CSE 143

Dynamic Dispatch and Virtual Functions

[Chapter 8 pp.354-370]

Substituting Derived Classes

- Recall that an instance of a derived class can always be substituted for an instance of a base class
  - Derived class guaranteed to have (at least) the same data and interface as base class
- But you may not get the behaviour you want!

```
//client function (not a method)
void printPoint( Point pt )
{
    pt.print( cout );
    //the question: which print?
}

Point p( 1.0, 9.0 );
ColorPoint cp( 6.0, 7.0, red );

printPoint( p );
p = cp; //information lost
printPoint( p );
printPoint( cp );
```


Pointers And Inheritance

- You can also substitute a *pointer* to a derived class for a *pointer* to a base class
  - There's still that guarantee about data and interface
  - Also holds for reference types
  - **No information disappears!!**
- Unfortunately, we still have the same problems…

```
//client function
void printPoint( Point *ptr )
{
    ofstream ofs( "point.out" );
    ptr->print( ofs );
    ofs.close();
}

Point *pptr = new Point( 1.0, 9.0 );
ColorPoint *cpptr =
    new ColorPoint( 6.0, 7.0, red );

printPoint( pptr );
printPoint( cpptr );

pptr = cpptr;
printPoint ( pptr);
```


Static And Dynamic Types

- In C++, every variable has a *static* and a *dynamic* type
  - Static type is declared type of variable
    - Every variable has a single static type that never changes
  - Dynamic type is type of object the variable actually contains or refers to
    - Dynamic type can change during the program!
- Up to now, these have always been identical
  - But not any more!

```
Point *myPointPointer = new ColorPoint( 3.14, 2.78, green );
```


"Dispatch"

- "Dispatching" is the act of deciding which piece of code to execute when a method is called
- Static dispatch means that the decision is made statically, *i.e.* at compile time
  - Decision made based on static (declared) type of receiver

```
Point *myPointPointer = new ColorPoint( 3.14, 2.78, green );

myPointPointer->print( cout );
  // myPointPointer is a Point*, so call Point::print
```


Dynamic Dispatch

- C++ has a mechanism for declaring individual methods as dynamically dispatched
  - If an overriding function exists, call it
  - The decision is made at run-time
  - Sometimes called "late binding".
- The mechanism: In base class, label the function with `virtual` keyword
  - Overriding versions in subclasses don't need the `virtual` keyword
    - but please use the keyword anyway for better style

Example Of Dynamic Dispatch

```
class Point { //base class
public:
    virtual void print( ostream& os );
    …
};

class ColorPoint : public Point { //derived class
public:
    virtual void print( ostream& os );
    …
};

//in a client
Point *p = new ColorPoint( 3.13, 5.66, ochre );

p->print( cout );
     // calls ColorPoint::print( )
```

07/22/01 R-7

Dynamically-Dispatched Calls

```
Point *p = new ColorPoint( 3.13, 5.66, ochre );
p->print( cout );
```

- The compiler notices that `Point::print` is defined as `virtual`
- Instead of just calling `Point::print`, it inserts extra code to look at information attached to the object by `new` to decide what function to call
- This is slightly slower than static dispatch
  - Almost always too minor a speed penalty to worry about

07/22/01 R-8

When Does This Happen?

- Dynamic dispatch ONLY happens when BOTH of these two conditions are met:
  1. The object is accessed through a pointer (or reference)
  2. The method is virtual
- In ALL other cases, you get static dispatch
- Two common useful cases
  - **Parameters**: Objects passed by pointer or reference to a function
  - **Arrays**: an array of pointers to objects

07/22/01 R-9

Example Application

- An array of pointers to objects derived from the same base class:

  *mammal * zoo[20];* // An array of 20 pointers.
- All the objects pointed to are mammals, but some might be dogs, people, aardvarks, hedgehogs, etc.
- Each class might have its own methods for behavior like “scream” “fight” “laugh”, etc.
  - If I write *zoo[i]->laugh()* I want to get the appropriate behavior for that type of animal

    `Won't happen unless laugh is virtual in mammal class`

07/22/01 R-10

Contrast

- ***mammal mlist[20];***
  - all array elements are of the same type
  - Everything in the list is treated as a mammal, period

    `regardless of whether methods are virtual or not`
- ***mammal * vmlist[20];***
  - Each critter behaves like "mammal" for the non-virtual functions, and like its own particular kind of mammal for the virtual methods.

07/22/01 R-11

Virtual Destructors

- If a class contains a destructor and is used as a base class, then the destructor should be declared virtual
  - Ensures that correct destructor is called when a pointer to that class is deleted, even if there are no other virtual functions

```
Class XYZ {
public:
        ...
        virtual ~XYZ();
        ...
};
```

- Puzzle: *constructors* are ***never*** virtual!! (Why?)

07/22/01 R-12

Abstract vs Concrete Classes

- Some classes are so abstract that instances of them shouldn't even exist
  - What does it mean to have an instance of `widget`? of `pushbutton`? of `Animal`?
- It may not make sense to attempt to fully implement all functions in such a class
  - What should `pushbutton::clicked()` do?
- An *abstract class* is one that should not or can not be instantiated - it only defines an interface
  - declaration of public methods, partial implementation
- A *concrete class* can have instances

Abstract Class in C++

- "abstract" and "concrete" are not keywords in C++
- Abstract classes are recognized by being classes with unimplementable methods
  - "pure virtual functions" (next slide)
- Such a class is only intended to be used as a base class

Pure Virtual Functions

- Syntax: append "`= 0`" to base method declaration

```
class pushbutton : public widget {
public:
  virtual void clicked() = 0;
};
```

- A "pure virtual" function is not implemented in the base class
  - must implement in derived classes
- Compiler guarantees that class with pure virtual functions cannot be instantiated
- If you call a pure virtual function, you'll use the version from some derived class

```
pushbutton *b = new quitbutton;
b->clicked();
```


Draw the Hierarchy

```
class animal {...
  virtual dance ( ) = 0;
... };

class mammal : public
  animal {...
  dance ( );
  walk ( );
...};
```

```
class hedgehog : public
  mammal {...
      // no "dance" method
      dig ( );
      walk ( );
      walk (int, int);
...};

class seaUrchin : public
  animal {...
      dance ( );
      sting ( );
};
```


What's Legal / Which function is called? (continued)

- animal annie;
- hedgehog * hp;
- hp->walk ();
- animal * ap = hp;
- ap -> dance();
- ap->walk();
- mammal * mp = hp;
- mp->walk();

Example Hierarchy

```
class person {...
  virtual walk ( ) = 0;
  virtual run ( );
...};

class student : public person
  {...
  enroll ( );
  virtual walk ( );
...};
```

```
class freshman : public
  student {...
      enroll ( );
      virtual run ( );
...};
```

What's Legal / Which function is called? (continued)

- person paula;
- student *stu = new freshman();
- stu->enroll();
- student sara = *stu;
- sara.run();
- person *pp = stu;
- pp->run();
- pp->walk();
- freshman *fred = pp;
- fred->enroll();

07/22/01 R-19

Draw hierarchy & call graph

Start your drawing at plug::dispatch()

```
class plug {
  public:
    virtual void boof()
      { bang(); }
    virtual void bang()
      { nalg(); }
    void dispatch()
      { trog->boof(); }
  protected:
    plug *trog;
};
```

```
class lir : public plug {
  public:
    virtual void boof()
      { biff(); }
}

class vop : public plug {
  public:
    virtual void bang()
      { whing(); }
  protected:
    int log;
}
```

07/22/01 R-20