CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2008

Lecture 25— C++ Overriding and Wrap-up;

Manual Memory-Management Idioms

Method overriding, part 1

If a derived class defines a method with the same name and argument types as one defined in the base class (perhaps because of an ancestor), it *overrides* (i.e., replaces) rather than *extends*.

If you want to use the base-class code, you specify the base class when making a method call.

- Like `super` in Java (no such keyword in C++ since there may be multiple inheritance)

Warning: the title of this slide is *part 1*.

Casting and subtyping

An object of a derived class *cannot* be cast to an object of a base class.

- For the same reason a `struct T1 { int x, y, z; }` cannot be cast to type `struct T2 { int x, y; }` (different size)

A *pointer to* an object of a derived class *can* be cast to a pointer to an object of a base class.

- For the same reason a `struct T1 *` can be cast to type `struct T2 *` (point to a prefix of the memory)
- (Story not so simple with multiple inheritance)

After such an *upcast*, field-access works fine (prefix), but what do method calls mean in the presence of overriding...

An important example

```
class A {
public:
  void m1() { cout << "a1"; }
  virtual void m2() { cout << "a2"; }
};
class B : public A {
  void m1() { cout << "b1"; }
  void m2() { cout << "b2"; }
};
void f() {
  A* x = new B();
  x->m1();
  x->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 (see CSE341):

- Non-virtual – can look at the code to know what you're calling
- Virtual – easier to extend code already written

The implementations are the same and different:

- Same: Methods just become functions 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; usually what you want.

Downcasts

Old news:

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

New news:

- C++ has "all the above" (several different kinds of casts)
- 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.
- Worth learning about the differences on your own

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.
- Makes sense with dynamic dispatch.
- Unlike Java, no need/way to mark the class specially.
- Funny syntax in base class; override as usual:

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

- Side-comment: with multiple inheritance and pure-virtual methods, no need for a separate notion of Java-style interfaces.

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
- Plus all the stuff we didn't get to, especially templates, exceptions, and operator overloading.
- Maybe later: why objects are better than code-pointers / coding up object-like idioms in C

Memory-management idioms

Review: Java and C memory-management rules

Idioms for memory-management:

- Garbage collection
- Unique pointers
- Reference Counting (later)
- Arenas (a.k.a. regions) (later)

Note: Same "problems" with file-handles, network-connections, Java-style iterators, ...

Note: *Idioms* are not tools, rules, or language-features, rather "common time-tested approaches"

- Those are important to learn too.

Java rules

- Space for local variables lasts until end of method-call, but no problem because cannot get pointer into stack
- All "objects" are in the heap; they conceptually live forever.
  - Really get reclaimed when they are *unreachable* (from a stack variables or global variable).
  - Static fields are global variables.

Consequences:

- You rarely think about memory-management.
- You *can* run out of memory without needing to (e.g., long dead list in a global), but you still get a *safe* exception.
- No dangling-pointer dereferences!
- Extra behind-the-scenes space and time for doing the collection.

C rules

- Space for local variables lasts until end of function-call, may lead to dangling pointers into the stack.
- Objects into the heap live until `free(p)` is called, where `p` points to the beginning of the object.
- Therefore, unreachable objects can never be reclaimed.
- `malloc` returns `NULL` if it cannot find space.
- If you do the following, HYCSBWK:
  1. Call `free` with a stack pointer or middle pointer.
  2. Call `free` twice with the same pointer.
  3. Dereference a pointer to an object that has been freed.
- Usually 1–2 screw up the `malloc`/`free` library and 3 screws up an application when the space is being used for another object.

Garbage Collection for C

Yes, there are garbage collectors for C (and C++)!

`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

- redefines `free` to do nothing
- unlike a Java GC, *conservatively* thinks an `int` might be a pointer.

Questions to ask yourself in any application:

- Why do I want manual memory management?
- Why do I want C?

Good (and rare!) answers against GC: Tight control over performance; even short pauses unacceptable; need to free reachable data.

Good (and rare!) answers for C: Need tight control over data representation and/or pointers into the stack.

Other answer for C: need easy interoperability with lots of existing code

Why is it hard?

This is not really the hard part:

```
free(p);
...
p->x = 37; // dangling-pointer dereference
```

These are:

```
p = q; // if p was last reference and q!=p, leak!
```

```
lst1 = append(lst1,lst2);
free_list(lst2); // user function, assume it
                 // frees all elements of list
length(lst1); // dangling-pointer dereference
              // if append does not copy!
```

There are an infinite number of *safe idioms*, but only a few are known to be simple enough to get right in large systems...

Idiom 1: Unique Pointers

Ensure there is exactly one pointer to an object. Then you can call `free` on the pointer whenever, and set the pointer's location to `NULL` to be "extra careful".

Furthermore, you *must* free pointers before losing references to them.

Hard parts:

1. May make no sense for the data-structure/algorithm.
2. May lead to extra space because sharing is not allowed.
3. Easy to lose references (e.g., `p=q;`).
4. Easy to duplicate references (e.g., `p=q;`) (must follow with `q=NULL;`).
5. A pain to return unfreed function arguments.

Relaxing Uniqueness

This does not preserve uniqueness:

```
void g(int *p1, int*p2) { ... }
void f(int *p1, int*p2) {
  if(...)
    g(p1,p1);
  else
    g(p1,p2);
  ...
  free(p1);
  free(p2);
}
```

Wrong if g frees an argument or stores an alias somewhere else.

Also notice true-branch creates aliases just in the callee.

Relaxing Uniqueness

Instead, have some "unconsumed" pointers:

- Callee won't free them
- They will be unique again when function exits

More often what you want, but changes the contract:

- Callee must *not* free
- Callee must not store the pointer anywhere else (in a global, in a field of an object pointed to by another pointer, etc.)