CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 16
March 29, 2016
Polymorphic Derivation
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:

```cpp
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

```cpp
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 \texttt{B* bp; U* up = new U; V* vp = new V}.

Can write \texttt{bp = up;} or \texttt{bp = vp;}

Why does this make sense?

\begin{itemize}
\item *up has an embedded instance of \texttt{B}.
\item *vp has an embedded instance of \texttt{B}.
\end{itemize}

If \texttt{bp = up}, then \texttt{bp} points to the embedded \texttt{B}-instance of object \texttt{*up}. The rest of \texttt{*up} is inaccessible because of object slicing.
Ordinary derivation

In our previous example

```cpp
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.

```cpp
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 $\theta$.

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
```
Run-time variability

Two types are closely related; differ only slightly.

Example: Company has several different kinds of employees.

- Employee base class has a large and complicated payroll function.
- Payroll is same for all kinds of employees except for a function \texttt{pay()} that computes the actual weekly pay.
- Each employee kind has its own \texttt{pay()} function.
- Big payroll function is in base class.
- It calls \texttt{pay()} to get the actual pay for this Employee.
Pure virtual functions

Suppose we don’t want \texttt{B::f()} and never create instances of \texttt{B}. We make \texttt{B::f()} into a \textit{pure virtual function} by writing \texttt{=0}.

```cpp
class B { public: virtual int f()=0; ... }; 
class U : B { virtual int f(); ... }; 
class V : B { virtual int f(); ... }; 
B* bp;
```

A pure virtual function is sometimes called a \textit{promise}. It tells the compiler that a construct like \texttt{bp->f()} is legal. The compiler requires every derived class to contain a method \texttt{f()}. 

Abstract classes

An **abstract class** is a class with one or more pure virtual functions.

An abstract class cannot be instantiated. It can only be used as the base for another class.

The destructor can never be a pure virtual function but will generally be **virtual**.

A **pure abstract class** is one where all member functions are pure virtual (except for the destructor) and there are no data members,

Pure abstract classes define an **interface à la Java**.

An interface allows user-supplied code to integrate into a large system.