

=====

Data Abstraction and Object Orientation

Recall discussion of scoping and encapsulation from Chap. 3

Historical development of abstraction mechanisms is roughly:

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, C#, Scala, Swift, Ruby, Python, ...
object-based	Self, JavaScript
type extensions	Oberon, Modula-3, Ada 95

Except that objects originated with Simula 67 but were otherwise ignored for most of the '70s, while people continued to refine modules (Simula 67 didn't have data hiding).

The 3 key factors in OO programming (as codified by Wegner):

encapsulation (data hiding)

modules do this, too—e.g., packages in Java and namespaces in C++—but they don't usually give you multiple instances

inheritance

dynamic method binding

this is crucial and often doesn't get looked at carefully—and the default in C++ is different from what you may be used to in Java

Visibility rules

Public and Private parts of an object declaration/definition.

(Some other options in some languages—e.g., package in Java or protected in Java or C++ [which treat them slightly differently])

C++ distinguishes among

public	visible to anybody
protected	visible only to this class and its descendants
private	visible only to this class

Default is public for structs and private for classes.

C++ base classes can also be public, private, or protected. E.g.

```
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*
```

Java rules are slightly different:

public:	visible to anybody
(package)	visible only to this class and classes in the same package
protected	visible only to this class, its descendants, and classes in the same package
private	visible only to this class

Package is the default; it's what you get with no specifier ('package' isn't a keyword).

Recall that a **declaration** introduces a name, and enough information about it to allow it to be **used**, at least in limited contexts. A **definition** provides enough information for the compiler to implement the object.

2 reasons to put things in the declaration:

(1) so programmers know how to use the object

Many module-based languages separate modules into pieces: one for the declaration and one for the definition, usually placed in separate files for the purpose of separate compilation.

Declaration modules may be compiled into symbol table data, or they may be textually "included" into user and definition modules. The latter option is a more structured, formal version of the typical ".h" and ".c" files of C.

(2) so the compiler knows how to generate code for uses of the object

At the very least the compiler needs to know how to invoke the methods of the object. If it must allocate space for the object it also needs to know its size (this is a particular challenge in languages with a value model of variables). To figure out the size, the compiler will often need to know information that the programmer does **not** need to know, such as the types (sizes) of private data members.

This can get awkward. It's part of the reason why some newer languages (e.g., Java & C#) dispense with separate declaration and implementation modules. The compiler peruses the single body of code and extracts what users of it need. If you want teams to develop in parallel, you start by creating skeleton versions, which each team uses as an interface specification while they flesh out their own part.

A few C++ and Java features you may not have learned:

C++ destructors

These are the opposite of constructors. Mostly they're needed for explicit space management. Java can get by without them because it has garbage collection. Given the availability of destructors, C++ and Rust programmers have invented other clever uses for them, e.g. for locking:

```
std::mutex my_lock;
...
{
    std::lock_guard m(&my_lock);
    // m is a dummy object whose constructor acquires
    // the lock passed as an argument, and then keeps a
    // pointer to this lock in a private data member.

    // code that we'd like to have executed atomically
    // at end of scope, m's destructor automatically releases my_lock
}
```

This idiom is sometimes called RAI: "resource acquisition is initialization."

unexpected constructor calls

Constructors are relatively straightforward in a language with a reference model of variables. With a value model, however, we have to arrange to call them at elaboration time for declared objects and sometimes (for temporaries) in the middle of expressions as well.

Consider an object constructed in an argument list:

```
void foo(my_class o) { ... }
...
my_class o2(args);    // constructed here
```

```
foo(o2);                // passed by value
foo(my_class(args));    // constructed w/in arg list
```

Because `foo`'s argument is passed by value, the calling sequence needs to invoke the copy constructor. In the first call, this makes good sense. In the second call it seems like a shame, because what's being copied is a temporary that will be destroyed immediately after being passed. (If you put print statements or other side effects in the constructor and destructor [bad idea!], you may be able to see this happen. Or not: the compiler is allowed to elide calls to copy constructors when the copied object will never be used again.)

A similar situation happens when returning:

```
foo e;                    foo f(args) {
...                        foo rtn;
e = f(args); // copy?      ...
                           return rtn; // copy?
```

Returning `rtn` from `f` may entail 0, 1, or 2 copies, depending on how smart the compiler is. The copies at both ends can be eliminated “for sure” in C++11 using **move constructors**, which use **rvalue references** (indicated in the argument list of the constructor itself with a double ampersand `&&`). A move constructor will be used whenever the compiler knows that the copied-from object will never be used again. Typically that constructor will modify the copied-from object's state so that its destructor won't free stuff we still need.

rvalue references are also used for **move assignment** methods. Programmers are free to use them for other purposes, as well, but this requires great care—it's easy to end up with bugs analogous to dangling references.

initialization

Straightforward in Java because all object-typed variables are references. Data members of object types are simply initialized to null; you specify arguments to the constructors when you call `new`, explicitly. Arguments for the superclass constructor, if any, can be provided in a pseudo-call, which must be the first statement of the constructor:

```
public child(a, b, c) {
    super(a, b);
    ...
}
```

If you don't provide the `super()` call, the compiler inserts a call to the zero-arg constructor (which must exist).

Harder in C++ (or Eiffel, or Ada, ...) because of expanded (elaborated) objects—**not** referenced w/ pointers: actually **there**, “in place”.

C++ requires that every object be initialized by a call to a constructor. The rules for doing this for expanded objects are quite complex. For example:

objects as members

```
foo::foo(args) : base(args0),  
               member1(args1), member2(args2) { ...    // C++
```

`args0`, `args1`, `args2`, etc. need to be specified in terms of `args`. The reason these things end up in the header of `foo` is that they get executed **before** `foo`'s constructor does, and the designers consider it good style to make that clear in the header of `foo::foo`.

Commonly the arg lists are singletons (for copy constructors), and you might be tempted to replace the code

```
foo::foo(a, b, c) : member1(a), member2(b) { ...
```

with

```
foo::foo(a, b, c) {  
    member1 = a;  
    member2 = b;
```

but this is **not** the same: the latter option calls zero-arg constructors for `member1` and `member2` **before** calling `foo::foo()`, and **then** calls `operator=`.

Note that the constructors for base classes are called **before** the constructors for children (with multiple inheritance, they're called in the order the specified in the header of the child). Destructors for base classes are called **after** the destructors for children.

In general, the C++ compiler will generate default versions of any needed zero-arg, copy and move constructors (and `operator=`) that weren't provided by the programmer. These just construct their sub-members and, for the copy case, copy members of built-in types. Automatic generation can be disabled by explicitly **deleting** the constructor:

```
class glarch {
public:
    glarch() = delete;
```

In this case, if a zero-arg constructor is needed, the compiler will produce a compile-time error message.

initialization v. assignment — not the same!

```
foo::operator=(&foo) v. foo::foo(&foo)
```

```
foo b;
    // calls no-arg constructor
foo f = b;
    // calls one-arg "copy constructor".
    // This is syntactic sugar for foo f(b);
```

```
foo b, f;    // calls no-arg constructor
f = b;       // calls operator=
```

classes as members

Called “inner” classes in Java.

Q: if Inner is a member of Outer, can Inner’s methods see Outer’s members, and if so, which instance do they see?

```
class A {
    int i;
    class B {
        method foo()
            i := 3    // is this allowed?
```

C++ and C# say no, inner classes can see only static fields of the parent. Java says yes, instances of inner class belong to an instance of the outer class, and can access data members of that instance. This capability provides much (most?) of the power of nested subroutines, which C++ and Java lack.

Java can be thought of as having four kinds of inner classes.

Static inner classes are only “sorta” inner: they have limited visibility, but they don’t need an outer class instance to exist.

Member classes (class instance within a class instance) contain a hidden reference to the parent object.

Local classes (class instance within a method of a class instance) contain the hidden reference AND copies of the method's parameters and final locals (but **not** the non-final locals—so there's still no static chain).

Anonymous inner classes are like local classes, but can have only one instance.

virtual functions

Virtual functions provide C++'s dynamic method binding: you don't know at compile time what type the object referred to by a variable will be at run time.

Key question: if `child` is derived from `parent` and I have a `parent* p` (or a `parent& p`) that points (refers) to an object that's actually a `child`, what member function do I get when I call `p->f` (`p.f`)? By default in C++ I get `p's f`, because `p's` type is `parent*`. But if `f` is a virtual function, I get `c's f`.

Also note: If a C++ virtual function has a "`0`" body in the parent class, then the function is said to be a "pure" virtual function and the parent class is said to be "abstract". In Java you prepend the method declaration with the "abstract" keyword. You can't declare objects of an abstract class; you have to declare them to be of derived classes. Moreover any derived class **must** provide a body for the pure virtual function(s) (unless it too is supposed to be abstract).

Simula had (abstract) virtual functions all the way back in 1967. In most modern OO languages, (Java, Scala, Ruby, Python, ...) **all** member functions are virtual. Interestingly, C# is like C++: non-virtual is the default.

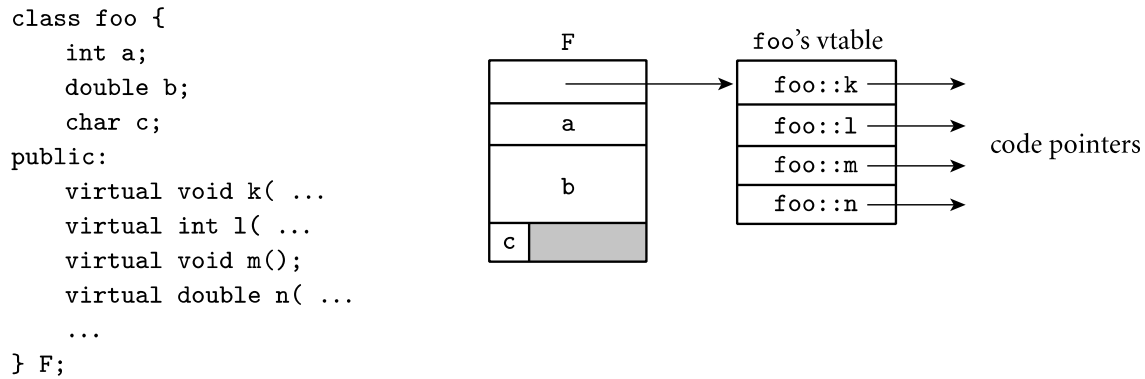
BTW: note that inheritance does not obviate the need for generics. You might think: hey, I can define an abstract list class and then derive `int_list`, `person_list`, etc. from it, but the problem is you won't be able to talk about the elements because you won't know their types. That's what generics are for: abstracting over types. See the lecture on polymorphism (generics).

===== Implementation of classes

Data members of classes are implemented just like structs (records). With (single) inheritance, derived classes have extra fields at the end. A pointer to the parent and a pointer to the child contain the same address—the child just knows that the struct goes farther than the parent does.

Non-virtual functions require no extra space at run time; the compiler just calls the appropriate version, based on type of variable. Member functions are passed an extra, hidden, initial parameter: 'this' (called 'current' in Eiffel and 'self' in Smalltalk).

Virtual functions are the only thing that requires any trickiness. They are implemented by creating a dispatch table (“vtable”) for the class and putting a pointer to that table in the data of the object. Objects of a derived class have a different vtable. In that table, functions defined in the parent come first, though some of the pointers point to overridden versions.



The C++ philosophy is to avoid run-time overhead whenever possible. (Sort of the legacy from C). That’s why non-virtual functions are the default. Most other OO languages have more run-time support.

Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info. The standard implementation technique is to put a pointer to the type info at the beginning of the vtable. Of course you only have a vtable in C++ if your class has virtual functions. That’s why you can’t do a `dynamic_cast` on a pointer whose static type doesn’t have virtual functions.

Interface (trait, mix-in) inheritance

Simpler to implement than true multiple inheritance. Each class can have one “real” parent and an arbitrary number of **interfaces**, each of which is fully abstract: no data members (other than statics); no non-pure-virtual methods.

In a dynamic language like Python or Ruby, interface routines are simply included in the (dynamically searched) method table. With just-in-time compilation, one common implementation adds a dictionary of interface methods to the end of the vtable (or to an *itable* pointed at by the vtable). The first call through a given call site does a (slow) lookup in the table and then *modifies the code* to embed what it found, for use the next time execution reaches the same point in the code. If the `this` parameter is in `r1` (and arguments have been computed into various other registers or the stack), then call site code can look something like this:

```

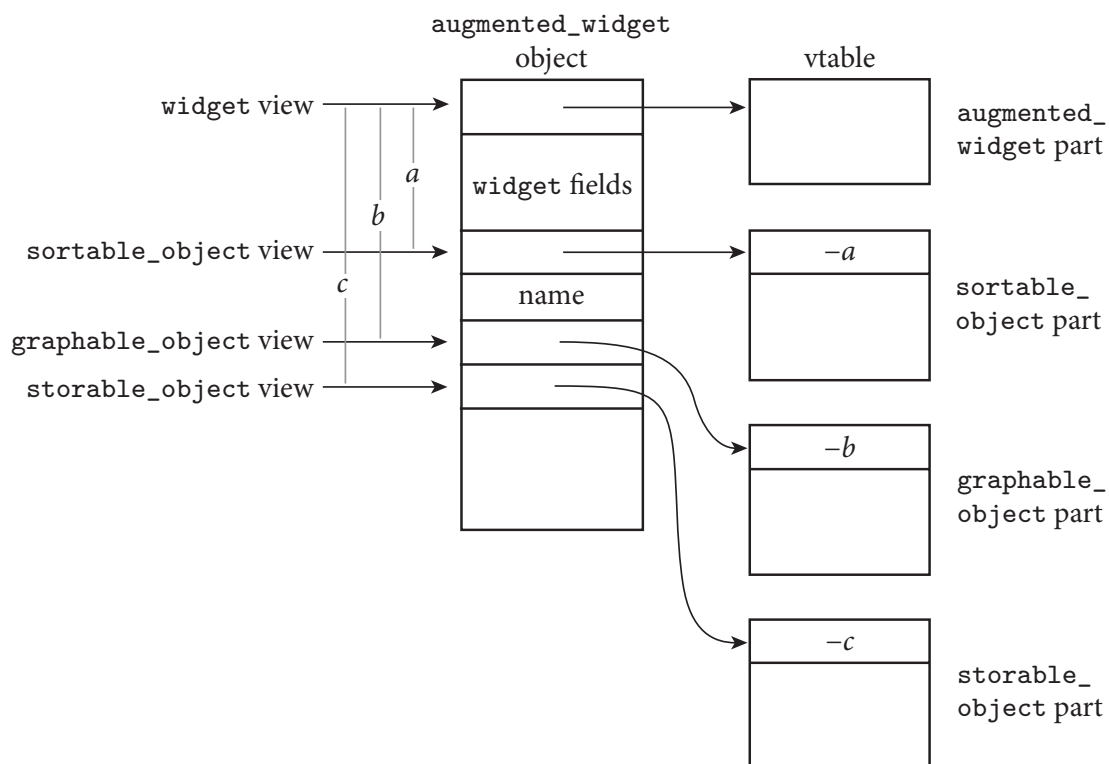
— assume object address is in r1
— and method args are in parameter-passing registers
r2 := *r1          — vtable address
if r2 = cached_class† goto cache_hit
r3 := interface id
r4 := method id
call dispatch      — slow path: ordinary dispatch
goto next          — branch to after the call
cache_hit:
  call cached_method† — fast path
next:              — code after call

```

†NB: `cached_class` and `cached_method` are embedded in the code

If the cache misses too many times, the VM may update it. This depends on self-modifying code, which is possible in a VM but generally not in security-conscious modern systems with ahead-of-time compilation. If `cached_class` and `cached_method` have to be moved to the data segment, the fast path changes from three instructions to five, and a static approach to vtables becomes more attractive.

The alternative mechanism includes an extra vtable pointer among the data fields of a C object for each interface implemented by C. Here's a possible layout for an `augmented_widget` object that implements `sortable_object`, `graphable_object`, and `storable_object` interfaces:



Each interface vtable begins with a “this correction” value that indicates the distance from the (class-specific) interface vtable pointer back to the beginning of the object. Now the calling sequence looks something like this:

```
— assume object address is in r1
— and method args are in parameter-passing registers
r2 := *r1           -- vtable address
r3 := *r2           -- this correction
r3 += r1            -- this
call *(r2+k)        -- method address
```

Note that in the self-modifying, JIT-compiled implementation, a variable of interface type holds a reference to the concrete object. The run-time system caches, in the code, the expected concrete type of the object and the address of its implementation of the method in question. In the ahead-of-time compiled implementation, a variable of interface type holds a reference to the *interface vtable field* of the object. It reconstructs the address of the concrete object using the this correction in the vtable.

A third mechanism may be used in compiled languages like Rust. Recall that a trait may be added to a pre-existing type that has already been used in code that cannot be recompiled. A variable of trait type may then be represented by a pair of references—one to the underlying object and another to the trait vtable. If r1 contains a pointer to this variable, then a call to, say, the 3rd method of the trait would look something like this on a 64-bit machine:

```
r2 := *(r1+8)       -- vtable address
r1 := *r1           -- address of object
call *r2            -- this in r1; other args in other regs
```

NB: classic Java also allows static fields in Interfaces.

Starting with Java 8, Interfaces can have

- static methods (straightforward: no access to this)

- default methods

 - Designed to allow extension of an interface without rewriting all existing uses of that interface.

 - Implementation is a little tricky.

 - no access to members other than methods & static fields of interface itself
 - does** need access to vtable, however: for each class that needs the default code, the compiler generates a static, class-specific forwarding routine to set up the right this parameter.

For true multiple inheritance, see the PLP companion site.

=====

Smalltalk and Scripting Languages

Long considered the canonical object-oriented language. Based on the thesis work of Alan Kay at Utah in the late 1960's. Went through 5 generations at Xerox PARC, where Kay worked after graduating. Active development ended with Smalltalk-80.

Smalltalk is interesting in its own right, and also for historical reasons. It carried the OO torch from the Simula of the 60s to the C++ of the 80s and 90s and the Java, C#, Python, Ruby, etc. of the 90s and 00s. Smalltalk is HEAVILY integrated into its programming environment. Things like typefaces are part of the syntax of the language.

Everything in Smalltalk is anthropomorphized. "3 + 4" is syntax for sending the message "+ 4" to the object 3, which returns a reference to the object 7. Even control flow is conceptualized as messages. For example:

```
total = 0
  ifTrue: [average <- 0]
  ifFalse: [average <- sum // total]
```

sends an "= 0" mesasage to the object total, which returns a reference to either the object TRUE or the object FALSE, which is then passed an "ifTrue: ... ifFalse ..." message. Similarly

```
count <- 0.
sum <- 0.
[count <= 20]
  whileTrue: [sum <- sum + count.
              count <- count + 1]
```

sends a "whileTrue: ..." message to a block that would return TRUE or FALSE if sent a "value: ..." message. Similarly

```
3 timesRepeat: [...]
```

```
1 to: 100 by: 10 do: [:i | total <- total + (a at: i)]
```

A simple example method for factorial, understood by integers:

```
factorial
  self = 0
  ifTrue: [^1].
```

```
self < 0
  ifTrue: [self error 'Factorial not defined']
  ifFalse: [^ self * (self-1) factorial]
```

The OO syntax (and semantics) of Objective C, Ruby, and Swift is highly reminiscent of Smalltalk.

In Ruby the expression

```
4 * 3 < 16
```

is equivalent to

```
4.*(3).<(16)
```

which in turn is equivalent to

```
4.send('*', 3).send('<', 16)
```

that is,

```
send a "'*', 3" message to 4
then send a "'<', 16" message to t,
  where t is what you got back from the first send
```

Note, however, that this is more than syntactic sugar: message notation evaluates left-to-right, without regard to traditional notions of precedence. So

```
16 > 4 * 3
```

is equivalent to

```
16.>(4).*(3)
```

or

```
16.send('>', 4).send('*', 3)
```

which groups as

```
(16.>(4)).*(3)
```

or

```
(16.send('>', 4)).send('*', 3)
```

which produces a run-time type error ("undefined method `*' for true:TrueClass"). If you want the equivalent of the infix evaluation order, you have to parenthesize explicitly:

```
16.>(4.*(3))
```

or

```
16.send('>', 4.send('*', 3))
```

As we saw in Chap. 6, Ruby has pass-a-lambda iterators:

```
sum = 0                                => 0
[ 1, 2, 3 ].each { |i| sum += i }      => [1, 2, 3] # array itself
sum                                    => 6
```

Here the (parameterized) brace-enclosed block is passed to the each method as a parameter.

There's also more conventional-looking syntax:

```
sum = 0
for i in [1, 2, 3] do      # 'do' is optional
  sum += i
end
sum
```

The for loop is syntactic sugar for a call to each.

Here's a more OO alternative:

```
sum = 0
1.upto 3 { |i| sum += i }
sum
```

or instead of using braces:

```
sum = 0
1.upto 3 do |i| sum += i end
sum
```

You can write your own iterators using 'yield'.

```
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
```

```
end
...
[1, 3, 5, 7, 9].find { |v| v*v > 30 } => 7
```

Think of `yield` as invoking the block that was juxtaposed (“associated”) with the call to the iterator.

Notice that we’ve defined a new method of the built-in `Array` class. (Actually, `Array` already has a `find` method, but we can redefine it, and it probably looks like this anyway.)

Blocks can also be turned into first-class closures, with unlimited extent:

```
def nTimes(aThing)
  # note lack of type declaration—dynamically typed, as in Lisp
  return proc { |n| aThing * n }
end
```

In recent Ruby, `->` is a synonym for `proc`

```
p1 = nTimes(3)
p2 = nTimes("foo")
p1.call(4)           => 12
p2.call(4)           => "foofoofoofoo"
```

Object orientation in Perl 5 is kind of a kludge; Perl 6 is supposed to be better

JavaScript has an unusual system based on “prototype objects”. It’s an “object-based” language, as opposed to object-oriented. (The original object-based language was `Self`.)

JavaScript, Python, and Ruby all allow new fields to be added to an object at run time.

JavaScript and Ruby allow new methods to be added.

Python and Ruby allow class bodies to be elaborated – conditional compilation. In Ruby:

```
class My_class
  def initialize(a, b)
    @a = a; @b = b;
  end
  if expensive_function()
    def get()
```

```
        return @a
    end
else
    def get()
        return @b
    end
end
end
end
```

type extensions

- build on structs/records
- language already has modules
- Ada 95, Modula-3, Oberon
 - single inheritance
 - no constructors or destructors
 - explicit 'this' parameter
 - all methods virtual in Modula-3 (also Oberon, I think)
- static by default in Ada 95; virtual when desired; but not a property of the method;
 - rather, determined by access through "class-wide" parameter or pointer
- Fortran 2003

Exist OO extensions to Common Lisp (CLOS), Rexx (Object Rexx), Tcl (Incr Tcl).

OCaml is of course object-oriented (that's what the 'O' is for). For simplicity, I don't usually use any of the OO features in course assignments.