CPSC 427: Object-Oriented Programming

Michael J. Fischer

Preview Lecture 15
October 24, 2018

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. A simulation such as PS4 might want to use different random number implementations, e.g., one using `random()` and another reading numbers from a file.
- 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

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
```

Uses of polymorphism: 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 `pay()` that computes the actual weekly pay.
- Each employee kind has its own `pay()` function.
- Big payroll function is in base class.
- It calls `pay()` to get the actual pay for this `Employee`.

Pure virtual functions

Suppose we don't want `B::f()` and we never create instances of the base class `B`.
Rather, we want every derived class to provide a definition for `f()`.
We make `B::f()` into a pure virtual function by writing =0.

```
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 promise.
It tells the compiler that a construct like `bp->f()` is legal.
The compiler requires every derived class to contain a method `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.

Introduction to the C++ Standard Library

A bit of history

C++ standardization.

- C++ standardization began in 1989.
- ISO and ANSI standards were issued in 1998, nearly a decade later.
- The standard covers both the C++ language and the standard library (everything in namespace `std`).
- The standardization process continues as the language evolves and new features are added.

The standard library was derived from several different sources.

STL (Standard Template Library) portion of the C++ standard was derived from an earlier STL produced by Silicon Graphics (SGI).

Some useful classes

Here are some useful classes, some of which you have already seen:

- `string` – a character string designed to act as much as possible like the primitive data types such as `int` and `double`.
- `iostream`, `ifstream`, `ofstream` — buffered reading and writing of character streams.
- `istringstream` – permits input from an in-memory string-like object.
- `vector<T>` – creates a growable array of objects of type `T`, where `T` can be any type.

Class `stringstream`

A `stringstream` object (in the default case) acts like an `ostream` object.

It can be used just like you would use `cout`.

The characters go into an internal buffer rather than to a file or device.

The buffer can be retrieved as a `string` using the `str()` member function.

stringstream example

Example: Creating a label from an integer.

```
#include <sstream>
...
int examScore=94;
stringstream ss;
string label;
ss << "Score=" << examScore;
label = ss.str();
cout << label << endl;
```

This prints `Score=94`.

vector

`vector<T> myvec` is something like the C array `T myvec[]`.

The element type `T` can be any primitive, object, or pointer type.

One big difference is that a `vector` starts empty (in the default case) and it grows as elements are appended to the end.

Useful functions:

- `myvec.push_back( item )` appends `item` to the end.
- `myvec.size()` returns the number of objects in `myvec`
- `myvec[k]` returns the object in myvec with index `k` (assuming it exists.) Indices run from 0 to size()-1.

Other operations on `vectors`

Other operations include creating an empty vector, inserting, deleting, and copying elements, scanning through the vector, and so forth.

Liberal use is made of operator definitions to make vectors behave as much like other C++ objects as possible.

Vectors implement value semantics, meaning type `T` objects are moved freely within the vectors.

This implies that class `T` should support move constructors and assignment.

Alternatively, one can store pointers in the vector instead.

vector examples

You must #include <vector>.

Elements can be accessed using standard subscript notion.

Inserting at the beginning or middle of a vector takes time $O(n)$.

Example:

```
vector<int> tbl(10);  // creates length 10 vector of int
tbl[5] = 7;           // stores 7 in slot #5
cout << tbl[5];       // prints 7
tbl[10] = 4;          // illegal, but not checked!!!
cout << tbl.at(5);    // prints 7
tbl.at(10) = 4;       // illegal and throws an exception
tbl.push_back(4);     // creates tbl[10] and stores 4
cout << tbl.at(10);   // prints 4
```