Template Class: Generic

a way to write my class (one class) that works with any datatype

sum(int,int);
sum(float,float);

```
//standalone Function to accept different types
template<typename T>//compiler type name T
T sum (T n1,T n2)
{
    return n1+n2;
}
cout<<sum<complex>(c1,c2);//overload +operator << overloading to handle
printing of complex using cout
cout<<sum<int>(5,9);
cout<<sum<float>(8.5,6.2);
```

Template (generic) Class to handle different datatypes at runtime

Template Class Stack

```
template<class T>
class Stack
{
   int top;
   int capacity;
   T* data;
   static inline int counter=0;
public:
   Stack(int c):top(0),capacity(c)
   {   counter++;
      data=new T[capacity];
   }
   ~Stack(){counter--;delete[] data;}
   static int showCounter(){return counter;}
   //utilities
   bool isEmpty(){return top==0;}
   bool isFull(){return top==capacity;}
   void push(T n)
   {
      if(isFull())
         cout<<"stack is full"<<endl;
      else
      {
         data[top]=n;
         top++;
      }
   }
   //signature only inside class
   T pop();
};
```

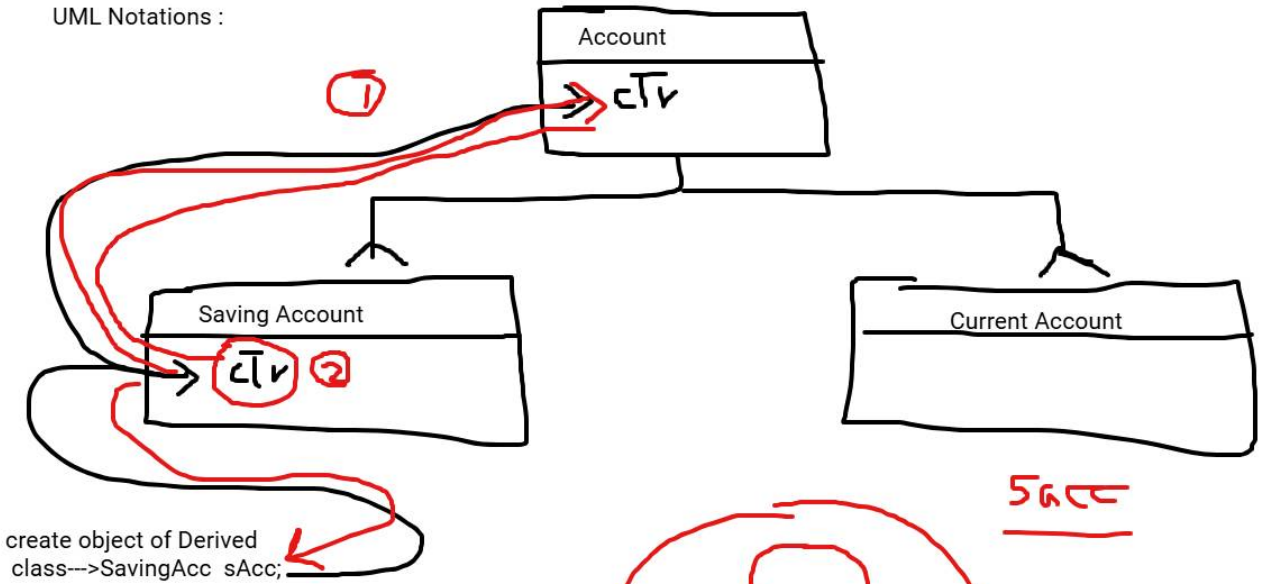
```
//memeber implemented outside class musd add Stack<T>:: before fn
name
template<class T>
T Stack<T>::pop()
{
      T retValue=0;
      if(isEmpty())
         cout<<"stack is empty"<<endl;
      else
      {
         top--;
         retValue=data[top];
      }
      return retValue;
}
```

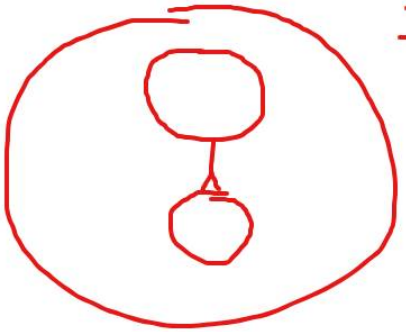
```
int main()
{
   cout << "Hello Template Stack!" << endl;
   Stack<int> s1(10);
   Stack<int> s2(9);
   //Stack<Complex> cs;
   s1.push(8);
   s1.push(50);
   Stack<char> schars(50);
   schars.push('A');
   //cout<<Stack::showCounter();//error mut specify typename
   cout<<s1.pop()<<"# of int Stacks is
:"<<Stack<int>::showCounter()<<endl;
   cout<<schars.pop()<<"# of char Stacks is
:"<<Stack<char>::showCounter()<<endl;

   return 0;
}
```

Why Inheritance:
-to extend classes by adding new Features
Ex: Acoount ----> Saving Acc or Current Acc specialization technique

-Gathering common Features of classes (class Base) Generalization Technique

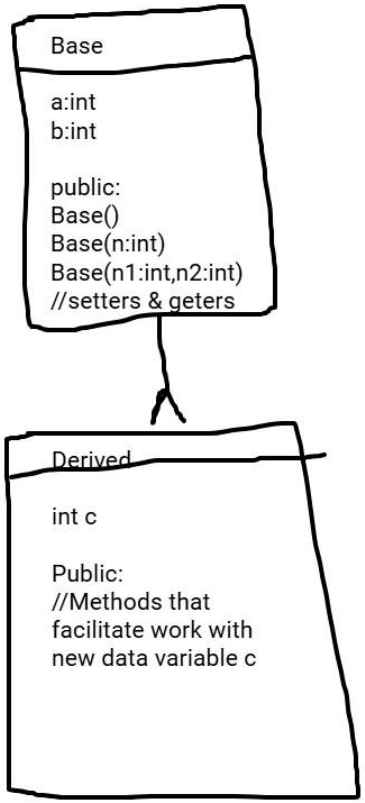
UML Notations :





Concept:

constructor Chaining -->inheritance


Base
a:int
b:int
public:
Base()
Base(n:int)
Base(n1:int,n2:int)
//setters & geters
Derived
int c
Public:
//Methods that
facilitate work with
new data variable c

```
class Base
{
private:
   //can't be acceced through derived class
protected:
   int a,b;
public :
   Base()=default;
   Base(int n):a(n),b(n){}
   Base(int n1,int n2):a(n1),b(n2){}
   void setA(int n){a=n;}
   void setB(int n){b=n;}
   int  getA(){return a;}
   int  getB(){return b;}
   int calcSum(){return a+b;}
};
//child class
class Derived:public Base//public here refere inheritance mode (type)
//protected mode / private mode
{
   int c;
public:
   Derived(){c=0;}//be default call default ctr of mybase class
   Derived(int n):Base(n),c(n)//order to call parametrized ctr of Base
   {}
   Derived(int n1,int n2,int n3):Base(n1,n2),c(n3){}
   void setC(int n){c=n;}
   int  getC(){return c;}
   int calcSum()//(POV : polymorphism (overriding)
   //fn with the same name as function in my base class
   //{return getA()+getB()+c;} //first way if a,b private
   {
      return Base::calcSum()+c;//second way if a,b private
   }
   //{return a+b+c;}
   //a,b private//a,b protected members in base can be accessed from
derived
};
```

```
int main()
{
   cout << "Hello Inheritance Part1!" << endl;
   Derived d;
   d.setA(5);
   d.setB(6);
   d.setC(1);
   Derived d1(5);
   Derived d2(5,6,4);
   Base b(4,8);
   cout<<"sum of d is:"<<d.calcSum()<<endl;
   cout<<"sum of d1 is:"<<d1.calcSum()<<endl;
   cout<<"sum of d2 is:"<<d2.calcSum()<<endl;
   cout<<"sum of b is:"<<b.calcSum()<<endl;

   //d1.a;
   //d1.c;
   return 0;
}
```

Note: usually compiler looks at methods at object that is calling it first then looks in base class ,it goes from bottom to up.

what we done here by creating calcsum in derived --> actually not called overriding it called method hiding ...