POINTERS, STRUCTURES AND INTRODUCTION TO DATA STRUCTURES

2.1 POINTERS

Pointer is a variable that holds the address of another variable. Pointers are used for the indirect manipulation of the variable. Every pointer has an associated data type. The pointer data type indicates the type of the variable whose address the pointer stores. This type is useful in interpreting the memory at a particular address. Pointers are declared using * operator.

General Form:

```
datatype *ptr;
```

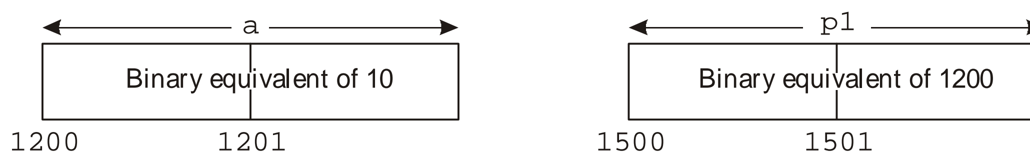
Here, *datatype* can by any basic/derived/user defined data type. And *ptr* is name of the pointer variable.

For example, consider the following declarations and statements.

```
int     a=10, *p1;
float   b=4.5, *p2;

p1= &a;
p2=&b;
```

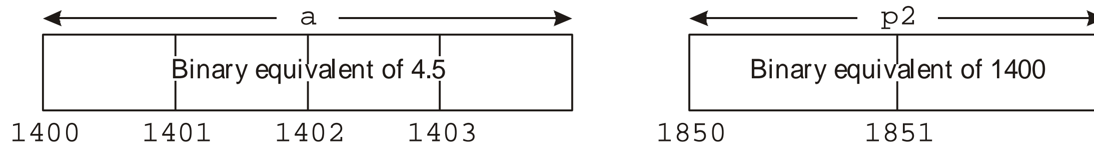
Here, p1 is a pointer variable which holds the address of an integer variable *a* and p2 is a pointer to hold the address of a floating point variable b.

The operator * is known as ***dereference operator*** and means ***value at the address***. The operator **&** is known as ***address of operator***. Now, the memory for *a* and *p1* may be allocated as follows:



The variable *a* occupies two bytes of memory as it is an integer. Suppose that the address of *a* is 1200 (the address is system-dependent and it may vary even for several runs of the same program on the same system). Now, as p1 is assigned the address of *a*, it will have the value as 1200. As p1 stores the address, and the address being an integer, p1 occupies two bytes of memory having its own address.

Similarly for the variables b and p2 memory allocation will be:



Here, the variable b occupies four bytes as it is a floating point variable. Assume that the address of b is 1400. Now, p2 is assigned the address of b. So, p2 will have the value 1400. Note that 1400 is an integer value. Hence, the space occupied by p2 is only two bytes.

The above fact indicates that a pointer variable always occupies only two bytes even if it is pointing to any of the basic data types like int, float, char or any of the derived data types like structures, unions etc.

As a pointer variable is also having an address, we can have a pointer to store the address of another pointer. This is called as a pointer to a pointer. For example, with the above given set of declarations, we can declare two more variables like:

```
int      **ptr1;
float    **ptr2;

ptr1 = &p1;
ptr2 = &p2;
```

Now, ptr1 is called as *a pointer to a pointer to an integer*. This will be having the address of p1 i.e. here, 1500. And ptr2 is called as *a pointer to a pointer to a float*. ptr2 will be having the address of p2 i.e. here, 1850. Proceeding in the same manner, we can have a pointer to a pointer to a pointer etc.

Consider a simple example to illustrate the usage of pointers.

```
#include<stdio.h>
void main()
{
        int i, x, *pi;
        float f, *pf;

        pi=&i;
        pf=&f;
        printf(”Enter i:”);
        scanf(”%d”, &i);
        x = *pi + 2;
        *pf=4.6;
        printf(“Address of i =%u”, pi);
        printf(“\nAddress of f =%u”, &f);
        printf(“\n Values: i= %d,  f= %f,  x=%d”, *pi, *pf, x);
}
```

The output would be -

Enter i:12

Address of i: 0x8fa4fff4 **(Note: address may be different for every execution)**

Address of f: 0x8fa4ffee

Values: i=12, f= 4.6 x= 14

Observed in the above example that, the address of any variable can be printed using either the pointer name or variable name with & symbol (that is, pi and &f both can be used to print the address). Similarly, to print the value of any variable, one can use either just name of the variable or pointer name preceded with symbol * (That is, f or *pf are same).

2.1.1 Initialization of Pointers

It is known from the discussion about variables in Chapter 1 that, when a variable is declared it will be containing some garbage value unless it is initialized. Similarly, an un-initialized pointer contains a garbage address. An un-initialized pointer containing garbage address is known as ***dangling pointer.*** Dangling pointer may contain any address which might have been already referenced by some other variable of any program or OS. An un-intentional manipulation of such pointer may cause the working of other programs or even OS. Thus, it is always advised to initialize any pointer at the time of its declaration as –

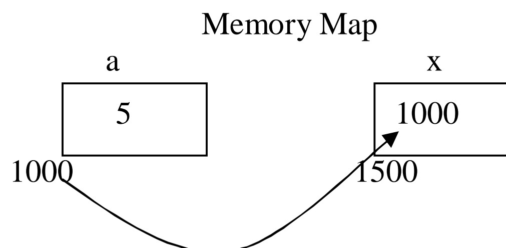
```
data_type *p = NULL;
```

Here, data_type is any basic/derived data type. NULL indicates that the pointer is *pointing no-where in the memory.*

2.1.2 Pointer as Function Arguments

Pointer can be passed as an argument to a function, and this methodology is known as *call-by-address/reference*. We will now see one simple example for passing a pointer to function.

```
#include<stdio.h>
void fun(int *x)
{
        *x=5;
}

void main()
{
        int a;
        fun(&a);        //address of a is being passed
        printf("a= %d", a);
}
```



In the above example, initially, the variable *a* contains garbage value. Then, the address of *a* is being passed to the function *fun()* which has the parameter as *int *x.* The internal meaning of this parameter passing is actually means,

```
int *x=&a;
```

Hence, a pointer *x* is created which stores the address of *a.* Now, inside *fun()*, the statement

```
*x = 5;
```

Indicates

value at the address(x) =5
or value at the address(1000) =5

Hence, the variable *a* gets the value 5.

2.1.3 Arrays and Pointers

It has been discussed in Unit 3 about accessing array elements using array subscripts. But, there is a speciality about arrays when it comes to pointers. Name of the array itself gives the address of the array and hence allowing the programmer to play with both array and pointer together. Since array is a collection of elements, to access all the elements through pointers, we need to traverse the array. Even for this, only starting address is sufficient. The starting address of an array is known as ***base address*** of the array. The base address of an array can be assigned to a pointer as –

```
        int a[10],*p;
        p=a;
or      p=&a[0];
```

Thus, either the name of array or the address of first element can be assigned to a pointer. To access the elements of array, we need to understand the arithmetic operations on pointers.

2.1.4 Pointer Arithmetic

Arithmetic operations are possible on pointers in the following situations:

1. A pointer can be incremented or decremented. That is, if p is a pointer, then the statements like

   ```
   p++;   or   p—;
   ```

   are valid. When incremented(decremented), the pointer will increment(decrement) by the quantity equal to the size of the data it holds. For example,

   ```
   int *p;
   float *ptr;
   p++; ptr--;
   ```

Now, p is incremented by 2 bytes and ptr is decremented by 4 bytes. the same rule is applied when a constant value is added to/subtracted from a pointer as –

```
int *p;
p=p+3;
```

now, p is incremented by 3 positions, that is, 6 bytes.

2. Addition or subtraction of pointers with integers is allowed. That is, if p and q are pointers of same data type, then the statements

   q = p–2; or q = p+5; etc.

   are valid.

3. If two pointers are of same data type, then their subtraction is possible. For example, if p, q and r are pointers of same data type, then the statements like

   r = p – q;

   are valid.

4. Multiplication of pointers with de-referenced values is allowed. For example,

   a = (*p1) * (*p2);

   is valid, where p1 and p2 are pointers and a is a variable of the same data type as of p1 and p2.

5. Pointers can be compared with another pointer using relational operators. That is, the expressions like:

   p >= p2, p1 != p2 etc.

   are valid.

6. Multiplication or division of a pointer with either another pointer or with an integer is not allowed. For example,

   p1/p2, p1 * p2, p1/5, p1*3 etc.

   are invalid.

7. Addition of two pointers is not allowed. That is,

   p1 + p2 can not be done.

If the pointer is addressed to an array element, then using pointer arithmetic like addition or subtraction, we can access the next/previous elements of the array. For example:

Illustration of pointer to array

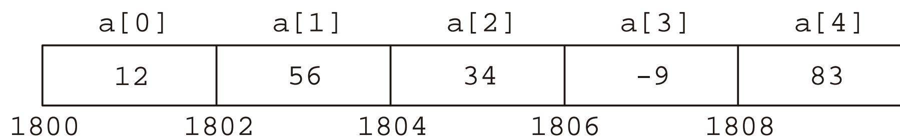
```
#include<stdio.h>

void main()
{
    int a[5]={12, 56, 34, -9, 83};
    int i=0, *pa;
```

```
        /* address of the first element of the array a, that is the address of a[0] is
        stored in pa. This is known as base address of the array a[].
        */

        pa=a;                   //or pa=&a[0];

        for(i=0;i<5;i++)
                printf("%d\n", *(pa+i));
}
```

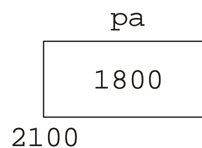
The output would be:

12 56 34 –9 83

In this program, when we declare an array a[5] and initialize it, the memory is allocated as:



And the memory allocation for the pointer pa will be:



Now, inside the *for* loop, when the statement

printf("%d\n", ***(pa+i)**);

gets executed for the first time, value of *i* is 0. Hence, the compiler de-references the address **(pa+i)** using the *value at the address(*)* operator. So, the value stored at the address 1800 i.e. 12 is printed.

In the next iteration of the loop, *i* will be 1. But, as per pointer arithmetic, **pa+1** is 1800+2bytes. Thus, the pointer points towards the next location with the address 1802, but not 1801 as normal variable. So, the array element at the address 1802 (i.e., 56) gets printed. The process continues till all the array elements get printed.

Note

1. It can be observed that all the following notations are same:

   a[i] , *(a+i), *(i+a), i[a]

   All these expressions means i^{th} element of an array a.

2. Consider the declarations:

   int a[3] = {12, 56, 78};

```
int *p ;
p = a;
```

Now, there is a difference between the statements:

```
printf("%d", *(p+1)); and
printf("%d", (*p+1));
```

The first expression will increment the pointer variable and so, forces the pointer to point towards next array element. Then the pointer is de-referenced and the value 56 gets printed. But the second expression will de-reference the pointer to get the value 12 and then 1 is added to it. So, the output will be 13.

So, the programmer should take care while incrementing the pointer variable and de-referencing it.

2.1.5 Functions Returning Pointers

A function can normally return any basic data type (like int, float etc) or user defined data type (like structure, union etc). Also, a function can return a derived data type like pointer. The pointer/address returned by the function can be a pointer to any data type. Consider an example:

```
#include<stdio.h>

int * fun()            //function return type is pointer to integer
{
        int a=5, *p;
        p=&a;

        return p;      //p is pointer to int,  same as return type of function
}

void main()
{
        int *x;                 //pointer to int type

        x= fun();               // x receives the address of a from fun()
        printf("%d", *x);       // value at address x is 5

}
```

The comments given in the above program clearly explains the program. In the same manner, one can return **float *, char *** etc. from the functions.

2.1.6 Pointer to Functions

When a function returns some value, it will be available within the name of the function. But, a function name refers to a memory location. So, we can have a pointer to a function also. That is, a function can be passed to another function as an argument.

The general form of declaring a function as a pointer is :

ret_type (*fn_name)(argument_list)

Here,

ret_type	is the return type of the function
fn_name	is valid name given to a pointer to the function
argument_list	is the list of parameters of the function fn_name, if any

To illustrate this concept, consider the program for the simulation of simple calculator.

Simulation of calculator using pointer to functions

```
#include<stdio.h>
#include<conio.h>
/* function that uses a pointer to another function as one parameter and two integer
parameters. This function is used to call different functions depending on the operator
   chosen by the user. */

float operate(float (*pf)(int, int), int a, int b)
{
        float val;

        val=(*pf)(a, b);        //call for the function using pointer to that function
        return val;
}
float add(int p, int q)         //function to add two integers
{
        return (p+q);
}

float sub(int p, int q)         //function for subtraction
{
        return (p-q);
}

float mul(int p, int q) //function for multiplication
{
        return (p*q);
}

float divide(int p, int q)      //function for division
{
        return (float)p/q;
}
```

```
void main()
{
        int x, y;
        char op;
        float result;

        printf("Enter two integers");
        scanf("%d%d", &x,&y);
        printf("\n Enter the operator");
        fflush(stdin);
        scanf("%c", &op);

        switch(op)
        {
      /* depending on the operator entered, the appropriate function is passed as a
      parameter to the operate() function */
                case '+': result = operate(add, x, y);
                          break;
                case '-': result = operate(sub, x, y);
                          break;
                case '*': result = operate(mul, x, y);
                          break;
                case '/': result = operate(divide, x, y);
                          break;
                default : printf("Invalid operator");
                          break;
        }
        printf("\n The result is: %f", result);
}
```

The output would be:

```
Enter two integers :
5
4
Enter the operators : *
The result is 20
```

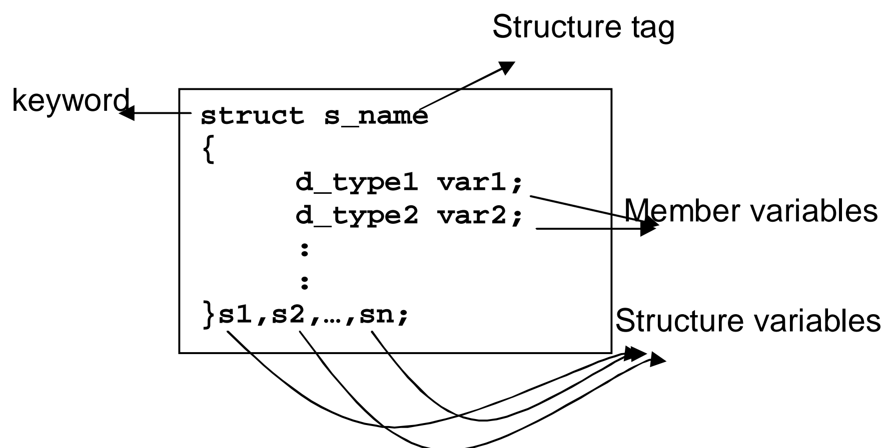
In this program, operate() is a function having one of the arguments as a pointer to some other function.

2.2 STRUCTURES

Along with built-in data types like int, float, char, double C/C++ provides the programmer to define his/her own data-type. These are known as *user-defined data types*. A programmer can derive a new data type based on existing data types. Structures, Unions and Enumerations are considered to be major user-defined data types.

We know that an array is a collection of related elements of same data type. But, if the related elements belong to different data types, then it is not possible to form an array. Consider a situation of storing the information about students in a class. The information may include name of the student, age of the student and marks of the student. It can be easily observed that name is a character array (or string), age might be an integer and marks obtained may be a floating point number. Though name, age and marks are of same student, we can not combine them into an array. To overcome this problem with arrays, C/C++ allows the programmer to combine elements of different data types into a single entity called as a *structure.*

Structure is a collection of inter-related elements of different data types. The syntax of structure is:



For example:

```
struct student
{
        int age;
        float marks;
}s1,s2;
```

Here, student is name of the structure or structure tag.
age, marks are member variables of a structure.
s1, s2 are variables of new data type *struct student*

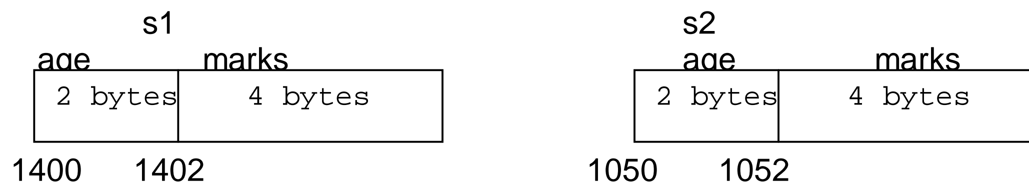
2.2.1 Declaration of Structure Variables

Structure declaration is considered to as defining a new data type. So, to make use of it, the programmer has to declare a variable of this new data type. There are two ways of declaring a structure variable. The programmer can declare variables of structure at the time of structure declaration itself as shown in the above example. Otherwise, one can declare them separately. For example –

```
struct student
{
        int age;
        float marks;
};

struct student s1,s2;
```

Here, struct student is name of new data type and hence using it, one can declare the variables s1 and s2.

Note that declaration of structure is just a prototype and it will not occupy any space in the memory. When a variable of structure gets declared, memory will be allocated for those variables. The memory space required for one structure variable will be the sum of the memories required for all member variables. In the above example, each of the structure variables s1 and s2 takes 6 bytes of memory (2 bytes for age and 4 bytes for marks). The memory map can be given as –



Thus, s1 and s2 are considered to be separate variables.

Note:
After declaring some of the structure variables along with the structure definition, it is possible to declare some more structure variables in a separate statement as shown below:

```
struct student
{
        int age;
        float marks;
}s1,s2;

struct student s3,s4;
```

2.2.2 Structure Initialization

Just like any variable or an array, a structure variable also can be initialized at the time of its declaration. Values for member variables are given in a comma-separated list as we do for arrays. Consider an example:

```
struct student
{
        int age;
        float marks;
}s1 = {21, 84.5};

struct student s2 = {20, 91.3};
```

NOTE:

1. Providing lesser number of values during initialization will make the rest of the variables to hold the value zero. That is,

```
struct test
{
        int a,b,c;
}t = {12, 45};
```

Now, the values of a, b and c will be 12, 45 and 0 respectively.

2. It is not possible to give initial value to a member variable within a structure definition. Because, structure definition is just a logical entity and member variables will not be having any physical location until the structure variables are declared. Thus, following attempt is ***erroneous.***

```
struct test
{
        int a=10                //error
        float b= 12.5;          //error
};
```

2.2.3 Accessing Member Variables

Member variables of a structure can not be accessed just by referring to their names. The reason is obvious. Refer to the memory map given in section 6.2.1**.** It can be observed that memory location for the member variable age of s1 is different from that of s2. Similarly for the member marks. Hence, it can be understood that each of the member variable is associated with a structure variable. Or in other words, every structure variable has its own copy of member variables. Thus, for accessing any member variable, its associated structure variable also should be specified. This is done with the help of ***dot (.) operator.***

The syntax is:

Structure_variable.member_variable

Note that a member variable of a structure can be used in arithmetic/logical/relational expressions. The following example illustrates the accessibility and usage of member variables.

Example for accessing member variable

```
#include<stdio.h>

struct student
{
        int age;
        float marks;
}s1, s2;

void main()
{
        s1.age=21;
        s2.age=s1.age;
        printf("Enter marks of students:");
        scanf("%f%f", &s1.marks, &s2.marks);

        printf("\nAge1= %d   Age2=%d", s1.age, s2.age);
        printf("\nMarks1= %f   Marks2=%f", s1.marks, s2.marks);

        if(s1.marks>s2.marks)
                printf("\n First Rank is Student1");
        else
                printf("\n First Rank is Student2");
}
```

The output would be:

```
Enter marks of students: 81.5      92.3
Age1=21                       Age2=21
Marks1=81.5         Marks2=92.3
First Rank is Student2
```

2.2.4 Structure Assignment

In some situations, the programmer may need two structure variables to have same values for all the member variables. Then, instead of assigning each member separately, it is possible to assign whole structure variable to another structure variable of same type. For example:

```
struct student
{
        int age;
        float marks;
}s1, s2={21,92.3};
```

Now, the statement

s1=s2;

will assign values of all members of s2 to the corresponding members of s1.

2.2.5 Arrays within Structure

A member variable of a structure can be an array of any data type. Consider the following example:

Example for array as a member of structure

```
#include<stdio.h>

struct student
{
	char name[20];			//array as a member
	int age;
	float marks;
}s1, s2={“Jaggu”,22,92.3};


void main()
{
	s1.age=21;
	printf(”Enter name of student:”);
	scanf(“%s”, s1.name);

	printf(”\nEnter marks of student:”);
	scanf(“%f”, &s1.marks);

	printf(”\n Student Information:\n”);
	printf (”\n Name \t\t Age \t\t Marks \n”);
	printf (“%s \t %d \t %f \n”, s1.name, s1.age, s1.marks);
	printf (“%s \t %d \t %f \n”, s2.name, s2.age, s2.marks);
}
```

The output would be:

```
Enter name of student: Ramu
Enter marks of students: 81.5
Student Information:
Name		Age		Marks
Ramu		21		81.5
Jaggu		22		92.3
```

2.2.6 Arrays of Structures

We know that array is a collection of elements of same data type and the structure is a collection of items of different data types. Some times, it may be necessary to have several structure variables. For example, if the information (like name, age and marks) about 60 students studying in 6th Semester is required, creating 60 structure variables is absurd. Instead, one can create an array of structure variables. Memory allocation for array of structures will be in contiguous locations. This concept is illustrated in the following example:

Example for array of structures

```
#include<stdio.h>

struct student
{
	char name[20];
	int age;
	float marks;
}s[2];		//array of structures

void main()
{
	int i;

	for(i=0;i<2;i++)
	{
		printf(”Enter name of student”);
		scanf(“%s”, s[i].name);

		printf(”\nEnter age of student”);
		scanf(“%d”,&s[i].age);

		printf(”\nEnter marks of student”);
		scanf(“%f”, &s[i].marks);
	}

	printf(”\n Student Information:\n”);
	printf (”\n Name \t Age \t Marks \n”);

	for(i=0;i<2;i++)
		printf(“\n%s \t %d \t %f”, s[i].name, s[i].age, s[i].marks);
}
```

The output would be:

```
	Enter name of student: Vidya
	Enter age of student: 21
	Enter marks of student: 81.5
```

```
Enter name of student: Jaggu
Enter age of student: 22
Enter marks of student: 92.3

Student Information:
Name        Age         Marks
Vidya       21          81.5
Jaggu       22          92.3
```

In the above example, memory allocation for the array s[2] might be as follows:

s[0].name	s[0].age	s[0].marks	s[1].name	s[1].age	s[1].marks
20 bytes	2 bytes	4 bytes	20 bytes	2 bytes	4 bytes
1000	1020	1022	1026	1046	1048

Note that array of structure variables can also be initialized just like a normal array. For example:

```
struct student
{
        int age;
        float marks;
}s[3] = {
                {21, 81.5},
                {22, 92.3},
                {25, 88.4}
           };
```

Then, the individual elements will be assigned the values as –

```
s[0].age=21             s[1].age=22             s[2].age=25
s[0].marks=81.5         s[1].marks=92.3         s[2].marks=88.4
```

Note also that, un-initialized members will take the value as zero.

2.2.7 Nested Structures

A member variable of a structure may be a variable of type another structure. This is known as nesting of structures. For example:

```
struct address
{
        int d_no;               ------------  2 bytes  ┐
        char street[20];        ------------ 20 bytes  ├ 42 bytes
        char city[20];          ------------ 20 bytes  ┘
};
```

```
struct employee
{
        char name[20];          ------------   20 bytes  ⎫
        int d_no;               ------------    2 bytes  ⎬  68
        float sal;              ------------    4 bytes  ⎪  bytes
        struct address add;     ------------   42 bytes  ⎭
}emp;
```

The total memory allocated for the variable emp would be 68 bytes. The memory map can be:

emp.name	emp.num	emp.sal	emp.add.d_no	emp.add.street	emp.add.city
20 bytes	2 bytes	4 bytes	2 bytes	20 bytes	20 bytes
1000	1020	1022	1026	1028	1048

Here, the structure address contains the members d_no, street and city. Then the structure employee contains the members name, d_no, sal and add. Here, add is a variable of structure address. To access the members of the inner structure one has to use the following type of statements:

```
emp.add.d_no
emp.add.street
emp.add.city   etc.
```

2.2.8 Passing Structure to Functions

Just like a normal variable, it is possible to pass a structure variable as a function parameter either using call-by-value method or call-by-address method. In this section, only a call-by-value method is discussed and the latter is discussed in **Chapter 6.** Passing of structure to a function can be done in two different ways viz.

- **Member variables as arguments**: Individual members of a structure can be passed to function as arguments. In this situation, members of structure are treated like any normal variable. Consider an example –

  Structure member as a function argument

```
#include<stdio.h>

struct add
{
    int a, b;
};
```

```
int sum(int x, int y)
{
    return (x+y) ;
}

void main()
{
    struct add var ;
    int s ;

    printf("Enter two numbers :" );
    scanf("%d%d", &var.a, &var.b);
    s=sum(var.a, var.b);

    printf("Sum is: %d ",s);
}
```

The output would be –

```
    Enter two numbers : 5          8
    Sum is: 13
```

- **Whole structure variable as an argument:** Instead of passing each member of a structure to a function, entire structure variable can be passed. For example:

Structure variable as a function argument

```
#include<stdio.h>

struct add
{
    int a, b;
};

int sum(struct add A)    //parameter is a structure variable
{
    return (A.x + A.y) ;
}

void main()
{
    struct add var ;
    int s ;

    printf("Enter two numbers :" );
    scanf("%d%d", &var.a, &var.b);

    s=sum(var);
```

```
        printf(”Sum is: %d “,s);
    }
```

The output would be –

```
        Enter two numbers : 5         8
        Sum is: 13
```

2.2.9 Pointers and Structures

We can create a pointer to store the address of a structure variable. To access the member variables of a structure using a pointer, we need to use either an indirectional operator -> or the combination of * and dot(.). Let us consider an example:

```
#include<stdio.h>

struct student
{
        char name[20];
        int age;
};

void main()
{
        struct student s={“Ramu”, 22};
        struct student *p;

        p= &s;
        printf(“Student name =%s”, p ->name);
        printf(“\nStudent age =%d”, (*p).age);
}
```

In the above example, note the usage:

```
        (*p).age
```

Here, the pointer *p* is dereferenced first, and then dot operator is used to get the member variable *age.* Instead of two operators * and dot, we can use single indirectional operator (arrow mark) to achieve the same. That is, we can use

```
        p->age,  p->name etc.
```

We can even create a pointer to array of structures also. For example (with respect to structure declared in the above example):

```
                struct student s[10], *p;
                p = s;      // base address of the array s is assigned to p
```

Then, for accessing array members, we can use:

```
        for(i=0; i<n;i++)
```

```
printf("%s %d", p[i]->name, p[i].age);
```

The similar approach can be used while passing structures to functions via call by address method.

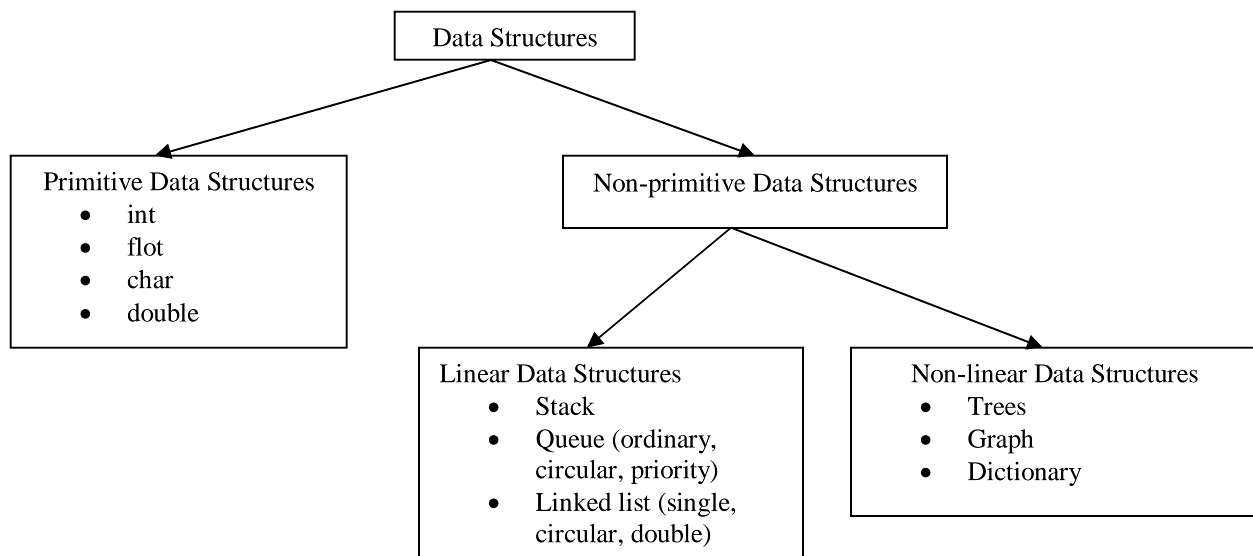
2.3 CLASSIFICATION OF DATA STRUCTURES

The study of data structures in C deals with the study of ***how the data is organized in the memory, how efficiently the data can be retrieved from the memory, how the data is manipulated in the memory and the possible ways in which different data items are logically related.*** Thus, we can understand that the study of data structure involves the study of memory as well. Irrespective of the programming language, the structure of the data can be studied. C is one programming language which throws light into in-depth of this concept, as C facilitates hardware interaction and memory management to the programmers.

The study of data structures also involves the study of how to implement the developed data structures using the available data structures in C. Since the problems that arise which implementing high-level data structures are quite complex, the study will allow to investigate the C language more thoroughly and to gain valuable experience in the use of this language. While implementing data structure, one should take care of efficiency, which involves two facts viz. time and space. That is, a careful evaluation of time complexity and space complexity should be made before data structure implementation.

Irrespective of the programming language, the structure of the data can be studied. C is one programming language which throws light into in-depth of this concept, as C facilitates hardware interaction and memory management to the programmers.

Data structures can be categorized as below –



Basic data types like int, float etc. are called as ***primitive*** data structures, because values in these type of variables are directly accessible. The data types (or data structures) in which values are not directly accessible, instead, they are pointers (or references) are called as ***non-primitive*** data structures. In C, non-primitive data structures can be further divided into – ***linear*** and ***non-linear*** data structures. When the relationship between data-elements is linear, it is called as linear data structure. For example, stack, queue, linked list etc. If the relationship between data-elements is hierarchical, then it is called as non-linear data structure. For example, trees, graphs, sets, dictionaries etc.

Most of the data structures involves following operations –
- o Creation of data structure
- o Insertion of elements (at front, back or any position)
- o Deletion (from front, back or any position)
- o Display the contents
- o Searching for particular element
- o Sorting the elements

2.4 ABSTRACT DATA TYPES (ADT)

The term abstract data type refers to the basic mathematical concept that defines the data type. ADT will specify the logical properties of a data type. It is a useful tool for implementers and programmers who wish to use the data type correctly. Whenever any new data type (user-defined data type) is to be created, a prototype of its nature, the possible operations on it etc. have to be thought of. In such a situation, ADT helps in forming a prototype. Note that, the representation of ADT do not follow the syntax of any programming language.

Even though there are several methods to specify ADT, we use the semiformal method, which will adopt C notations.

For illustration, consider rational number in the form –

$$\left\{ \frac{p}{q} / \ p, q \in Z \text{ and } q \neq 0 \right\}$$

The above equation indicates that any rational number will be in the form of p divided by q, where p and q are integers (the set Z) and the denominator q is not equal to zero.

In this case, the sum of two rational numbers (a1/a2) and (b1/b2) would be –

$$\frac{a_1}{a_2} + \frac{b_1}{b_2} = \frac{a_1 * b_2 + b_1 * a_2}{a_2 * b_2}$$

The specification of any ADT consists of two parts –

- **value definition**
  - here, we specify the set of possible values taken by the ADT along with some conditions or constraints with which the ADT bounds.
- **operator definition**
  - here, various operations which are imposed on ADT are defined. This part contains 3 sections viz. a header, the preconditions (which is optional) and the postconditions. The term 'abstract' in the header indicates that this is not a C function; rather it is an ADT operator definition. This term also indicates that the role of an ADT is a purely logical definition of a new data type.

The following listing gives the ADT for rational numbers. The value definition here indicates the constraints on rational number. The operator definition parts contains the definition for various operations like creation of rational number, addtion and multiplication of rational numbers and for checking the equality of two rational numbers.

```
// value definition
abstract typedef<integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

//Operator definition
abstract RATIONAL createrational (a, b)
int a,b;
precondition b!=0;
postcondition   createrational [0] = = a;
                        createrational [1] = = b;

abstract RATIONAL add(a,b)
RATIONAL a, b;
postcondition add[0] = =  a[0]*b[1] + b[0]*a[1];
                     add[1] = = a[1]*b[1];
abstract RATIONAL mul(a,b)
RATIONAL a, b;
postcondition mul[0] = = a[0]*b[0];
                     mul[1] = = a[1]*b[1];

abstract equal(a, b)
RATIONAL a,b;
postcondition
            equal = = (a[0]*b[1] = = b[0]*a[1]);
```

2.4.1 Array as an ADT

An array is a derived data type. Hence, it can be represented as an ADT. The following listing gives ADT for an array. It contains the value definition indicating how an array should look like. The operator definition part indicates two major operations of the array viz. extracting an element from the array and storing an element into the array.

```
//value definition part
Abstract typedef <<eltype, ub>> ARRTYPE (ub, eltype);
Condition type(ub)==int;

//operator definition part
abstract eltype extract(a,i)
ARRTYPE(ub, eltype) a;
int i;
precondition 0<= i <ub;
postcondition extract==a_i


abstract store(a,i, elt)
ARRTYPE(ub, eltype) a;
int i;
eltype elt;
precondition 0<= i <ub;
Postcondition a[i]==elt;
```

2.4.2 Sequences as Value Definitions

While developing the specification for various data types, to specify the value of an ADT, we use set notation or the notation of sequences. Basically, a sequence is an ordered set of elements denoted by,

$$S = <S_0, S_1, \ldots S_{n-1}>$$

If S contains n elements then S is said to be of length n. We assume the functions len(S), first(S), last (S) etc to denote length, first element and last element respectively. A sequence with zero length is called *nilseq.*

Various syntaxes for ADT specifications are as below –

- To define an ADT viz. adt1 whose values are the elements of a sequence, we write –

  abstract typedef<<type1>> adt1;

  Here, *type1* indicates the data type of elements in the sequence.

- To denote an ADT taking the values of different data types type1, type2, etc. we write –

  abstract typedef<type1,type2,….,type n> adt2;

- To denote an ADT having a sequence of length n, where all elements are of same data type, we write –

  abstract typedef<<type, n>> adt3;

Two sequences are said to be equal if their corresponding elements are equal. If S is a sequence, the function sub(S,i,j) refers to the subsequence of S starting at the position i

in S and consisting of j consecutive elements. The concatenation of two sequences S1 and S2 is the sequence consisting of all elements of S1 followed by the elements of S2.

2.4.3 String as an ADT

Using the sequence notation, the specification of ADT for the varying-length character string can be illustrated. Normally, these are four basic operations for strings viz.

- length is a function that returns length of string
- concat is returns concatenation of two strings
- pos is returns the first position of one string in the other
- substr is returns a substring of given string.

```
abstract typdef <<char>> STRING;

abstract length(s)
STRING s;
postcondition length = len(s);

abstract STRING concat(s1,s2)
STRING s1, s2;
postcondition concat == s1 + s2;

abstract STRING substr(s1, i, j)
STRING s1;
int i, j;
postcondition substr == sub(s1, i, j)
```