Notes for CSC 2/454, Nov. 25 and 30, 2020

==========

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 namespases 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 & 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
Cosntructors 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 is allowed to elide calls 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 ***rvalue 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 stuff we still need.

rvalue references are also used for ***move assignment*** methods. Programmers are free to use them for 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++ 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) { ...
```

`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 in C++!
`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 class. 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 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.

Simula also had virtual functions (all of which were abstract).
In most modern OO languages, (Java, C#, Scala, Ruby, Python, ...) ***all*** member functions are virtual, so you don't need the keyword.

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`.
In Java all methods are virtual.

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).

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).