CSE 333

Lecture 17 - inheritance, dispatch, slicing

Steve Gribble

Department of Computer Science & Engineering

University of Washington

Administrivia

Midterm on Monday

- reminder that Colin, Aryan are proctoring the midterm

HW3 is due in 12 days

- as usual, start early!!

Goals for today

One last run at inheritance

- virtual functions and dynamic dispatch
- slicing

Using STL containers to store base/derived classes

- pointers
- wrapper classes

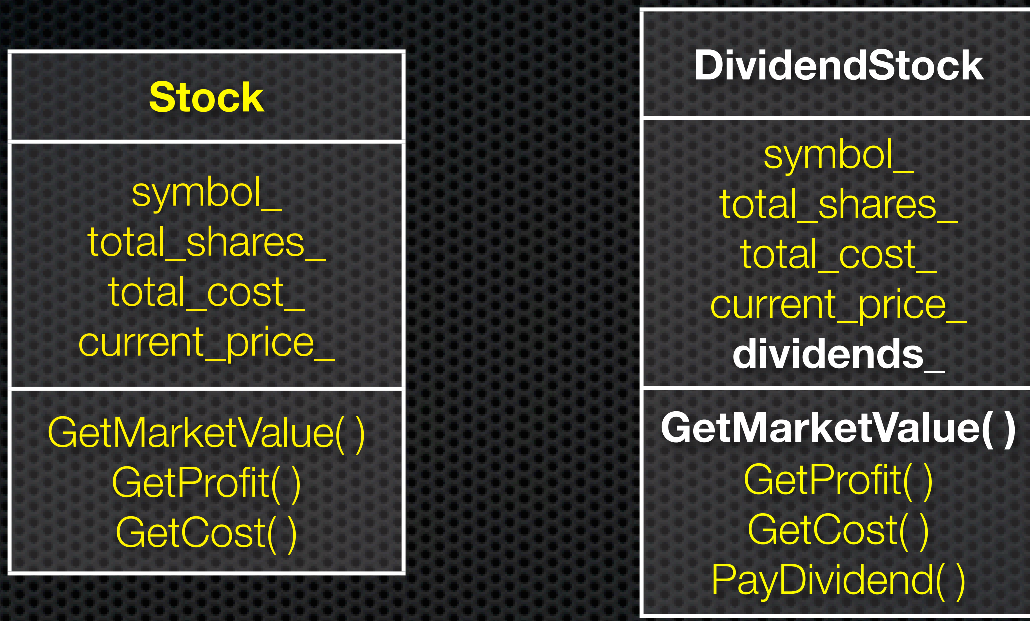
Public inheritance

```
#include "BaseClass.h"

class DerivedClass : public BaseClass {
   ...
};
```

- "public" inheritance
  - anything that is [*public, protected*] in the base is [*public, protected*] in the derived class
- derived class inherits **almost** all behavior from the base class
  - not constructors and destructors
  - not the assignment operator or copy constructor

Revisiting the portfolio example



Without inheritance (separate class per type)

- lots of redundancy
- no type relationship between the classes

Revisiting the portfolio example

A derived class:

- ***inherits*** the behavior and state of the base class
- **overrides** some of the base class's member functions
- **extends** the base class with new member functions, variables

(implement better_design/)

Static dispatch

When a member function is invoked on an object

- the code that is invoked is decided at compile time, based on the compile-time visible type of the callee

```
double DividendStock::GetMarketValue() const {
  return get_shares() * get_share_price() + _dividends;
}

double DividendStock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

DividendStock.cc

```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

Stock.cc

Static dispatch

```
DividendStock dividend();

DividendStock *ds = &dividend;
Stock *s = &dividend;

// invokes Stock::GetProfit(), since that function is
// inherited (i.e, not overridden).  Stock::GetProfit()
// invokes Stock::GetMarketValue(), since C++ uses
// static dispatch by default.
ds->GetProfit();

// invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// invokes Stock::GetMarketValue()
s->GetMarketValue();
```

(see even_better_design/)

Dynamic dispatch

When a member function is invoked on an object

- the code that is invoked is decided at run time, and is the **most-derived function** accessible to the object's visible type

```
double DividendStock::GetMarketValue() const {
  return get_shares() * get_share_price() + _dividends;
}

double DividendStock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

DividendStock.cc

```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
  return DividendStock::GetMarketValue() - GetCost();
}
```

Stock.cc

Dynamic dispatch

```
DividendStock dividend();

DividendStock *ds = &dividend;
Stock *s = &dividend;

// invokes Stock::GetProfit(), since that function is
// inherited (i.e, not overridden).  Stock::GetProfit()
// invokes Dividend::GetMarketValue(), since that is
// the most-derived accessible function.
ds->GetProfit();

// invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// invokes DividendStock::GetMarketValue()
s->GetMarketValue();
```

But how can this possibly work??

The compiler produces Stock.o from Stock.cc

- while doing this, it can't know that DividendStock exists
  - so, how does the code emitted for Stock::GetProfit( ) know to invoke Stock::GetMarketValue( ) some of the time, and DividendStock::GetMarketValue( ) other times???!?

```
virtual double Stock::GetMarketValue() const;

virtual double Stock::GetProfit() const;
```

Stock.h

```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

Stock.cc

vtables and the vptr

If a member function is virtual, the compiler emits:

- a "vtable", or virtual function table, **for each class**
  - it contains an function pointer for each virtual function in the class
  - the pointer points to the most-derived function for that class
- a "vptr", or virtual table pointer, **for each object instance**
  - the vptr is a pointer to a virtual table, and it is essentially a hidden member variable inserted by the compiler
  - when the object's constructor is invoked, the vptr is initialized to point to the virtual table for the object's class
  - thus, the vptr "remembers" what class the object is

vtable/vptr example

```
class Base {
 public:
  virtual void fn1() {};
  virtual void fn2() {};
};

class Dr1: public Base {
 public:
  virtual void fn1() {};
};

class Dr2: public Base {
 public:
  virtual void fn2() {};
};
```

```
// what needs to work

Base b;
Dr1  d1;
Dr2  d2;

Base *bptr = &b;
Base *d1ptr = &d1;
Base *d2ptr = &d2;

bptr->fn1();    // Base::fn1()
bptr->fn2();    // Base::fn2()

d1ptr->fn1();   // Dr1::fn1();
d1ptr->fn2();   // Base::fn2();

d2.fn1();       // Base::fn1();
d2ptr->fn1();   // Base::fn1();
d2ptr->fn2();   // Dr2::fn2();
```

vtable/vptr example

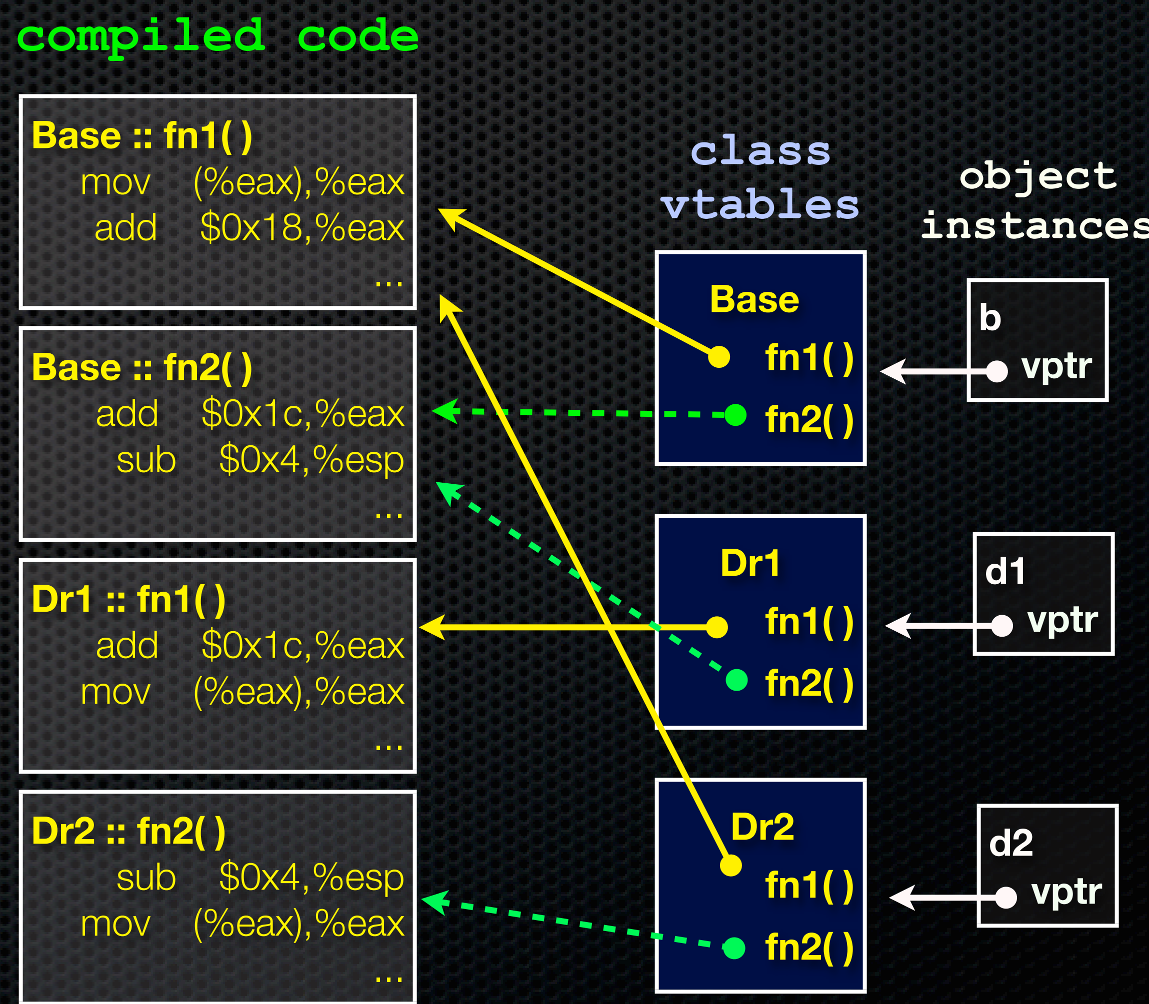
```
// what happens

Base b;
Dr1  d1;
Dr2  d2;

Base *d2ptr = &d2;

d2.fn1();
// d2.vptr -->
// Dr2.vtable.fn1 -->
// Base::fn1()


d2ptr->fn2();
// d2ptr -->
// d2.vptr -->
// Dr2.vtable.fn1 ->
// Base::fn1()
```



actual code

```
class Base {
 public:
  virtual void fn1() {};
  virtual void fn2() {};
};

class Dr1: public Base {
 public:
  virtual void fn1() {};
};

main() {
 Dr1   d1;
 d1.fn1();
 Base *ptr = &d1;
 ptr->fn1();
}
```

vtable.cc

Let's compile this and use objdump to see what g++ emits!

- g++ -g vtable.cc
- objdump -CDSRTtx a.out | less

Inheritance and constructors

A derived class **does not inherit** the base class's constructor

- the derived class ***must*** have its own constructor
  - if you don't provide one, C++ synthesizes a default constructor for you
    - it initializes derived class's member variables to zero-equivalents and invokes the default constructor of the base class
    - if the base class has no default constructor, a compiler error
- a constructor of the base class is invoked after the constructor of the derived class
  - you can specify which base class constructor in the initialization list of the derived class, or C++ will invoke default constructor of base class

Examples

```
// Base has no default constructor
class Base {
 public:
  Base(int x) : y(x) { }
  int y;
};

// Compiler error when you try
// to instantiate a D1, as D1's
// synthesized default constructor
// needs to invoke Base's default
// constructor.
class D1 : public Base {
 public:
  int z;
};

// Works.
class D2 : public Base {
 public:
  D2(int z) : Base(z+1) {
   this->z = z;
  }
  int z;
};
```

badcons.cc

```
// Base has a default constructor.
class Base {
 public:
  int y;
};

// Works.
class D1 : public Base {
 public:
  int z;
};

// Works.
class D2 : public Base {
 public:
  D2(int z) {
   this->z = z;
  }
  int z;
};
```

goodcons.cc

Destructors

When the destructor of a derived class is invoked...

- the destructor of the base class is invoked after the destructor of the derived class finishes

Note that static dispatch of destructors is almost always a mistake!

- good habit to always defined a destructor as virtual
  - empty if you have no work to do

baddestruct.cc

```
class Base {
 public:
  Base() { x = new int; }
  ~Base() { delete x; }
  int *x;
};

class D1 : public Base {
 public:
  D1() { y = new int; }
  ~D1() { delete y; }
  int *y;
};

Base *b = new Base;
Base *dptr = (Base *) new D1;

delete b;     // ok
delete dptr; // leaks D1::y
```

Slicing -- C++'s revenge

C++ allows you to...

- assign to..
  - an instance of a base class...
  - the value of a derived class

slicing.cc

```
class Base {
 public:
  Base(int x) : x_(x) { }
  int x_;
};

class Dr : public Base {
 public:
  Dr(int y) : Base(16), y_(y) { }
  int y_;
};

main() {
  Base b(1);
  Dr d(2);
  b = d;      // what happens to y_?
  // d = b; // compiler error
}
```

Given this, STL containers?? :(

STL stores **copies of values** in containers, not pointers to object instances

- so, what if you have a class hierarchy, and want to store mixes of object types in a single container?
    - e.g., Stock and DividendStock in the same list
- you get sliced! :(

```
class Stock {
  ...
};

class DivStock : public Stock {
  ...
};

main() {
  Stock         s;
  DivStock     ds;
  list<Stock> li;

  l.push_back(s);    // OK
  l.push_back(ds);   // OUCH!
}
```

STL + inheritance: use pointers?

Store pointers to heap-allocated objects in STL containers

- no slicing :)
  - you have to remember to delete your objects before destroying the container :(
  - sort( ) does the wrong thing :(:(

```
#include <list>
using namespace std;

class Integer {
 public:
  Integer(int x) : x_(x) { }
 private:
  int x_;
};

main() {
  list<Integer *> li;
  Integer *i1 = new Integer(2);
  Integer *i2 = new Integer(3);

  li.push_back(i1);
  li.push_back(i2);
  li.sort();  // waaaaaah!!
}
```

An idea...

Create a wrapper class?

- contains a pointer to the thing we actually want to store in the STL container
  - e.g., Stock* or DividendStock*
- overrides "<" so sort works
- calls delete in its destructor
- but...STL makes many copies
  - lots of destructors are invoked
  - argh!!!! $#@#$!

```
#include <vector>
#include <algorithm>

using namespace std;

class Integer {
 public:
  Integer(int *x) : x_(x) { }
  ~Integer() { delete x_; }

  bool operator<(const Integer &rhs)
   const {
    return *x_ < *(rhs.x_);
  }
 private:
  int *x_;
};

main() {
  vector<Integer> v;
  Integer i1(new int(2));
  Integer i2(new int(3));

  v.push_back(i1);  // ok...
  v.push_back(i2);  // hmm....
  // much pain...
  sort(v.begin(), v.end());
}
```

What we really want...

A smarter wrapper

- contains a pointer, similar to the last slide
- overrides the copy constructor, assignment operator
  - to track # of copies of the wrapped pointer that have been made
  - a “reference count”
- has a smart destructor
  - decrements the reference count
  - calls delete if reference count falls to zero
- overrides `->` and `*` so it feels like a pointer

smart pointers

We'll pick this topic up after the midterm. :)

Exercise 1

Design a class hierarchy to represent shapes:

- examples of shapes: Circle, Triangle, Square

Implement methods that:

- construct shapes
- move a shape (i.e., add (x, y) to the shape position)
- returns the centroid of the shape
- returns the area of the shape
- Print( ), which prints out the details of a shape

Exercise 2

Implement a program that:

- uses your exercise 1
  - constructs a vector of shapes
  - sorts the vector according to the area of the shape
  - prints out each member of the vector
- notes:
  - to avoid slicing, you'll have to store pointers in the vector
  - to be able to sort, you'll have to implement a wrapper for the pointers, and you'll have to override the "<" operator

See you on Friday!