n);
        printf ("\nString in reverse order: \n");
        for (i=0; i<n; i++)
        {
                putchar (*p);
                p--;
        }
```

```
}
```
Output:
Length of the string is 5
String in reverse order: ADINO

5.9 Passing Array to Function.
If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.
Way-1
Formal parameters as a pointer –
```
void myFunction(int *param) {
  .
  .
  .
}
```
Way-2
Formal parameters as a sized array –
```
void myFunction(int param[10])
{
  .
  .
  .
}
```
Way-3
Formal parameters as an unsized array –
```
void myFunction(int param[])
{
  .
  .
  .
}
```
Example
Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows –
```
double getAverage(int arr[], int size)
{
  int i;
  double avg;
  double sum = 0;
  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }
  avg = sum / size;
  return avg;
}
```
Now, let us call the above function as follows –
```
#include <stdio.h>
 /* function declaration */
double getAverage(int arr[], int size);
```

```
int main ()
{
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   avg = getAverage( balance, 5 ) ;
   printf( "Average value is: %f ", avg );
     return 0;
}
```

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

```
Average value is: 214.400000
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.

Structure, Union, and Enumerated Data Types: Introduction – Nested Structures – Arrays of Structures – Structures and Functions - Unions – Enumerated Data Types

Introduction:

C allows the programmer to create custom data types in five different ways. These are:

- The **structure**, which is a collection of variables under one name.
- Types created with **typedef**, which defines a new name for an existing type.
- The **union**, which enables the same piece of memory to be defined as two or more types of variables.
- The **enumeration**, which is a list of symbols.
- The **bit-field**, a variation of the structure, which allows easy access to the bits of a word.

Structure:

Structure is a user-defined data type in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together. In structure, data is stored in form of **records**.

Defining a structure

- struct keyword is used to define a structure.
- struct defines a new data type which is a collection of primary and derived data types.
- **Syntax:**
    - struct [structure_tag]
    - {
        - //member variable 1
        - //member variable 2
        - //member variable 3 ...
    - }[structure_variables];
- As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.
- After the closing curly brace, we can specify one or more structure variables, again this is optional.
- **Note:** The closing curly brace in the structure type declaration must be followed by a semicolon(;).

Example of Structure:

- struct Student
- {
    - char name[25];

        - int age;
        - char branch[10]; //
        - Char gender; //F for female and M for male
    - };
- Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender.
- These fields are called **structure elements or members**.
- Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc.
- **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables

- It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined.
- **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype.
- Structure variables can be declared in following two ways:

Declaring Structure variables separately

```
struct Student
{
        char name[25];
        int age;
        char branch[10];
        char gender; //F for female and M for male
}; struct Student S1, S2;
//declaring variables of struct Student
```

Declaring Structure variables with structure definition

```
struct Student
{
        char name[25];
        int age;
        char branch[10]; //F for female and M for male
        char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student.

Accessing Structure Members

- Structure members can be accessed and assigned values in a number of ways.
- Structure members have no meaning individually without the structure.
- In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot. operator also called **period** or **member access** operator.

For example:

```
#include<stdio.h>
#include<string.h>
struct Student
{
        char name[25];
        int age;
        char branch[10];
        char gender; //F for female and M for male
};
```

```
int main()
{
        struct Student s1; /* s1 is a variable of Student type and age is a member
        of Student */
        s1.age = 18; /* using string function to add name */ strcpy(s1.name,
        "Viraaj"); /* displaying the stored values */ printf("Name of Student 1:
        %s\n", s1.name);
        printf("Age of Student 1: %d\n", s1.age);
         return 0;
        }
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

Structure Initialization

- Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
        float height;
        int weight;
        int age;
        Char name[10];
}; struct Patient p1 = { 180.75 , 73, 23, "munwar" }; //initializationor,
struct Patient p1;
p1.height = 180.75; //initialization of each member separately
p1.weight = 73;
p1.age = 23;
P1.gender='F';
```

Array of Structure

- We can also declare an array of **structure** variables.
- In which each element of the array will represent a **structure** variable.
  - **Example :** struct employee emp[5];
- The below program defines an array emp of size 5.
- Each element of the array emp is of type Employee.

```
#include<stdio.h>
struct Employee
{
        char ename[10];
        int sal;
};
struct Employee emp[5];
int i, j;
void ask()
{
        for(i = 0; i < 3; i++)
        {
        printf("\nEnter %dst Employee record:\n", i+1);
        printf("\nEmployee name:\t");
        scanf("%s", emp[i].ename);
        printf("\nEnter Salary:\t");
```

```
        scanf("%d", &emp[i].sal); }
        printf("\nDisplaying Employee record:\n");
        for(i = 0; i < 3; i++)
        {
                printf("\nEmployee name is %s", emp[i].ename);
                printf("\nSlary is %d", emp[i].sal);
        }
}
void main()
{
        ask();
}
```

Nested Structures

- Nesting of structures, is also permitted in C language.
- Nested structures means, that one structure has another stucture as member variable.
- **Example:**

```
struct Student
{
char[30] name;
int age; /* here Address is a structure */
struct Address
{
char[50] locality;
char[50] city;
int pincode;
}addr;
};
```

Structure as Function Arguments

- We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

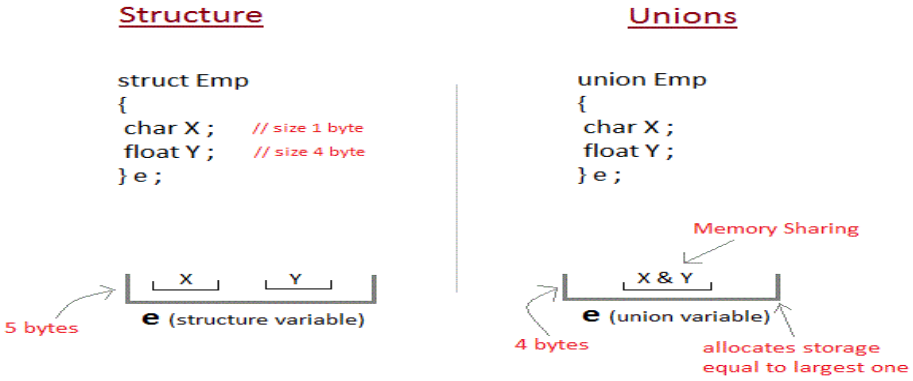
- **Example:**

```
#include<stdio.h>
struct Student
{
        char name[10];
        int roll;
};
void show(struct Student st);
void main()
{
        struct Student std;
        printf("\nEnter Student record:\n");
        printf("\nStudent name:\t");
        scanf("%s", std.name);
        printf("\nEnter Student rollno.:\t");
        scanf("%d", &std.roll);
        show(std);
}
```

```
void show(struct Student st)
{
        printf("\nstudent name is %s", st.name);
        printf("\nroll is %d", st.roll);
}
```

C Unions

- **Unions** are conceptually similar to **structures**.
- The syntax to declare/define a union is also similar to that of a structure.
- The only differences is in terms of storage.
- In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



- This implies that although a **union** may contain many members of different types, **it cannot handle all the members at the same time**. A **union** is declared using the union keyword.

```
union item
        {
                int m;
                float x;
                char c;
        }It1;
```

- In the union declared above the member x requires **4 bytes** which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

Accessing a Union Member in C

- Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
        int a;
        float b;
        char c;
}t;
t.a; //to access members of union t
t.b;
```

t.c;

Example

```
#include <stdio.h>
union item
{
        int a;
        float b;
        char ch;
};
int main( )
{
        union item it;
        it.a = 12;
        it.b = 20.2;
        it.ch = 'z';
        printf("%d\n", it.a);
        printf("%f\n", it.b);
        printf("%c\n", it.ch);
        return 0;
}
```

Arrays of Unions Variables

- Like array of structures we can also create array of unions and access them in similar way. When array of unions are created it will create each element of the array as individual unions with all the features of union. That means, each array element will be allocated a memory which is equivalent to the maximum size of the union member and any one of the union member will be accessed by array element.

union category

{

int intClass;

char chrDeptId**[**10**]**;

};

- union category catg **[**10**]**; // creates an array of unions with 10 elements of union type
- Members of array of unions are accessed using '.' operator on union variable name along with the index to specify which array element that we are accessing.
- catg[0].intClass = 10;
- catg[5].chrDeptId = "DEPT_001";
- Here note that, each element of the union array need not access the same member of the union. It can have any member of the union as array element at any point in

time. In above example, first element of the union array accesses intClass while 6th element of union array has chrDeptId as its member.

Unions inside Structures

In the below code, we have union embedded within a structure. We know, the fields of a union will share memory, so in main program we ask the user which data he/she would like to store and depending on the user choice the appropriate field will be used. By this way we can use the memory efficiently.

```
#include <stdio.h>
struct student
{
union
{               //anonymous union (unnamed union)
        char name[10];
        int roll;
};
int mark;
};
 int main()
{
        struct student stud;
        char choice;
        printf("\n You can enter your name or roll number ");
        printf("\n Do you want to enter the name (y or n): ");
        scanf("%c",&choice);
        if(choice=='y'||choice=='Y')
        {
                printf("\n Enter name: ");
                scanf("%s",stud.name);
                printf("\n Name:%s",stud.name);
        }
        else
        {
                printf("\n Enter roll number");
                scanf("%d",&stud.roll);
                printf("\n Roll:%d",stud.roll);
        }
        printf("\n Enter marks");
        scanf("%d",&stud.mark);
        printf("\n Marks:%d",stud.mark);
        return 0;
}
```

Output

```
You can enter your name or roll number Do you want to enter the name (y or n) : y
Enter name: john
Name:john
Enter marks: 45
 Marks:45
```

Structures inside Unions

Example for defining a structure inside union is given below :

```
#include<stdio.h>
int main() {
struct student
{
        char name[30];
        int rollno;
        float percentage;
};
union details
{
        struct student s1;
};
union details set;
printf("Enter details:");
printf("\nEnter name : ");
scanf("%s", set.s1.name);
printf("\nEnter roll no : ");
scanf("%d", &set.s1.rollno);
printf("\nEnter percentage :");
scanf("%f", &set.s1.percentage);
printf("\nThe student details are : \n");
printf("\name : %s", set.s1.name);
printf("\nRollno : %d", set.s1.rollno);
printf("\nPercentage : %f", set.s1.percentage);
return 0;
}
```

Enumerated Data Types

- Enumeration is a user defined datatype in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain. The keyword "enum" is used to declare an enumeration.
- Here is the syntax of enum in C language,
- enum enum_name{const1, const2, ... };
- The enum keyword is also used to define the variables of enum type. There are two ways to define the variables of enum type as follows.
    - enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};
    - enum week day;
- Here is an example of enum in C language,
- **Example**

```
#include<stdio.h>
enum week
{
        Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun
};
enum day
{
        Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund
};
int main()
{
```

```
        printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon ,
        Tue, Wed, Thur, Fri, Sat, Sun);
        printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond ,
        Tues, Wedn, Thurs, Frid, Satu, Sund);
    return 0;
    }
```

Output

```
The value of enum week: 10111213101617
The default value of enum day: 0123181112
```

- In the above program, two enums are declared as week and day outside the main() function
- In the main() function, the values of enum elements are printed.

```
    enum week
    {
        Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun
    };
    enum day
    {
        Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund
    };
    int main()
    {
        printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon ,
        Tue, Wed, Thur, Fri, Sat, Sun);
        printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond ,
        Tues, Wedn, Thurs, Frid, Satu,    Sund);
}
```

Structure Vs. Union

Structure	**Union**
You can use a struct keyword to define a structure.	You can use a union keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.

It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

Advantages of structure

- Here are pros/benefits for using structure:
  - Structures gather more than one piece of data about the same subject together in the same place.
  - It is helpful when you want to gather the data of similar data types and parameters like first name, last name, etc.
  - It is very easy to maintain as we can represent the whole record by using a single name.
  - In structure, we can pass complete set of records to any function using a single parameter.
  - You can use an array of structure to store more records with similar types.

Advantages of union

- Here, are pros/benefits for using union:
  - It occupies less memory compared to structure.
  - When you use union, only the last variable can be directly accessed.
  - Union is used when you have to use the same memory location for two or more data members.
  - It enables you to hold data of only one data member.
  - Its allocated space is equal to maximum size of the data member.

Disadvantages of structure

- Here are cons/drawbacks for using structure:
  - If the complexity of IT project goes beyond the limit, it becomes hard to manage.
  - Change of one data structure in a code necessitates changes at many other places. Therefore, the changes become hard to track.
  - Structure is slower because it requires storage space for all the data.
  - You can retrieve any member at a time in structure whereas you can access one member at a time in the union.
  - Structure occupies space for each and every member written in inner parameters while union occupies space for a member having the highest size written in inner parameters.
  - Structure supports flexible array. Union does not support a flexible array.

Disadvantages of union

- Here, are cons/drawbacks for using union:
  - You can use only one union member at a time.
  - All the union variables cannot be initialized or used with varying values at a time.
  - Union assigns one common storage space for all its members.

Files: Introduction to Files – Using Files in C – Reading Data from Files – Writing Data from Files – Detecting the End-of-file –Close a file – Random Access Files – Binary Files – Command line arguments.

Files:

File is a bunch of bytes stored in a particular area on some storage devices like floppy disk, hard disk, magnetic tape and cd-rom etc., which helps for the permanent storage.

There are 2 ways of accessing the files. These are:

- Sequential access: the data can be stored or read back sequentially.
- Random access: the data can be access randomly.

If a file can support random access (sometimes referred to as position requests), opening a file also initializes the file position indicator to the start of a file. This indicator is incremented as each character is read from, or written to, the file.

The close operation disassociates a file from a specific stream. If the file was opened for output, the close operation will write the contents (if any) of the stream to the device. This process is usually called flushing the stream.

All files are closed automatically when the program terminates, but not when it crashes. Each stream associated with a file has a file control structure of type ***FILE***.

4.2.1. Streams:

Even though different devices are involved (terminals, disk drives, etc), the buffered file system transforms each into a logical device called a **stream**. Because streams are device-independent, the same function can write to a disk file or to another device, such as a console. There are two types of streams:

Text Streams: A text stream is a sequence of characters. In a text stream, certain character translations may occur (for example, a newline may be converted to a carriage return/line-feed pair). This means that there may not be a one-to-one relationship between the characters written and those in the external device.

Binary Streams: A binary stream is a sequence of bytes with a one-to-one correspondence to those on the external device (i.e, no translations occur). The number of bytes written or read is the same as the number on the external device. However, an implementation-defined number of bytes may be appended to a binary stream (for example, to pad the information so that it fills a sector on a disk).

4.2.2. File Input and Output functions:

The ANSI file system comprises several interrelated functions. These are:

Function	Description
fopen()	Opens a file.

fclose()	Closes a file.
putc()	Writes a character.
fputc()	Writes a character.
getc()	Reads a character.
fgetc()	Reads a character.
fseek()	Seeks a specified byte in a file.
fprintf()	Is to a file what printf() is to the console.
fscanf()	Is to a file what scanf() is to a console.
feof()	Returns TRUE if end-of-file is reached.
ferror()	Returns TRUE if an error is detected.
rewind()	Resets file position to beginning of file.
remove()	Erases a file.
fflush()	Flushes a file.

Most of these functions begin with the letter "f". The header file stdio.h provides the prototypes for the I/O function and defines these three types:

```
typedef  unsigned long    size_t
typedef  unsigned long    fpos_t
typedef  struct _FILE      FILE
```

stdio.h also defines the following:

```
EOF          -1          /* value returned at end of file */
SEEK_SET      0          /* from beginning of file  */
SEEK_CUR      1          /* from current position  */
SEEK_END      2          /* from end of file  */
```

The latter three are used with fseek() function which performs random access on a file.

4.2.3. The File Pointer:

C treats a file just like a stream of characters and allows input and output as a stream of characters. To store data in file we have to create a buffer area. This buffer area allows information to be read or written on to a data file. The buffer area is automatically created as soon as the file pointer is declared. The general form of declaring a file is:

```
FILE *fp;
```

FILE is a defined data type, all files should be declared as type FILE before they are used. FILE should be compulsorily written in upper case. The pointer fp is referred to as the stream pointer. This pointer contains all the information about the file, which is subsequently used as a communication link between the system and the program.

A file pointer fp is a variable of type FILE that is defined in stdio.h.

4.2.4. Opening a File:

fopen() opens a stream for use, links a file with that stream and returns a pointer associated with that file. The prototype of fopen() function is as follows:

FILE *fopen (const char * *filename*, const char * *mode*);

Where, filename is a pointer to a string of characters that make a valid filename (and may include a path specification) and mode determines how the file will be opened. The legal values for mode are as follows:

Value	Description
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append a binary file for read/write.

A file may be opened in text or binary mode. In most implementations, in text mode, CR/LF sequences are translated to newline characters on input. On output, the reverse occurs. No such translation occurs on binary files.
The following opens a file named TEST for writing:

```
FILE *fp;
fp = fopen ("test", "w");
```

However, because fopen() returns a null pointer if an error occurs when a file is opened, this is better written as:

```
FILE *fp;
if ((fp = fopen ("test", "w")) == NULL)
{
        printf("cannot open file\n");
        exit(1);
}
```

4.2.5. Closing a File:

fclose() closes the stream, writes any data remaining in the disk buffer to the file, does a formal operating system level close on the file, and frees the associated file control block. fclose() has this prototype:

int fclose (FILE **fp*);

A return value of zero signifies a successful operation. Generally, fclose() will fail only when a disk has been prematurely removed or a disk is full.

4.2.6. Writing a Character:

Characters are written using putc() or its equivalent fputc(). The prototype for putc() is:

```
int putc (int ch, FILE *fp);
```

where ch is the character to be output. For historical reasons, ch is defined as an int, but only the low order byte is used.
If the putc() operation is successful, it returns the character written, otherwise it returns EOF.

4.2.7. Reading a Character:

Characters are read using getc() or its equivalent fgetc(). The prototype for getc() is:

```
int getc(FILE *fp);
```

For historical reasons, getc() returns an integer, but the high order byte is zero. getc() returns an EOF when the end of file has been reached. The following code reads a text file to the end:

```
do
{
        ch = getc (fp);
} while(ch != EOF);
```

4.2.8. Using feof():

As previously stated, the buffered file system can also operate on binary data. When a file is opened for binary input, an integer value equal to the EOF mark may be read, causing the EOF condition. To solve this problem, C includes the function feof(), which determines when the end of the file is reached when reading binary data.
The prototype is:

```
int feof (FILE *fp);
```

The following code reads a binary file until end of file is encountered:

```
while (! feof (fp))
        ch = getc(fp);
```

Of course, this method can be applied to text files as well as binary files.

4.2.9. Working With Strings - fputs() and fgets():

In addition to getc() and putc(), C supports the related functions fputs() and fgets(), which read and write character strings. They have the following prototypes:

```
int fputs (const char *str, FILE *fp);

char *fgets (char *str, int length, FILE *fp);
```

The function fputs() works like puts() but writes the string to the specified stream. The fgets() function reads a string until either a newline character is read or length-1 characters have been read. If a newline is read, it will be part of the string (unlike gets()). The resultant string will be null-terminated.

4.2.10. rewind ():

rewind() resets the file position indicator to the beginning of the file. The syntax of rewind() is:

```
rewind(fptr);
```

where, fptr is a file pointer.

4.2.11. ferror ():

ferror() determines whether a file operation has produced an error. It returns TRUE if an error has occurred, otherwise it returns FALSE. ferror() should be called immediately after each file operation, otherwise an error may be lost.

4.2.12. Erasing Files:

remove () erases a specified file. It returns zero if successful.

4.2.13. Flushing a Stream:

fflush () flushes the contents of an output stream. fflush() writes the contents of any unbuffered data to the file associated with fp. It returns zero if successful.

4.2.14. fread () and fwrite ():

To read and write data types which are longer than one byte, the ANSI standard provides fread() and fwrite(). These functions allow the reading and writing of blocks of any type of data. The prototypes are:

size_t fread (void *_buffer_, size_t _num_bytes_, size_t _count_, FILE *_fp_);

size_t fwrite (const void *_buffer_, size_t _num_bytes_, size_t _count_, FILE *_fp_);

For fread(), *buffer* is a pointer to a region of memory which will receive the data from the file. For fwrite(), *buffer* is a pointer to the information which will be written. The buffer may be simply the memory used to hold the variable, for example, &l for a long integer.

The number of bytes to be read/written is specified by *num_bytes*. count determines how many items (each *num_bytes* in length) are read or written.

fread() returns the number of items read. This value may be less than *count* if the end of file is reached or an error occurs. fwrite() returns the number of items written.

One of the most useful applications of fread() and fwrite() involves reading and writing user-defined data types, especially structures. For example, given this structure:

```
struct struct_type
{
        float balance;
        char name[80];
} cust;
```

The following statement writes the contents of cust:

```
fwrite (&cust, sizeof(struct struct_type), 1, fp);
```

4.2.15. fseek() and Random Access I/O:

Random read and write operations may be performed with the help of fseek(), which sets the file position locator. The prototype is:

```
int fseek(FILE *fp, long numbytes, int origin);
```

in which *numbytes* is the number of bytes from the origin, which will become the new current position, and origin is one of the following macros defined in stdio.h:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End-of-file	SEEK_END

fseek() returns 0 when successful and a non-zero value if an error occurs. fseek() may be used to seek in multiples of any type of data by simply multiplying the size of the data by the number of the item to be reached, for example:

```
fseek (fp, 9*sizeof (struct list), SEEK_SET);
```

Which seeks the tenth address.

4.2.16. fprint() and fscanf():

fprint() and fscanf() behave exactly like print() and scanf() except that they operate with files. The prototypes are:

```
int fprintf (FILE *fp, const char *control_string, ...);
int fscanf (FILE *fp, const char *control_string, ...);
```

Although these functions are often the easiest way to read and write assorted data, they are not always the most efficient. Because formatted ASCII data is being written as it would appear on the screen (instead of in binary), extra overhead is incurred with each call. If speed or file size is of concern, use fread() and fwrite().

4.2.17. The Standard Streams:

Whenever a C program starts execution, three streams are opened automatically. These are:

- Standard input (stdin)
- Standard output (stdout)
- Standard error (stderr)

Normally, these streams refer to the console, but they may be redirected by the operating system to some other device or environment. Because the standard streams are file pointers, they may be used to provide buffered I/O operations on the console, for example:

```
putchar(char c)
{
    putc(c, stdout);
}
```

4.3. Command Line Arguments:

Some times it is very useful to pass information into a program when we run it from the command prompt. The general method to pass information into main() function is through the use of command line arguments.

A command line argument is the information that follows the program's name on the command prompt of the operating system.

For example: TC program_name

There are three special built_in_arguments to main(). They are:

- The first argument is argc (argument count) must be an integer value, which represents the number arguments in the command prompt. It will always be at least one because the name of the program qualifies as the first argument.
- The second argument argv (argument vector) is a pointer to an array of strings.
- The third argument env (environment data) is used to access the DOS environmental parameters active at the time the program begins execution.

When an array is used as an argument to function, only the address of the array is passed, not a copy of the entire array. When you call a function with an array name, a pointer to the first element in the array is passed into a function. (In C, an array name without as index is a pointer to the first element in the array).

Each of the command line arguments must be separated by a space or a tab. If you need to pass a command line argument that contains space, you must place it between quotes as:

"this is one argument"

Dclaration of argv must be done properly, A common method is:

```
char *argv[];
```

That is, as a array of undetermined length.
The env parameter is declared the same as the argv parameter, it is a pointer to an array of strings that contain environmental setting.

Example 4.3.1:

```
/*      The following program will print "hello tom".       */
# include <stdio.h>
# include <process.h>
main( int argc, char *argv[])
{
        if(argc!=2)
        {
                printf("You forgot to type your name\n");
                exit(0);
        }
        printf("Hello %s ", argv[1]);
}
```

if the name of this program is name.exe, at the command prompt you have to type as:

- **name tom**

Output:

hello tom

Example 4.3.2:

```
/*      This program prints current environment settings          */
# include <stdio.h>
main (int argc, char *argv[], char *env[])
{
        int i;
        for(i=0; env[i]; i++)
                printf("%s\n", env[i]);
}
```

We must declare both argc and argv parameters even if they are not used because the parameter declarations are position dependent.

4.4. Example Programs on File I/O and Command Line Arguments:

The following programs demonstrate the use of C's file I/O functions.

Example 4.4.1:
Program on fopen(), fclose(), getc(), putc(). Specify filename on command line. Input chars on keyboard until $ is entered. File is then closed. File is then re-opened and read.

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
        FILE *fp;                                       /* file pointer  */
        char ch;
```

```
        if(argc!=2)
        {
                printf("You forgot to enter the file name\n");
                exit(1);
        }

        if((fp=fopen(argv[1], "w"))==NULL)              /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
        do                                          /* get keyboard chars until `$'  */
        {
                ch = getchar();
                putc(ch, fp);
        } while (ch! = '$');

        fclose(fp);                                     /* close file  */

        if((fp=fopen(argv[1], "r"))==NULL)              /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
        ch = getc(fp);                                  /* read one char  */

        while(ch != EOF)
        {
                putchar(ch);                            /* print on screen  */
                ch = getc(fp);                          /* read another char */
        }

        fclose(fp);                                     /* close file  */
}
```

Example 4.4.2:
Program on feof() to check for EOF condition in Binary Files. Specify filenames for input and output at command prompt. The program copies the source file to the destination file. feof() checks for end of file condition. The feof() can also be used for text files.

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
        FILE *in, *out;                                 /* file pointers  */
        char ch;

        if(argc != 3)
        {
                printf("You forgot to enter the filenames\n");
                exit(1);
        }
```

```
        if((in=fopen(argv[1], "rb"))==NULL)              /* open source file  */
        {                                                /* for read binary  */
                printf("Cannot open source file\n");
                exit(1);
        }

        if((out=fopen(argv[2], "wb"))==NULL)              /* open dest file  */
        {                                                 /* for write binary  */
                printf("Cannot open destination file\n");
                exit(1);
        }
        while(! feof(in))                                 /* here it is  */
        {
                ch = getc(in);

                if(! feof(in))                            /* and again  */
                        putc(ch, out);
        }

        fclose(in);                                      /* close files  */
        fclose(out);
}
```

Example 4.4.3:
Program on fputs(), fgets and rewind(). Strings are entered from keyboard until blank line is entered. Strings are then written to a file called 'testfile'. Since gets() does not store the newline character, '\n' is added before the string is written so that the file can be read more easily. The file is then rewind, input and displayed.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
        FILE *fp;                                        /* file pointer  */
        char str[80];

        if((fp=fopen("testfile", "w+"))==NULL)      /* open file for text read & write  */
        {
                printf("Cannot open file\n");
                exit(1);
        }

        do                                          /* get strings until CR and write to file  */
        {
                printf("Enter a string (CR to quit):\n");
                gets(str);
                strcat(str, "\n");
                fputs(str, fp);
        } while(*str != '\n');

        rewind(fp);                                      /* rewind  */
```

```
        while(! feof(fp))                                   /* read and display file  */
        {
                fgets(str, 79, fp);
                printf(str);
        }

        fclose(fp);                                         /* close file  */
}
```

Example 4.4.4:
Program on fread() and fwrite() (for Data Types Longer than Byte) which writes, then reads back, a double, an int and a long. Notice how sizeof () is used to determine the length of each data type. These functions are useful for reading and writing user-defined data types, especially structures.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{

        FILE *fp;
        double d=12.23;
        int i=101;
        long l=123023;
        if((fp=fopen("testfile", "wb+"))==NULL)         /* open for binary read & write */
        {
                printf("Cannot open file\n");
                exit(1);
        }

/* parameters are: *buffer, number of bytes, count of items, file pointer  */

        fwrite(&d, sizeof(double), 1, fp);
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&l, sizeof(long), 1, fp);
        rewind(fp);
        fread(&d, sizeof(double), 1, fp);
        fread(&i, sizeof(int), 1, fp);
        fread(&l, sizeof(long), 1, fp);
        printf("%2.3f   %d   %ld\n",d,i,l);
        fclose(fp);
}
```

Example 4.4.5:
Program on fprintf() and fscanf(). Reads a string and an integer from the keyboard, writes them to a file, then reads the file and displays the data. These functions are easy to write mixed data to files but are very slow compared with fread() and fwrite().

```
#include <stdio.h>
#include <stdlib.h>
#include <exec/io.h>
void main (void)
{

        FILE *fp;
```

```
        char s[ 80];
        int t;
        if ((fp=fopen("testfile", "w"))==NULL)              /* open for text write  */
        {
                        printf ("Cannot open file\n");
                        exit (1);
        }

        printf ("Enter a string and a number: ");
                        /* parameters are: FILE *fp, const char *control_string, ...  */
         fscanf (stdin, "%s%d", s, &t);
        fprintf (fp, "%s %d", s, t);
        fclose (fp);

        if ((fp=fopen("testfile", "r"))==NULL)              /* open for text read */
        {
                printf ("Cannot open file\n");
                 exit (1);
        }

        fscanf (fp,"%s%d",s,&t);
        fprintf (stdout, "%s %d\n", s, t);

        fclose (fp)
}
```