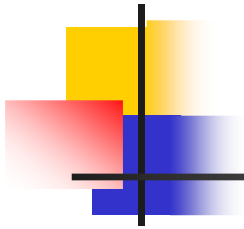
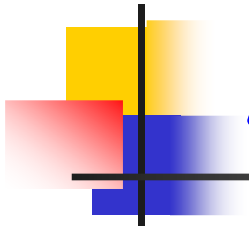


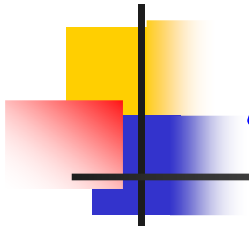
Lecture-5

- Abstract classes
- this keyword
- Setting member values
- const member functions
- Passing arguments by reference
- Function overriding and function overloading
- Standard Template Libraries (STL)
- Templates



Abstract classes

- Consider an object of Account.
- It makes sense to have
  - A specific type (e.g., checking) of account
  - Not just a generic account object.
- A user should be able to create
  - Specific object types.
  - NOT generic objects.
- An abstract class is the generic class.



Abstract classes … contd.

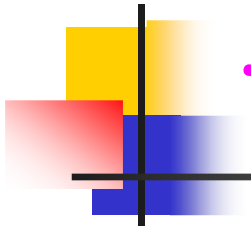
- Properties of abstract classes.
  - Defines a generic base class
  - Class definition has attributes and methods
  - Other classes are derived from it.
  - Derived classes implement the methods defined in abstract class.
  - Can NOT instantiate objects of base class.
  - Can instantiate only objects of derived classes.

How do we create abstract classes?

- Set ANY virtual function to 0.

```
class Account
{
    virtual double computeInterest () = 0;
}
class CheckingAccount : public Account
{
    double computeInterest ( ) { … }
}

Account x; // Will NOT work.
CheckingAccount y; // Will work.
```



this keyword

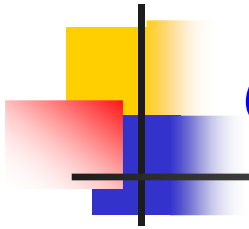
- this refers to the addrss of the current object
- E.g.

```
class Account
{
   private:
      int balance;
   public setBalance (int amount)
    {
       this->balance = amount;
    }
};
```

Initializing member values

```
class Account
  {
     private:
       int balance;
     public:
       Account ( ) : balance (0)
         { }
       Account (int amount) :
   balance (amount) { }
  };
```

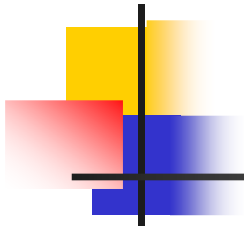
```
class checkingAccount : public
    Account
{
  checkingAccount (int amount)
   :  Account (amount) { }
}
```



const member functions

```
class myClass
{
   int a;
   ...
   void f1( ) const
    { a = 3; } // Not allowed
};
```

- const functions can't change any attributes of myClass.
- f1 can't change a in the above example



Making arguments const

```
class myClass
{
   int a;
   …
   void f1(const int& i )
    {  i = 3; } // not allowed
```

- Cannot change the value of const arguments

Passing args. to a function … by value

- Compiler creates its own copy.
- Any changes made inside the function are not reflected after the function.

```
class myClass
{
        void f1(int i ) // i is passed by value
         { i = 3; }
};
int x = 5;
myClass obj;
obj.f1( x );
cout << "value of x: " << x << endl; // x is still 5
```

Passing args. to a function … by reference

- Compiler takes the original object.
- Any changes made inside the function are reflected after the function.

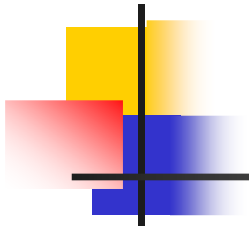
```
class myClass
{
        void f1(int& i ) // i is passed by reference.
         { i = 3; }
};
int x = 5;
myClass obj;
obj.f1( x );
cout << "value of x: " << x << endl; // x is 3
```

Function overriding

- Derived class can redefine (override) any function defined in the base class.
- E.g. computeInterest below is overridden by checkingAccount class.

```
class Account
   {
      protected:
        double balance;
      public:
        void computeInterest( )
         {
            balance = balance +
                0.01 * balance;
         }
   };
```

```
class checkingAccount : public
   Account
 {
    public:
        void computeInterest ( )
          {
             balance = balance +
                0.03 * balance;
          }
 };
```



Function overloading

- Possible to have multiple member functions of the same name with different parameters

⇒ Function overloading

```
class myClass
{
   // f1 – overloaded function
   void f1 (int i);
   void f1 (int i, int j);
   void f2 (int i, double j);
}
```

Templates – motivation

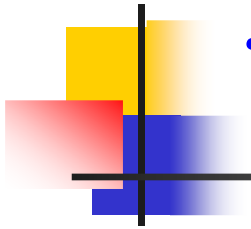
- Consider a function to sort integers

```
int findMaxInt (int num1, int num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- Consider a function to sort floats

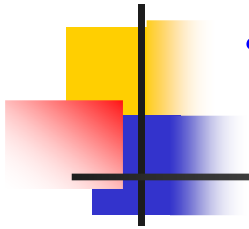
```
float findMaxFloat (float num1, float num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- Same code, but two functions are needed
  - One for each data type



Templates motivation … cont.

- Problem here
  - One max. function is needed for each data type.
  - But the code finding max itself is the same.
- Templates are used to solve this issue.
- Templates
  - Define functions with generic types
  - Define generic classes.
  - Same code can be used with any data type.
  - No need for code repetition.



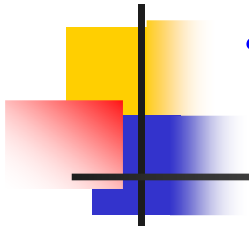
Template functions

- Define functions with generic types

```
template <class anyType>
        function definition
```

- Example

```
template <class anyType>
    anyType GetMax (anyType a, anyType b)
    {
            if (a > b) return a;
            return b;
    }
```



Template classes

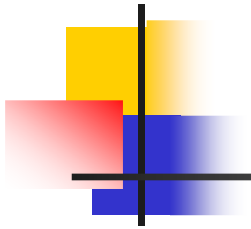
- Define any class
- Use template definition to define any generic data type.
- Rest of the code remains the same.

```
template <class anyType>
class someClass
{
    anyType a;
    someClass (…) // constructor
    ~someClass( ) // destructor
//    member data & methods // can use anyType
};
```

Template class example

```
template <class anyType>
class myClass
{
   private:
     anyType value1, value2;

    public:
       myClass (anyType i, anyType j) : value1(i), value2(j)
   {  }
       ~myClass() { }
       anyType findMaxValue ()
       {
            if (value1  > value2) return (value1);
            return (value2);
       }
};
```



Template class example … contd.

```
main()
{
   myClass <int> intObject (2, 3);
   int maxInt = intObject.findMaxValue();
   cout << "maxInt: " << maxInt << endl;

   myClass <float> floatObject (4.3, 3.2);
   float maxFloat = floatObject.findMaxValue();
   cout << "maxFloat: " << maxFloat << endl;
}
```