Agenda: Classes & Objects

- Introduction to classes & objects
- Creation & destruction of objects
- Data Members
- Member Functions
- this Pointer
- Constructor &Destructor
- Static class member
- Friend class and functions
- Namespace

Working with Classes & Objects

- A Class is a collection of data member and member functions together.
- Class prevents the direct access of the data members.

Syntax:

```
class ClassName
{
   public:
     variable declarations;    // Data member
     function declarations;   // Member functions
}
```

- The members can be declared as public or private.
- If the members are declared as **private** that means the members can be accessed only from within the class and they cannot be accessed from outside of the class
- If the members are declared as **public** then the members can be accessed from both within and outside the class.
- Class is divided into two parts:
  - **Class member declaration** specifies type and scope of data member which are being present in class.
  - **Class function definition** specifies how the member functions are being implemented.
- An object is a real entity of the class and is the one which actually is allocated memory at runtime.
- Instantiating an object implies that all the Data Members of the Object are allocated memory as one unit.

Data Members: Data members are set of variables of different data types of a class which defines the functionality of a class.

Member Functions:

- Member functions are operators and functions that are declared as members of a class.
- The definition of a member function is within the scope of its enclosing class.

Whenever a member function is added to the class two things should be taken into considerations:

1. Prototype of the function should be declared inside the class.
2. Implementation of the function should be placed outside of the class.
3. Function name should be qualified by class name using Scope resolution operator.

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

class Account
{
private:
	int Id;
	string Name;
	double Balance;
public:
	void GetData(); //Print the values of data members
	void SetData(); //To read the values for data members
};

void Account::GetData()
{
	cout << "---------Account Details---------" << endl;
	cout << "Id: " << Id << endl;
	cout << "Name: " << Name << endl;
	cout << "Balance: " << Balance << endl;
	cout << "---------------------------------" << endl;
}
void Account::SetData()
{
	cout << "Enter Id: ";
	cin >> Id;
	cout << "Enter Name: ";
```

```
        cin >> Name;
        cout << "Enter Balance: ";
        cin >> Balance;
}

int main()
{
        Account a1;
        Account a2;
        a1.SetData();
        a1.GetData();
        a2.SetData();
        a2.GetData();
}
```

Creating and Destroying Objects

- Object can be created either in Global memory or Stack memory orHeap memory.
- Object when created as a global variable is allocated memory in Global memory area and would remain alive through out the lifetime of an application. Only when the application terminates the object is destroyed.
- Object when created as a local variable in scope of a method is allocated memory on stack of that method.
- Object when created on heap is allocates the memory dynamically and its called dynamic memory allocation.

Syntax for instantiating the object:

```
<ClassName><ObjectName>; //Is Instantiaged either on Global or Stack memory
```

Dynamic memory allocation technique is mainly used when it is not known in advance (at the compile time) that how much amount of memory space is being required.

C++ provides two new operators 'new'&'delete' to allocate and de-allocate memory respectively.

- 'new' operator is responsible for creating an object.
- Object is destroyed dynamically by using the keyword “delete” on heap memory. “delete” operator requires the address of the object to destroy the object.

```
<ClassName> *<pointerToObject> = new <ClassName>; //On Heap.
delete pointerToObject
```

```
#include <iostream>
```

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

class Account
{
private:
	int Id;
	string Name;
	double Balance;
public:
	void GetData(); //Print the values of data members
	void SetData(); //To read the values for data members
	Account()
	{
		cout << "Object is created" << endl;
	}
	~Account()
	{
		cout << "Object is destroyed" << endl;
	}
};

void Account::GetData()
{
	cout << "---------Account Details---------" << endl;
	cout << "Id: " << Id << endl;
	cout << "Name: " << Name << endl;
	cout << "Balance: " << Balance << endl;
	cout << "---------------------------------" << endl;
}
void Account::SetData()
{
	cout << "Enter Id: ";
	cin >> Id;
	cout << "Enter Name: ";
	cin >> Name;
```

```
        cout << "Enter Balance: ";
        cin >> Balance;
}

Account a1; //Global Object
void Foo()
{
        Account *pAccount; //Pointer but not initialized
        pAccount = new Account(); //Object created on heap

        //Syntax to access members using Pointer to an object
        (*pAccount).GetData();
        // is same as
        pAccount->GetData();
        // (*). is same as ->

        delete pAccount; //Object is destroyed and memory deallocated from heap
}

int main()
{
        cout << "Enter Main" << endl;
        Account a2; //Local object
        Foo();
        cout << "Exit Main" << endl;
}
```


Member Functions and this Pointer

Important points while writing methods in a class:

- A method requires parameter's only for that data which it doesn't have direct access to but is needed for implementation of logic. Thus while writing a method in a class if that datais already the data member in the same class **do not** set to it any data as **parameter or return type.**
- Inside a method "**this"** is **pointer to the current object** on which the method is invoked.
- The object's address is available from within the member function as the 'this' pointer.
- It is legal, though unnecessary, to explicitly use this when referring to members of the class.

- If parameter/local variable and data member of a class have same name, the local variable takes the precedence over data member and if data member has to be accessed then "**this**" should be used to resolve name ambiguity.
- Whenever any of the business rules of an object or the integrity of the object is violated through a property or a method, it should respond by **throwing a runtime exception**.

Example:

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

class Account
{
private:
        int Id;
        string Name;
        double Balance;
public:
        void SetData(); //To read the values for data members
        void GetData(); //Print the values of data members
        void Deposit(double amount); //To deposit the amount
        void Withdraw(double amount); //To withdraw the amount
        double GetBalance();
};

void Account::GetData()
{
        cout << "---------Account Details---------" << endl;
        cout << "Id: " << Id << endl;
        cout << "Name: " << Name << endl;
        cout << "Balance: " << Balance << endl;
        cout << "---------------------------------" << endl;
}
void Account::SetData()
{
        cout << "Enter Id: ";
```

```
		cin >> Id;
		cout << "Enter Name: ";
		cin >> Name;
		cout << "Enter Balance: ";
		cin >> Balance;
}
void Account::Deposit(double amount)
{
		this->Balance += amount;
}
void Account::Withdraw(double amount)
{
		if (this->Balance - amount < 500)
			throw "Insufficient Funds";
		this->Balance -= amount;
}
double Account::GetBalance()
{
		return Balance;
}
int ShowMenu()
{
		int op;
		cout << "1. Create Object" << endl;
		cout << "2. Set Data" << endl;
		cout << "3. Get Data" << endl;
		cout << "4. Deposit" << endl;
		cout << "5. Withdraw" << endl;
		cout << "6. Destroy object" << endl;
		cout << "7. Exit" << endl;
		cout << "----------------------------" << endl;
		cout << "Enter your choice: ";
		cin >> op;
		return op;
}
```

```
int main()
{
	int op;
	double amount;
	Account *pAccount = NULL; //assigning null value to the pointer object
	do {
		op = ShowMenu();
		switch(op)
		{
			case 1:
				if (pAccount == NULL)
					pAccount = new Account();
				else
					cout << "Object already created" << endl;
				break;
			case 2:
				pAccount->SetData();
				break;
			case 3:
				pAccount->GetData();
				break;
			case 4:
				cout << "Enter the amount to Deposit: ";
				cin >> amount;
				pAccount->Deposit(amount);
				break;
			case 5:
				cout << "Enter the amount to Withdraw: ";
				cin >> amount;
				pAccount->Withdraw(amount);
				break;
			case 6:
				if (pAccount != NULL)
				{
					delete pAccount;
					pAccount = NULL;
```

```
                                        }
                                        else
                                                cout << "Object is not existing" << endl;
                                        break;
                        }
        } while (op != 7);
        return 0;
}
```

Constructor & Destructor

Constructors are special type functions which are called when an instance of the class is being created.

Ideally they are used for initializing the Data Member of the class.

Characteristics of Constructor:

- The Constructor has same name as that of the class.
- It does not have return type (not even void).
- Constructors are invoked automatically whenever the object of the class is being created.
- Constructors cannot be virtual.
- Constructor can be overloaded.
- If a class **doesn't have any form of constructor**, a **public default constructor** is automatically added to it by the language compiler.
- If required a Constructor can be declared as **private**.

Types of Constructors:

a) Default / Parameter-less Constructor:

A Constructor that accepts no arguments is called as the default constructor.

Syntax: ClassName()

{ }

b) **Parameterized Constructor:**

The parameterized constructor is a constructor which takes at least one argument and this parameter is used to initialize the data member of the class. This constructor is generally used to initialize the data members with different values for each newly created object.

Syntax: ClassName(datatype<parameter name>)

{

//Initializing the data member with the given parameter

}

c) **Copy Constructor:**

- When a constructor takes the argument which is the type of same class itself then this kind of constructor is called copy constructor.
- Copy Constructor is used to create a new object by **duplicating** the state of an existing object and this is done by copying the data members of existing object in new object data members.
- The parameter to the copy constructor will be the **reference of an object** of the same class.

Syntax: ClassName **(ClassName&<instance objectName>)**

{ }

Example:

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

class Account
{
private:
        int Id;
        string Name;
        double Balance;
public:
        Account(); //Default constructor
        Account(int id,string name,double balance); //Parameterized constructor
        Account(Account &a); //copy constructor
        void SetData(); //To read the values for data members
        void GetData(); //Print the values of data members
        void Deposit(double amount);
        void Withdraw(double amount);
        double GetBalance();
};

Account::Account()
{
        cout << "Parameter less default contructor used" << endl;
```

```
}
Account::Account(int id,string name,double balance)
{
	cout << "Parameterized contructor used" << endl;
	Id = id;
	Name = name;
	Balance = balance;
}
Account::Account(Account &a)
{
	cout << "Copy contructor used" << endl;
	this->Id = a.Id;
	this->Name = a.Name;
	this->Balance = a.Balance;
}
void Account::GetData()
{
	cout << "---------Account Details---------" << endl;
	cout << "Id: " << Id << endl;
	cout << "Name: " << Name << endl;
	cout << "Balance: " << Balance << endl;
	cout << "---------------------------------" << endl;
}
void Account::SetData()
{
	cout << "Enter Id: ";
	cin >> Id;
	cout << "Enter Name: ";
	cin >> Name;
	cout << "Enter Balance: ";
	cin >> Balance;
}
void Account::Deposit(double amount)
{
	this->Balance += amount;
}
```

```
void Account::Withdraw(double amount)
{
        if (this->Balance - amount < 500)
                throw "Insufficient Funds";
        this->Balance -= amount;
}
double Account::GetBalance()
{
        return Balance;
}

int main()
{
        Account a0(1,"A1",10000);
        a0.GetData();
        //while calling the copy constructor, object of the class is passed as a parameter
        Account a1(a0);
        a1.GetData();
}
```

Destructors:

- Like constructor, Destructor is a special type of function which is called when the object is getting destroyed.
- Whenever an object is created, the object consumes some memory which has to be destroyed after the lifetime of the object. In these situations, the destructors are used to destroy dynamically allocated memory.

Syntax: ~ClassName()
{}

Notes:

- Like constructors, destructors neither take any argument nor return any value.
- Destructor has no arguments.
- The name of the destructor is same as className but it should be preceded with a symbol tilde (~).
- Destructors are invoked implicitly by the compiler upon the exit of the program.

Example:

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

class CA
```

```
{
public:
    int A;
    CA()  //Constructor
    {
        cout << "In CA" << endl;
    }
    ~CA()  //Destructor
    {
        cout << "CA object is destroyed" << endl;
    }
};

class CB
{
public:
    CA *pA;
    CB()
    {
        cout << "In CB" << endl;
        pA = new CA();
    }
    ~CB()
    {
        cout << "CB object is destroyed" << endl;
        delete pA; //Destroys object of class CA pointed by pA
    }
};

int main()
{
    CB b;
}
```

Static Members

Static Data Member

- For every new instance of a class, all the instance members are allocated memory as one unit.
- Static data members of a class are allocated memory **only once** irrespective of the number of objects created.
- All static data is initialized to zero when the first object is created.
- These members are also called **class members**and are shared by all the objects of the class.
- The life time of these members is the entire program and hence they can be used as Global Variables also.
- We cannot initialize them in class declaration, it should be done outside the class using scope resolution operator.

  int CA::S1 = 10

Example:

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

class CA
{
public:
        int M1=0;
        static int S1;
        CA()
        {
                M1++;
                S1++;
        }
        ~CA()
        {
                S1--;
        }
        void Display()
        {
                cout << "Instance Member: " << M1 << endl;
                cout << "Static Member: " << S1 << endl;
        }
};
int CA::S1=0;
int main()
```

```
{
        CA a1,a2,a3;
        a3.Display();
        cout << "Static : " << CA::S1 << endl;
        cout << "Static : " << a1.S1 << endl;
        cout << "Static : " << a2.S1 << endl;
        cout << "Static : " << a3.S1 << endl;
        CA *pA = new CA();
        cout << "Before Delete - Static : " << CA::S1 << endl;
        delete pA;
        cout << "After Delete - Static : " << CA::S1 << endl;
        return 0;
}
```

Static Member Functions

- Static member functions are the functions which are called using the class without creating the object.
    - ClassName :: FunctionName();
- Static member function can access only static data member which are present in the class.
- Static member functions can not access non static data members of the same class.
- **this** pointer cannot be used in Static Member Functions.
- A static member function cannot access non-static members of the class without explicitly qualifying with an object.
- Though it can also be accessed by the object of the class along with the (.) membership operator, the compiler will automatically replace the object with class name.

Example:

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

class Account
{
private:
        int Id;
        string Name;
        double Balance;
```

```
public:
        void SetData(); //To read the values for data members
        void GetData(); //Print the values of data members
        static Account& GetAccountWithLessBalance(Account&a1,Account& a2);
        //Non Static version of above static function.
        Account& GetAccountWithLessBalance(Account& other);
};

Account& Account::GetAccountWithLessBalance(Account& a1,Account& a2)
{
        //this->Id = 10;
        //Id = 10;
        if (a1.Balance < a2.Balance)
                return a1;
        else
                return a2;
}
Account& Account::GetAccountWithLessBalance(Account& other)
{
        if (this->Balance < other.Balance)
                return *this;
        else
                return other;
}
void Account::GetData()
{
        cout << "---------Account Details---------" << endl;
        cout << "Id: " << Id << endl;
        cout << "Name: " << Name << endl;
        cout << "Balance: " << Balance << endl;
        cout << "---------------------------------" << endl;
}
void Account::SetData()
{
        cout << "Enter Id: ";
        cin >> Id;
```

```
        cout << "Enter Name: ";
        cin >> Name;
        cout << "Enter Balance: ";
        cin >> Balance;
}

int main()
{
        Account a0;
        Account a1;
        a0.SetData();
        a1.SetData();
        Account &a3 = Account::GetAccountWithLessBalance(a0,a1);
        a3.GetData();
        Account &a4 = a0.GetAccountWithLessBalance(a1);
        a4.GetData();
}
```

Singleton Class

Singleton class is a class which can be instantiated only once.

The Singleton Design pattern is used, where only one instance of an object is needed throughout the lifetime of an application.

Steps to write a Singleton class

1. Create a Private constructor so that we cannot create an object of the class in any function outside of the class.
2. Declare a private static member variable of same class.
3. Write a public member function which when called for the first time creates an object of the class and returns the reference to the same even for subsequent call to that method.

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

class Singleton
{
private:
        Singleton()
        {
```

```
                cout << "Object is created" <<  endl;
        }
        //Constrctor is declared as private so that object cannot be created outside the class
        static Singleton *pObject;
public:
        int InstanceMember;
        //Method is declared as static so that it can be called without using object
        static Singleton* GetInstance()
        {
                if (pObject == NULL)
                        pObject = new Singleton();
                return pObject;
        }
        static void DestroyInstance()
        {
                if (pObject != NULL)
                {
                        delete pObject;
                        pObject = NULL;
                }
        }
};
Singleton* Singleton::pObject = NULL;

int main()
{
        //Singleton s1; //Not allowed because constructor is private
        Singleton *pA1,*pA2;
        pA1 = Singleton::GetInstance();
        pA1->InstanceMember = 100;
        pA2 = Singleton::GetInstance();
        cout << pA2->InstanceMember << endl;
        Singleton::DestroyInstance();
        return 0;
}
```

Friend Function and Friend class

Friend functions:

- From the earlier topics, we know that only public member functions has right to access the private variables of the same class. In addition to it, in C++ we have a special type of function which can access the private variables of the class even though it is not a member function of the class. This type of function is called **Friend functions**.
- In other words, a friend function is a non- member function that can access the private data members of the class in which it is a friend.
- **Syntax:** **friend**<returnType><name of the function>()
  {
  }

Characteristics of Friend Function

- Friend function cannot access the data members directly. It can be accessed only by using object name and (.) membership operator with each data member.
- It can be declared either in private or public section of the class.
- Friend function can be invoked just like a normal function without the help of the object.
- Friend function mostly takes as parameter object of class type in which it is a friend.
- A member function of a class can be declared as a friend for other class. Whenever the member function of one class is to be made the friend for other class we have to use the scope resolution operator.

Friend Class

- Like friend functions, we can also declare a whole class to be a friend of another class.
- In this scenario, all the member functions of the friend class will have the access to all the private data members of the class for which it is a friend.

Note: By default friendship is not mutually exclusive i.e. if Class A is friend of Class B but Class B may not be a friend of Class A.

```
//Global function ShowA is friend of CA
//Global function ShowAB is friend of both CA and CB
//Member function of CB ShowAFromB is friend of CA
//Class CB is friend of CA and hence in all member functions of CB we can access private data of CA.
#include<iostream>
using namespace std;

class CA;
class CB
```

```
{
private:
    int MemB;
public:
    void SetData()
    {
        cout << "Enter value for CB.MemB: ";
        cin >> MemB;
    }
    friend void ShowAB(CA &a, CB &b);
    void ShowAFromB(CA a);
};

class CA
{
private:
    int MemA;
public:
    void SetData()
    {
        cout << "Enter value for CA.MemA: ";
        cin >> MemA;
    }
    friend void ShowA(CA &a);
    friend void ShowAB(CA &a, CB &b);
    //friend void CB::ShowAFromB(CA );
    friend class CB;
};
void ShowA(CA &a)
{
    cout << a.MemA;
}
void CB::ShowAFromB(CA a)
{
    cout << "In CB, CA.MemA: " << a.MemA << endl;
}
```

```
void ShowAB(CA &a, CB &b)
{
    cout << "Value of CA.MemA: " << a.MemA << endl;
    cout << "Value of CB.MemB: " << b.MemB << endl;
}

int main()
{
    CA a;
    a.SetData();
    ShowA(a);
    CB b;
    b.SetData();
    ShowAB(a,b);
    b.ShowAFromB(a);
    return 0;
}
```

Inline function

- Functions written inside the class along with declaration are called as Inline functions.
- We can use the keyword "inline" when the function definition is outside the class.
- Call to Inline functions in calling function is replaced with actual instructions of the called function.
- Inline functions are faster because you don't need to push and pop things on/off the stack like parameters and the return address.
- Inline functions are best for small functions that are called often and if used with large functions it can increase the size of output file i.e executable file generated by compiler.
- It's up to the compiler to decide to actually use the function as inline or not.

Syntax:

inline returnType className :: function()
{ }

Example:

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

```
class Student
{
        int roll;
        float marks;
public:
        void get();
        void show();
};
// function defined as inline
inline void Student :: get()
{
        cout<<"Enter roll & marks: "<<endl;
        cin>>roll>>marks;
}
inline void Student :: show()
{
        cout<<"Roll is: "<<roll<<endl;
        cout<<"Marks are: "<<marks;
}

int main()
{
        Student s1;
        s1.get();
        s1.show();
        return 0;
}#include<iostream>
using namespace std;

class Student
{
        int roll;
        float marks;
public:
        void get(); //to get the data from the student
        void show(); //to print the data
```

```
};
// function defined as inline
inline void Student :: get()
{
        cout<<"Enter roll & marks: "<<endl;
        cin>>roll>>marks;
}
inline void Student :: show()
{
        cout<<"Roll is: "<<roll<<endl;
        cout<<"Marks are: "<<marks;
}

int main()
{
        Student s1;
        s1.get();
        s1.show();
        return 0;
}
```

Namespace

- C++ allows variables to be declared as local or global inside classes or functions.
- In addition to this C++ provides a new keyword called namespace that are responsible for providing a scope to the global identifiers.
- The most common example of namespace is the C++ standard library i.e. the reason namespace std is used in the program.
- **Syntax for defining namespace**

```
namespace namespaceName
{
        Declaration of var, class, function;
}
```

- The namespace name should be the valid identifier. The only difference between the namespace and class is namespace does not end with a semicolon.
- The main advantage of using namespace is namespace minimizes the name collision. In order to minimize it, it takes the help of the namespace signature.

- Namespace can also be defined inside another namespace,this is called as nesting namespace.
- If a variable is to be used frequently in a program then rather than accessing it by scope resolution operator, it is better to use 2 different methods:

  Using directive

  Syntax: using namespace namespaceName

Example:

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

namespace A
{
        class Number
        {
                int n;
                public:
                Number(int a)
                {
                        n = a;
                }
                void show()
                {
                        cout<<n;
                }
        };
}

int main()
{
        using namespace A;  // including namespace in main
        Number n1(30);
        // or you can also write like this
        //A:: number n1(30);
        n1.show();
        return 0;
}
```