Class Template & Virtual Base Class & friend class

Class Template

Class Templates like function templates,

Class templates are useful when a class defines something that is independent of the data type

```
template <class   T>
class class-name {
    .
    .
    .
}
```

```
template<class T>
class A
{
   public:
   T num1 = 11;
   T num2 = 20;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
   }
};

int main()
{
   A<int> d;
   d.add();
   return 0;
}
```

```
template<class T1, class T2>
   class A
   {
      T1 a;
      T2 b;
      public:
     A(T1 x,T2 y)
     {
        a = x;
        b = y;
     }
        void display()
       {
            std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
    };
```

```
int main()
   {
       A<int,float> d(5,6.5);
       d.display();
       return 0;
   }
```

```
template<class T, int size>
class A
{
   public:
   T arr[size];
   void insert()
   {   int i =1;
      for (int j=0;j<size;j++)
      {
         arr[j] = i;     i++;
      }
   }
 void display()
   {
      for(int i=0;i<size;i++)  {
         << arr[i] << " ";   }
   }
};
```

```
int main()
{
   A<int,10> t1;
   t1.insert();
   t1.display();
   return 0;
}
```

```
template <class T> class info {
public:
        info(T   A)
        {
                cout << "\n"<< "A = " << A << " size of data in bytes:" << sizeof(A);
        }
};

int main()
{
                info<char> p('x');      //passing character value by creating an objects
                info<int> q(22);        // passing integer value by creating an object
                info<float> r(2.25);    //passing float value by creating an object
      return 0;
}
```

```
template <typename T>
class Array {
private:
        T* ptr;
        int size;
public:
        Array(T arr[], int s);
        void print();
};
template <typename T> Array<T>::Array(T arr[], int s)
{
        ptr = new T[s];
        size = s;
        for (int i = 0; i < size; i++)
                ptr[i] = arr[i];
}
```

```
template <typename T> void
Array<T>::print()
{
        for (int i = 0; i < size; i++)
                cout << " " << *(ptr + i);
        cout << endl;
}

int main()
{
        int arr[5] = { 1, 2, 3, 4, 5 };
        Array<int> a(arr, 5);
        a.print();
        return 0;
}
```

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

template <class T, class U>
class A {
        T x;
        U y;

public:
        A() { cout << "Constructor Called" << endl; }
};

int main()
{
        A<char, char> a;
        A<int, double> b;
        return 0;
}
```

default value for template arguments

```
template <class T, class U = char>
class A {
public:
        T x;
        U y;
        A() { cout << "Constructor Called" << endl; }
};

int main()
{
        A<char> a; // This will call A<char, char>
        return 0;
}
```

Write a c++ class template “calculator” , the class has two data members and a method to display the results. Other methods might be add(), subtract(), multiply () and Divide().

```
template <class T>
class Calculator {
  private:
   T num1, num2;
  public:
   Calculator(T n1, T n2) {
      num1 = n1;
      num2 = n2;
   }
   void displayResult() {
      cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
      cout << num1 << " + " << num2 << " = " << add() << endl;
      cout << num1 << " - " << num2 << " = " << subtract() << endl;
      cout << num1 << " * " << num2 << " = " << multiply() << endl;
      cout << num1 << " / " << num2 << " = " << divide() << endl;
   }
   T add() { return num1 + num2; }
   T subtract() { return num1 - num2; }
   T multiply() { return num1 * num2; }
   T divide() { return num1 / num2; }
};
```

```
int main() {
   Calculator<int> intCalc(2, 1);
   Calculator<float> floatCalc(2.4, 1.2);

   cout << "Int results:" << endl;
   intCalc.displayResult();

   cout << endl<< "Float results:" << endl;
   floatCalc.displayResult();

   return 0;
}
```

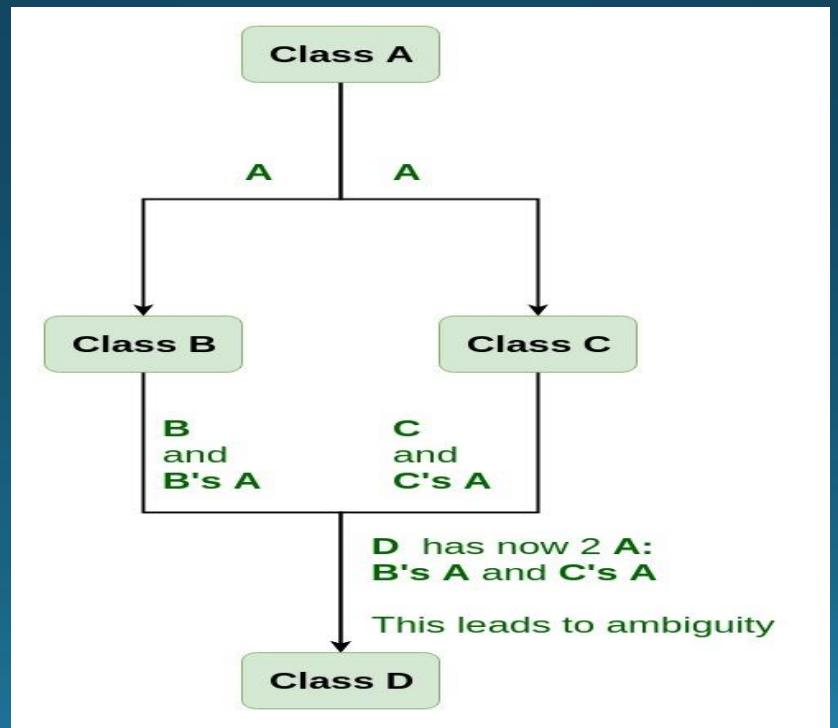
```
template <class T, class U, class V = char>
class ClassTemplate {
  private:
   T var1;
   U var2;
   V var3;
  public:
   ClassTemplate(T v1, U v2, V v3) :
   var1(v1), var2(v2), var3(v3) {}  // constructor
   void printVar() {
      cout << "var1 = " << var1 << endl;
      cout << "var2 = " << var2 << endl;
      cout << "var3 = " << var3 << endl;
   }
};
}
```

```
int main() {
ClassTemplate<int, double> obj1(7, 7.7, 'c');
   cout << "obj1 values: " << endl;
   obj1.printVar();


   // create object with int, double and bool types
   ClassTemplate<double, char, bool> obj2(8.8, 'a',
false);
   cout << "\nobj2 values: " << endl;
   obj2.printVar();


   return 0;
```

Virtual Base Class

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes: Consider the situation where we have one class A . This class A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.



```
class A {
public:
        void show()
        {       cout << "Hello form A \n";  }
};

class B : public A {  };

class C : public A {  };

class D : public B, public C {
};

int main()
{
        D object;
        object.show();
}
```

- Virtual can be written before or after the public.
- Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class.
- Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances.
- When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members.
- A single copy of its data members is shared by all the base classes that use virtual base

```
class A {
public:
        int a;
        A()     {    a = 10;   }
};
class B : public virtual A {  };
class C : public virtual A {   };
class D : public B, public C {   };

int main()
{
        D object;
        cout << "a = " << object.a << endl;

        return 0;
}
```

```
class A {
public:
        void show()
        {       cout << "Hello from A \n";     }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main()
{
        D object;
        object.show();
}
```

Friend class

- A **friend class** can access private and protected members of other class in which it is declared as friend.
- It is sometimes useful to allow a particular class to access private members of other class.
- A friend class can access both private and protected members of the class in which it has been declared as friend.

```
class A
{
        int x;
        public:
        A()
        {
                x=10;
        }
        friend class B; //friend class
};
class B
{
        public:
                void display(A &t)
                {
                cout<<endl<<"The value of x="<<t.x;
                }
};
```

```
main()
{
        A a;
        B b;
        b.display(a);
        return 0;
}
```

```
class ClassB;
class ClassA {
   private:
      int numA;
      friend class ClassB;
   public:
   ClassA() : numA(12) {}
};
class ClassB {
   private:
      int numB;
   public:
ClassB() : numB(1) {}
int add() {
      ClassA objectA;
      return objectA.numA + numB;
   }
};
```

```
int main() {
   ClassB objectB;
   cout << "Sum: " << objectB.add();
   return 0;
}
```

```
class MyClass
{
        friend class SecondClass;
        public:
                MyClass() : Secret(0){}
                void printMember()
                {
                        cout << Secret << endl;
                }
        private:
                int Secret;
};
class SecondClass
{
        public:
                void change( MyClass&
yourclass, int x )
                {
                        yourclass.Secret = x;
                }
};
```

```
int main()
{
        MyClass my_class;
        SecondClass sec_class;
        my_class.printMember();
        sec_class.change( my_class, 5 );
        my_class.printMember();
        return 0;
}
```

RTTI(Run-Time Type Information)

- In C++, RTTI (Run-time type information) is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.
- Runtime Casts

The runtime cast, which checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:

- Upcasting: When a pointer or a reference of a derived class object is treated as a base class pointer.
- Downcasting: When a base class pointer or reference is converted to a derived class pointer.

- Using ‘dynamic_cast‘: In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class.
- On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.

```
class B {
        virtual void fun() {   }
};
class D : public B {
};
int main()
{
        B* b = new D; // Base class pointer
        D* d = dynamic_cast<D*>(b); // Derived class pointer
        if (d != NULL)
                cout << "works";
        else
                cout << "cannot cast B* to D*";
        getchar();
        return 0;
}
```