

## =====

## 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 O-O 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. 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 &amp; 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.

Typically if you modify a definition module you have to recompile only that definition module. If you change the private portions of a declaration module (the parts the compiler depends on), you have to recompile both the definition module and the user modules, but you don't have to change the **source** of user modules.-----  
A few **C++** and **Java** features you may not have learned:**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++ 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
```

**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 in the middle of expressions for temporaries 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 will copy the copied object to copy constructors when the copied object will never be used again.)

A similar situation happens when returning:

```
my_class o3 = bar(args);
```

Here bar is a method that presumably returns an object of type my\_class. The declaration of o3 will invoke the copy constructor, but copying from a temporary that was created back in bar just for the sake of returning.

The copies in both of these examples can be eliminated "for sure" in C++11 using **move constructors**, which use **value references** (indicated in an argument list 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 the stuff still needed.Value 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 null: you specify members of object types are simply initialized to null, you specify arguments to the constructors when you call 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, c);
    ...
}
```

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

Harder in C++ 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:

```
foo::foo(args1, args2, args3, args4, args5, args6, args7, args8, args9, args10, args11, args12, args13, args14, args15, args16, args17, args18, args19, args20, args21, args22, args23, args24, args25, args26, args27, args28, args29, args30, args31, args32, args33, args34, args35, args36, args37, args38, args39, args40, args41, args42, args43, args44, args45, args46, args47, args48, args49, args50, args51, args52, args53, args54, args55, args56, args57, args58, args59, args60, args61, args62, args63, args64, args65, args66, args67, args68, args69, args70, args71, args72, args73, args74, args75, args76, args77, args78, args79, args80, args81, args82, args83, args84, args85, args86, args87, args88, args89, args90, args91, args92, args93, args94, args95, args96, args97, args98, args99, args100, args101, args102, args103, args104, args105, args106, args107, args108, args109, args110, args111, args112, args113, args114, args115, args116, args117, args118, args119, args120, args121, args122, args123, args124, args125, args126, args127, args128, args129, args130, args131, args132, args133, args134, args135, args136, args137, args138, args139, args140, args141, args142, args143, args144, args145, args146, args147, args148, args149, args150, args151, args152, args153, args154, args155, args156, args157, args158, args159, args160, args161, args162, args163, args164, args165, args166, args167, args168, args169, args170, args171, args172, args173, args174, args175, args176, args177, args178, args179, args180, args181, args182, args183, args184, args185, args186, args187, args188, args189, args190, args191, args192, args193, args194, args195, args196, args197, args198, args199, args200, args201, args202, args203, args204, args205, args206, args207, args208, args209, args210, args211, args212, args213, args214, args215, args216, args217, args218, args219, args220, args221, args222, args223, args224, args225, args226, args227, args228, args229, args230, args231, args232, args233, args234, args235, args236, args237, args238, args239, args240, args241, args242, args243, args244, args245, args246, args247, args248, args249, args250, args251, args252, args253, args254, args255, args256, args257, args258, args259, args260, args261, args262, args263, args264, args265, args266, args267, args268, args269, args270, args271, args272, args273, args274, args275, args276, args277, args278, args279, args280, args281, args282, args283, args284, args285, args286, args287, args288, args289, args290, args291, args292, args293, args294, args295, args296, args297, args298, args299, args300, args301, args302, args303, args304, args305, args306, args307, args308, args309, args310, args311, args312, args313, args314, args315, args316, args317, args318, args319, args320, args321, args322, args323, args324, args325, args326, args327, args328, args329, args330, args331, args332, args333, args334, args335, args336, args337, args338, args339, args340, args341, args342, args343, args344, args345, args346, args347, args348, args349, args350, args351, args352, args353, args354, args355, args356, args357, args358, args359, args360, args361, args362, args363, args364, args365, args366, args367, args368, args369, args370, args371, args372, args373, args374, args375, args376, args377, args378, args379, args380, args381, args382, args383, args384, args385, args386, args387, args388, args389, args390, args391, args392, args393, args394, args395, args396, args397, args398, args399, args400, args401, args402, args403, args404, args405, args406, args407, args408, args409, args410, args411, args412, args413, args414, args415, args416, args417, args418, args419, args420, args421, args422, args423, args424, args425, args426, args427, args428, args429, args430, args431, args432, args433, args434, args435, args436, args437, args438, args439, args440, args441, args442, args443, args444, args445, args446, args447, args448, args449, args450, args451, args452, args453, args454, args455, args456, args457, args458, args459, args460, args461, args462, args463, args464, args465, args466, args467, args468, args469, args470, args471, args472, args473, args474, args475, args476, args477, args478, args479, args480, args481, args482, args483, args484, args485, args486, args487, args488, args489, args490, args491, args492, args493, args494, args495, args496, args497, args498, args499, args500, args501, args502, args503, args504, args505, args506, args507, args508, args509, args510, args511, args512, args513, args514, args515, args516, args517, args518, args519, args520, args521, args522, args523, args524, args525, args526, args527, args528, args529, args530, args531, args532, args533, args534, args535, args536, args537, args538, args539, args540, args541, args542, args543, args544, args545, args546, args547, args548, args549, args550, args551, args552, args553, args554, args555, args556, args557, args558, args559, args560, args561, args562, args563, args564, args565, args566, args567, args568, args569, args570, args571, args572, args573, args574, args575, args576, args577, args578, args579, args580, args581, args582, args583, args584, args585, args586, args587, args588, args589, args589, args590, args591, args592, args593, args594, args595, args596, args597, args598, args599, args599, args600, args601, args602, args603, args604, args605, args606, args607, args608, args609, args609, args610, args611, args612, args613, args614, args615, args616, args617, args618, args619, args619, args620, args621, args622, args623, args624, args625, args626, args627, args628, args629, args629, args630, args631, args632, args633, args634, args635, args636, args637, args638, args639, args639, args640, args641, args642, args643, args644, args645, args646, args647, args648, args649, args649, args650, args651, args652, args653, args654, args655, args656, args657, args658, args659, args659, args660, args661, args662, args663, args664, args665, args666, args667, args668, args669, args669, args670, args671, args672, args673, args674, args675, args676, args677, args678, args679, args679, args680, args681, args682, args683, args684, args685, args686, args687, args688, args689, args689, args690, args691, args692, args693, args694, args695, args696, args697, args698, args699, args699, args700, args701, args702, args703, args704, args705, args706, args707, args708, args709, args709, args710, args711, args712, args713, args714, args715, args716, args717, args718, args719, args719, args720, args721, args722, args723, args724, args725, args726, args727, args728, args729, args729, args730, args731, args732, args733, args734, args735, args736, args737, args738, args739, args739, args740, args741, args742, args743, args744, args745, args746, args747, args748, args749, args749, args750, args751, args752, args753, args754, args755, args756, args757, args758, args759, args759, args760, args761, args762, args763, args764, args765, args766, args767, args768, args769, args769, args770, args771, args772, args773, args774, args775, args776, args777, args778, args779, args779, args780, args781, args782, args783, args784, args785, args786, args787, args788, args789, args789, args790, args791, args792, args793, args794, args795, args796, args797, args798, args799, args799, args800, args801, args802, args803, args804, args805, args806, args807, args808, args809, args809, args810, args811, args812, args813, args814, args815, args816, args817, args818, args819, args819, args820, args821, args822, args823, args824, args825, args826, args827, args828, args829, args829, args830, args831, args832, args833, args834, args835, args836, args837, args838, args839, args839, args840, args841, args842, args843, args844, args845, args846, args847, args848, args849, args849, args850, args851, args852, args853, args854, args855, args856, args857, args858, args859, args859, args860, args861, args862, args863, args864, args865, args866, args867, args868, args869, args869, args870, args871, args872, args873, args874, args875, args876, args877, args878, args879, args879, args880, args881
```