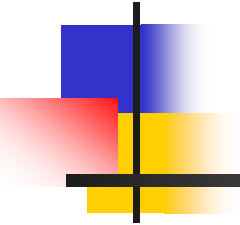


COMS W3101 Programming Language: C++ (Fall 2014)



Ramana Isukapalli
ramana@cs.columbia.edu



Lecture-3

- Constructor and destructor review
- Data and Member functions review
- Data encapsulation
 - public, private and protected members
- friend functions and friend classes
- Inheritance
- Function overloading and function overriding



Class methods

Syntax:

```
<ret_type> class::functionName(args)
{
    // code
}
```

Method code can be present in class definition

- Outside the class definition
- In a separate file

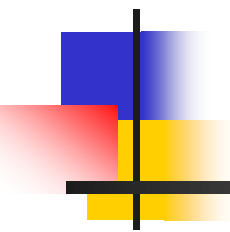
Example

```
void account::withdrawMoney (int amount)
{
    // code
}
```



A simple "account" example

```
class account
{
    private:
        int user_SSN;           // data
        int accountNumber;     // data
    public:
        void withdrawMoney (int amount); // method
        void depositMoney (int amount);  // method:
        void computeInterest( );         // method
};
account x; // x is an object of class "account"
```



Constructor and Destructor



Constructor and destructor ... contd.

Constructor

- o A function with the same name as the class
- o Called when an object is created
- o A class can have more than one constructor

Destructor

- o Called when an object is cleaned up (goes out of scope)
- o One class can have only one destructor

Examples

account x; // constructor code is called

account *y = new account; // constructor code is called

delete (y); // destructor code is called



Constructor and destructor

Constructor code

```
account::account( )  
{ user_ssn = -1; accountNumber = -1; }  
  
account::account( ) : user_ssn (-1),  
                    accountNumber(-1) { }  
  
account::account (int ssn, int acctNum)  
{  
    user_ssn = ssn;  
    accountNumber = acctNum;  
}
```

Destructor code

```
account::~~account( )  
{ // Any memory/resource cleanup, etc. }
```



Initializing member values

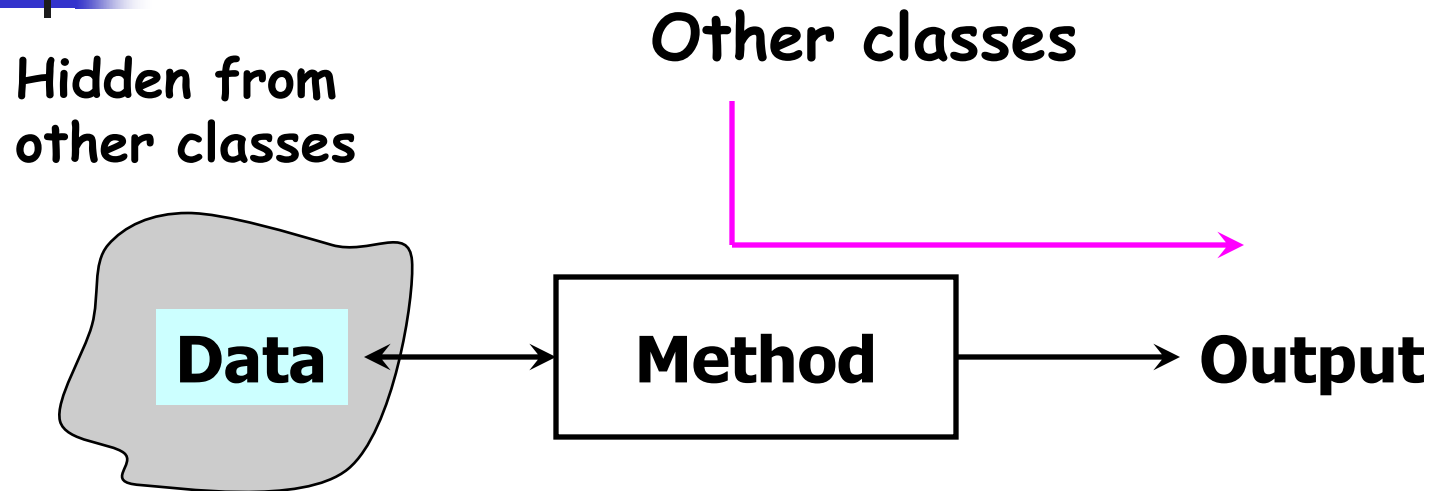
```
class Account
{
    private:
        int balance;
    public:
        Account ( ) : balance (0)
        { }
        Account (int amount) :
        balance (amount) { }
};
```

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




Data encapsulation

Data encapsulation ... contd.



- Methods act on data to provide output.
- User needs to see only method, not data.
- User should not be affected by
 - Implementation details of methods.
 - Changes in implementation of methods.



Data encapsulation

- Provide access restrictions to member data and functions
 - From other classes and functions.
- Implemented y using access modifiers
 - **public**, **private** and **protected**
- Other classes, functions need to know **what** methods are implemented
 - **Not how** they are implemented



Account example ... contd.

- **class** has both “**data**” and “**methods**”.
- Attributes and methods are “**members**” of a class
- An instance of a class is an **object**.
- A class should typically correspond to some meaningful entity.
- A class uses methods to interact with other classes/functions.
- **private** members accessible only to the class (and friends)
- **public** members are accessible to every class and functions



Back to data encapsulation

- How can data be hidden?
 - Only class should have access to data
 - Class methods use data
- Define every class member to be one of
 - **public** - accessible to every class, function
 - **private** - accessible only to class and **friends**
 - **protected** - accessible only to class, friends and children

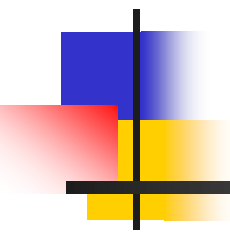


Data encapsulation in account example

- In an object of account
 - `user_ssn` and `accountNumber` are declared **private**
 - Accessible only to account objects (and **friends**)
 - Methods are **public**
 - Anyone can access them.

■ Example

```
void function1 ( ) // function, not defined in Account
{
    account x;
    x.user_ssn = 123; // Will NOT work
    x.computeInterest ( ); // Will work
}
```



friend functions and friend classes



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);
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
{
    ...
    friend class bank
}
```

```
class bank
{
    ...
}
```

Members of bank have access to private members of account



Examples

- Valid usage in an external function
 - `account acct(123456, 5672);`
 - `checkingAccount ca;`
 - `acct.deposit (700);`
 - `acct.withdraw (300);`
 - `ca.deposit (1000);`
 - `ca.showAllChecksCleared()`
- Invalid usage in derived class
 - `ca.user_SSN = 1234; // Can't access user_SSN`
 - `ca.accountNumber = 567;`



Inheritance



Inheritance - base class & derived class

- **Base class**

```
class account
{
    int user_SSN;
    int accountNumber;
public:
    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:
    account ( ) { }
    account (int ssn, acctNum);
    ~account( ) { }
    void deposit (int amount)
    void withdraw (int amount);
    double computeInterest( );
};
```

■ Derived (or child) class-1

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

■ Derived (or child) class-2

```
class IRA_account : public account
{
public:
    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, 5672);`
 - `checkingAccount ca;`
 - `acct.deposit (700);`
 - `acct.withdraw (300);`
 - `ca.deposit (1000);`
 - `ca.showAllChecksCleared()`
- Invalid usage in derived class
 - `ca.user_SSN = 1234; // Can't access user_SSN`
 - `ca.accountNumber = 567;`



Function overloading and function overriding



Function overloading

- Functions with the same name but with
 - Different number of arguments or
 - Different types of arguments

E.g. `int add (int a, int b, int c) { return (a+b+c); }`
`int add (int a, int b) { return (a + b); }`
`double add (double a, double b) { return (a + b); }`

- Here "add" is an **overloaded** function



Function overriding

- Functions defined in parent class and re-implemented by the child class.

E.g. class Bird

```
{  
    int canFly( ) { return (1); }  
}
```

class Penguin

```
{  
    int canFly ( ) { return (0); }  
}
```

Here, "canFly" is an **overridden** by the child class, Penguin