UNIT - 3

CLASSES & OBJECTS

CLASSES & OBJECTS

Introduction

Object oriented programming paradigm makes it easy to solve real-world problems. Classes and objects are the fundamental concepts of object oriented paradigm. They make development of large and complex systems easier and help to produce software which is easy to understand, modular, modify, and reusable.

Syntactically structures and classes are similar. But a major difference between them is, by default all the members of a structure are *public*. Whereas, by default all the members of a class are *private*.

A rule of thumb is, use structures to store less amount of data and classes to store data as well as functions to process that data.

Class Declaration

A class is a template or blueprint for creating objects. Class is the basic mechanism to implement encapsulation. A class combines variables (data) and functions (behavior) into a single entity. By declaring a class we are creating a new user-defined type. Syntax for declaring a class is as follows:

```
class ClassName
{
        access-specifier:
                type variable1;
                type variable2;
                ....
                type function1(params-list);
                type function2(params-list);
                ....
};
```

In the above class declaration syntax *class* is a keyword which is used to declare classes. The body of the class starts with { and ends with };. A class can have variables and functions. These are together called as class members. Access specifier can be either *public*, *private*, or *protected*.

Private: All class members declared as private are only accessible within the class and are not accessible outside the class. Data hiding is implemented through *private* access specifier. By default access specifier for class members is *private*.

Public: All class members declared as public are accessible inside and outside the class.

Example for class declaration is as follows:

```
class Student
{
	private:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details();
		void show_details();
};
```

Function Definition

Inside Class

A function can be defined inside a class like we define functions in our C++ programs. Functions defined inside a class are by default treated as inline functions (provided they don't contain any loops, switch, goto, static variables, or recursive code). An example of function defined inside the class is as follows:

```
class Student
{
	private:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details()
		{
			cout<<"Enter student name: ";
			cin>>name;
			cout<<"Enter student regdno: ";
			cin>>regdno;
			cout<<"Enter student branch: ";
			cin>>branch;
			cout<<"Enter student age: ";
			cin>>age;
		}
		void show_details();
};
```

Outside Class

Functions defined outside the class will contain scope resolution operator (::) to notify the compiler that this function belongs to a certain class.

Uses of scope resolution operator in a function definition are:

- When a function with same name is available in different classes, this operator resolves the scope of a function to a certain class.
- To restrict access to private data of a class to non-member functions.
- Allows a member function of a class to call another member function of the same class without using dot operator.

Syntax of defining a function outside the class is as follows:

```
return-type class-name :: function-name(params-list)
{
	//statement(s)
}
```

An example of defining a function outside the class is as follows:

```
class Student
{
	private:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details();
		void show_details();
};
void Student::get_details()
{
	cout<<"Enter student name: ";
	cin>>name;
	cout<<"Enter student regdno: ";
	cin>>regdno;
	cout<<"Enter student branch: ";
	cin>>branch;
	cout<<"Enter student age: ";
	cin>>age;
}
```

Creating Objects

For using the classes that we declared in our programs, we need to create objects. An object is an instance of a class. The process of creating an object for a class in called as **instantiation**. Memory is only allocated for an object. Not to a class. Object creation syntax is as follows:

```
ClassName object-name;
```

Based on our student class example, objects can be created as follows:

Student std1, std2;

In the above example, *std1* and *std2* are objects of class *Student*.

Accessing Object Members

After creating an object, we can access the data and functions using the dot operator as shown below:

object-name.variable-name; (or)

object-name.function-name(params-list);

Based on our student class example, we can access the *get_details()* function as shown below:

```
std1.get_details();
std2.get_details();
```

The complete program which demonstrates creating class, creating objects, and accessing object members is as follows:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details();
		void show_details();
};
void Student::get_details()
{
	cout<<"Enter student name: ";
	cin>>name;
	cout<<"Enter student regdno: ";
	cin>>regdno;
	cout<<"Enter student branch: ";
	cin>>branch;
	cout<<"Enter student age: ";
	cin>>age;
}
```

```
void Student::show_details()
{
	cout<<"-----Student Details-----"<<endl;
	cout<<"Student name: "<<name<<endl;
	cout<<"Student regdno: "<<regdno<<endl;
	cout<<"Student branch: "<<branch<<endl;
	cout<<"Student age: "<<age<<endl;
}
int main()
{
	Student std1, std2;
	std1.get_details();
	std1.show_details();
	return 0;
}
```

Input and output for the above program are as follows:

```
Enter student name: teja
Enter student regdno: 15pa1a0501
Enter student branch: teja
Enter student age: 22
-----Student Details-----
Student name: teja
Student regdno: 15pa1a0501
Student branch: teja
Student age: 22
```

In the above program class name is *Student* and object names are *std1* and *std2*.

Nested Functions

A member function can call another member function directly without any need of dot operator. To call another member function, we can simply write the function name followed by parameter values enclosed in parentheses. An example for calling another member function is as follows:

```
#include<iostream>
using namespace std;
class Square
{
	public:
		int side;
	public:
		float area(float);
		float peri(float);
		void show_details();
};
```

```
float Square::area(float s)
{
	return s*s;
}
float Square::peri(float s)
{
	return 4*s;
}
void Square::show_details()
{
	cout<<"Side is: "<<side<<endl;
	cout<<"Area of sqaure is: "<<area(side)<<endl;
	cout<<"Perimeter of sqaure is: "<<peri(side)<<endl;
}
int main()
{
	Square s1;
	s1.side = 2;
	s1.show_details();
	return 0;
}
```

Input and output for the above program is as follows:

```
Side is: 2
Area of sqaure is: 4
Perimeter of sqaure is: 8
```

In the above program, the function *show_details()* contains nested call to *area()* and *peri()* functions.

Inline Member Functions

Making a function inline avoids the overhead of a function call. By default functions defined inside a class are inline. To get the benefits of making a function inline, functions defined outside the class can also be made inline. Syntax for defining a function as inline is as follows:

```
inline return-type ClassName :: function-name(params-list)
{
	//Body of function
}
```

Consider the following example which demonstrates inline member functions:

```
#include<iostream>
using namespace std;
class Square
```

```
{
	public:
		int side;
	public:
		float area(float);
		float peri(float);
		void show_details();
};
inline float Square::area(float s)
{
	cout<<"Area of square is: "<<(s*s)<<endl;
}
inline float Square::peri(float s)
{
	cout<<"Perimeter of square is: "<<(4*s)<<endl;
}
void Square::show_details()
{
	cout<<"Side is: "<<side<<endl;
	area(side);
	peri(side);
}
int main()
{
	Square s1;
	s1.side = 2;
	s1.show_details();
	return 0;
}
```

Input and output for the above program is as follows:

```
Side is: 2
Area of sqaure is: 4
Perimeter of sqaure is: 8
```

Memory Allocation for Classes and Objects

Class is not allocated any memory. This is partially true. The functions in a class are allocated memory which are shared by all the objects of a class. Only the data in a class is allocated memory when an object is created. Every object has its own memory for data (variables). For example consider the following *Student* class and its functions:

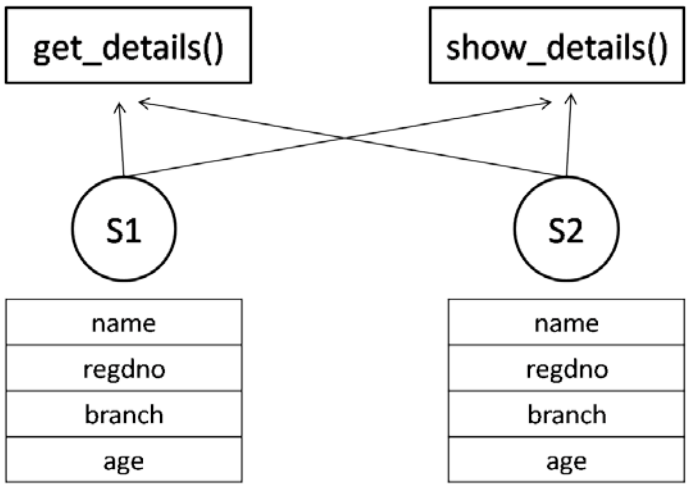
```
class Student
{
	private:
		string name;
		string regdno;
```

```
                string branch;
                int age;
        public:
                void get_details();
                void show_details();
};
void Student::get_details()
{
        cout<<"Enter student name: ";
        cin>>name;
        cout<<"Enter student regdno: ";
        cin>>regdno;
        cout<<"Enter student branch: ";
        cin>>branch;
        cout<<"Enter student age: ";
        cin>>age;
}
void Student::show_details()
{
        cout<<"-----Student Details-----"<<endl;
        cout<<"Student name: "<<name<<endl;
        cout<<"Student regdno: "<<regdno<<endl;
        cout<<"Student branch: "<<branch<<endl;
        cout<<"Student age: "<<age<<endl;
}
```

Now let's create two objects for the above *Student* class as follows:

Student std1, std2;

Memory representation for the objects s1 and s2 will be as shown in the below diagram:



Static Members

C++ allows us to declare variables or functions as static members by using the keyword *static*.

Static Data

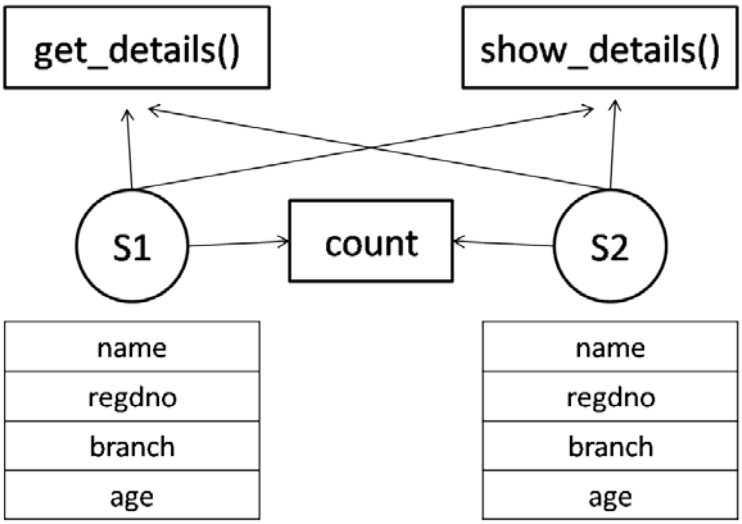
Syntax for declaring a static variable is as follows:

static data-type variable-name;

Static data members will have the following properties:

- Only one copy of static data element will be created irrespective of the no. of objects created.
- All objects share the same copy of the static data element.
- All static data members are initialized to 0 when the first object for a class is created.
- Static data members are only visible within the class. But their lifetime is throughout the program.

If needed, static data members can be accessed using the class name along with scope resolution operator. Let's add a static member *count* to our *Student* class. The memory representation of two objects will be as shown below:



Student class with *count* static member is as follows:

```
#include<iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                string branch;
                int age;
                static int count;
        public:
                void get_details();
                void show_details();
                void show_count()
                {
```

```
                cout<<"No.of students: "<<count++<<endl;
        }
};
int Student::count = 1;
int main()
{
        Student std1, std2;
        std1.show_count();
        std2.show_count();
        return 0;
}
```

Output of the above program is as follows:

```
No.of students: 1
No.of students: 2
```

Static Functions

Functions of a class can also be declared as static. Static functions have following properties:

- It can access only static members declared in the same class.
- As it is not part of an object, it is called using the class name along with scope resolution operator.
- As static functions does not belong to a object, *this* pointer cannot be used with them.
- A static function cannot be declared as a virtual function.
- A static function can be declared with *const, volatile* type qualifiers.

Syntax for creating a static member function is as follows:

```
static return-type function-name(params-list)
{
        //body of function
}
```

Syntax for accessing a static member function outside the class is as follows:

ClassName :: function-name(param-values list);

Following example demonstrates creating and using a static function:

```
#include<iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                string branch;
                int age;
```

```
		static int count;
	public:
		void get_details();
		void show_details();
		static void show_count()
		{
			cout<<"No.of students: "<<count++<<endl;
		}
};
int Student::count = 1;
int main()
{
	Student std1, std2;
	Student::show_count();
	Student::show_count();
	return 0;
}
```

Output for the above program is as follows:

```
No.of students: 1
No.of students: 2
```

Note: Two member functions cannot be declared with one function as static and another as non-static having the same name and same parameters list.

Static Objects

By default object data members contains garbage values. To initialize the data members to their default values, we can declare the objects as static. Syntax for creating static objects is as follows:

static ClassName object-name;

Below example demonstrates working with static objects:

```
#include<iostream>
using namespace std;
class Student
{
	public:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details();
		void show_details();
};
int main()
```

```
{
	static Student std1, std2;
	cout<<"Age of student 1 is: "<<std1.age<<endl;
	cout<<"Age of student 2 is: "<<std2.age<<endl;
	return 0;
}
```

Output for the above program is as follows:

```
Age of student 1 is: 0
Age of student 2 is: 0
```

In the above program, *std1* and *std2* are static objects. So all their data members like *name*, *age*, etc. will be initialized to their default values.

Array of Objects

As we can create array of basic data types, we can also create an array of user-defined types. An array of objects can be used to maintain details of a group of objects which are stored contiguously in memory. Syntax for creating an array of objects is as follows:

ClassName array-name[size];

For example we can create an array of objects for *Student* class which can be used to maintain details of a set of students. A single object in an array can be accessed using the subscript or index. Below program demonstrates the use of an array of objects:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		string branch;
		int age;
	public:
		void get_details();
		void show_details();
};
void Student::get_details()
{
	cout<<"Enter student name: ";
	cin>>name;
	cout<<"Enter student regdno: ";
	cin>>regdno;
	cout<<"Enter student branch: ";
	cin>>branch;
```

```
	cout<<"Enter student age: ";
	cin>>age;
}
void Student::show_details()
{
	cout<<"-----Student Details-----"<<endl;
	cout<<"Student name: "<<name<<endl;
	cout<<"Student regdno: "<<regdno<<endl;
	cout<<"Student branch: "<<branch<<endl;
	cout<<"Student age: "<<age<<endl;
}
int main()
{
	Student s[3];
	for(int i=0; i<3; i++)
	{
		cout<<"Enter student "<<i+1<<" details: "<<endl;
		s[i].get_details();
	}
	for(int i=0; i<3; i++)
	{
		cout<<"Student "<<i+1<<" details are: "<<endl;
		s[i].show_details();
	}
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter student 1 details:
Enter student name: ramesh
Enter student regdno: 101
Enter student branch: cse
Enter student age: 23
Enter student 2 details:
Enter student name: suresh
Enter student regdno: 102
Enter student branch: cse
Enter student age: 23
Enter student 3 details:
Enter student name: mahesh
Enter student regdno: 103
Enter student branch: cse
Enter student age: 24
Student 1 details are:
-----Student Details-----
Student name: ramesh
Student regdno: 101
Student branch: cse
Student age: 23
```

```
Student 2 details are:
-----Student Details-----
Student name: suresh
Student regdno: 102
Student branch: cse
Student age: 23
Student 3 details are:
-----Student Details-----
Student name: mahesh
Student regdno: 103
Student branch: cse
Student age: 24
```

Dynamic memory allocation for array of objects

In the previous topic memory for array of objects was static as the memory was allocated at compile time. To allocate memory dynamically or at run time, we use an array of pointers which can be created as follows:

*ClassName *array-name[size];*

Memory allocated for an array of pointers is far less than memory allocated for an array of objects. We can create an object at run time using the *new* operator as follows:

array-name[index] = new ClassName;

After dynamically allocating memory for an object we access the members using the -> operator as follows:

array-name[index]->member;

Below example demonstrates how to dynamically allocate memory for an array of objects:

```
#include<iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                string branch;
                int age;
        public:
                void get_details();
                void show_details();
};
void Student::get_details()
{
        cout<<"Enter student name: ";
```

```
	cin>>name;
	cout<<"Enter stduent regdno: ";
	cin>>regdno;
	cout<<"Enter student branch: ";
	cin>>branch;
	cout<<"Enter student age: ";
	cin>>age;
}
void Student::show_details()
{
	cout<<"-----Student Details-----"<<endl;
	cout<<"Student name: "<<name<<endl;
	cout<<"Student regdno: "<<regdno<<endl;
	cout<<"Student branch: "<<branch<<endl;
	cout<<"Student age: "<<age<<endl;
}
int main()
{
	Student *s[3];
	for(int i=0; i<3; i++)
	{
		s[i] = new Student;
		cout<<"Enter stduent "<<i+1<<" details: "<<endl;
		s[i]->get_details();
	}
	for(int i=0; i<3; i++)
	{
		cout<<"Student "<<i+1<<" details are: "<<endl;
		s[i]->show_details();
	}
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter student 1 details:
Enter student name: ramesh
Enter student regdno: 101
Enter student branch: cse
Enter student age: 23
Enter student 2 details:
Enter student name: suresh
Enter student regdno: 102
Enter student branch: cse
Enter student age: 23
Enter student 3 details:
Enter student name: mahesh
Enter student regdno: 103
Enter student branch: cse
Enter student age: 24
```

```
Student 1 details are:
-----Student Details-----
Student name: ramesh
Student regdno: 101
Student branch: cse
Student age: 23
Student 2 details are:
-----Student Details-----
Student name: suresh
Student regdno: 102
Student branch: cse
Student age: 23
Student 3 details are:
-----Student Details-----
Student name: mahesh
Student regdno: 103
Student branch: cse
Student age: 24
```

Objects As Function Arguments

Like variables, objects can also be passed using pass-by-value, pass-by-reference, and pass-by-address. In pass-by-value we pass the copy of an object to a function. In pass-by-reference we pass a reference to the existing object. In pass-by-address we pass the address of an existing object. Following program demonstrates all three methods of passing objects as function arguments:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
	public:
		string branch;
		void get_details();
		void show_details();
		//Pass-by-value
		void set_branch(Student s)
		{
			branch = s.branch;
		}
		//Pass-by-reference
		void set_branch(Student &s, int x)
		{
			branch = s.branch;
```

```
		}
		//Pass-by-address
		void set_branch(Student *s)
		{
			branch = s->branch;
		}
};
int main()
{
	Student s1, s2, s3, s4;
	s1.branch = "cse";
	s2.set_branch(s1);
	s3.set_branch(s1, 10);
	s4.set_branch(&s1);
	return 0;
}
```

Returning Objects

We can not only pass objects as function arguments but can also return objects from functions. We can return an object in the following three ways:

1. Returning by value which makes a duplicate copy of the local object.
2. Returning by using a reference to the object. In this way the address of the object is passed implicitly to the calling function.
3. Returning by using the 'this pointer' which explicitly sends the address of the object to the calling function.

Following program demonstrates the three ways of returning objects from a function:

```
#include<iostream>
using namespace std;
class Comparer
{
	public:
		int value;
		Comparer compare_val(Comparer);
		Comparer &compare_ref(Comparer);
		Comparer &compare_add(Comparer);
};
//Return by value
Comparer Comparer::compare_val(Comparer c)
{
	Comparer x;
	x.value = 10;
	if(x.value < c.value)
		return x;
	else
```

```
		return c;
}
//Return by reference
Comparer &Comparer::compare_ref(Comparer c)
{
	Comparer x;
	x.value = 10;
	if(x.value < c.value)
		return x;
	else
		return c;
}
//Return by address
Comparer &Comparer::compare_add(Comparer c)
{
	Comparer x;
	x.value = 10;
	if(x.value < c.value)
		return x;
	else
		return *this;
}
int main()
{
	Comparer obj;
	obj.value = 20;
	obj = obj.compare_val(obj);
	cout<<"Least value is: "<<obj.value<<endl;
	obj.value = 5;
	obj = obj.compare_ref(obj);
	cout<<"Least value is: "<<obj.value<<endl;
	obj.value = 2;
	obj = obj.compare_ref(obj);
	cout<<"Least value is: "<<obj.value<<endl;
	return 0;
}
```

Output for the above program is as follows:

```
Least value is: 10
Least value is: 5
Least value is: 2
```

this Pointer

For every object in C++, there is an implicit pointer called *this* pointer. It is a constant pointer which always points to the current object. Uses of *this* pointer are as follows:

- When data member names and function argument names are same, *this* pointer can be used to distinguish a data member by writing *this->member-name*.
- It can be used inside a member function to return the current object by address by writing *return *this*.

The *this* pointer cannot be used along with static functions as static functions are not part of any object. Following program demonstrates the use of *this* pointer:

```
#include<iostream>
using namespace std;
class Circle
{
	private:
		int r;
	public:
		void area(int);
		void peri(int);
};
void Circle::area(int r)
{
	this->r = r;
	cout<<"Area of circle is: "<<(3.1415*this->r*this->r)<<endl;
}
void Circle::peri(int r)
{
	this->r = r;
	cout<<"Perimeter of circle is: "<<(2*3.1415*this->r)<<endl;
}
int main()
{
	Circle c;
	c.area(10);
	c.peri(10);
	return 0;
}
```

Output for the above program is as follows:

```
Area of circle is: 314.15
Perimeter of circle is: 62.83
```

Modify the *area()* function in the above program with the following code:

```
void Circle::area(int r)
{
	r = r;
	cout<<"Radius of circle is: "<<this->r<<endl;
	cout<<"Area of circle is: "<<(3.1415*this->r*this->r)<<endl;
}
```

Now again compile and execute the program. Observe the value of radius in the output. Is it correct?

Constant Parameters and Members

Constant Member Functions

Member functions in a class can be declared as constant if that member function has no necessity of modifying any data members. A member function can be declared as constant as follows:

return-type function-name(params-list) const
{
//body of function
}

Following program demonstrates the use of constant member functions:

```
#include<iostream>
using namespace std;
class Circle
{
	private:
		int r;
	public:
		void area() const;
		void peri() const;
		void set_radius(int);
};
void Circle::area() const
{
	//r = 20;
	cout<<"Area of circle is: "<<(3.1415*r*r)<<endl;
}
void Circle::peri() const
{
	//r = 20;
	cout<<"Perimeter of circle is: "<<(2*3.1415*r)<<endl;
}
void Circle::set_radius(int radius)
{
	r = radius;
}
int main()
{
	Circle c;
	c.set_radius(10);
	c.area();
```

```
	c.peri();
	return 0;
}
```

Output for the above program is as follows:

Area of circle is: 314.15
Perimeter of circle is: 62.83

In the above program there is no need for *area()* and *peri()* functions to modify the data member *r*. So, they are declared as constant member functions.

Constant Parameters

To prevent functions from performing unintended operations on data members of an object, we can declare the object arguments as constant arguments as follows:

const ClassName object-name

Following program demonstrates the use of constant parameters:

```
#include<iostream>
using namespace std;
class Circle
{
	private:
		int r;
	public:
		void area() const;
		void peri() const;
		void set_radius(int);
		void get_radius(const Circle);
};
void Circle::area() const
{
	//r = 20;
	cout<<"Area of circle is: "<<(3.1415*r*r)<<endl;
}
void Circle::peri() const
{
	//r = 20;
	cout<<"Perimeter of circle is: "<<(2*3.1415*r)<<endl;
}
void Circle::set_radius(int radius)
{
	r = radius;
}
void Circle::get_radius(const Circle c)
{
	//c.r = 30;
```

```
	cout<<"Radius of circle is: "<<r<<endl;
}
int main()
{
	Circle c;
	c.set_radius(10);
	c.get_radius(c);
	c.area();
	c.peri();
	return 0;
}
```

Output of the above program is as follows:

```
Radius of circle is: 10
Area of circle is: 314.15
Perimeter of circle is: 62.83
```

In the above program inside the *get_radius()* function, *Circle* object has been declared as constant. So performing *c.r = 30* is invalid as modifying constant values is illegal.

Pointers within a Class

Pointers can be used inside a class for the following reasons:

- Pointers save memory consumed by objects of a class.
- Pointers can be used to allocate memory dynamically i.e., at run time.

Following program demonstrates the use of pointers within a class:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		int *marks;
	public:
		string branch;
		void get_details();
		void show_details();
		void get_marks();
};
void Student::get_marks()
{
	int n;
```

```
        cout<<"Enter no.of subjects: ";
        cin>>n;
        marks = new int[n];
        for(int i=0; i<n; i++)
        {
                cout<<"Enter subject "<<(i+1)<<" marks: ";
                cin>>marks[i];
        }
}
int main()
{
        Student s1;
        s1.get_marks();
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter no.of subjects: 6
Enter subject 1 marks: 20
Enter subject 2 marks: 15
Enter subject 3 marks: 16
Enter subject 4 marks: 19
Enter subject 5 marks: 20
Enter subject 6 marks: 18
```

In the above program *marks* is a pointer inside the class *Student*. It is used to allocate memory dynamically inside the *get_marks()* function with the help of *new* operator.

Local Classes

A class which is declared inside a function is called a local class. A local class is accessible only within the function it is declared. Following guidelines should be followed while using local classes:

- Local classes can access global variables only along with scope resolution operator.
- Local classes can access static variables declared inside a function.
- Local classes cannot access auto variables declared inside a function.
- Local classes cannot have static variables.
- Member functions must be defined inside the class.
- Private members of the class cannot be accessed by the enclosing function if it is not declared as a friend function.

Below example demonstrates a local class:

```
#include<iostream>
using namespace std;
const float PI = 3.1415;
```

```
int main()
{
	class Circle
	{
		public:
			int r;
			void area()
			{
				cout<<"Area of circle is: "<<(::PI*r*r);
			}
			void set_radius(int radius)
			{
				r = radius;
			}
	};
	Circle c;
	c.set_radius(10);
	c.area();
	return 0;
}
```

Output of the above program is as follows:

Area of circle is: 314.15

Nested Classes

C++ allows programmers to declare one class inside another class. Such classes are called nested classes. When a class B is declared inside class A, class B cannot access the members of class A. But class A can access the members of class B through an object of class B. Following are the properties of a nested class:

- A nested class is declared inside another class.
- The scope of inner class is restricted by the outer class.
- While declaring an object of inner class, the name of the inner class must be preceded by the outer class name and scope resolution operator.

Following program demonstrates the use of nested classes:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string regdno;
		string branch;
		int age;
	public:
```

```
		class Name
		{
			private:
				string fname;
				string mname;
				string lname;
			public:
				string get_name()
				{
					return fname+" "+mname+" "+lname;
				}
				void set_name(string f, string m, string l)
				{
					fname = f;
					mname = m;
					lname = l;
				}
		};
};
int main()
{
	Student::Name n;
	n.set_name("P", "S", "Suryateja");
	cout<<"Name is: "<<n.get_name();
	return 0;
}
```

Output for the above program is as follows:

Name is: P S Suryateja

Object Composition

In real-world programs an object is made up of several parts. For example a car is made up of several parts (objects) like engine, wheel, door etc. This kind of whole part relationship is known as composition which is also known as has-a relationship. Composition allows us to create separate class for each task. Following are the advantages or benefits of composition:

- Each class can be simple and straightforward.
- A class can focus on performing one specific task.
- Classes will be easier to write, debug, understand, and usable by other people.
- Lowers the overall complexity of the whole object.
- The complex class can delegate the control to smaller classes when needed.

Following program demonstrates composition:

```
#include<iostream>
using namespace std;
```

```
class Engine
{
	public:
		int power;
};
class Car
{
	public:
		Engine e;
		string company;
		string color;
		void show_details()
		{
			cout<<"Compnay is: "<<company<<endl;
			cout<<"Color is: "<<color<<endl;
			cout<<"Engine horse power is: "<<e.power;
		}
};
int main()
{
	Car c;
	c.e.power = 500;
	c.company = "hyundai";
	c.color = "black";
	c.show_details();
	return 0;
}
```

Output for the above program is as follows:

```
Compnay is: hyundai
Color is: black
Engine horse power is: 500
```

Friend Functions

A friend function is a non-member function which can access the private and protected members of a class. To declare an external function as a friend of the class, the function prototype preceded with *friend* keyword should be included in that class. Syntax for creating a friend function is as follows:

```
class ClassName
{
	...
	friend return-type function-name(params-list);
	...
};
```

Following are the properties of a friend function:

- Friend function is a normal external function which is given special access to the private and protected members of a class.
- Friend functions cannot be called using the dot operator or -> operator.
- Friend function cannot be considered as a member function.
- A function can be declared as friend in any number of classes.
- As friend functions are non-member functions, we cannot use *this* pointer with them.
- The keyword *friend* is used only in the declaration. Not in the definition.
- Friend function can access the members of a class using an object of that class.

Following program demonstrates using a friend function:

```
#include<iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		void set_name(string);
		friend void show_details(Student);
};
void Student::set_name(string n)
{
	name = n;
}
void show_details(Student s)
{
	cout<<"Name of the student is: "<<s.name;
}
int main()
{
	Student s1;
	s1.set_name("Mahesh");
	show_details(s1);
	return 0;
}
```

Output of the above program is as follows:

Name of the student is: Mahesh

In the above program *show_details()* function was able to access the private data member *name* as it is a friend of *Student* class.

Friend Class

A friend class is a class which can access the private and protected members of another class. If a class B has to be declared as a *friend* of class A, it is done as follows:

```
class A
{
        friend class B;
        ...
        ...
};
```

Following are properties of a friend class:

- A class can be friend of any number of classes.
- When a class A becomes the friend of class B, class A can access all the members of class B. But class B can access only the public members of class A.
- Friendship is not transitive. That means a friend of friend is not a friend unless specified explicitly.

Following program demonstrates a friend class:

```
#include<iostream>
using namespace std;
class A
{
        friend class B;
        private:
                int x;
        public:
                void set_x(int n)
                {
                        x = n;
                }
};
class B
{
        private:
                int y;
        public:
                void set_y(int n)
                {
                        y = n;
                }
                void show(A obj)
                {
                        cout<<"x = "<<obj.x<<endl;
                        cout<<"y = "<<y;
                }
};
```

```
int main()
{
        A a;
        a.set_x(10);
        B b;
        b.set_y(20);
        b.show(a);
        return 0;
}
```

Output for the above program is as follows:

```
x = 10
y = 20
```

In the above program class B has been declared as a *friend* of class A. That is why it is able to access private member *x* of class *A*.

Bit Fields in Classes

A field or member inside a class which allows programmers to save memory are known as bit fields. As we know a integer variable occupies 32 bits (on a 32-bit machine). If we only want to store two values like 0 and 1, only one bit is sufficient. Remaining 31-bits are wasted. In order to reduce such wastage of memory, we can use bit fields. Syntax of a bit field is as follows:

type-specifier declarator: width;

In the above syntax *declarator* is an identifier (bit-field name), *type-specifier* is the data type of the declarator and *width* is the size of bit field.

Following are the properties of bit-fields:

- C++ does not allow arrays of bit fields, pointers to bit fields, and functions returning bit fields.
- The declarator is optional and is used only to name the bit field.
- The address operator (&) cannot be used with bit fields.
- When a value that is out of range is assigned to a bit field, the lower-order bit pattern is preserved.
- Bit fields with a size of 0 must be unnamed.

Following program demonstrates bit field in a class:

```
#include<iostream>
using namespace std;
class Student
{
        private:
```

```
		string name;
		string regdno;
		int age;
		string branch;
		unsigned int status : 1; //size of status field is 1-bit
	public:
		void set_status(int x)
		{
			status = x;
		}
		int get_status()
		{
			return status;
		}
};

int main()
{
	Student s1;
	s1.set_status(1);
	cout<<"Status of student is: "<<s1.get_status();
	return 0;
}
```

Output of the above program is as follows:

Status of student is: 1

In the above program size of *status* field is only 1-bit. It can store either a 0 or 1.

Volatile Objects and Member Functions

An object which can be modified by some unknown forces (like hardware) other than the program itself can be declared as *volatile*. Compiler doesn't apply any optimizations for such volatile objects. Syntax for declaring an volatile object is as follows:

volatile ClassName object-name;

A member function can be declared as *volatile* to make the access to member variables to be volatile. A volatile object can access only volatile functions. Syntax for creating a volatile function is as follows:

return-type function-name(params-list) volatile;

Following program demonstrates both volatile objects and volatile programs:

```
#include<iostream>
using namespace std;
```

```
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		void set_name(string);
		void get_name(Student) volatile;
};
void Student::set_name(string n)
{
	name = n;
}
void Student::get_name(Student s) volatile
{
	cout<<"Name is: "<<s.name;
}
int main()
{
	Student s1;
	s1.set_name("Mahesh");
	volatile Student s2;
	s2.get_name(s1);
	return 0;
}
```

Output of the above program is as follows:

Name is: Mahesh

In the above program volatile function is *get_name()* and volatile object is *s2*. The object *s2* can access only volatile member functions.