MODULE 1

Introduction to Data Structures

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

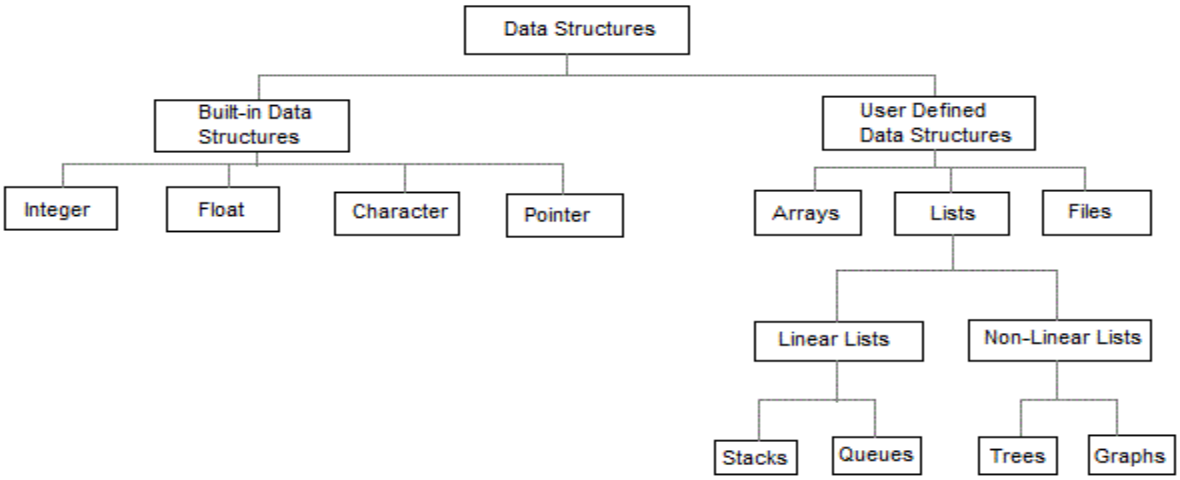
Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

Basic Concepts - Operations that can be performed on data structures

Following operations can be performed on the data structures:

1. Traversing

2. Searching

3. Inserting

4. Deleting

5. Sorting

6. Merging

1. Traversing- It is used to access each data item exactly once so that it can be processed.

2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

3. Inserting- It is used to add a new data item in the given collection of data items.

4. Deleting- It is used to delete an existing data item from the given collection of data items.

5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

STRUCTURES:

Arrays allow defining type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:

```
struct  < Tag >
 {
 Data type  Variable1;
 Data type   Variable2;
    …

Data type variableN;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
 struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program:

```
#include <stdio.h>
#include <string.h>

struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main( ) {

  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "guru");
  strcpy( Book1.subject, "C Programming ");
  Book1.book_id = 1000;

  /* book 2 specification */
  strcpy( Book2.title, "Telephone Billing");
  strcpy( Book2.author, "prasad");
  strcpy( Book2.subject, "Telephone Billing ");
  Book2.book_id = 2000;

  /* print Book1 info */
  printf( "Book 1 title : %s\n", Book1.title);
  printf( "Book 1 author : %s\n", Book1.author);
  printf( "Book 1 subject : %s\n", Book1.subject);
  printf( "Book 1 book_id : %d\n", Book1.book_id);

  /* print Book2 info */
  printf( "Book 2 title : %s\n", Book2.title);
  printf( "Book 2 author : %s\n", Book2.author);
  printf( "Book 2 subject : %s\n", Book2.subject);
  printf( "Book 2 book_id : %d\n", Book2.book_id);

  return 0;
```

```
}
```

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

```
Book 1 title : C Programming
Book 1 author : guru
Book 1 subject : C Programming
Book 1 book_id : 1000
Book 2 title : Telephone Billing
Book 2 author : prasad
Book 2 subject : Telephone Billing
Book 2 book_id : 2000
```

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```
#include <stdio.h>
#include <string.h>

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );


int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "guru");
   strcpy( Book1.subject, "C Programming ");
   Book1.book_id = 1000;

   /* book 2 specification */
   strcpy( Book2.title, "Telephone Billing");
   strcpy( Book2.author, "prasad");
```

```
         strcpy( Book2.subject, "Telephone Billing ");
         Book2.book_id = 2000;

  /* print Book1 info */
       printBook( Book1 );

      /* Print Book2 info */
        printBook( Book2 );

    return 0;
}

void printBook( struct Books book ) {

  printf( "Book title : %s\n", book.title);
  printf( "Book author : %s\n", book.author);
  printf( "Book subject : %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);
}
```

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

```
Book title : C Programming
Book author : guru
Book subject : C Programming
Book book_id : 1000
Book title : Telephone Billing
Book author : prasad
Book subject : Telephone Billing
Book book_id : 2000
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable:

```
struct Books  * pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

```
pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows: pointer->title;

UNION:

A union is like a structure in which all members are stored at the same address. Members of a union can only be accessed one at a time. The union data type was invented to prevent memory fragmentation. The union data type prevents fragmentation by creating a standard size for certain data. Just like with structures, the members of unions can be accessed with the dot (.) and arrow (->) operators. Take a look at this example:

```
typedef  union
{
double PI;
int  b;
}MYUNION;

        int main()
            {
             MYUNION numbers;
              numbers.PI = 3.14;
             numbers.b = 50;
              return 0;
               }
```

If you would want to print the values of PI and b, only b value will be printed properly since single memory is reserved to hold the data members' value of union. Hence, 50 will be displayed correctly.

Differences between structures and unions

Structure	Union
Structure declaration starts with keyword struct	Union declaration starts with keyword union
Structure reserves memory for each data member separately	Union reserves memory i.e., equal to maximum data member size amongst all.
Any data member value can be accessed at any time	Only one data member can be accessed at a time
Ex: struct Book{ int isbn; float price; char title[20]; }book; Total memory reserved will be sizeof(int)+sizeof(flaot)+(20*sizeof(char)	Ex: union Book{ int isbn; float price; char title[20]; }book; Total memory reserved will be Max(sizeof(int)+sizeof(flaot)+(20*sizeof(char))

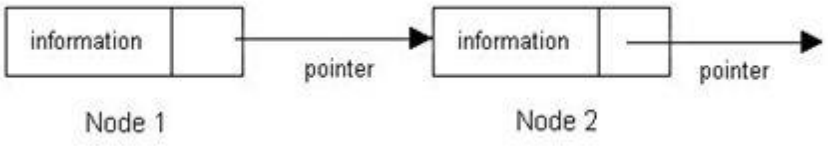
SELF REFRENTIAL STRUCTURES

A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind. A chain of such structures can thus be expressed as follows.

```
struct  <Tag>
 {
          member 1;
          member 2;
          . . .
          struct  <Tag>  *pointer;
};
```

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. .Several such inter-connected nodes create a chain of structures.

The following figure depicts the composition of such a node. The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.



Such self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. Unlike a *static data structure* such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted. Operations like insertion or deletion of nodes in a self-referential structure involve simple and straight forward alteration of pointers.

Example for self referential structure.

```
struct  Node
        {
       int data;                  /* information member*/
       struct  Node  *next;   /* link field to next information: refer above figure */
        };
    struct Node *start = NULL;
```

Note: The second member points to a node of same type.

Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ...,

numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

Type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double bal [3]={1.0,2.0,3.0}

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double bal [3]={1.0,2.0,3.0}

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

bal[2]=4.0

The above statement assigns the 3rd element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10^{th} element from the array and assign the value to salary variable.

Operations on arrays

```
void create(int a[ ] )
{/* n is global */
printf("\n Enter the elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
}
```

Function to display elements of the array

```
void display(int a[ ])
{
int i;
printf("\n elements are");
for(i=0;i<n;i++)
printf("\n %d",a[i]);
}
```

Function to Insert a element on to a array

```
void insert(int a[ ], int pos,int ele)
{
for(i=n-1; i>=pos;i--)
      a[i+1]=a[i]; //move the elements down
 a[pos]=ele;
n=n+1;
}
/*Function to delete a element from a array*/

int  delete(int a[ ], int pos)
{ /* assuming valid position */
int ele = a[pos];
  for(i=pos;i<n-1;i++)
      a[i]=a[i+1];    //move the elements up
  n= n-1;
return ele;
}
```

Binary search

```
#define COMPARE(x,y)(((x)==(y)?0:((x)>(y))?1:-1)

int bin_search(int key,int a[],int n)
{
int low,high,mid;
low=0;
high=n-1;
while (low<=high)
{
mid=(low+high)/2;
switch (COMPARE (key,a[mid]) ))
  {
case 0 :return mid;//key is in middle
case -1: high=mid-1;//key is on the left side of table
         break;
case 1: low=mid+1;//key is in the right side of the table;
   }
 }
         return -1;  //unsucessfull search
}
```

Memory allocation functions

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under `"stdlib.h"` for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type `void` which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size)

Here, *ptr* is pointer of cast-type. The `malloc()` function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of `int` 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```
This statement will allocate contiguous space in memory for an array of *n* elements. For example:
```
ptr=(float*)calloc(25,sizeof(float));
```
This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement causes the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

ptr=realloc(ptr,newsize);
Here, *ptr* is reallocated with size of newsize.

```
void  main()
{
   int *ptr,i,n1,n2;
   printf("Enter size of array: ");
   scanf("%d",&n1);
   ptr=(int*)malloc(n1*sizeof(int));

   printf("\nEnter new size of array: ");
   scanf("%d",&n2);
   ptr=realloc(ptr,n2);
   for(i=0;i<n2;++i)
       printf("%u\t",ptr+i);  /* display address of new block*/
         return 0;
 }
```

1D-Dynamically Allocated arrays

Usually array at the time of its declaration gets it predefined size. Bounding to this size, programmers will have to find out solution for several problems. For example, if SIZE value is 100, the program can be used to sort a collection up to 100 numbers. If the user wishes to sort more than 100 numbers, we have to change the definition of SIZE using some larger value and recompile the program. It becomes little tedious for someone to rethink on this number. Hence the concept, Dynamic Arrays.
Let us follow the below program to understand the concept of dynamically allocated arrays with the help of Dynamic Memory Allocation functions.

```
void main ( )
{
int i, *list, n=10, p;
list = (int *) malloc(n* sizeof(int));  /* dynamic allocation*/

printf("Enter array elements\n");
for(i=0;i<n;i++)
scanf("%d", (list+i) );

printf("Entered array elements\n");
for(i=0;i<n;i++)
printf("%d\n",*(list+i));
free(list);  /* deallocation*/
}
```

2D – Dynamically Allocated Arrays: Example

```
#define SIZE 10
int main()
 {
int *p[SIZE];  /* array of pointers one for each row of 2D*/
int row=4, col=3, i, j;
                                /* allocate memory */
p = (int *) malloc(row*sizeof(int )); /* row pointers */

for(i = 0; i < row; i++)
  p[i] = (int *) malloc (col* (sizeof(int));   /* Row data */

/* Assign values to array elements */
for(i = 0; i < row; i++)
   for(j = 0; j < col; j++)
     scanf("%d", (*p+i)+j) ;   /* read into dynamically created matrix

for(i = 0; i < row; i++)         /* To print matrix*/
for(j = 0; j < col; j++)
    printf("%d",*(*p+i)+j);
                                    /* deallocation */
for(i = 0; i < row; i++)
free(p[i]);                        /* free all rows */

free (p);                         /* free Row pointers */
}
```