We have learned that an array is a homogeneous collection. But every program does not need homogeneous collection, some program needs heterogeneous collection. In C++ a structure is used to represent heterogeneous collection. First we declare (create/define) a structure and then a variable of that structure type is created. **Structure** is a collection of **heterogeneous** data type stored under one name and memory is allocated contiguously. A structure is declared by using the keyword **`struct`** followed by a programmer-specified name followed by the structure declaration within pair of curly braces. Closing curly brace is terminated by semi-colon (;). A structure declaration contains the structure members – its **data members**. Data members of a structure are also known as **field**. An example is given below:

```
struct StructName
{
   DataType1 Member1;
   DataType2 Member2;
   :
};
void main()
{
   StructName StructVar;
   //More C++ statements
}
```

`StructName` is the name of the structure. `DataMember1`, `DataMember2`, ... are the field of the structure `StructName`. Data type of data members could fundamental data type or derived data type. In the `main()` function `StructVar` is created of the type `StructName`. A programming example is given below:

```
struct item
{
   int code;
   char name[20];
   double price;
};
void main()
{
   item prod;
   //More C++ code
}
```

Structure name is **`item`** and it has three data members (three fields) – **`int code`**, **`char name[20]`** and **`double price`**. Three data members are of three different data types. Data members **`code`** and **`price`** are of fundamental data type while data member **`name`** is string type – derived type. In the **`main`**() function a structure variable **`prod`** is created of the structure type **`item`**.

Variable `prod` created in the main() function is allocated memory. The size of a structure variable (memory allocated to a structure variable) is calculated in term of bytes – is the sum total of bytes allocated to the data members of the structure variable. Variable `prod` has three data members – `code`, `name[20]` and `price`. Data type **`int`** is allocated 4 bytes, an array of 20 **char** (string) is allocated 20 bytes and data type **`double`** is allocated 8 bytes. They all add up to 32 bytes. Variable `prod` will be allocated 32 bytes of memory. Once variable `prod` is created, we need to access the data members of variable `prod` – so that value can be **stored** in the data members or value stored in the data members can be **read** for display (or for **process**). General rule to access a data member of a structure variable is

```
StructVarName.DataMemberName
```

Note the use of dot operator (.) also called **membership** operator of a structure variable. Dot operator (.) is binary operator. On the left hand side of the dot operator (.) is the structure variable name and on the right hand side of dot (.) operator is the data member name of the structure variable. An example is given below:

```
prod.code
prod.name
prod.price
```

prod.code accesses data member **code**, **prod.name** accesses data member **name** and **prod.price** accesses data member **price**.

In a C++ program `StructVarName.DataMemberName` will be treated as a variable name. That is prod.code will be treated like an integer variable, prod.name will be treated like a string and prod.price will be treated like a floating point variable.

Storing values in a structure variable

Values can be stored in structure variable by inputting value from a keyboard. An example is given below:

```
item p;
cout<<"Code ? "; cin>>p.code;
cout<<"Code ? "; cin>>p.name;
Or,
cout<<"Code ? "; gets(p.name);
cout<<"Price? "; cin>>p.price;
cin>>p;
```

Values are stored in the data members by taking input from a keyboard. While inputting a string we may use **cin** or **gets**(). **cin>>StructVarName;** is not allowed, that is, **cin>>p;** will flag syntax error.

Values can be stored in structure variable by using assignment operator (=) for the fundamental data type and using `strcpy`() for the string type. An example is given below:

```
item p;
p.code=10;
strcpy(p.name,"DOVE SOAP");
p.price=42.75;
```

Values can be stored in structure variable by using an initializer. Initializer can only be used at the point of creation of a structure variable.

- `item p1 = {1001, "BASMATI RICE", 75.5};`
  `p1.code` will contain value `1001`, `p1.name` will contain value `"BASMATI RICE"` and `p1.price` will contain value `75.5`. This is an ideal case, where data members are fully initialized, that is, values inside the initializer are exactly equal to number of data members and the order of the values inside the initializer matches with order of declaration of the data members.

- `item p2 = {1002, "LIPTON TEA"};`
  `p2.code` will contain value `1002`, `p2.name` will contain value `"LIPTON TEA"` and `p2.price` will contain value `0.0`. In this example data members are partially initialized. Since last data member is not initialized, by default it will have value `0`.

- `item p3 = {};`
  `p3.code` will contain value `0`, `p2.name` will contain `""` and `p2.price` will contain value `0.0`.

- `item p4 = {1003, "GLUCOSE-D", 80.0, 25};`
  Syntax error, variable `p4` will not be created since number of values inside the initializer exceeds number of data members.

- `item p5 = {"GLUCOSE-D", 1004, 80.0};`
  Syntax error, since values inside the initializer appear in the wrong order.

Examples of incorrect structure declarations are given below:

1.
```
struct item
{
   int code=1007;
   char name[20]="HP Laptop";
   double price=40000;
};
```

Syntax error because values cannot be assigned to data members of **structure type**. Values only can be assigned to data members of a structure variable.

2.
```
struct item
{
   int code;
   char name[20];
   double price;
}
```

Syntax error because semi-colon is missing after closing curly bracket.

3.
```
item
{
   int code;
   char name[20];
   double price;
};
```

Syntax error, keyword **`struct`** is missing before the structure name.

Displaying a structure variable

Values stored in a structure variable can be displayed using `cout` or `printf`(). A string type data member can be displayed with the `puts`(). An example is given below:

```
cout<<"Code ="<<p.code<<endl;
cout<<"Name ="<<p.name<<endl;
```
Or,
```
cout<<"Name ="; puts(p.name);
cout<<"Price="<<p.price<<endl;
printf("Code =%i\n", p.code);
printf("Name =%s\n", p.name);
printf("Price=%lf\n", p.price);
cout<<p<<endl;
```

Each data member of a structure variable has to be displayed separately. **`cout<<StructVarName;`** is not allowed, that is, **`cout<<p;`** will flag syntax error.

Complete programs are given below showing the use of structure in a C++ program:

1.
```
#include<iostream.h>
#include<stdio.h>
struct item
{
   int code;
   char name[20];
   double price;
};
```

```
void main()
{
   item prod;
   cout<<"Product Code ? "; cin>>prod.code;
   cout<<"Product Name ? "; gets(prod.name);
   cout<<"Product Price? "; cin>>prod.price;
   cout<<"Product Code ="<<prod.code<<endl;
   cout<<"Product Name ="<<prod.name<<endl;
   cout<<"Product Price="<<prod.price<<endl;
}
```

Running of the program produces following output

```
Product Code ? 1023
Product Name ? Lipton Ice Tea Lemon
Product Price? 150
Product Code =1023
Product Name =Lipton Ice Tea Lemon
Product Price=150
```

2.

```
#include<iostream.h>
#include<stdio.h>
struct employee
{
   int eno;
   char name[20];
   double basic;     //Basic Salary
   double da;        //Dearness Allowance=50% of Basic
   double hra;       //House Rent=30% of Basic
   double ca;        //City Allowance=20% of Basic
   double gross;     //Gross Salary=basic+da+hra+gross
};
void main()
{
   employee a;
   cout<<"Employee Number? "; cin>>a.eno;
   cout<<"Employee Name  ? "; gets(a.name);
   cout<<"Basic Salary   ? "; cin>>a.basic;

   a.da=0.5*a.basic;
   a.hra=0.3*a.basic;
   a.ca=0.2*a.basic;
   a.gross=a.basic+a.da+a.hra+a.ca;

   cout<<"Employee Number   ="<<a.eno<<endl;
   cout<<"Employee Name     ="<<a.name<<endl;
   cout<<"Basic Salary      ="<<a.basic<<endl;
   cout<<"Dearness Allowance="<<a.da<<endl;
   cout<<"House Rent        ="<<a.hra<<endl;
   cout<<"City Allowance    ="<<a.ca<<endl;
   cout<<"Gross Salary      ="<<a.gross<<endl;
}
```

Input of data into structure variable

Process of inputted data

Display of inputted data and processed data on the screen

Running of the program produces following output

```
Employee Number? 1246
Employee Name  ? Nimesh Kumar Sampat
Basic Salary   ? 45000
Employee Number   =1246
Employee Name     = Nimesh Kumar Sampat
Basic Salary      =45000
Dearness Allowance=22500
House Rent        =13500
City Allowance    =9000
Gross Salary      =90000
```

Nested Structure

When a structure contains another structure as its member, it is called **nested** structure. We need at least two structures, outer structure and inner structure. Outer structure will contain inner structure as its member. When declaring the two structures, inner structure has to be declared first. An inner structure can be created inside outer structure. An example is given below:

```
struct date
{
   int dd, mm, yy;
};
struct student
{
   int roll;
   char name[20];
   date doj;
};
```

Inner structure name is `date`. Outer structure name is `student`. Outer structure `student` contains `doj` of the type `date` as its data member.

Or,

```
struct student
{
   int roll;
   char name[20];
   struct date
   {
      int dd, mm, yy;
   } doj;
};
```

Outer structure name is `student`. Outer structure `student` contains structure declaration of `doj`, which is of the structure type `date`.

Accessing the data members of a nested structure variable is very similar to accessing data member of a structure variable. General rule to access member of nested structure and complete program showing its use is given below:

```
OuterStuctVarName.OuterStructDataMember.InnerStructDataMember
```

```
#include<iostream.h>
#include<stdio.h>
struct date
{
   int dd, mm, yy;
};
```

```
struct student
{
   int roll;
   char name[20];
   date dob;
};
void main()
{
   student s;;
   cout<<"Roll ? "; cin>>s.roll;
   cout<<"Name ? "; gets(s.name);
   cout<<"Day  ? "; cin>>s.dob.dd;
   cout<<"Month? "; cin>>s.dob.mm;
   cout<<"Year ? "; cin>>s.dob.yy;
   cout<<"Roll     ="<<s.roll<<endl;
   cout<<"Name     ="<<s.name<<endl;
   cout<<"Birth Day="<<s.dob.dd<<'-'<<s.dob.mm<<'-'<<s.dob.yy;
}
```

Running of the program produces following output

```
Roll ? 12
Name ? Allen John Mathew
Birth Day
Day  ? 3
Month? 7
Year ? 1990
Roll     =12
Name     =Allen John Mathew
Birth Day=3-7-1990
```

```
#include<iostream.h>
#include<stdio.h>
struct date
{
   int dd, mm, yy;
};
struct employee
{
   int code;
   char name[25];
   double salary;
   date doj;
};
void main()
{
   employee t;
   cout<<"Code        ? "; cin>>t.code;
   cout<<"Name        ? "; gets(t.name);
   cout<<"Salary      ? "; cin>>t.salary;
   cout<<"Day    [1-31]? "; cin>>t.doj.dd;
   cout<<"Month  [1-12]? "; cin>>t.doj.mm;
   cout<<"Year  [YYYY]? "; cin>>t.doj.yy;
   cout<<"Code       ="<<t.code<<endl;
   cout<<"Name       ="<<t.name<<endl;
   cout<<"Salary     ="<<t.salary<<endl;
   cout<<"Join Date  ="<<t.doj.dd<<'-'<<t.doj.mm<<'-'
                       <<t.doj.yy<<endl;
}
```

Running of the program displays following output:

```
Inputting Employee's Details
Code       ? 45
Name       ? Pradeep Kumar Sharma
Designation? Manager IT
Salary     ? 74000
Teacher Join Date
Day   [1-31]? 4
Month [1-12]? 9
Year  [YYYY]? 2002
Displaying Employee's Details
Code       =45
Name       =Pradeep Kumar Sharma
Designation=Manager IT
Salary     =74000
Join Date  =4-9-2002
```

A structure is declared before the `main()` function as a global identifier so that the same structure declaration can be used to create structure variable in the `main()` function and in any other functions in the program. An example is given below:

```
struct teacher
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
};
void funct()
{
   teacher t1, t2;
   //More C++ code
}
void main()
{
   teacher t3, t4;
   //More C++ code
   {
      teacher t5, t6;
      //More C++ code
   }
   //More C++ code
}
```

Structure **teacher** is declared globally. Generally a structure is declared as a global identifier so that it can be used throughout the program.

Global structure **teacher** creates 2 local structure variables **t1** and **t2** inside the function **funct**().

Global structure **teacher** creates two local structure variables **t3** and **t4** inside the **main**() function. Two more local structure variables **t3** and **t4** are created inside the nested block.

But a structure can be declared locally inside a block. In that case, structure declaration can create variable of that structure declaration inside that particular block and blocks nested below. An example is given below:

```
void main()
{
   struct teacher
   {
      int tcode;
      char name[20];
      char subject[20];
      double salary;
   };
   teacher a1, a2;
   //More C++ Code
   {
      teacher b1, b2;
      //More C++ Code
   }
}
void funct()
{
   teacher t1, t2;
   //More C++ Code
}
```

Structure **teacher** is declared locally inside the **main**() function. Structure **teacher** can be used to create structure variable inside **main**() function and block nested below **main**() function.

Local structure **teacher** creates two local structure variables **a1** and **a2** inside the **main**() function. It also creates two more structure variables **b1** and **b2** inside the nested block.

Syntax error because the structure **teacher** is declared locally inside the **main**() function.

Normally structure declaration starts with keyword **struct** followed by a structure name; then a block containing all the data members and block is terminated by a semi-colon. The structure name becomes a data type. Structure name is used to create structure variables in the program. But it is possible to create structure without a structure name. An example is given below:

```
struct
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
};
```

Perfectly correct structure declaration but is of no use in the program since the structure is without any name.

But structure declaration without a name (type less structure) is of no use in a C++ program unless structure variable(s) is (are) created along with the structure declaration. An example is given below:

```
struct
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
} t1, t2;
```

A nameless (type less) structure is created and at the end structure declaration, structure variables **t1** and **t2** are also created so that **t1** and **t2** can used in the program.

Global and Local structure variable: Generally a structure is created as a global identifier but structure variables are created locally (inside a block). In all our previous examples, structure variables are created locally. Using a global variable goes against the paradigm of Object Oriented Programming but that does not stop a programmer to create global structure variable. Some examples are given below showing how to create global structure variable.

Example #1:

```
struct teacher
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
};

teacher t1;

void main()
{
   teacher t2;
   //More C++ Code
}
```

Example #2:

```
struct teacher
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
} t1;

void main()
{
   teacher t2;
   //More C++ Code
}
```

Example #3:

```
struct
{
   int tcode;
   char name[20];
   char subject[20];
   double salary;
} t1, t2;

void main()
{
   //More C++ Code
}
```

Function and Structures

Just like fundamental data and array type, a structure type can be passed as parameter to a function. A structure type can either be passed as value **parameter** or a **reference** parameter to a function. By **default**, structure type is passed by **value** to a function. A complete program is given below showing the use of structure as parameter to a function.

```
#include<iostream.h>
#include<stdio.h>
struct contact
{
  char name[20];
  int phone, mobile;
};
void inputdata(contact& c1)
{
  cout<<"Name  ? "; gets(c1.name);
  cout<<"Phone ? "; cin>>c1.phone;
  cout<<"Mobile? "; cin>>c1.mobile;
}
void viewdata(contact c2)
{
  cout<<"Name  ="<<c2.name<<endl;
  cout<<"Phone ="<<c2.phone<<endl;
  cout<<"Mobile="<<c2.mobile<<endl;
}
void main()
{
  contact co;
  inputdata(co);
  viewdata(co);
}
```

Formal parameter **c1** is passed as **reference** since input of values inside the function **inputdata**() must update actual parameter **co**. Formal parameter **c2** is passed as **value** parameter since function **viewdata**() only displays the value stored in **co**.

Running of the program produces following output

```
Name  ? Tarun Kumar
Phone ? 25649873
Mobile? 99325681
Name  = Tarun Kumar
Phone =25649873
Mobile=99325681
```

If we edit the function inputdata() and make formal parameter c1, a value parameter (by removing &), then the running of the program produces following output

```
Name  ? Tarun Kumar
Phone ? 25649873
Mobile? 99325681
Name  =☻|~g&s
Phone =4243884
Mobile=-19464465
```

Actual parameter co is passed by value to inputdata() function (formal parameter c1 is copy of actual parameter co). Updating formal parameter c1 does not update actual parameter co.

We have learned how pass a structure type as parameter to a function. Now we are going to learn that return value of a function could be structure type as well. An example is given below:

```
#include<iostream.h>
struct Time
{
   int hh, mm;
};
void InputTime(Time& t)
{
   cout<<"Hour  ? "; cin>>t.hh;
   cout<<"Minute? "; cin>>t.mm;
}
void ShowTime(Time t)
{
   cout<<"Time="<<t.hh<<':'<<t.mm<<endl;
}
Time AddTime(Time t1, Time t2)
{
   Time t;
   int m=t1.mm+t2.mm
   t.hh=t1.hh+t2.hh+m/60;
   t.mm=m%60;
   return t;
}
void main()
{
   Time t1, t2;
   InputTime(t1);
   InputTime(t2);
   Time t3=AddTime(t1,t2);
   ShowTime(t1);
   ShowTime(t2);
   ShowTime(t3);
}
```

Return value of the function **AddTime**() is **Time** where **Time** is a structure. Statement **return t**, returns value stored in structure variable **t** to the calling function.

Running of the program **twice**, produces following output

```
Hour  ? 2
Minute? 30
Hour  ? 3
Minute? 10
Time=2:30
Time=3:10
Time=5:40
Hour  ? 4
Minute? 50
Hour  ? 3
Minute? 40
Time=4:50
Time=3:40
Time=8:30
```

Array of structure: So far we have created one or two structure(s) in our program (in our example). Suppose we want to represent many structure variables of same structure type in computer's main storage, then we have to create an array of structures. To create an array of structures, first we have to create a structure and then we will create an array of that structures. Each element of the array will be a structure variable.

```
struct StructName
{
  //DataMembers
};
StructName ArrName[SIZE];
```

Name of the structure is **StructName**. Name of the array is **Arr**. **Arr** is an array of structure. **SIZE** is a constant representing number of elements in the array. **SIZE** could either be a user defined constant or a constant like **10** or **20**. Every elements of array **Arr** will represent structure variable.

Syntax to access an element of an array of objects is:

```
ArrName[Index]

ArrName[0], ArrName[1], ArrName[2],…
```

are the elements of the array `ArrName[]`. Each element of `ArrName[]` represents a structure variable. Syntax to access data members of an array of structures is given below:

```
ArrName[Index].DataMember
```

Programming example is given below explaining the concept of array of structures:

```
struct student
{
  int roll;
  char name[20];
  double mark;
};
student arr[5];
```

Structure **student** has 3 data members – **roll**, **name** and **mark**. Array **arr[5]** is an array of **student**.

`arr[0]`, `arr[1]`, `arr[2]`, `arr[3]` and `arr[4]` are the 5 elements of the array. Each element of the array `arr[]` is a structure variable. List below shows how to access data members of each element of array `arr[]`.

arr[0].roll arr[0].name arr[0].mark	arr[1].roll arr[1].name arr[1].mark	arr[2].roll arr[2].name arr[2].mark	arr[3].roll arr[3].name arr[3].mark	arr[4].roll arr[4].name arr[4].mark

A complete program is given below showing use of array of objects.

```
#include<iostream.h>
#include<stdio.h>
const MAX=20;
struct student
{
  int roll;
  char name[20];
  double mark;
};
```

```
void input(student& stu)
{
  cout<<"Roll? "; cin>>stu.roll;
  cout<<"Name? "; gets(stu.name);
  cout<<"Mark? "; cin>>stu.mark;
}
void display(student stu)
{
  cout<<"Roll= "<<stu.roll<<endl;
  cout<<"Name= "<<stu.name<<endl;
  cout<<"Mark= "<<stu.mark<<endl;
}
void main()
{
  student arr[MAX];
  for (int k=0; k<MAX; k++)
    input(arr[k]);
  double hm=0, sum=0;
  for (int x=0; x<MAX; x++)
  {
    display(arr[x]);
    sum+=arr[x].mark;
    if (hm<arr[x].mark)
      hm=arr[x].mark;
  }
  double avg=sum/MAX;
  cout<<"Highest="<<hm<<" & Average="<<avg<<endl;
}
```

Or,

```
void arrinput(student a[], int n)
{
  for (int k=0; k<n; k++)
    input(arr[k]);
}
void arrdisplay(student a[], int n)
{
  for (int k=0; k<n; k++)
    display(arr[k]);
}
void arrhiav(student a[], int n)
{
  double hm=0, sum=0;
  for (int k=0; k<MAX; k++)
  {
    sum+=arr[k].marks;
    if (hm>arr[k].mark)
      hm=arr[k].mark;
  }
  double avg=sum/n;
  cout<<"Highest="<<hm<<" & Average="<<avg<<endl;
}
```

```
void main()
{
   student arr[MAX];
   arrinput(arr, MAX);
   arrdisplay(arr, MAX);
   arrhiav(arr, MAX);
}
```

An initializer can be used with an array of structures to initialize data members of each element of an array of structure. Examples are given below:

```
struct student
{
   int roll;
   char name[20];
   double mark;
};
```

Array `arr[]`, array of structure `student` is fully initialized.

```
student arr[5]=  {  {10, "Bina Varghese", 50.0},
                    {13, "Dinesh Thakur", 80.5},
                    {22, "Manish Shrama", 75.0},
                    {24, "Pamela Gupta", 90.5},
                    {35, "Rashid Ali", 65.5}
                 };
```

Array `arr[]`, array of structure `item` is partially initialized.

```
student arr[5]=  {  {10, "Bina Varghese", 50.0},
                    {13, "Dinesh Thakur", 80.5},
                    {22, "Manish Shrama", 75.0},
                    {24, "Pamela Gupta"}
                 };
```

`arr[3].mark` will contain the value `0` (zero).
`arr[4].roll` will contain the value `0`.
`arr[4].name` will contain the value `""` (Nul string or Empty string).
`arr[4].mark` will contain the value `0` (zero).

If values inside the initializer exceeds the number of elements in the array or exceeds the number of data members of structure, then the compiler flags a syntax error. Example is given below:

```
student arr[5]=  {  {10, "Bina Varghese", 50.0},
                    {13, "Dinesh Thakur", 80.5},
                    {22, "Manish Shrama", 75.0},
                    {24, "Pamela Gupta", 90.5},
                    {35, "Rashid Ali", 65.5},
                    {36, "Tarun Jain", 86.0},
                    {40, "Sudipta Lamba", 72.0, "A1"}
                 };
```

```
void bubblesortroll(student arr[], int n)
{
   for(int x=1; x<n; x++)
      for(int k=0; k<n-x; k++)
         if (arr[k].roll>arr[k+1].roll)
         {
            student t=arr[k];
            arr[k]=arr[k+1];
            arr[k+1]=t;
         }
}
```

Function to sort an array of **student** on **roll** using **bubble** sort.

```
void selectionsortroll(student arr[], int n)
{
   for(int x=0; x<n-1; x++)
   {
      student min=arr[x];
      int pos=x;
      for(int k=x+1; k<n; k++)
         if (arr[k].roll<min.roll)
         {
            min=arr[k];
            pos=k;
         }
      arr[pos]=arr[x];
      arr[x]=min;
   }
}
```

Function to sort an array of **student** on **roll** using **selection** sort.

```
void insertionsortroll(student arr[], int n)
{
   for(int x=1; x<n; x++)
   {
      student t=arr[x];
      int k=x-1;
      while(k>=0 && t.roll<arr[k].roll)
      {
         arr[k+1]=arr[k];
         k--;
      }
      arr[k+1]=t;
   }
}
```

Function to sort an array of **student** on **roll** using **insertion** sort.

```
int linearsearchroll(student arr[], int n, int roll)
{
   int x=0, found=0;
   while (x<n && found==0)
      if (roll==arr[x].roll)
         found=1;
      else
         x++;
   return found;
}
```

Function to locate for a **roll** in an array of **student** using **linear** search. Return value of the function is **int**. If search is successful function returns value **1** otherwise **0**.

```
void linearsearchroll(student arr[], int n, int roll)
{
   int x=0, found=0;
   while (x<n && found==0)
      if (roll ==arr[x].roll)
         found=1;
      else
         x++;
   if (found==1)
      cout<<roll<<" Found in the array\n";
   else
      cout<<roll<<" Does not Exist in the array\n";
}
```

Function to locate for a `roll` in an array of `student` using **linear** search. Return value of the function is **void**. Status of search is displayed in the function.

```
int binarysearchroll(student arr[], int n, int roll)
{
   int lb=0, ub=n-1;
   int found=0, mid;
   while (lb<=ub && found==0)
   {
      mid=(ub+lb)/2;
      if (roll<arr[mid].roll)
         ub=mid-1;
      else
      if (roll>arr[mid].roll)
         lb=mid+1;
      else
         found=1;
   }
   return found;
}
```

Function to locate for a `roll` in a sorted array of `student` using **binary** search. Return value of the function is **int**. If search is successful function returns value `1` otherwise `0`.

```
void binarysearchroll(student arr[], int n, int roll)
{
   int lb=0, ub=n-1;
   int found=0, mid;
   while (lb<=ub && found==0)
   {
      mid=(ub+lb)/2;
      if (roll<arr[mid].roll)
         ub=mid-1;
      else
      if (roll>arr[mid].roll)
         lb=mid+1;
      else
         found=1;
   }
   if (found==1)
      cout<<roll<<" Found in the array\n";
   else
      cout<<roll<<" Does not Exist in the array\n";
}
```

Function to locate for a `roll` in a sorted array of `student` using **binary** search. Return value of the function is **void**. Status of search is displayed in the function.

```
void bubblesortname(student arr[], int n)
{
   for(int x=1; x<n; x++)
      for(int k=0; k<n-x; k++)
         if (strcmp(arr[k].name, arr[k+1].name)>0)
         {
            student t=arr[k];
            arr[k]=arr[k+1];
            arr[k+1]=t;
         }
}
```

Function to sort an array of `student` on `name` using **bubble** sort.

```
void selectionsortname(student arr[], int n)
{
   for(int x=0; x<n-1; x++)
   {
      student min=arr[x];
      int pos=x;
      for(int k=x+1; k<n; k++)
         if (strcmp(arr[k].name, min.name)<0)
         {
            min=arr[k];
            pos=k;
         }
      arr[pos]=arr[x];
      arr[x]=min;
   }
}
```

Function to sort an array of `student` on `name` using **selection** sort.

```
void insertionsortname(student arr[], int n)
{
   for(int x=1; x<n; x++)
   {
      student t=arr[x];
      int k=x-1;
      while(k>=0 && strcmp(t.name, arr[k].name)<0)
      {
         arr[k+1]=arr[k];
         k--;
      }
      arr[k+1]=t;
   }
}
```

Function to sort an array of `student` on `name` using **insertion** sort.

```
int linearsearchroll(student arr[], int n, int roll)
{
   int x=0, found=0;
   while (x<n && found==0)
      if (arr[x].roll==roll)
         found=1;
      else
         x++;
   return found;
}
```

Function to locate for a `roll` in an array of `student` using **linear** search. Return value of the function is **int**. If search is successful function returns value `1` otherwise `0`.

```
void linearsearchname(student arr[], int n, char name[])
{
   int x=0, found=0;
   while (x<n && found==0)
      if (strcmp(name, arr[x].name)==0)
         found=1;
      else
         x++;
   if (found==1)
      cout<<name<<" Found in the array\n";
   else
      cout<<name<<" Does not Exist in the array\n";
}
```

Function to locate for a **name** in an array of **student** using **linear** search. Return value of the function is **void**. Status of search is displayed in the function.

```
int binarysearchroll(student arr[], int n, int roll)
{
   int lb=0, ub=n-1, found=0, mid;
   while (lb<=ub && found==0)
   {
      mid=(ub+lb)/2;
      if (arr[mid].roll>roll)
         ub=mid-1;
      else
      if (arr[mid].roll<roll)
         lb=mid+1;
      else
         found=1;
   }
   return found;
}
```

Function to locate for a **roll** in a sorted array of **student** using **binary** search. Return value of the function is **int**. If search is successful function returns value 1 otherwise 0.

```
void binarysearchname(student arr[], int n, char name[])
{
   int lb=0, ub=n-1, found=0, mid;
   while (lb<=ub && found==0)
   {
      mid=(ub+lb)/2;
      if (strcmp(name, arr[mid].name)<0)
         ub=mid-1;
      else
      if (strcmp(name, arr[mid].name)>0)
         lb=mid+1;
      else
         found=1;
   }
   if (found==1)
      cout<<name<<" Found in the array\n";
   else
      cout<<name<<" Does not Exist in the array\n";
}
```

Function to locate for a **name** in a sorted array of **student** using **binary** search. Return value of the function is **void**. Status of search is displayed in the function.

```
void mergeroll(student a[],student b[],student c[],int n1,int n2)
{
   int i=0, j=0, k=0;
```

```
    while (i<n1 && j<n2)
      if (a[i].roll<b[j].roll)
        c[k++]=a[i++];
      else
        c[k++]=b[j++];
    while (i<n1)
      c[k++]=a[i++];
    while (j<n2)
      c[k++]=b[j++];
  }
```

Function to **merge** two arrays of **student** sorted on **roll** to obtain the third array also sorted on **roll**. All three arrays are sorted in **ascending** order on **roll**.

```
  void mergename(student a[],student b[],student c[],int n1,int n2)
  {
    int i=0, j=0, k=0;
    while (i<n1 && j<n2)
      if (strcmp(a[i].name,b[j].name)<0)
        c[k++]=a[i++];
      else
        c[k++]=b[j++];
    while (i<n1)
      c[k++]=a[i++];
    while (j<n2)
      c[k++]=b[j++];
  }
```

Function to **merge** two arrays of **student** sorted on **name** to obtain the third array also sorted on **name**. All three arrays are sorted in **ascending** order on **name**.

```
  void arrinsert(student arr[], int& n, int pos, student item)
  {
    if (n==MAX)
      cout<<"Overflow\n";
    else
    {
      for (int x=n-1; x>=pos; x--)
        arr[x+1]=arr[x];
      arr[pos]=item;
      n++;
      cout<<item.roll<<','<<item.name<<','<<item.mark;
      cout<<" inserted in the array\n";
    }
  }
```

Function to **insert** **student** in an array of **student**. Constant **MAX** is the size of the array. Array name, number of elements currently in the array, position for insertion and the **student** that is to be inserted are passed as parameters to the function.

```
  void arrdelete(student arr[], int& n, int pos)
  {
    if (n==0)
      cout<<"Underflow\n";
    else
    {
      student item=arr[pos];
      for (int x=pos+1; x<n; x++)
        arr[x-1]=arr[x];
      n--;
      cout<<item.roll<<','<<item.name<<','<<item.mark;
      cout<<" deleted from the array\n";
    }
  }
```

Function to **delete** from an array of **student**. Array name, number of elements currently in the array and position for deletion are passed as parameters to the function.