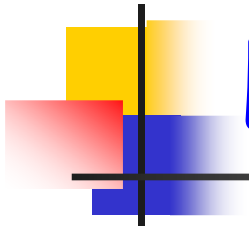
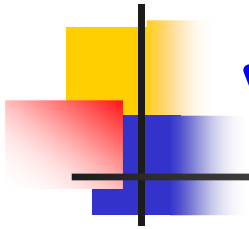


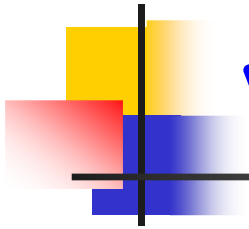
Lecture-5

- Polymorphism
  - Virtual functions
- Abstract classes
- Setting member values
- Passing arguments by reference
- Templates



virtual functions

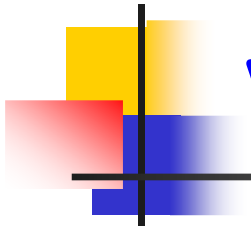
- Function "double computeInterest( )" is defined in both base and child classes.
  - Supposed to return different values
    - virtual double Account::computeInterest ( )
      { return 0; }
    - double CheckingAccount::computeInterest ( )
      { return 10.0; }
    - double IRA_Account::computeInterest ( )
      { return 100.0; }



virtual functions … contd.

```
main()
{
   Account *x = new CheckingAccount();
   x→computeInterest( );
   // Will this return 0 or 10.0?
}
```

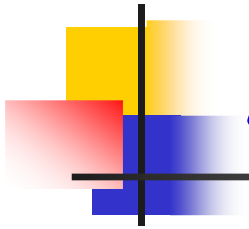
- This will return
  - 0, if the function is NOT virtual
  - 10.0, if the function is defined virtual



Why are virtual functions needed?

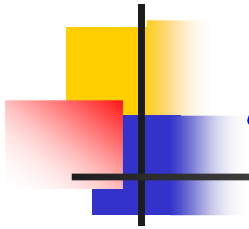
- Mainly to enforce class specific functional implementation.
- Should not call base class function from a child object.
- An account object may take different "forms" at different times
  - Checking account, IRA account, etc.
  - computeInterest( ) should compute derived class specific function.

⇒ Polymorphism



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.
  - Pure virtual function – value of function = 0
  - NO BODY for function

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

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

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
```

Templates – motivation

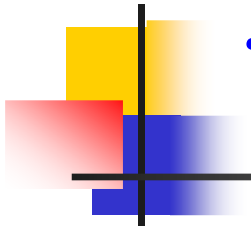
- Consider a function to sort integers

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

- Consider a function to sort floats

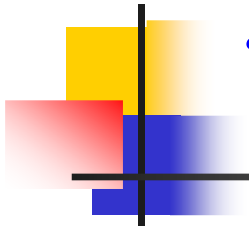
```
float findMaxFloat (float num1, float num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- 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.



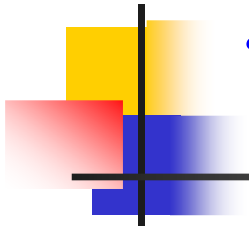
Template functions

- 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

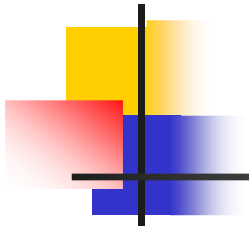
- Define any class
- Use template definition to define any generic data type.
- Rest of the code remains the same.

```
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;
}
```