Unit 2: Classes and Methods

2.1 Review of Structures

In C programming, a structure can hold only data members. Whereas, a structure in C++ is very much similar to class which can hold data members as well as member functions. The only subtle difference between a structure and a class in C++ is that the members are public by default in structure whereas private in class. Thus, a structure can be used in place of class in C++. However, it is advisable to use structure when a plain old kind of structure with just data members is required.

Example:

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

struct Rectangle{
    float width;
    float height;
};

float GetRectangleArea(struct Rectangle R) {
    float area = R.width*R.height;
    return area;
}

int main() {
    struct Rectangle rect1;
    cout << "Enter the width : ";
    cin >> rect1.width;
    cout << "Enter the height: ";
    cin >> rect1.height;
    cout << "Area of the rectangle = ";
    cout << GetRectangleArea(rect1);
    return 0;
}
```

Difference between Class and Structure in C++

Class	Structure (C++)
Members of a class are private by default when no access specifier is used.	Members of a structure are public by default when no access specifier is used
Inheritance is private by default	Inheritance is public by default
Declared using the **class** keyword	Declared using the **struct** keyword
It is a reference type meaning that the values are stored in heap area.	It is value type meaning that the actual values are stored in stack area.
An instance of a class is called an object	An instance of a structure is called a structure variable
It is used to encapsulate data and related functions together into a single unit.	Though a C++ structure also supports inclusion of member functions, generally it is used for grouping of logically related data.

2.2 Classes and Instances

Class

- A class is an abstract description of the state and behavior of a collection of similar objects.
    - State is represented by a set of data members also called properties.
    - Behavior is implemented as a collection of member functions.
- A class can be thought of as a blue print for creating instances of objects of similar kind.

Instance

- An instance of a class is a particular object which has its own state.
- All instances belonging to a particular class exhibit similar behavior.

A class is analogous to a data type whereas an instance is analogous to a particular variable.

Example:
Here, Employee is a class with:
- Data members (properties): ID and Salary
- Member functions (methods):
  SetData(), ShowData(), and CalculateTax()

Ramesh and Suresh in main( ) are instances (objects) of Employee class.

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

class Employee{
  private:
    int     ID;
    float   Salary;

  public:
    void SetData(int i, float sal) {
        ID     = i;
        Salary = sal;
    }
```

```
    float CalculateTax(float percentage)
    {
        float tax;
        tax = Salary * percentage/100;
        return tax;
    }

    void ShowData()
    {
        cout << endl << "Employee ID : " << ID;
        cout << endl << "Salary      : " << Salary;
    }
};

int main()
{
    Employee Ramesh, Suresh;
    Ramesh.SetData(1001, 40000.0);
    Suresh.SetData(2001, 50000.0);

    Ramesh.ShowData();
    cout << endl << "Tax amount  : ";
    cout << Ramesh.CalculateTax(5.0);

    cout << endl;

    Suresh.ShowData();
    cout << endl << "Tax amount  : ";
    cout << Suresh.CalculateTax(6.0);

    return 0;
}
```

2.3 State, Behavior, Method, and Responsibility

State
The state of an object represents all the information held within it. State, in general, may not be static and may change over time. State of an object is represented by data members (properties) in a class. Each object instantiated from a class holds different state information for the same set of properties.

Behavior
The behavior of an object is reflected by a set of actions it can perform. The complete description of all the behaviors for an object is called protocol. A set of behaviors is implemented as member functions in a class and all Objects belonging to a same class exhibit similar behavior.

Method
Methods are a set of standard operations by which an object fulfills its responsibilities. In other words, a method is a procedure or function associated with a class which is invoked for a particular object by message-passing.

Responsibility
Behavior of a component is described in terms of responsibilities.

2.4 Encapsulation and Data Hiding

Encapsulation
Encapsulation is the process of integrating all the required ingredients or features within a component while exposing only its responsibilities to the external world.

Data hiding
Data hiding is an encapsulation technique that seeks to abstract away the implementation details concerning what data values are maintained for an object exposing only its behavior.

Encapsulation or data hiding is achieved in C++ by using the following access specifiers to control the visibility of properties (data members) and methods (member functions) of a class.

- Private
- Public
- Protected

Access Specifiers

Access specifiers in C++ are the three keywords: private, public, and protected which are used inside a class to control the visibility or accessibility of members of a class from outside the class.

Private: Properties and methods defined under private access specifier are accessible only from the member functions of the same class.

Public: Properties and methods defined under public access specifier are accessible only from the member functions of the same class as well as from outside the class.

Protected: Properties and methods defined under protected access specifier are accessible only from the member functions of the same class or derived/inherited classes.

Functions

2.5 Friend Function

A function that is not a member function of a class but declared as "friend" by the class itself becomes a friend function of the class. A friend function is granted access to private and protected members of the class along with public members of the class.

A friend function can be:

- An independent (standalone) function
- A member function of another class

One of the most important aspects of OOP is encapsulation and data hiding which requires that the functions external to a class should not be allowed to access an object's private or protected features. However, this sometimes leads to inconvenience. Hence, in some scenario, even if the use of friend function breaches the concept of encapsulation and data hiding, programmers tend to use friend functions but to limited purpose.
A friend function is useful in (but not limited to) following scenarios:

- When a function has to access private/protected members of two or more objects of same or different classes.
- To allow a member function of another class to access private/protected members of a given class.

It should be noted that friendship is not inherited from base class to derived class.

Example: To add salaries of two Employee objects using a friend function

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

class Employee{
private:
    string Name;
    float  Salary;
```

```
public:
    Employee(string n = "", float s = 0)
    {
        Name   = n;
        Salary = s;
    }

    void Show()
    {
        cout << endl << "Name   : " << Name;
        cout << endl << "Salary : " << Salary;
    }

    friend float AddSalaries(Employee, Employee);
};


float AddSalaries(Employee a, Employee b)
{
    float  TotalSalary = a.Salary + b.Salary;
    return TotalSalary;
}


int main()
{
    Employee e1("Ramesh", 45000.0);
    Employee e2("Suresh", 54000.0);

    e1.Show();
    e2.Show();

    float total = AddSalaries(e1, e2);
    cout << endl << "Total Salary : " << total;

    return 0;
}
```

2.6 Inline function

Inline function is a mechanism used by the C++ compiler to speed up the execution time of a function call composed of very few lines of code.

Long sections of repeated code are generally better when implemented as normal functions. However, when the function body is just a few lines of code, execution time is hampered due to the following overhead that needs to be followed during a function call.

- Save the current state of CPU registers
- Push arguments onto the stack from the calling function
- Jump to the function code
- Pop arguments back out of the stack from the called function
- Push the results on to the stack from the called function
- Return to the calling function
- Pop the results back from the calling function
- Restore the CPU registers

When the function body is very small, the execution speed can be improved by repeating the code at such places rather than making function calls, getting rid of such overhead. But, this would result into losing the benefits of program organization and clarity that is provided by the use of functions.

The **solution** to this problem is **using inline function**. When making a function inline, we would use the function like a normal function but when the code is compiled, actual code from the function body is inserted at the place from where the function is called. In addition to the elimination of overhead involved during a function call, this results into well organized and easy to read code among other advantages given by functions.

This can be done simply by prefixing the keyword inline before the function definition. It should be noted that the function must be defined before the function call thus making the function declaration (prototyping) unnecessary. The inline keyword is just a request to the compiler. Sometimes, the compiler will ignore the request and compile the inline function as a normal function. This may happen when:

- the function is recursive
- the function contains a loop (for, while, etc)
- the function contains switch or goto statements

- the function contains static variables

Functions defined inside a class are readily considered inline by the compiler. Functions defined outside a class can be made inline by prefixing the function definition by inline keyword.

Disadvantages of Inline function:

- Use of too many inline functions would make the size of binary executable file larger.
- Memory thrashing may occur leading to the degradation in the performance
- Increases the compile-time overhead because changes made to an inline function needs to be replicated every place from where the function is called.
- Variables created inside the inline function would consume additional resources.
- Reduction in instruction cache hit rate.

Example:

```
#include <iostream>
using namespace std;
inline float PoundsToKG(float pounds)
{
	return 0.4536*pounds;
}

int main()
{
	cout << endl << PoundsToKG(2.0);
	cout << endl << PoundsToKG(3.5);
	return 0;
}
```

Here, the function call `PoundsToKG(2.0)` is converted to `0.4536*2.0` and `PoundsToKG(2.0)` is converted to `0.4536*3.5` during the program compilation.

2.7 Static Data and Static Function

Static data member

- Static data members are class members that are declared using the static keyword.
- Only one copy of a static member is created for the entire class which is shared by all the objects of that class, irrespective of the number of objects created.
- It should be declared in global scope and initialized before any object of the class is created, even before the starting of main function.
- A static data member is visible only within the class, but its lifetime is throughout the entire program.

Static member function

- When a static data member has to be accessed through a member function, it is more natural to access the static data member via the class name rather than through an object of that class because static data members belong to the whole class rather than a single object.
- To make this possible, we declare such a member function as static.
- Static member function of a class can be called using the function name prefixed by class name and the scope resolution operator (::).
- Static member functions can be called this way even when there exists no object of the respective class.
- It should be noted that non-static member functions can access members. Whereas, static member functions are not allowed to access only static members.

Example: Static data member and function to track the count of objects.

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

class CAR{
  private:
    string  Model;
    float   Mileage;
    static int Count;
```

```
  public:
    CAR(string theModel, float theMileage)   {
        Model    = theModel;
        Mileage  = theMileage;
        Count++;
    }

    void showData()  {
        cout << endl << "Model   : " << Model;
        cout << endl << "Mileage : " << Mileage;
    }

    static void showCount()  {
        Cout << endl << "Total No. of CARs: " << Count;
    }
};

// initialization of static data member
// (outside the class in global scope)
// {0} is the initializer which is optional
// because static members default to zero
int CAR::Count{0};

int main()
{
    CAR::showCount();

    CAR c1("Mazda RX4", 21.0);
    c1.showData();
    c1.showCount();
    CAR::showCount();
    cout<<endl;

    CAR c2("Honda Civic", 30.4);
    c2.showData();
    c2.showCount();
    CAR::showCount();
    cout<<endl;
    return 0;
}
```

2.8 Reference Variable

- A reference variable is an alias, that is, another name for an already existing variable.
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- References are often confused with pointers but the distinction between references and pointers are:
  - ❖ Unlike pointers which can be initialized at any time, a reference must be initialized when it is created.
  - ❖ Unlike pointers which can be pointed to another object/variable at any time, a reference cannot be changed to refer to another object/variable once it is initialized.
  - ❖ Unlike pointers which can have NULL values, a NULL reference is not possible. i.e., A reference must be connected to a valid object/variable.
  - ❖ A pointer variable is declared using the * operator before the variable name. Whereas, a reference variable is declared using the & operator before the variable.
  - ❖ The data item pointed to by a pointer variable is accessed using the indirection operator (*). Whereas, the data item referenced to by a reference variable does not require any extra operator and is directly accessed through the reference variable as if it was a normal variable.

Example: Reference variable

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

int main()
{
    int  x = 55;
    int &r = x;
    // r is declared as a reference to x
    // any changes to r directly affects x

    cout << endl << endl << "Initially:";
    cout << endl << "x = "<< x;
    cout << endl << "r = "<< r;
```

```
    x = 66;
    cout << endl << endl << "After assigning 66 to x:";
    cout << endl << "x = " << x;
    cout << endl << "r = " << r;

    r = 77;
    cout << endl <<endl << "After assigning 77 to r:";
    cout << endl << "x = "<< x;
    cout << endl << "r = "<< r;

    return 0;
}
```

Pass by reference

- Like pointers, we can also pass the local variables of one function to another function by reference.
- Advantages / applications of passing by reference:
  - ❖ Modify the passed parameters in a function: If a function receives a reference to a variable, it can modify the value of the variable. For example, in the following example variables are swapped using references.
  - ❖ Avoiding a copy of large structures: Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use pass-by-references to avoid this.
- The syntax of passing a variable by reference is much simpler and safer than passing by pointer. However, it may not be as powerful/versatile as passing by pointer.

Example: Pass by reference to demonstrate swapping of two int variables

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

void SwapByRef(int &, int &);

int main()
{
    int a=12, b=34;
```

```
    cout << endl << "Before call to SwapByRef:";
    cout << endl << "a = " <<a << "\t" << "b = " << b;

    SwapByRef(a, b);

    cout << endl << "After call to SwapByRef:";
    cout << endl << "a = " << a << "\t" << "b = " << b;

    return 0;
}

void SwapByRef(int & x, int & y)
{
    int z;
    z = x;
    x = y;
    y = z;
}
```

2.9 Default Argument

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function does not provide a value for the argument.
- In case a value is passed, the default value is overridden by the passed parameter.
- Once the default value is used for an argument in the function declaration, all subsequent arguments following it must also have a default value.
- Default arguments can be applied to standalone functions as well as member functions of a class.
- Advantages:
  - ❖ Useful when we want to increase the capabilities of an existing function by adding other arguments (with default values), without the need to change existing function calls.
  - ❖ Helps in reducing the program size.
  - ❖ Maintains consistency of a program.

Example 1: Default argument in function declaration.

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

float CalculateVAT(float, float = 13);

int main()
{
    cout << endl << "VAT for Rs. 1000 @ 13% = ";
    cout << CalculateVAT(1000.0);

    cout << endl << "VAT for Rs. 2000 @ 15% = ";
    cout << CalculateVAT(2000.0, 15.0);

    return 0;
}

float CalculateVAT(float amount, float percentage) {
    float vatAmount;
    vatAmount = amount * percentage / 100.0;
    return vatAmount;
}
```

Example 2: Default argument directly in function definition when there is no function declaration.

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

float CalculateVAT(float amount, float percentage=13) {
    float vatAmount;
    vatAmount = amount * percentage / 100.0;
    return vatAmount;
}

int main()
{
    cout << endl << "VAT for Rs. 1000 @ 13% = ";
    cout << CalculateVAT(1000.0);

    cout << endl << "VAT for Rs. 2000 @ 15% = ";
    cout << CalculateVAT(2000.0, 15.0);

    return0;
}
```