CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 9
September 29, 2011

Derivation

Construction, Initialization, and Destruction

Polymorphic Derivation

Derivation

Class relationships

Classes relate to and collaborate with other classes.

Many ways in which one class relates to other.

We first explore *derivation*, where one class modifies and extends another.

What is derivation?

One class can be derived from another.

Syntax:

```
class A {
public:
   int x;
   ...
};
class B : public A {
   int y;
   ...
};
```

A is the base class; B is the derived class.

B inherits the members from A.

Instances

A base class instance is contained in each derived class instance.

Similar to composition, except for inheritance.

Function members are also inherited.

Data and function members can be overridden in the derived class.

Derivation is a powerful tool for allowing variations to a design.

Some uses of derivation

Derivation has several uses.

- To allow a family of related classes to share common parts.
- To describe abstract interfaces à la Java.
- To allow generic methods with run-time dispatching.
- To provide a clean interface between existing, non-modifiable code and added user code.

Example: Parallelogram

```
class Parallelogram {
protected:           // allows access by children
    double base;     // length of base
    double side;     // length of side
    double angle;    // angle between base and side
public:
    Parallelogram() {}         // null default constructor
    Parallelogram(double b, double s, double a);
    double area() const;       // computes area
    double perimeter() const;  // computes perimeter
    ostream& print( ostream& out ) const;
};
```

Example: Rectangle

```
class Rectangle : public Parallelogram {
public:
   Rectangle( double b, double s ) {
      base = b;
      side = s;
      angle = pi/2.0; // assumes pi is defined elsewhere
   }
};
```

New class `Rectangle` inherits `area()`, `perimeter()`, and `print()` functions from `Parallelogram`.

Example: Square

```
class Square : public Rectangle {
public:
   Square( double b ) : Rectangle(b, b) {} // uses ctor
   bool inscribable( Square& s ) const {
      double diag = sqrt( 2.0 )*side;  // this diagonal
      return side <= s.side && diag >= s.side;
   }
   double area() const { return side*side; }
};
```

New class `Square` inherits the `perimeter()`, and `print()` functions from `Parallelogram` (via `Rectangle`).

It **overrides** the function `area()`.

Notes on `Square`

Features of `Square`.

- The ctor allows parameters to be supplied to the `Rectangle` constructor.
- The function `inscribable()` **extends** `Rectangle`, adding new functionality.
  It returns `true` if this square can be inscribed in square `s`.
- The function `area` overrides the less-efficient definition in `Parallelogram`.

Construction, Initialization, and Destruction

Structure of an object

A simple object is like a `struct` in C.
It consists of a block of storage large enough to contain all of its data members.

An object of a derived class contains an instance of the base class followed by the data members of the derived class.

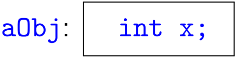
Example:

```
  class B : A { ...};
  B bObj;
```

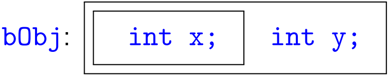
Then "inside" of `bObj` is an `A`-instance!

Example of object of a derived class

The declaration `A aObj` creates a variable of type `A` and storage size large enough to contain all of `A`'s data members (plus perhaps some padding).


aObj: int x;

The declaration `B bObj` creates a variable of type `B` and storage size large enough to contain all of `A`'s data members plus all of `B`'s data members.


bObj: int x; int y;

The inner box denotes an `A`-instance.

Referencing a composed object

Contrast the previous example to

```
class B { A aObj; ...};
B bObj;
```

Here `B` composes `A`.

The embedded `A` object can be referenced using data member name `aObj`, e.g., `bObj.aObj`.

Referencing a base object

How do we reference the base object embedded in a derived class?

Example:

```
class A { public: int x; int y; ...};
class B : A { int y; ...};
B bObj;
```

- The data members of `A` can be referenced directly by name.
  `x` refers to data member `x` in class `A`.
  `y` refers to data member `y` in class `B`.
  `A::y` refers to data member `y` in class `A`.
- `this` points to the whole object.
  Its type is `B*`.
  It can be coerced to type `A*`.

Initializing an object

Whenever a class object is created, one of its constructors is called.

If not specified otherwise, the default constructor is called.
This is the one that takes no arguments.

If you do not define the default constructor, then the null constructor (which does nothing) is used.

This applies not only to the “outer” object but also to all of its embedded objects.

Construction rules

The rule for an object of a simple class is:

1. Call the constructor/initializer for each data member object in sequence.
2. Call the constructor for the class.

The rule for an object of a derived class is:

1. Call the constructor for the base class recursively.
2. Call the constructor/initializer for each data member object of the derived class in sequence.
3. Call the constructor for the derived class.

Destruction rules

When an object is deleted, the destructors are called in the opposite order.

The rule for an object of a derived class is:

1. Call the destructor for the dervied class.
2. Call the destructor for each data member object of the derived class in reverse sequence.
3. Call the destructor for the base class.

Constructor ctors

Ctors (short for constructor/initializors) allow one to supply parameters to implicitly-called constructors.

Example:

```
class B : A {
  B( int n ) : A(n) {};
      // Calls A constructor with argument n
};
```

Initialization ctors

Ctors also can be used to initialze primitive (non-class) variables.

Example:

```
class  B {
  int x;
  const int y;
  B( int n ) : x(n), y(n+1) {}; // Initializes x and y
};
```

Multiple ctors are separated by commas.

Ctors present must be in the same order as the construction takes place – base class ctor first, then data member ctors in the same order as their declarations in the class.

Initialization not same as assignment

Previous example using ctors is not the same as writing

```
B( int n ) { y=n+1; x=n; };
```

- The order of initialization differs.
- `const` variables can be initialized but not assgined to.
- Initialization uses the constructor (for class objects).
- Initialization from another instance of the same type uses the copy constructor.

Copy constructors

- A copy constructor is automatically defined for each new class `A` and has prototype `A(const A&)`. It initializes a newly created `A` object by making a shallow copy of its argument.
- Copy constructors are used for call-by-value parameters.
- Assignment uses `operator=()`, which by default copies the data members but does not call the copy constructor.
- The results of the implicitly-defined assignment and copy constructors are the same, but they can be redefined to be different.

Polymorphic Derivation

Polymorphism and Type Hierarchies

Consider following simple type hierarchy:

```
class B     { public: int f(); ... };
class U : B { int f(); ... };
class V : B { int f(); ... };
```

We have a base class `B` and derived classes `U` and `V`.

Declare `B* bp; U* up = new U; V* vp = new V`.
Can write `bp = up;` or `bp = vp;`.

Why does this make sense?
`*up` has an embedded instance of `B`.
`*vp` has an embedded instance of `B`.

Relationships: A `U` is a `B` (and more). A `V` is a `B` (and more).

Polymorphic pointers

Recall:

```
class B     { public: int f(); ... };
class U : B { int f(); ... };
class V : B { int f(); ... };
B* bp;
```

bp can point to objects of type B, type U, or type V.
Say bp is a polymorphic pointer.

Want bp->f() to refer to U::f() if bp contains a U pointer.
Want bp->f() to refer to V::f() if bp contains a V pointer.
In this example, bp->f() always refers to B::f().

Virtual functions

Solution: Polymorphic derivation

```
class B     { public: virtual int f(); ... };
class U : B { virtual int f(); ... };
class V : B { virtual int f(); ... };
B* bp;
```

A virtual function is dispatched at run time to the class of the actual object.

bp->f() refers to U::f() if bp points to a U.
bp->f() refers to V::f() if bp points to a V.
bp->f() refers to B::f() if bp points to a B.

Here, the type refers to the allocation type.

Unions and type tags

We can regard `bp` as a pointer to the union of types `B`, `U` and `V`.

To know which of `B::f()`, `U::f()` or `V::f()` to use for the call `bp->f()` requires runtime type tags.

If a class has `virtual` functions, the compiler adds a type tag field to each object.
This takes space at run time.

The compiler also generates a vtable to use in dispatching calls on virtual functions.

Virtual destructors

Consider `delete bp;`, where `bp` points to a `U` but has type `B*`.

The `U` destructor will *not* be called unless destructor `B::~B()` is declared to be `virtual`.

Note: The base class destructor is always called, *whether or not it is* `virtual`.

In this way, destructors are different from other member methods.

Conclusion: If a derived class has a non-empty destructor, the *base class* destructor should be declared `virtual`.