Lecture-15

Data Structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types of heterogeneous nature. For instance, the elements storing the student's information (such as roll number, name, class, marks grade etc) needs to be processed together under one roof. Such a collection of heterogeneous nature of data is called a structure. The individual members which are called fields or members can be accessed and processed separately.

Structures are one of the two building blocks in understanding of *class* and *objects*. A structure is a collection of data while a class is a collection of data and functions. This means a structure along with related functions, make a class. In a structure, all the data members are public by default, where as in class members can be declared public as well as private. By default all the members of a class are private.

15.1 Declaration of Data structures

A data structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types (compare with arrays where members of same type) and different lengths. Second difference between arrays and structure is, members of arrays are referred by their position where as in structure they are referred by its unique name.

Data structures are declared in C++ using the following syntax:

```
struct structure_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as

structure_name is created and can be used in the rest of the program as if it was any other type. For example:

```
struct product {
  int weight;
  float price;
} ;

product apple;
product banana, melon;
```

We have first declared a structure type called product with two members: weight and price, each of a different fundamental type. We have then used this name of the structure type (product) to declare three objects of that type: apple, banana and melon as we would have done with any fundamental data type.

Once declared, product has become a new valid type name like the fundamental ones int, char or short and from that point on we are able to declare objects (variables) of this compound new type, like we have done with apple, banana and melon.

Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field object_name to directly declare objects of the structure type. For example, we can also declare the structure objects apple, banana and melon at the moment we define the data structure type this way:

```
struct product {
  int weight;
  float price;

} apple, banana, melon;
```

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like apple, banana and melon) from a single structure type (product).

Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.weight
apple.price
banana.weight
banana.price
```

melon.weight

melon.price

Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weight and melon.weight are of type int, while apple.price, banana.price and melon.price are of type float.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

Program 15.1 :

```
#include <iostream>
using namespace std;
void main (void)
{
      struct sample{
            int x;
            float y;
      };
      struct sample a;
      a.x=10;
      a.y=20.20;
      cout<<" contents of x="<<a.x<<endl;
      cout<<" contents of y="<<a.y<<endl;
}
 /* contents of x=10
     contnts of y=20.0 */
```

The example shows how we can use the members of an object as regular variables. For example, the member a.x is a valid variable of type int, and a.y is a valid variable of type float. In this example line struct sample a; can be written as part of structure, called object.

It is possible to define a structure variable name in the structure type declaration itself as in the following example.

```
#include <iostream>
using namespace std;
void main (void)
{
      struct sample{
            int x;
            float y;
      }a;
      a.x=10;
      a.y=20.20;
      cout<<" contents of x="<<a.x<<endl;
      cout<<" contents of y="<<a.y<<endl;
}
 /* contents of x=10
```

```
     contnts of y=20.0 */
```

Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them:

Program 15.2

```
//Structure initialisation
#include <iostream>
using namespace std;
void main (void)
{
      struct school{
            long int rollno;
            int age;
            char sex;
            float height;
            float weight;
      };
      school student = {95001,24,'M',167.9,56.7};
      cout<<" contents of structure\n";
      cout<<" Roll No="<<student.rollno<<endl;
      cout<<" Age ="<<student.age<<endl;
      cout<<" Sex="<<student.sex<<endl;
      cout<<" Height ="<<student.height<<endl;
      cout<<" Weight ="<<student.weight<<endl;
}
 /* contents of structure
      Roll No= 95001
      Age = 24
      Sex= M
      Height =167.9
      Weight = 56.7 */
```

Other declaration

A field or member of a structure is a unique name for the particular structure. Same field or member name can be given to other structures with different data type. The C++ compiler will treat each structure member as a separate variable and will reserve the memory space according to the data type of the member.

In the program 15.3 we will use to structures with same variables with different data type and print the output to see that these variable work differently in two structures.

Program 15.3 :

```
#include <iostream>
using namespace std;
void main (void)
{
```

```
      struct first{
            int a;
            float b;
            char c;
      };
      struct second{
            char a;
            int b;
            float c;
      };
      first one;
      second two;
      one.a=23;
      one.b=11.89;
      one.c='f';
    two.a='m';
      two.b=5;
      two.c=56.9;
      cout<<" contents of first structure \n";
      cout<<one.a<<'\t'<<one.b<<'\t'<<one.c<<endl;
      cout<<" contents of second structure \n";
      cout<<two.a<<'\t'<<two.b<<'\t'<<two.c<<endl;
}
/* contents of first structure
 23 11.89  f
contents of second structure
m  5  56.9
```

15.2 Nested Structures

A structure element may be simple or complex. The simple elements are any of the fundamental data type of C++ i.e. int, float, char or double. However a structure may consist of an element that itself is complex i.e it is made up of fundamental type, e.g. arrays, structures etc. A structure consisting of such comlex elements is called a complex structure. Here e first discuss structures inside another structure. This type of structures are called nested structures.

A code for structure nested inside another structure is as follows:

```
      struct addr                    //structure tag
                {
                  int houseno;
                  char area[26];
                  char city [26];
                  char state [26];
               };
                Struct emp
                  {
                   int empno;
                    char name [26];
```

```
                    char design [16];
                    addr address;          //another structure
                     float basic;
                     };
                     emp worker;        //create structure variable
```

The structure **emp** has been defined having several elements including a structure **address** too. The element address is an object of structure addr. While defining such structures , one should make sure that inner structure is defined before the structure here this structure is included as element.

If we want to access the city where object worker of structure emp lives we will use

worker.address.city

Similarly to initialize house number of the worker of structure emp, we write

worker.address.houseno =889

Folloing program demonstrate the functioning of nested arrays

Program 15.4

```
// Nested structures

#include <iostream>
#include <conio.h>  //for clrscr()
 using namespace std;
#include<stdio.h>   //for gets

struct addr     //global definition
{
      int houseno;
      char area[26];
      char city[20];
      char state[20];
};
struct emp
{
      int empno;
      char name[20];
      char desig[20];
      addr address;    //another structure
      float basic;
}worker;

  int main()
  {
      //  clrscr();
        cout<<"\n"<<"Enter Employee no:\n";
        cin >>worker.empno;
        cout<<"\n"<<"Name:";
```

```
        gets(worker.name);  //to read white spaces as >> cannot read
them
        cout<<"\n"<<"Designation:"<<"\n";
        gets(worker.desig);
       cout<<"\n"<<"Address:";
              cout<<"\n"<<"House no.:";
              cin>>worker.address.houseno;
         cout<<"\n"<<"Area:";
        gets(worker.address.area);
         cout<<"\n"<<"City:";
               gets(worker.address.city);
         cout<<"\n"<<"State:";
               gets(worker.address.state);
         cout<<"\n"<<"Basic Pay:";
             cin>>worker.basic;
             //Output
      cout<<"\n"<<"Employee no:\n"<<worker.empno;
        cout<<"\n"<<"Name:"<<worker.name;
        cout<<"\n"<<"Designation:"<<worker.desig;
       cout<<"\n"<<"Address:"<<worker.address.houseno;
             cout<<worker.address.area<<"\n";
         cout<<"\n"<<"City:"<<worker.address.city;
                     cout<<"\n"<<"State:"<<worker.address.state;
                  cout<<"\n"<<"Basic Pay:"<<worker.basic;
                 return 0;
  }//end of main
```

15.3 Structures and arrays.

In the present section we will store student data of whole class or college in the form of arrays using structures. This has been demonstrated in the following example.

```
struct school{
     int rollno;
    int age;
     char sex;
      float height;
      float weight;
};
  school student [300];
```

The student [300] is a structure variable. It can accommodate the structure student upto 300. Each record can be accessed and processed separately like individual element of an array.

Initialisation of this array can be done as follows:

```
struct school{
     int rollno;
    int age;
     char sex;
```

```
    float height;
    float weight;
}
  school student [3] = {
                        {95001,24,'M',167.9,56.7}
                         {95002,25,'F',157.9,46.7}
                         {95003,26,'M',187.9,76.7}
                         };
```

15.4 Arrays within a structure

So far the discussion has been limited to members of a structure which have been declared as an ordinary type such as int, float, char etc. However member of a structure can be an array also. Following program demonstrates this property of a structure.

Program 15.5

```
//Array within structure
#include <iostream>
using namespace std;
#define MAX 4

void main (void)
{
 struct school{
      char name[20];
     int rollno;
   int age;
    char sex;
     float height;
     float weight;
};
  school student [MAX] = {
                        {"Ram",95001,24,'M',167.9,56.7}
                         {"Sham",95002,25,'F',157.9,46.7}
                         {"Krishan",95003,26,'M',187.9,76.7}
                         {"Mahadevan",95004,26,'M',188.9,79.7}
                         };
   for(i=0; i<= MAX-1; ++i) {

      cout<<"Conteents of structure:"<<i+1<<endl;
  cout<<"Name:"<<student[i].name<<endl;
  cout<<"Rollno:"<<student[i].rollno<<endl;
  cout<<"Age:"<<student[i].age<<endl;
    cout<<"Sex:"<<student[i].sex<<endl;
```

```
  cout<<"Height:"<<student[i].height<<endl;
   cout<<"Weight:"<<student[i].weight<<endl;
}
}
```

15.5 Structures and Functions.

A structure can be passed to a function as a single variable. The scope of a structure declaration should be an external storage class whenever a function in the main program is using a structure data types. The field or member data should be same throughout the program either in the main or in a function. Program 15.4 displays the contents of a structure using function definition.

Program15.4.

```
// Stucture with functions
#include <iostream>
#include <fstream>
using namespace std;


struct date{
      int day;
      int month;
      int year;
};
void main (void)
{
      date today;
      void display (struct date one);
      //Fuction declaration

      today.day=10;
      today.month=03;
      today.year=2007;
      display(today);
}

void display (struct date one)//Structure past to a function
{
      ofstream outfile("result.txt", ios::out);
      outfile << "Today's date is = "<<one.day<<"/" ;
      outfile<<one.month;
      outfile<<"/"<<one.year<<endl;
}
OUTPT
//Today's date is = 10/3/2007
```

15.6 Pointers to structures

Like any other type, structures can be pointed by its own type of pointers: A pointer can be used to hold the address of the the structure variable as well.

The following declaration is valid.

```
struct movies_t {
  string title;
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

Here amovie is an object of structure type movies_t, and pmovie is a pointer to point to objects of structure type movies_t. So, the following code would also be valid:

```
pmovie = &amovie;
```

The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address).

Following example will further illustrate the use of pointers in structures.

Program 15.5

```
#include <iostream>
#include <fstream>
using namespace std;
void main(void)
{ struct sample {
      int x;
      int y;
};
sample *ptr;
sample one;
ptr =&one;
(*ptr).x=10;
(*ptr).y=20;
cout<<"contents of x=" <<(*ptr).x<<endl;
cout<<"contents of y=" <<(*ptr).y<<endl;
cout<<"contents of x=" <<one.x<<endl;
cout<<"contents of x=" <<one.y<<endl;
}
```

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (->):

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

```
Enter title: Invasion of
the body snatchers
Enter year: 1978

You have entered:
Invasion of the body
snatchers (1978)
```

The previous code includes an important introduction: the arrow operator (->). This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

```
pmovie->title
```

Which is for all purposes equivalent to:

```
(*pmovie).title
```

Both expressions pmovie->title and (*pmovie).title are valid and both mean that we are evaluating the member title of the data structure pointed by a pointer called pmovie. It must be clearly differentiated from:

```
*pmovie.title
```

which is equivalent to:

```
*(pmovie.title)
```

And that would access the value pointed by a hypothetical pointer member called title. The following panel summarizes possible combinations of pointers and structure members:

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)