Data Abstraction and Object Orientation

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Object-Oriented Programming

- Control or PROCESS abstraction (subroutines!) is a very old idea
- Data abstraction (Object-Oriented (OO)) is somewhat newer
  - An Abstract Data Type is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation
  - How did we do it before OO?
    - We created and manipulated a data structures.
    - "Manager functions": pre-OO formalism

      l = list_create()

      list_append(l, o)
      - C libraries still use this paradigm: GTK, Linux Kernel, etc.
- In this chapter, object = instance of a class.

Object-Oriented Programming

- Why abstractions?
  - easier to think about - hide what doesn't matter
  - protection - prevent access to things you shouldn't see
  - plug compatibility
    - replacement of pieces, often without recompilation, definitely without rewriting libraries
    - division of labor in software projects

Object-Oriented Programming

- We talked about data abstraction some back in the unit on naming and scoping
  - historical development of abstraction mechanisms:
    - Static set of variables — Basic
    - Locals — Fortran
    - Statics — Fortran, Algol 60, C
    - Modules — Modula-2, Ada 83
    - Module types — Euclid
    - Objects — Smalltalk, C++, Eiffel, Java Oberon, Modula-3, Ada 95

Object-Oriented Programming

- The 3 key factors in OO programming
  - Encapsulation (data hiding)
  - Inheritance
  - Dynamic method binding

Object-Oriented Programming

- OOP is currently ascendant. It cross-cuts paradigms:
  - Imperative OO (C++, Java, C#, Python, etc.)
  - Functional OO (Ocaml, CLOS, etc.)
  - Logical OO (Flora-2)
- OO adds:
  - Convenient syntax,
  - Inheritance,
  - Dynamic method binding,
  - Encapsulation.

Object-Oriented Programming

- Benefits of OOP:
  1. Reduced Conceptual Load
     - Give a concept a name
  2. Fault containment
     - Object can enforce an interface that must be followed
  3. Independence
     - Reusability

Object-Oriented Programming

- The language needs a way of defining a class:
  - Name,
  - Superclasses (incl. Interfaces),
  - Fields,
  - Methods.

  Fields + Methods = Members of the class
- Note: classes do not need to be defined in a single file:
  - C++ allows a definition to be split up over multiple files,
  - Java allows more than one class per file (one is public)

Object-Oriented Programming

- A class needs ways to:
  - Allocate objects (constructors and the `new` operator)
    - Some languages allow to allocate in all 3 areas, not just heap
  - Change fields
  - Invoke methods

Protection

- OO supports data hiding / protection:
  - Keep implementation details from leaking into the larger program
- The 4 kinds of Visibility protection in Java:
  - Public
  - Protected
  - Default (visible to classes in same module)
  - Private
- Others are possible: C++ has Friend
- Private inheritance: we can inherit code from a parent, without advertising that we are substitutable for that superclass

Protection

- A C++ friend class in C++ can access the "private" and "protected" members of the class in which it is declared as a friend:

```
class B {
    friend class A; // A is a friend of B
private:
    int i;
};
class A {
public:
    A(B b) { // the object has to be passed as a parameter to the function
        b.i = 0; // legal access due to friendship
    }
};
```

Encapsulation and Inheritance

- Example:
  class circle : public shape { …
  anybody can convert (assign) a circle* into a shape*

  class circle : protected shape { …
  only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

  class circle : private shape { …
  only members and friends of circle can convert (assign) a circle* into a shape*

Accessors

- Field access is bad in OOP: allows an external program to change any value.
- Accessors: small subroutines that are used to access.
  - constrain how we can change our implementation.
  - getters + setters = accessors and mutators for all the fields.
- Less convenient than fields, but preferred for safety.
- Good compromise: declare properties, which look like fields, but invoke methods (C#, Python)

Constructors and Destructors

- Objects often need to be initialized before they are used.
  - This is because Objects represent things with semantics.
- Constructors are used to initialize objects.
- Languages often support multiple constructors.
  - Overloading on type and # of arguments.
  - Named constructors.
    - Example: a Coordinate class.
      - double x, double y OR
      - double angle, double radius
- Important constructors:
  - Zero Argument,
  - Copying: must take a reference.

In C++ (objects use value semantics):
class C { ... }
C a; <- calls 0-argument constructor.
C b = a; <- calls copying constructor.
Differs from: C b;
b = a;

Constructors and Destructors

- In languages where objects are values, those values need to be initialized as part of our constructor. (By calling their constructor.)

```
class A : public B {
  C d;
  A () { }
}
```

  - Leads to 2 further constructor calls: to B, then C.

Constructors and Destructors

- Destructors:
  - Called when an object goes away, to free up resourced used by it.
  - Interact poorly with GC, especially in the presence of cycles.

Inheritance

- We can create a subclass that inherits from a superclass.
  - Inherits fields and methods from base class.
- Do we need a root class?
  - C++ is fine without it.
  - Java, Smalltalk, C#, etc have one.
- OO requires scoping rules to determine where we look for fields:
  - The instance,
  - The class (statics),
  - Superclasses,
  - Global Scope,
  - The method (local vars.)

Inheritance

- General-purpose base classes.
  - When we don't have generics in a language, use a base class and subclasses to fake it.
  - Not as good as generics/Parametric Polymorphism, because it means that we have to use multiple classes, one for each type, e.g., IntList, PersonList, etc.

Multiple Inheritance

- In C++, you can say

  class professor : public teacher, public researcher {

  ...

  }

  Here you get all the members of teacher and all the members of researcher

  - If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them

Multiple Inheritance

- You can of course create your own member in the merged class

```
professor::print () {
    teacher::print ();
    researcher::print (); …
}
```

Or you could get both:

```
professor::tprint () {
    teacher::print ();
}
professor::rprint () {
    researcher::print ();
}
```

Nested Inner Classes

- Inner Class = class defined inside another class:
  - Need to decide which fields such a class sees
    - Nothing - no special relationship to outer class (Python)
    - Statics Only (C++/C#).
    - Associated with every instance (Java)
      - Needs link to instance of enclosing class.

Metaclasses

- In some languages, a class is an object (Python, Flora-2, not Java)
  - Class of the class = metaclass
    - Constructing a metaclass

Extension Methods

- It's possible to extend a class without inheriting from it
  - C#: Extension method:

```
public static string[] split(this string s) {   ....
}
```

  - Can use like:

```
s = "Hello World";
s.split()
```

  - Really syntactic sugar: Defines a new class with a static method, and then calls static method.

User-Defined Objects

- How do we store the fields in objects?

```
class A {
    int v;
    int w;
     String s;
}
```

- The object variable = (reference == pointer == address)
- The field == constant offset from pointer.
  - 1 opcode on all but most RISC CPUs.
- What about inheritance?
  - The object contains all fields starting from root ancestor.
- An alternative is to represent objects as maps (Python, Javascript): key = field name, maps to field values

User-Defined Objects

- Objects as maps (Python, Javascript): key = field name, maps to field values
  - Pro: allows for reflection to be done easily.
  - Pro: allows for adding of fields at runtime.
  - Con: requires an expensive map lookup per field.

Method Binding

- Can use a subclass in place of the superclass: Subtype polymorphism:

  class C: def foo(): …

  class D(C): def foo(): …

- When both define method foo, which method do we call:
  - static method binding - needs static typing - call method based on static type - method call == function call
  - dynamic method binding - use class of the object - call from most derived type → more expensive.
    - Dynamic == virtual methods in C++.
    - Dynamic == default in Java.
    - C# requires you to decide if you're overriding or replacing.

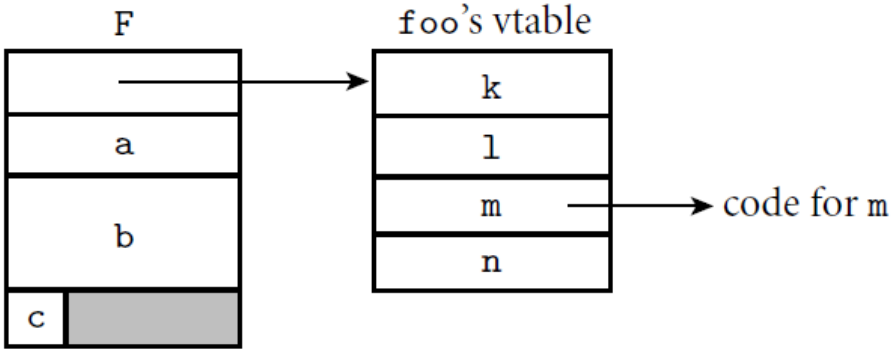
Method Binding

- Pure virtual = No implementation for method.
  - Abstract in Java.
  - Means that we won't be able to instantiate the class.
  - Java Interfaces = Really Classes without non-abstract methods
    - Also, no base class == no diamond problem.
- Dynamic method binding:
  - Virtual Dispatch

    class C: a() b()

    class D(C): c() b()
  - The subclass includes methods in the same order = the vtable
  - Multiple inheritance problem.
    - no-canonical order to put the entries in.
    - multiple vtables.

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```



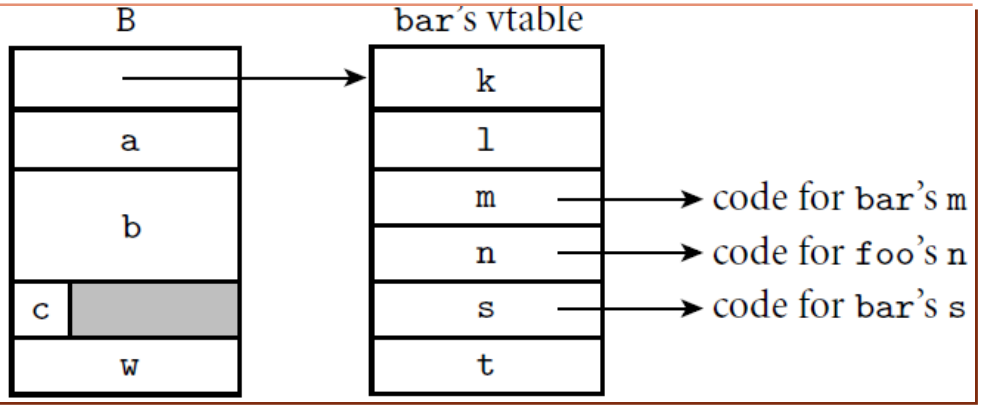
The representation of object F begins with the address of the vtable for class foo.

All objects of this class will point to the same vtable.

The vtable itself consists of an array of addresses, one for the code of each virtual method of the class.

The remainder of F consists of the representations of its fields.

```
class bar : public foo {
    int w;
public:
    void m();  //override
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



Single inheritance

```
class Shape:
  def name():     print "Shape"
  def color():    print "Blue"
class Rectangle(Shape):
  def name():     print "Rectangle"
  def color():    print "Red"
class Square(Shape)
  def name():     print  "Square"
Square s = new Square()
Rectangle r = s
Shape sh = s
s.name()
r.name()
sh.name()
```

Static Method Binding: Square Rectangle Shape

Dynamic Method Binding: Square Square Square

Garbage Collection

- Automatic heap memory management.
  - Alternative is to require explicit deletes.
  - What is garbage? What is not garbage?
    - Set of roots: - Registers - Stack - Statics - etc.
    - Objects reachable from roots are live. - Recursively search objects for other objects. - Some objects don't need to be searched. (Strings, Arrays of basic types, etc.)
    - Unreachable objects are dead, can be freed.
  - Reference counts: Each object contains a field giving the number of objects (incl. stack frames) referring to it.
    - When a reference is taken to object, inc refcount.
    - When a reference is removed, dec refcount.
    - When drops to 0, deallocate.
  - Problem with circular references.

Garbage Collection

- Mark and Sweep:
  - Walk references, flag useful objects.
    - Requires space in header for flag.
  - Walk objects, find unflagged, dealloc.
    - Requires ability to walk objects.
  - Interesting technique: Pointer reversal.
    - Stores path to parent var in place of pointer to child obj.
- Stop and Copy (Compacting)
- Generational Collection:
  - Objects live short or long time.
  - Old objects rarely refer to much newer objects.
  - Requires MMU or some sort of write barrier.
- Incremental GC

Object-Oriented Programming

- SMALLTALK is the canonical object-oriented language
  - It has all three of the characteristics listed above
  - It's based on the thesis work of Alan Kay at Utah in the late 1960‘s
  - It went through 5 generations at Xerox PARC, where Kay worked after graduating
  - Smalltalk-80 is the current standard