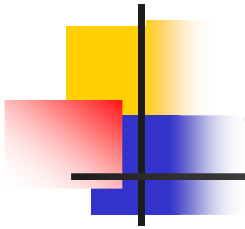
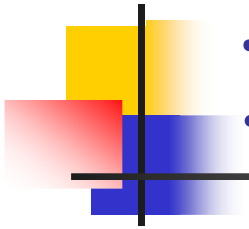


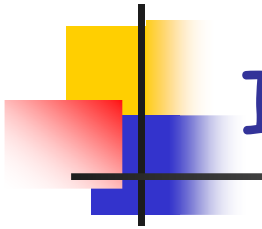
Lecture-3

- Inheritance
- public, private and protected members
- const member functions
- friend functions
- friend classes
- Virtual functions
- Polymorphism
- Abstract classes



Inheritance

- Let's take the account example again
- There are many types of accounts
  - Checking, saving, money market, IRA, etc.
- All accounts may have
  - Some common members.
    - Account number, user SSN, etc.
  - Some class specific members.
- Method implementation may be
  - Same in different classes
  - Different in different classes.



Inheritance – base class & derived class

- **Base class**

```
class account
{
   int user_SSN;
   int accountNumber;
   void deposit (int amount);
   void withdraw (int amount);
   double computeInterest ( );
};
```

- **Derived class or child class**

```
class checkingAccount : public account // checkingAccount is
{                                      // derived from account
   int lastCheckCleared;               // not present in account
   void showAllChecksCleared( );// not present in account
   double computeInterest( ); // defined in both classes
};
```

Inheritance – base class and derived classes

- Base Class

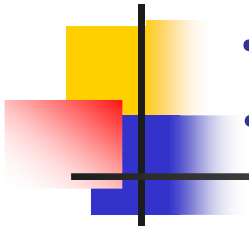
```
class account
{
private:
    int user_SSN;
    int accountNumber;
public:
    void deposit (int amount)
    void withdraw (int amount);
    double computeInterest( );
};
```

- Derived (or child) class-1

```
class checkingAccount : public
    account
{
    int lastCheckCleared;
    void showChecksCleared ( );
    double computeInterest ( )
};
```

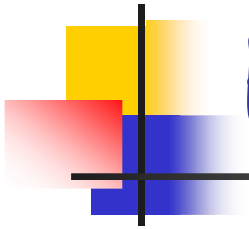
- Derived (or child) class-2

```
class IRA_account : public account
{
    void buyFund (int fund_ID);
    void sellFund (int fund_ID);
    double computeInterest ( );
};
```



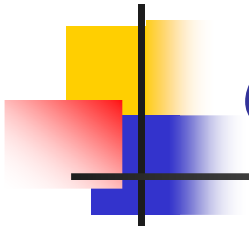
Inheritance – continued.

- Important points to note:
  - Derived classes have access to members of base classes in this example.
  - Derived classes can have their own members.
    - E.g. lastCheckCleared, showAllChecksCleared( ), buyFund( ), sellFund( ), etc.
    - Members of one derived class are not accessible to another



Examples

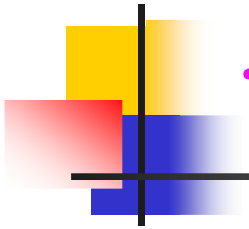
- Valid usage in an external function
  - account acct(123456, abc);
  - checkingAccount
  - acct.deposit (700);
  - acct.withdraw (300);
  - checkingAccount.deposit (1000);
  - checkingAccount
- Invalid usage in an external function
  - acct.user_SSN = 1234; // Can't access user_SSN
  - acct.accountNumber = 567;



const member functions

```
class myClass
{
   int a;
   …
   void function_1( ) const;
};
```

- const functions can't change any attributes of myClass.
- function_1 can't change a in the above example



friend functions

- What if a function genuinely needs to have access to private data?
  - E.g. showAccountInfo (Account acct)
- Need to give access ONLY to that function, not others.
- Use friend function definition
- friend functions of a class have access to private members of the class.

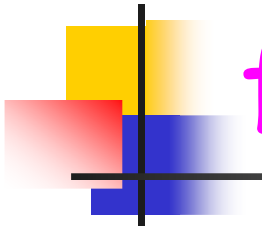
Example – friend function

```
class account
{
private:
    int user_SSN;
    int accountNumber;
public:
    void deposit (int amount)
    void withdraw (int amount);
    double computeInterest ( );
friend showAccountInfo
    (class Account)
};
```

```
void showAccountInfo
    (Account acct)
{
    cout << user_SSN << endl;
    cout << accountNumber
         << endl;
}
```

This is valid.

Friend function can access private members.

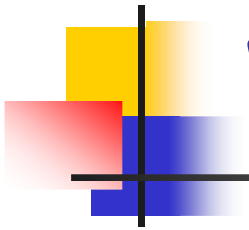


friend class

- Concept of friend can be extended to a class from a function.
- A class gives access to its private members to its friend classes.

```
class account                    class bank
{                                {
   …                                    …
   friend class bank             }
}
```

Members of bank have access to private members of account



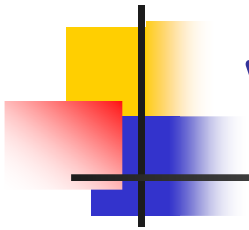
Virtual functions

- 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?
}
```

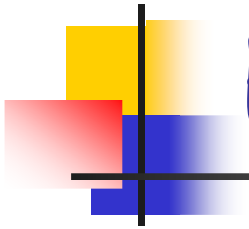
- This will return
  - 0, if the function is NOT virtual
  - 10.0, if the function is defined virtual



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.

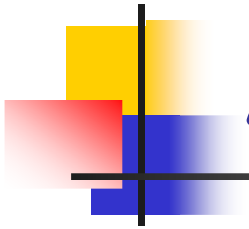
⇒ Polymorphism



Example

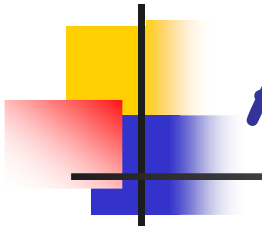
```
Account  *x;
for (each object obj)
{
   x = &obj;      // polymorphism
   i->computeInterest( );
}
```

Here computeInterest( ) of derived classes should be called.



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.
```