Inheritance

• Often, Classes Are Made From Existing Classes
• Base Class
  – starting point for defining a set of classes
  – most general attributes and methods defined here
• Derived Class
  – extends the definition of a base class in some way

protected Qualifier

Virtual Functions
Introduction to Inheritance

• Object-Oriented Programming
  – Powerful Programming Technique
  – Provides Abstraction Dimension Called Inheritance
• General Form Of Class Is Defined First
  – Specialized Versions Then Inherit Properties Of The General Class
  – And Add/Modify Functionality As Needed
"Is-A" vs. "Has-A" Relationships

- Inheritance
  - Considered An "Is-A" Class Relationship
    - An HourlyEmployee "Is A Kind Of" Employee
    - A Convertible "Is A Kind Of" Automobile
- Aggregation
  - Considered A "Has-A" Class Relationship
    - One Class Has Data Of Another Class Type
      - Automobile "Has A" Steering Wheel, Engine, Tailpipe

Inheritance Terminology

- Simulates Family Relationships
- Parent Class
  - Refers To Base Class
- Child Class
  - Refers To Derived Class
- Ancestor Class
  - Parent And Their Parents…
- Descendant Class
  - Children And Their Children…

Inheritance Basics

- New Class Inherited From Another Class
- Base Class
  - "General" Class From Which Others Derive
- Derived Class
  - New Class
  - Automatically Has Its Base Class’s:
    - Member variables
    - Methods
  - Additional Methods And Members Can Be Added
Derived Classes

- Consider the concept of "Employees"
  - All have names and social security numbers
- Composed of:
  - Salaried employees
  - Hourly employees
- Each is a "subset" of employees
  - Another might be those paid a fixed wage per week or month...

Introducing the Employee Class

- Many members of the "Employee" class apply to all types of employees
  - Data elements
    - SSN
    - Name
    - Pay
  - And their associated accessors and mutators

Textbook Example

```cpp
// This is the header file hourlyemployee.h
// This is the interface for the class HourlyEmployee.

// hourlyemployee.h

#include <string>
#include "employee.h"

using std::string;

namespace SwtichEmployees
{
```
Textbook Example

```cpp
class HourlyEmployee : public Employee
{
public:
  HourlyEmployee();
  HourlyEmployee(string theName, string theNumber, double wageRate, double theHours);
  void setRate(double newWageRate);
  double getRate() const;
  void setHours(double newHours);
  double getHours() const;
private:
  double wageRate;
  double hours;
};
#endif //HOURLYEMPLOYEE_H
```

HourlyEmployee Class Additions

- An “Additive Model” – Don’t Repeat What’s Already There!
- HourlyEmployee adds:
  - Constructors
  - wageRate, hours member variables
  - setRate(), getRate(), setHours(), getHours()

Methods

Derived Class Constructor Example

- HourlyEmployee Constructor:
  HourlyEmployee(string theName, string theNumber, double theWageRate, double theHours)
  Employee(theName, theNumber),
  wageRate(theWageRate), hours(theHours)
  }
  //Deliberately empty
}

- The Portion After : Is An "initialization section"
  - Invokes The Parent Class Constructor Which In This Case Is Employee
Another HourlyEmployee Constructor

• Another HourlyEmployee Constructor:
  HourlyEmployee::HourlyEmployee()
  : Employee(), wageRate(0), hours(0)
  {
      //Deliberately empty
  }
• Default, No Argument Parent Class Constructor Is Called
• You Should ALWAYS Invoke A Base Constructor
  – Lacking Any Call, C++ Calls The Base Default Constructor So It Better Have One Available!

Pitfall: Private Access Modifier

• Derived Class "Inherits" Private Member Variables And Private Methods
  – But Cannot Directly Access Them
  – private Really Is Private!

protected: Access Modifier

• It Is private To Driver Code And Other Classes
• It Is public To You And All Your Derived Classes
• Allows You To Plan Ahead For Inheritance Purposes…
Doors Example

- Let’s Design A Set Of Doors Classes For An Adventure Game
- What Are The Common Characteristics Of All Doors?

Door Object

- Knows:
  - Its Status (Open or Shut)
- Can Do:
  - Initialize Itself As Shut
  - Open Itself, If Possible
  - Close Itself
  - Tell Whether Or Not It Is Open

Class Door

- A Generic Base Class
class Door {
  public:
    Door();
  bool isOpen() const;
  void open();
  void close();
protected:
  bool isShut;
}
**protected Qualifier**

- **protected** is a compromise between private and public
  - **protected** is public to base classes
  - **protected** is public to friends of the base classes
  - **protected** is public to derived classes
  - **protected** is public to friends of derived classes
  - **protected** is private to other classes

---

**Time For Our First Demo!**

- Door.cpp

(See Handout For Example 1)

---

**Summarizing Our First Demo!**

- Inheritance is an important part of good OO design and implementation
Derived Lockable Door From Door

- A Derived Class
  
  ```cpp
  class LockableDoor : public Door {
    public:
      LockableDoor();
      bool isLocked() const;
      void open();
      void lock();  void unlock();
      protected:
        bool theLock;
  }
  ```

Lockable Door Object

- **Member Data**
  - isShut
  - theLock

- **Member Functions**
  - isLocked()
  - open()
  - lock()
  - unlock()
  - isOpen()
  - close()

Comparing Door And LockableDoor

- **Door**
  - isShut

- **LockableDoor**
  - isShut
  - theLock
Inheritance Behavior

• By Default, All Member Functions And Member Attributes Are Inherited Down To Derived Classes
  – happens without mentioning these functions and attributes in the derived class definition
• Any Member Function Or Member Attribute Can Be Redefined In The Derived Class
  – hides access to the base class versions

Example Redefinition

• LockableDoor’s open() Function

```cpp
void LockableDoor::open()
{
  if (!isLocked()) {
    Door::open();
  }
}
```

Using The Doors

```cpp
Door hallDoor;
LockableDoor frontDoor;

hallDoor.open();
frontDoor.lock();
frontDoor.open();
if (!frontDoor.isOpen())
  frontDoor.unlock();
```
Time For Our Next Demo!

• LockableDoor.cpp

(See Handout For Example 2)

Summarizing Our Next Demo!

• protected Members Are Accessible To Derived Classes
• Using The Scope Operator ::, You Can Specify Just Which Version Of A Function To Call

Derived Classes Can Be A Base Class
CombinationLockDoor

- Combinations Are Single Integers

```cpp
class CombinationLockDoor : public LockableDoor {
public:
    CombinationLockDoor( int combo = 0);
    void unlock( int combo );

protected:
    int thecombination;
}
```

CombinationLockDoor Object

- Member Data
  - isShut
  - thelock
  - thecombination

- Member Functions
  - isLocked()
  - open()
  - lock()
  - unlock( int )
  - isOpen()
  - close()

Time For Our Next Demo!

- CombinationLockDoor.cpp

(See Handout For Example 3)
Summarizing Our Next Demo!

- protected Members Are Accessible To Derived Classes
- Redefined Members Hide Access To The Parent Class Versions

Relationships Between Object

- IS-A
  - one class “is a kind of” another class
  - base class is a general class
  - derived class is a specialization of the general concept
- PART-OF
  - one class “is a part of” another class
  - often used to represent compound objects

Relationships Between Objects

```
COMPUTER IS-A MAC

COMPUTER PART-OF MONITOR
```
Relationships Between Objects

class Computer {
}
class Mac: public Computer {
}

IS-A

PART-OF

PasswordLockDoor

• Doors That Require A Password To Open Or Close
  – “open sesame”   “sounds like fun”
• Let’s Represent The Password As string
• The Password Is “Part-Of” A PasswordLockDoor

PasswordLockDoor Object

class PasswordLockDoor : public LockableDoor {
  public:
    PasswordLockDoor(const char c[] = "");
    void unlock(const char c[] = "");
  protected:
    string thepassword;
}
Object Diagram For Doors

```
DOOR
  LOCKABLEDOOR
  COMBINATIONLOCKDOOR
  PASSWORDLOCKDOOR
    string
```

PasswordLockDoor Object

- **Member Data**
  - isShut
  - thelock
  - thepassword

- **Member Functions**
  - isLocked()
  - open()
  - lock()
  - unlock(char[])
  - isOpen()
  - close()

Pointers To Base Classes

- **Pointers Can Be Made To Point To Derived Classes**

```cpp
typedef Door* DoorPtr;
DoorPtr p = new Door();
p->open();  // calls Door::open
...
p = new LockableDoor();
p->open();  // which open??
```
virtual Functions

- Late Binding Allows The Selection Of Which Implementation Of A Member Function To Execute To Be Determined At Run-Time
- C++ Performs Late Binding Via virtual Functions

virtual Functions

- A Generic Base Class
  ```cpp
class Door {
public:
  Door();
  bool isOpen() const;
  virtual void open();
  void close();
protected:
  bool isShut;
};
```

Time For Our Next Demo!

- VirtualFunctions.cpp

(See Handout For Example 4)
Summarizing Our Next Demo!

• **virtual** Functions Allow For Late Binding To Runtime Objects To Determine Which Version Of A Function Actually Gets Called

---

Summary

• Inheritance
• *protected* Qualifier
• Virtual Functions