Inheritance Primer

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

Base Class - Employee

class Employee {
public:
  Employee() {name = ssn = ""; netPay = 0.0; }
  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;
};
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.

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
  - Derived class constructor simply calls it
    - 'First' thing derived class constructor does
Derived Class Constructor Example

• Consider syntax for HourlyEmployee constructor:

```cpp
HourlyEmployee::HourlyEmployee
    (string tName, double tWageRate, double tHours)
    : Employee(tName, tNumber), wageRate(tWageRate), hours(tHours)
{}  
```

• Portion after ':' is “initialization section”
  - Includes invocation of base constructor

A Second Constructor

• A second constructor:
  – Default version of base class constructor is called (no arguments)

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

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)
{}
```
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 \rightarrow B \rightarrow 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

Example

```cpp
class Point {
public:
    Point(int = 0, int = 0); // default constructor
    ~Point();                // destructor
    int getX() const;       // destructor
    int getY() const;
    void print() const { cout << '[' << x << ', ' << y << ']
                          << endl; }
private:
    int x, y;
};
```
// default constructor
Point::Point( int xValue, int yValue )
  : x( xValue ), y( yValue ) {
  cout << "Point constructor: " ;
  print();
}

// destructor
Point::~Point( ) {
  cout << "Point destructor: " ;
  print();
}

Output message to demonstrate constructor function call order.

Output message to demonstrate destructor function call order.

Example

class Circle : public Point {
public:
  Circle( int = 0 , int = 0 , double = 0.0 );
  ~Circle( );
  void print( ) const { cout << '[' << getX( ) << ',' << getY( ) << ',' <<radius<< ']' << endl; }
private:
  double radius;
};

Example

Circle::Circle( int xValue, int yValue, double radiusValue )
  : Point( xValue, yValue ), radius(radiusValue)
{}
  cout << "Circle constructor: " ;
  print();
}
Circle::~Circle( ) {
  cout << "Circle destructor: " ;
  print();
}

Output message to demonstrate constructor function call order.
Example

```cpp
int main() {
    // begin new scope
    Point point(11, 22);
} // end scope
```

Point constructor: [11, 22]
Point destructor: [11, 22]

```cpp
Circle circle1(72, 29, 4.5);
```

Circle constructor: [72, 29, 4.5]
Circle destructor: [5, 5, 10]

Circle destructor: [72, 29, 4.5]
Point destructor: [72, 22]

Point destructor: [72, 22]
```

```cpp
Circle circle2(5, 5, 10);
```

Circle constructor: [5, 5, 10]
Circle destructor: [5, 5, 10]

Circle destructor: [5, 5, 10]
```

Access Class Members

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

- Member functions 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