Inheritance
Introduction

- Timepiece
  - Analog timepiece
    - Analog clock
      - Analog alarm clock
    - Sundial
    - Watch
      - Pocket watch
      - Analog wristwatch
  - Digital timepiece
    - Digital wristwatch
    - Digital clock
      - Digital alarm clock
Inheritance Basics

- New class inherited from another class
- Base class
  - ‘General’ class from which others derive
- Derived class
  - New class
  - Automatically has base class’s:
    - Member variables
    - Member functions
  - Can then add additional member functions and variables
Example

- Consider a class of ‘Employees’
  - Salaried employees
  - Hourly employees
- Don’t ‘need’ type of generic ‘employee’
  - Since no one’s just an ‘employee’
- General concept of employee helpful!
  - All have names
  - All have social security numbers
  - Associated functions for these ‘basics’ are same among all employees
class Employee {
public:
    Employee();
    Employee(string tName, string tSsn);
    string getName() const;
    string getSsn() const;
    double getNetPay() const;
    void setName(string newName);
    void setSsn(string newSsn);
    void setNetPay(double newNetPay);
    void printCheck() const;
private:
    string name;
    string ssn;
    double netPay;
};
Derived Class - HourlyEmployee

class HourlyEmployee : public Employee
{
public:
    HourlyEmployee();
    HourlyEmployee(string tName, 
            string tSsn, double tWageRate, 
            double tHours);
    void setRate(double newWageRate); 
    double getRate() const; 
    void setHours(double hoursWorked); 
    double getHours() const; 
    void printCheck();
private:
    double wageRate; 
    double hours; 
};

List the declaration of an inherited function if you want to change the definition.
Derived Class - SalariedEmployee

class SalariedEmployee
   : public Employee
{
 public:
   SalariedEmployee();
   SalariedEmployee(string tName,
                     string tSsn, double tWageRate,
                     double tWeeklySalary);
   void setSalary(double newSalary);
   double getSalary() const;
   void printCheck();
 private:
   double salary;
};
Derived Class

• Derived classes from Employee class
  – Automatically have all member variables
  – Automatically have all member functions

• Derived class interface only lists new or ‘to be redefined’ members
  – Since all others inherited are already defined
  – i.e.: ‘all’ employees have ssn, name, etc.

• HourlyEmployee adds:
  – Constructors
  – A member variable and two member functions

• SalariedEmployee adds:
  – Constructors
  – Two member variables and four member functions
Constructors

• Base class constructors are NOT inherited in derived classes!
  – But they can be invoked within derived class constructor, which is all we need!

• Base class constructor must initialize all base class member variables
  – Those inherited by derived class
  – So derived class constructor simply calls it
    • ‘First’ thing derived class constructor does
Derived Class Constructor Example

- Consider syntax for HourlyEmployee constructor:

```cpp
DerivedClass::DerivedClass (string tName, string tSsn, double tWageRate, double tHours): BaseClass(tName, tNumber),
    wageRate(tWageRate), hours(tHours)
{
}
```

- Portion after `:` is “initialization section”
  - Includes invocation of base constructor
A Second Constructor

- A second constructor:

```cpp
HourlyEmployee::HourlyEmployee()
    : Employee(), wageRate(0), hours(0)
{ }
```

- Default version of base class constructor is called (no arguments)
What if Constructor not Called

• Derived class constructor should always invoke one of the base class’s constructors
• If you do not:
  – Default base class constructor automatically called
• Equivalent constructor definition:

```cpp
HourlyEmployee::HourlyEmployee()
    : wageRate(0), hours(0)
{
}
```
Copy Constructor

• Consider:
  Derived::Derived(const Derived& Object)
    : Base(Object), ...

{...

• After `:` is invocation of base copy constructor
  – Note Object is of type Derived; but it’s also of type Base, so argument is valid
Destructors in Derived Classes

- When derived class destructor is invoked:
  - Automatically calls base class destructor!
  - So no need for explicit call
- So derived class destructors need only be concerned with derived class variables
  - And any data they ‘point’ to
  - Base class destructor handles inherited data automatically
Destructor Calling Order

• Consider:
  class B derives from class A
  class C derives from class B
  \[ A \leftarrow B \leftarrow C \]

• When object of class C goes out of scope:
  – Class C destructor called 1st
  – Then class B destructor called
  – Finally class A destructor is called

• Opposite of how constructors are called
Access Class Members

- **Base class**
  - Public section
  - Protected section
  - Private section

- **Derived class**

  The derived class has access to both the public members and the protected members of the base class.

Clients:
- Client has access
- Client has no access
- Client has no access
The private: Quantifier

- Derived class ‘inherits’ private member variables
  - But still cannot directly access them
  - Not even through derived class member functions!
- Private member variables can ONLY be accessed ‘by name’ in member functions of the class they’re defined in
- Same holds for base member functions
The **private**: Quantifier

- Larger impact here vs. member variables
  - Member variables can be accessed indirectly via accessor or mutator member functions
  - Member functions simply not available

- This is ‘reasonable’
  - Private member functions should be simply ‘helper’ functions
  - Should be used only in class they’re defined
The **protected**: Quantifier

- New classification of class members
- Allows access ‘by name’ in derived class
  - But nowhere else
  - Still no access ‘by name’ in other classes
- Considered ‘protected’ in derived class
  - To allow future derivations
- Many feel this ‘violates’ information hiding
The public: Quantifier

- Allows access ‘by name’ in derived class
  - And anywhere else
- Violates information hiding
Protected and Private Inheritance

• New inheritance ‘forms’
  – Both are rarely used

• Protected inheritance:
  class SalariedEmployee : protected Employee {...}
  – Public members in base class become protected in derived class

• Private inheritance:
  class SalariedEmployee : private Employee {...}
Redefinition of Member Functions

• Recall interface of derived class:
  – Contains declarations for new member functions
  – Also contains declarations for inherited member functions to be changed
    • Inherited member functions NOT declared:
      • Automatically inherited unchanged

• Implementation of derived class will:
  – Define new member functions
  – Redefine inherited functions as declared
Redefining vs. Overloading

• Very different!

• Redefining in derived class:
  – SAME parameter list
  – Essentially ‘re-writes’ same function

• Overloading:
  – Same function name, different parameter list
  – Defined ‘new’ function that takes different parameters
  – Overloaded functions must have different signatures
A Function’s Signature

• Recall definition of a ‘signature’:  
  – Function’s name  
  – Sequence of types in parameter list  
    • Including order, number, types

• Signature does NOT include:  
  – Return type  
  – const keyword
Overloading Example: Average

- Function computes average of 2 numbers:
  ```c
  double average(double n1, double n2)
  {
    return ((n1 + n2) / 2.0);
  }
  ```

- Now compute average of 3 numbers:
  ```c
  double average(double n1, double n2, double n3)
  {
    return ((n1 + n2 + n3) / 3.0);
  }
  ```

- Same name, two functions
Operator Overloading

• Operators +, -, %, ==, etc.
  – Really just functions!
• Simply ‘called’ with different syntax:
  x + 7
  – ‘+’ is binary operator with x & 7 as operands
    • We ‘like’ this notation as humans
• Think of it as:
  +(x, 7)
  – ‘+’ is the function name
  – x, 7 are the arguments
  – Function ‘+’ returns ‘sum’ of it’s arguments
Overloading ==

- If we declare
  Money m1(5,25), m2(5,25);
- Can we use
  if (m1==m2)
    cout << "m1 and m2 has the same mount of money";
Overloading as non-Member Functions

```cpp
bool operator==(const Money& m1, const Money& m2) {
    return (m1.getDollars()==m2.getDollars()) &&
            m1.getCents()==m2.getCents();
}
```

- **Equality operator,==**
  - Enables comparison of Money objects
  - Returns bool type for true/false equality

- **Usage**
  - if (amount1==amount2) → if == (amount1, amount2)
Overloading as Member Functions

```cpp
bool Money::operator==(const Money& m2)
{
    return ((dollars==m2.getDollars()) &&
            cents==m2.getCents());
}
```

- Previous examples: standalone functions
  - Defined outside a class
- When operator is member function:
  - Only ONE parameter, not two!
  - Calling object serves as 1\textsuperscript{st} parameter
Function Redefinition

Employee::printCheck()
{
    cout << "ERROR: Undifferentiated employee\n"
    exit(0);
}

SalariedEmployee::printCheck()
{
    setNetPay(salary);
    cout << "Pay to the order of" << getName() << 
         " the amount of " << getNetPay() << " Dollars\n";
}
Accessing Redefined Function

- When redefined in derived class, base class’s definition not ‘lost’
- Can specify it’s use:
  Employee JaneE;
  HourlyEmployee SallyH;
  JaneE.printCheck(); → calls Employee’s printCheck function
  SallyH.printCheck(); → calls HourlyEmployee printCheck function
  SallyH.Employee::printCheck(); → Calls Employee’s printCheck function!
“Is a” Relationships

- Inheritance
  - Considered an “Is a” class relationship
  - e.g.: An HourlyEmployee “is a” Employee
"Has a" Relationships

- A class contains objects of another class as its member data
  - Considered a "Has a" class relationship
  - e.g.: One class "has a" object of another class as its data
Multiple Inheritance

• Derived class can have more than one base class!
  – Syntax just includes all base classes separated by commas:
    class derivedMulti : public base1, base2
    {...

• Possibilities for ambiguity are endless!

• Dangerous undertaking!
  – Some believe should never be used
  – Certainly should only be used by experienced programmers!
Summary 1

• Inheritance provides code reuse
  – Allows one class to ‘derive’ from another, adding features

• Derived class objects inherit members of base class
  – And may add members

• Private member variables in base class cannot be accessed ‘by name’ in derived

• Private member functions are not inherited
Summary 2

- Can redefine inherited member functions
  - To perform differently in derived class
- Protected members in base class:
  - Can be accessed ‘by name’ in derived class member functions
- Constructors are not inherited
  - Are invoked from derived class’s constructor