Object-oriented programming: Inheritance in C++

Topics:
- Derived classes and inheritance
- Member function overriding

Motivation: Employee records

```cpp
class Employee {
public:
    string first_name;
    string last_name;
    short department;
};
```

Questions:
- Isn’t a Manager also an Employee?
- Can a Manager have another Manager in his/her group?

Derived classes

Defines new class by extending existing class
- Base class
  - Class that is extended
  - Defines function and data members that are inherited by the derived class
- Derived class
  - Inherits function and data members from base class
  - May add additional function/data members
  - May also override inherited function members with new methods

How to declare a derived class

```cpp
class DerivedClassName : kind BaseClassName
{
public:
    // New function/data members.
protected:
    // ...
};
```

Where:
- kind ∈ {public, private, protected}

Example

```cpp
class Manager : public Employee {
public:
    list<Employee*> group;
    short level;
};
```

Terminology:
- Manager called the derived class (also subclass)
- Employee called the base class (also superclass)
- Manager inherits from Employee

Pointers/references & inheritance

Pointer (resp. reference) to an object of a derived class can be used as pointer (resp. reference) to an object of its base.

```cpp
void foo()
{
    Manager m, *mPtr;
    Employee e, *ePtr;
    ePtr = &m; // OK
    mPtr = &e; // Error.
}
```
Advantages of inheritance

- Factor out code that is common in multiple classes
- Less code to maintain!
- Fix errors once
- Reuse functions that operate on base-class objects
- Invoke function whose formal parameter is of (reference or pointer) to class C with actual parameter of (reference or pointer) to class derived from C.
- Represent domain relationships explicitly in code

Example: Class Figure

```cpp
class Figure {  
public:  
    Figure (unsigned x, unsigned y): xLocation(x), yLocation(y) ()  
    unsigned getLocation() const { return xLocation; }  
    unsigned getYLocation() const { return yLocation; }
protected:  
    unsigned xLocation;  
    unsigned yLocation;
};
```

Example: Continued

```cpp
class Rectangle : public Figure {  
public:  
    Rectangle (unsigned x, unsigned y, unsigned length, unsigned height): Figure(x, y), lengthDimension(length), heightDimension(height) ()  
    unsigned getLength() const { return lengthDimension; }  
    unsigned getHeight() const { return heightDimension; }
protected:  
    unsigned lengthDimension;  
    unsigned heightDimension;
};
```

Example: Reusing functions

```cpp
bool nearsOrigin(const Figure& f1, const Figure& f2) {
    unsigned fx1 = f1.getXLocation();  
    unsigned fy1 = f1.getYLocation();  
    unsigned fx2 = f2.getXLocation();  
    unsigned fy2 = f2.getYLocation();
    return (fx1 * fx1 + fy1 * fy1) < (fx2 * fx2 + fy2 * fy2);  
}
```

Example: Reusing functions (cont)

```cpp
int main(void) {
    Figure fig1(20,30), fig2(30,50);  
    Rectangle rect1(10,20,15,20);  
    Rectangle rect2(5,40,10, 50); ...
    if( nearsOrigin(fig1, fig2) ) {  
    ...  
    if( nearsOrigin(rect1, rect2) ) {  
    ...  
    if( nearsOrigin(fig1, rect1) ) { ...
}
```

Question

How is it possible to pass a reference to a Rectangle object to a function that expects a reference to a Figure object?

Answer: Objects of derived class implemented as objects of base class with additional data appended at the end.
Derived class (implementation)

<table>
<thead>
<tr>
<th>FIG1</th>
<th>FIG1</th>
<th>REC1</th>
<th>REC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>xLocation</td>
<td>20</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>yLocation</td>
<td>30</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>lengthDimension</td>
<td>13</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>heightDimension</td>
<td>28</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Access control

Given:
- Base class A with (data or function) member m
- Derived class B, which inherits from A
- Function f that uses an object of class A
- Function g that uses an object of class B

<table>
<thead>
<tr>
<th>PRIVATE</th>
<th>PROTECTED</th>
<th>PUBLIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

Member-function over-riding

```c++
class Employee {
    private:
        string firstName, lastName;
    short department;
    public:
        void print() { const std::ostream& os | const
            class Manager : public Employee {
                private:
                    list<Employee>* group;
                short level;
                public:
                    void print() { const std::ostream& os | const
                        void Manager::print() { const std::ostream& os | const
                            os << "Name: " << firstName << " " << lastName << endl << "Dept: " << department;
                        }
                    }
                }
            }
        }
    }
}
```

Function over-riding (continued)

```c++
int main(void)

    Employee e; "John", "Eve", 235];
    Manager m; "Charles", "Howell", 235, 3];
    e.print(); // invokes Employee::print();
    m.print(); // invokes Manager::print();
    e.print(); // invokes Employee::print();
    m.print(); // invokes Manager::print();
    return 0;
```

Pop quiz

```c++
int main(void)

    void myPrint(Employee& empl)
    {
        empl.print();
    }

    int main(void)
    {
        Employee e; "John", "Eve", 235];
        Manager m; "Charles", "Howell", 235, 3];
        myPrint(m);
    }
```

Question: Which method is invoked when actual parameter is Manager?
Terminology

Distinction between *operation* and *method*
- Operations are over-ridden with new methods.
  - E.g., `print` is an operation; `Employee::print` and `Manager::print` are methods.

**Consider:** It would be useful to be able to write functions that invoke an operation on an object, as opposed to invoking a particular method.