CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 25 – C++ virtual functions

(Thanks to Hal Perkins)

An important example

```
class A {
  public:
    void m1() { cout << "a1" << endl; }
    virtual void m2() { cout << "a2" << endl; }
};
class B : public A {
  public:
    void m1() { cout << "b1" << endl; }
    void m2() { cout << "b2" << endl; }
};

int main() {
  B* b = new B();
  A* a = b;
  a->m1();
  a->m2();
  b->m1();
  b->m2();
}
```

In words…

- A non-virtual method-call is *resolved* using the (compile-time) type of the *receiver* expression
- A virtual method-call is *resolved* using the (run-time) class of the *receiver* object (what the expression evaluates to)
  - Like in Java
  - Called "dynamic dispatch"
- A method-call is virtual if the method called is marked **`virtual`** or overrides a virtual method
  - So "one virtual" somewhere up the base-class chain is enough, but it's probably better style to repeat it

More on two method-call rules

- For software-engineering, virtual and non-virtual each have advantages:
  - Non-virtual – can look at the code to know what you're calling (even if subclass defines the same function)
  - Virtual – easier to extend code already written
- The implementations of virtual/non-virtual are the same and different:
  - Same: a methods is a function with one extra argument: *this* (pointer to receiver)
  - Different:
    - Non-virtual: linker can plug in code pointer
    - Virtual: At run-time, look up code pointer via "secret field" in the object

Destructors revisited

```
class B : public A { ... }
...
B * b = new B();
A * a = b;
delete a;
```

- Will `B::~B()` get called (before `A::~A()`)?
- Only if `A::~A()` was declared **`virtual`**
  - Rule of thumb: Declare destructors virtual
    - (more precise: declare destructors virtual if you use the base class polymorphically)

Downcasts

- BaseClass* a = new DerivedClass()   // implicit upcast
- DerivedClass* b = (DerivedClass) a;   // downcast

Old news:

- C pointer-casts: unchecked; better know what you are doing
- Java: checked; may raise **`ClassCastException`** (checks "secret field")

New news:

- C++ has "all the above" (several different kinds of casts)
  - `static_cast, dynamic_cast, reinterpret_cast…`
  - Worth learning about the differences on your own
- If you use single-inheritance **and know what you are doing**, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts

An example inspired by hw7

```
_w = newwin…
MapEntity map[WIDTH][HEIGHT];

void draw_map() {
  for (int x=0; x<WIDTH; x++) {
    for (int y=0; y<HEIGHT; y++) {
      mvaddch(_w, y, x, map[x][y].symbol())
    }
  }
}
```

Pure virtual methods

A C++ “pure virtual” method is like a Java “abstract” method.

- Some subclass must override because there is **no definition in base class**
- Unlike Java, no need/way to mark the class specially
- to declare a pure virtual in the base class:

```
class C {
  virtual t0 m(t1,t2,...,tn) = 0;
  ...
};
```

- override as usual in subclass class
- Side-comment: with multiple inheritance and pure-virtual methods, C++ has no need for a separate notion of Java-style interfaces (as a Class with only pure virtual functions)

C++ summary

- Lots of new syntax and gotchas, but just a few new concepts:
  - Objects vs. pointers to objects
  - Destructors
  - virtual vs. non-virtual
  - pass-by-reference
- more stuff as there is time:
  - why objects are better than code-pointers – a.k.a. “coding up object-like idioms in C”
  - templates (serve a similar function as java generics), exceptions, and operator overloading

Quick break

- Why might pointers to functions be useful?

Function pointers

- "Pointers to code" are almost as useful as "pointers to data". (But the syntax is painful in C.)
- (Somewhat silly) example:

```
void app_arr(int len, int * arr, int (*f)(int)) {
   for(int k = 0; k < len; k++)
          arr[k] = (*f)(arr[k]);
}
int twox(int i) { return 2*i; }
int sqr(int i)    { return i*i; }
void twoXarr(int len, int* arr) {app_arr(len,arr,&twox);}
void sqr_arr(int len, int* arr)  { app_arr(len,arr,&sqr); }
```

- Now functions are "first-class citizens": they can be passed around as data
- app_arr is a *higher-order function*, that is, it takes a function as an argument

C function-pointer syntax

- C syntax: painful and confusing. Rough idea: The compiler "knows" what is code and what is a pointer to code, so you can write less than we did on the last slide:

```
arr[k] = (*f)(arr[k]);
  ⇒ arr[k] = f(arr[k]);
app_arr(len,arr,&twoX);
  ⇒ app_arr(len,arr,twoX);
```

- Examples: Compute integral with function (pointer) to integrate and bounds as parameters (int1.c, int2.c)

What is an object?

First Approximation

- An object consists of data and methods
  - Provides the correct (conceptual) model
  - Easy to explain
- But…
  - Doesn't make engineering sense — we don't want to replicate the (same) method bodies (function code) in every object

What is an object?

Second Approximation

- An object consists of data and pointers to methods
- The compiler adds an additional, implicit “this” parameter to every method holding a reference to the receiver object
  - Gives the method a way to refer to the instance variables of the correct receiver object
  - Actual method (function) code has no other connection to any particular object
- Avoids code duplication
- See BAccount1.c (C version of Baccount.cpp)

But. . .

- Still wastes space for pointers to every class function in every object, particularly if there is relatively little instance data, or if the class has a large number of methods

What is an object?

How it's really done (C++, Java, et al):

- There is a single "virtual function" table (vtable) for each class containing pointers to the methods of that class.
  - This is static, constant class data – does not change during execution; initialized at load/startup time
- An object consists of data and a pointer to its class vtable
- Method calls are indirect through the vtable
- Each method still has an implicit this parameter that refers to the receiving object
- Avoids code duplication
- Avoids method pointer duplication
- Costs an indirect pointer lookup during each function call
- Example: BAccount2.c

Inheritance and overriding

Basic ideas:

- We have a vtable for every class and subclass
- The vtable for a subclass points to the correct methods — either ones belonging to the base class that are inherited, or ones belonging to the subclass (added or overriding)
- Key idea: The initial part of the vtable for a subclass points to the methods that are inherited or overridden from the base class in exactly the same order they appear in the base class vtable
  - So compiled code can find the correct method at the same offset in the vtable whether it is overridden or not
- Use casts as needed to adjust references up and down the inheritance chain