CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 17
November 2, 2016

Functions Revisted (continued)

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`”.

Polymorphic Derivation

Some uses for derived classes.

- Code reuse. A base class can contain one copy of code that is be used by several derived variants through inheritance.
- Modularity. The functionality provided by a base class can be extended in a derived class. Example: `BSquare` extends `Square` by adding board coordinates and clusters.
- Generic programming and isolation. Demo 17-Craps-extended contains a simulator for the gambling game "craps" that can use different dice implementations.
- Polymorphic collections. A company has different kinds of employees with different rules for calculating their pay, each represented by a derived class with its own `calculatePay` function appropriate to that kind of employee.

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`.
A different method `f()` is defined in each.

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

A `U` can be used wherever a `B` is expected.

Example: Definition `f(B& x)  ...` ; call `U z; f(z);`

Inside of `f()`, only the `B`-part of `z` is visible. This is called **slicing**.

Pointers and slicing

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`.

If `bp = up`, then `bp` points to the embedded `B`-instance of object `*up`. The rest of `*up` is inaccessible because of object slicing.

Ordinary derivation

In our previous example

```
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`.

Want `bp->f()` to refer to `U::f()` if `bp` points to a `U` object.
Want `bp->f()` to refer to `V::f()` if `bp` points to a `V` object.

However, with ordinary derivation, `bp->f()` always refers to `B::f()`.

Polymorphic derivation

The keyword `virtual` allows for 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`.

Uses of polymorphism

Some uses of polymorphism:

- To define an extensible set of representations for a class.
- To allow containers to store mixtures of different but related types of objects.
- To support run-time variability of within a restricted set of related types.

Multiple representations

Might want different representations for an object.

Example: A point in the plane can be represented by either Cartesian or Polar coordinates.

A `Point` base class can provide abstract operations on points. E.g., `virtual int quadrant() const` returns the quadrant of `*this`.

For Cartesian coordinates, quadrant is determined by the signs of the x and y coordinates of the point.
For polar coordinates, quadrant is determined by the angle θ.

Both `Cartesian` and `Polar` derived classes should contain a method for `int quadrant() const`.

Heterogeneous containers

One might wish to have a stack of `Point` objects.

The element type of the stack would be `Point*`.

The actual values would have type either `Cartesian*` or `Polar*`.

The automatically generated type tags and dynamic dispatching obviates the need to cast the result of `pop()` to the correct type.

Example:

```
Stack st; Point* p;
p = st.pop();   // no need to cast result
p->quadrant(); // automatic dispatch
```