CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 15
March 10, 2016

Storage Model Revisited

Object parts

Recall that an object is a block of memory divided into parts, where each part is an instance of the base class or is an instance of a data member. We say that the parts are **embedded** in the "parent" or **outer** object.

Embedded objects are actual objects in their own right. A constructor is called when they are born, and their destructor is called when they are about to die. They can be pointed at and used wherever an object of their type is needed.

However, an embedded object is dependent on its parent and has the same lifetime and storage class as its parent.

Object parts (cont.)

Because an embedded object lives and dies with its parent, it can never be explicitly deleted, even if it lives in the heap.

Rather, it is deleted when its parent is deleted. Neverthtless, its destructor will be called just before it dies, just like for any other object!

Note that embedding order and derivation order are opposites. If class `Deriv` is derived from class `Base`, then `Base` is the parent of `Deriv` in the class hierarchy, but an instance of `Deriv` has an instance of `Base` embedded within it.

Functions Revisited

Global vs. member functions

A **global** function is one that takes zero or more *explicit* arguments.
Example: `f(a, b)` has two explicit arguments `a` and `b`.

A **member** function is one that takes an *implicit* argument along with zero or more *explicit* arguments.
Example: `c.g(a, b)` has two explicit arguments `a` and `b` and implicit argument `c`.
Example: `d->g(a, b)` has two explicit arguments `a` and `b` and implicit argument `*d`.

Note that an omitted implicit argument defaults to `(*this)`, which must make sense in the context.
Example: If `g` is a member function of class `MyClass`, then within `MyClass`, the call `g(a, b)` defaults to `(*this).g(a,b)` (or equivalently `this->g(a,b)`).

Defining global functions

There are three ways to define a global function.

1. Place the declaration at the top level of your code, outside of any class declarations. Most functions in C are of this kind.
2. Place the declaration inside a class definition, prefixed by the keyword `static`. This creates a global function whose *name* is qualified by the class name. It's visibility is controlled by the visibility keywords `public`, `protected`, and `private`.
3. Place the declaration at the top level and prefix its name by `static`. This creates a C-style static function whose name is visible only within the one compile module. Classes and static member functions provide a better way to provide modularity and control name visibility, so this should not be used in C++. It is retained only for compatibility with C.

Defining member functions

Placing a function declaration inside a class definition creates a member function.

Its definition is considered to be "inside" the class, whether or not it appears in the class or as an out-of-line function in a `.cpp` file.

Example:

```
class MyClass {
protected:
    double g(const int* a, unsigned b) const;
};
```

This defines a member function `g` with explicit parameters of type `const int*` and `unsigned` and implicit parameter of type `const MyClass&`.

Operator syntax

We have seen the `operator` keyword used to extend the meaning of operators.

Each binary operator $\oplus$ corresponds to a function whose name is `operator`$\oplus$, but the operator syntax $a \oplus b$ does not tell us whether to look for a global or a member function. Possibile meanings:

- Global function: `operator`$\oplus$`(a, b)`.
- Member function: `a.operator`$\oplus$`(b)`.

It could mean either, and the compiler sees if either one matches. If both match, it reports an ambiguity.

Operator extension as member function

Here's a sketch for how one might go about defining a complex number class.

```
class Complex {
private:
   double re;  // real part
   double im;  // imaginary part
public:
   Complex( double re, double im ) : re(re), im(im) {}
   Complex operator+(const Complex& b) const {
      return Complex( re+b.re, im+b.im );
   }
   Complex operator*(const Complex& b) const {
      return Complex( re*b.re - im*b.im, re*b.im + im*b.re );
   }
};
```

Operator extension as global function

We have seen one important example of a global operator extension when we define the output operator on a new class.

Given the choice, it is preferable to use a member operator function.

We use a global form of `operator<<` because the left hand operator is of predefined type `ostream`, and we can't add member functions to that class.

Prefix unary operator extensions

C++ has a number of prefix unary operators
`*`, `-`, `++`, `new`, `...`

The corresponding operator functions are
`operator*()`, `operator-()`, `operator++()`,
`operator new()`, `...`

Postfix unary operator extensions

C++ also has two postfix unary operators `++, --`.

The corresponding operator functions are `operator++(int), operator--(int)`.

This is a special case that breaks all the normal rules, but it works since `++` and `--` are not binary operators. The dummy `int` parameter should be ignored.

Ambiguous operator extensions

```
class Bar {
public:
    int operator+(int y) { return y+2; }
};

int operator+(Bar& b, int y) { return y+3; }

int main() {
    Bar b;
    cout << b+5 << endl;
}
```

Compiler reports `error: ambiguous overload for 'operator+' in 'b + 5'`.

Functional composition

Functional composition refers to using the result returned by one function as the argument for another.

Example: `g(f(x))`.

The type of `f(x)` (which is the result type declared in the definition of `f()`) must be **compatible** with the corresponding parameter type for *some* method of `g()`.

Types are compatible if they are the same, or if the result type can be converted to the corresponding parameter type.

Type compatibility

Here's what the compiler does when it sees the call `g(f(x))`.

1. It finds the type of `f(x)`. Call it `T`.
2. It looks for a method for `g` with **signature** `(T)`.
3. If it finds one, that method is selected.
4. If not, it searches the methods for `g` with signatures that are compatible with `(T)`, meaning that it is possible to convert `T` to the type required by the signature.
5. If it finds exactly one such method, then that is used.
6. If it fails to find one, it reports "no match", and it lists the candidates it tried.
7. If it finds more than one possible method, it reports "ambiguous".

Calling constructors implicitly

Normally, constructors are called implicitly when an object is created, whether by `new` (in the case of dynamic storage) or by having a declaration executed (in the case of automatic storage).

When several constructor methods are present, which is chosen depends on the arguments supplied, either explicity or through ctors, but the call itself is implicit.
Examples

- `MyClass b` creates a stack object and invokes the default constructor `MyClass()`.
- `MyClass b(4):` creates a stack object and invokes constructor `MyClass(4)`.
- `new MyClass(6)` creates a dynamic object and invokes constructor `MyClass(6)`.

Calling constructors explicitly

Constructors can also be called explicitly, just like ordinary global functions.

The meaning is to create a new temporary stack object, just as a new temporary is created to hold the result of `y+z` in the expression `x*(y+z)`.

As with all object construction, the constructor is called when the object is created, and the destructor is called when it is deleted.

Because the created object is temporary, it must be used immediately, after which it will be discarded.

This is how `throw Fatal("Error message")` works. `Fatal()` creates an exception object of type `Fatal` for use by `throw`.

Conversion using constructor

Now suppose `f()` returns an object of type `A&` and `g()` expects an argument of type `B`. What happens with `g(f())`?

Example 1:

```
class A;  // forward declaration

class B {
public:
    B(){}
    B(A& aa) { cout << "B constructor called" << endl; }
};
```

Compiler will use `B`'s constructor to build a `B&` from an `A&`.

Output is "`B constructor called`".

Conversion using a cast

Example 2:

```
class B;  // forward declaration

class A {
public:
    operator B() {
        cout << "operator B cast called" << endl;
        return *new B;
    }
};
```

Compiler will use `A::operator B()` to cast the `A&` returned by `f()` to the `B` expected by `g()`.

Output is "`operator B cast called`".

What if both options exist?

```
class A;  // forward declaration
class B { public:
    B(){}
    B(A& aa) { cout << "B constructor called" << endl; }
};
class A { public:
    operator B() {
        cout << "operator B cast called" << endl;
        return *new B;
    }
};
A& f() { return *new A; }
B& g(B aa) { return *new B; }
```

Compiler will complain "`error:  conversion from 'A' to 'B' is ambiguous`".