C++ Subclasses vs. Java

class Derived extends Base {...
class Derived: public Base {...

C++ also has: private {..., which screens out base-class methods. However, it violates general subtyping principles and is frowned upon. (Compare text, p190.)

C++ makes you think harder about visibility of non-public data to subclasses, because C++ lacks Java’s default/package visibility. Liberally making data protected can allow alien subclasses to violate INVs of the base class.

- Example: top in class StringStack.

So let’s examine communication when base data is private.
Base and Derived Constructors

- As with Java, base-class constructors must be called first—and the system will do so even if you don’t.
- Java: must call `super(...)` in first line in ctor body.
- C++: must call base-class ctor first in initializer list. E.g.:

```cpp
class Stack: public List {
    public
        Stack(int maxSize) : List(maxSize) { ...}
}
```

Without that, compiler would insert a call to `List()`, which might cause havoc...
Havoc

- If you declare a constructor with parameters, then this *disables* the zero-parameter ctor... unless you define it too!
- So if List() is disabled, you get a compile-time error message, with templates maybe screenfuls...
- If List() is not disabled, it will compile... which is usually far worse.
- Hence, always insert the base call, and always define a constructor.
- And use `explicit` with single-parameter ctors, else a typo
  
  ```
  int sz = myStack = size; when you meant
  int sz = myStack.size();
  will compile, and will re-construct myStack to empty data of that size!
  ```
Private Base Class Instance Variables

- Need to be initialized by the base-class ctor call.
- Having a protected setter method for them in the base class is better than having “protected” data, since the base-class designer writes the body and can monitor CLASS INVs.
- An “alien” subclass who overrides the protected method still can get its green hands on the crucial data.
- C++ friend requires the base-class designer to know the names of the friends in advance, so they are “terrestrial.” Good for emulating Java package-visibility, but use sparingly...
- Can friend global functions/operators as well as classes, e.g. operator<< in LinkArg.h is outside the class but allowed to see the private data.
Overriding

- The only difference from Java is that an overriding method must have the exact same return type, whereas Java has allowed a subclass return type since 2005.
  - The main hitch is how this relaxation would interact with `const` return types. For general reasons we will try to avoid them.
  - You can assign or pass a non-const return value to a `const` variable or parameter, but not vice-versa.
- And of course, to allow overriding the base-class method must be marked `virtual` (and the derived method almost-always is too).
- Pointer (or reference) variable + `virtual` = Java behavior, else you get `static binding`. (Not to be confused with `static` members.)
Constructors in Declarations and Expressions

Base* bp;  //pointer declaration
Base* bbp = new Base();  //pointer init on heap
Base bv;  //value *construction* if 0-param ctor
//Base bv();  //looks like 0-ary function dec.

Base foo() {...; return Base();}  //return Base; is error
Base* foo(){...; return new Base();} //OK, heap obj persists
//Base* bar() { Base b; return &b; } //"dangling pointer"

cout << Base();  //OK if ostream& operator<<(ostream&, Base)
class Derived: public Base {
    int newField;

public:
    explicit Derived(int x) : Base(), newField(x) { ... }
    virtual void meth() { ... }  //overrides Base::meth
};

Derived dv(3);//OK

bv = dv;       //VALUE LOSS, bv has no newField.
bv.meth();     //hence value variables give only static binding

bp = &dv;      //no problem, newField still accessible

bp->meth()     //OK, calls override if meth() is virtual
(*bp).meth()   //Same: pointer variable not "." is what matters

See HelloWorld.cpp also for cases involving C++ reference variables, but we will prefer pointer variables.
Suppose we have a `Base*` pointer `bp`, which may-or-may-not be holding a `Derived` object. In Java we can tell by testing:

```
if (bp instanceof Derived) {
...
```

In C++ this is a two-step process:

```
Derived* dp = dynamic_cast<Derived*>(bp);
if (dp) {
...
```

If `bp` is holding a `Derived` object, the cast succeeds and `dp` holds the same object. If not, then `dp` is a NULL pointer and the test `if (dp)` fails.

See text pp205–206. Lecture demo with `RealFn.h` and `Newton.cpp` includes a meaty example with how the `str()` methods decide when to introduce parentheses.
Overloading and Un-Shadowing

- Overloading rules are (basically) same as Java.
- Can overload where the only difference is that a parameter or the return is `const`!
- Ditto `register` and `volatile` and other C++ `type qualifiers`.
- A ginormous error message ending with the words “...discards qualifiers” usually means you violated a rule of `const`.
  - E.g. you took a `const` variable and tried to call a non-`const` method or assign it to a non-`const` variable, or made a `const` method call a non-`const` method from the same class.
- If a method `meth` would have been an override except for the absence of `virtual`, it creates an overload that `shadows` the base-class method.
- Can still invoke base-class method as `Base::meth(...)`, which translates Java `super.meth(....)` (example: `IntList2.java`).
Overloading Operators

In C++, we can overload operators via their long-form names such as:

- `operator==`: e.g. `operator==(a,b)` is `a == b`
- `operator=`: (long form in member-syntax only)
- `operator<<`: with `cout`, different meaning from shifting
- `operator++`: is prefix; postfix is `operator++(int)`
- `operator[]`: `arrp->operator[](i)` is `(*arrp)[i]`
- `operator()`: `(*fp)(x)` is same as `fp->operator()(x)`
- `operator*`: overload de-referencing for iterator class
- `operator->`: can even override this! but not member-access.

The whole Standard Template Library syntax is based on overloading the last five!

In an OS course, you may overload `operator new` and `operator delete`.

We’ve also mentioned conversion operators `operator Bar` for other types `Bar`, but will ignore them.
Object Scopes and Lifetimes

- **Globals and Statics**: “eternal”
  - should *guard* names by class::* or namespace::*
  - a *using* declaration *unguards* the latter.
- **Locals**, including all *value objects*.
  - Constructed w/o “new” on the system stack.
  - Are *reclaimed* when their declaring function/method (incl. main) exits.
  - No need to delete.
- **Heap Objects**
  - constructed via *new*
  - *held* by pointers, but themselves nameless
  - exist after their activation frame exits
  - In C++, need explicit *delete*-ion when no longer wanted.
- **(Web Objects** *persist* even after main/applet exits.)**
Reclaiming, Deleting, Copying

- **Reclaiming** a pointer variable `p` does not reclaim the object `obj` it points to. That needs `delete(p)`.
  - As covered before, `p` itself is a value object, whose value is an address—while `obj` can be a heap object or a value object (even another pointer!).

- Reclaiming an object `obj` activates its *destructor*.

- Every class starts off with a *default destructor*, which does nothing more than *reclaim* all of its fields.
  - but this does not `delete` any pointer fields it may have.

- **Copying** a pointer variable does not copy the object it points to.

- **Copying** an object `obj` by-default copies its fields. “*Shallow Copy*”
  - but does not copy the objects any pointer fields may point to.

- Every class `Foo` starts off with a *default copy constructor* `Foo(const Foo& other)` and a default `operator=(const Foo& rhs)`, which do shallow copy.
Simplified “Rule of 3”: A class Foo is non-reclaimable when it has one or more fields that are pointers, or containers of pointers. Then it should define the following three members ([... ] means optional):

1. [virtual] ~Foo() { ... } //destructor
2. Foo(const Foo& copyMe) [: <inits>] { ... } //copy ctor
3. Foo& operator=(const Foo& copyMe) { ... } //assignment

- The field vector<string>* elements; in the StringQueue or StringStack class is such a pointer field.
- A “raw array” field also counts as a pointer field.
- A vector field (without the *), however, counts as a value field. The vector will be reclaimed automatically.
- A vector of pointers, however, counts as a container-of-pointers field, and may need further action.
- Motivation for destructor is to free up memory when objects are no longer needed.
Default and “Skin-Deep” Big Three

So-called “default” versions of the “big 3” always exist:

- The default destructor *reclaims* each field.
- But, reclaiming a pointer leaves the object it points to untouched.
- The default copy-constructor copies each field, but not any objects “further down” that they point to.
- The default assignment operator assigns each field individually.
- For each *value* field, these actions will recursively call the corresponding “big 3” of the class the field belongs to.
The “skin-deep” destructor calls `delete` on every pointer field. Value fields need not be mentioned—they get reclaimed (too).

For every pointer field `Bar* p;` the skin-deep copy-ctor does an initialization `p(new Bar(*(copyMe.p)))` (this is doable without `friend-ing` because it is inside the class).

And the skin-deep `operator=` does `*p = *(copyMe.p)`; for every such field.

The skin-deep destructor is correct for `Queue` and is inherited by `Deque`:

```cpp
virtual ~Queue<T>() { delete elements; }
```
Problem with the skin-deep/“Next Hop” Destructor

- Calls `delete` on every pointer field.
- (Calls `delete[]` on raw-array fields, cf. KW::vector p251.)
- Other fields do not have to be mentioned—they still get reclaimed automatically.

Looks logical! But do we want to code it? Consider the “swath” of an object, defined as follows:

- Primitive object (int etc.): itself.
- Value class/struct object: itself + the swaths of all fields.
- Vector-or-array: itself + the swaths of all elements.
- Pointer: itself + the swath of whatever it points at.

If everyone has a next-hop destructor, `delete` will wipe the entire swath!...

- (…except for “double indirection,” when a pointer points at a pointer, e.g. `Cell** nextLink = &next;`)
Responsibility For Destruction

Consider a (templated!) Cell class for a linked list:

template <typename I> //I = Item_Type in text
class Cell { //cf. "Node" on p255
    friend class LinkedList<I>; //reason needed is below
    I data;
    Cell<I>* next;

public:
    Cell(I dataItem, Cell<I>* nextPtr)
        : data(dataItem), next(nextPtr)
    {
    }
    //virtual ~Cell() { delete(next); } //next-hop, bad here.
    virtual ~Cell() { } //omit virtual for "true structs"
};

With next-hop destruction, each Cell would “Delete Thy Neighbor”! Rather, a LinkedList class that manages the cells should do it...
When not to delete: Shared Data

- If two objects have the same sub-object in their swaths, and the former deletes it, it “munges” the latter!
- Example: The `Newton.cpp` client for `RealFn.h` builds function objects that share subterms, rather than always making new ones. It could have had this:

```cpp
MonicFn* x = new X();
MonicFn* log2x = new Log(2.0, x);
MonicFn* ps2 = new Times(new Constant(40.0), new Times(log2x, log2x));
MonicFn* xx = new Times(x, x);
```

- If deleting `xx` whacked `x`, then `log2x` and hence `ps2` would get corrupted.
- Can be solved by having each object monitor its reference count, but what a hassle!...
- ...a main reason newer languages are adopting garbage collection—but can you do it “in a heartbeat”?
A Linked List Destructor (cf. text, p273)

template <typename I>
class LinkedList {
    Cell<I>* head;

public:
    ...

    virtual ~LinkedList() { // INV: head = next cell to delete
        while (head != NULL) { // by INV, means no more to delete
            Cell<I>* curr = head; //delete(head) would Invalidate head
            head = head->next; //needs friending
            delete(curr);
        }
    }
    ...
};

(Aside: An auto_ptr type deletes neighbors, and could destruct the Cells after delete(head); Still managed by LinkedList so OK.)
Linked-List Destructor (cont’d)

- As in the text, this traverses the list and deletes in forward order.
- Works unchanged for doubly-linked list and DNode—the extra prev pointers are themselves value objects and are simply reclaimed.
- If Cell were nested inside LinkedList, we wouldn’t need to repeat the template parameter I
- Text puts DNode into a separate file and does manual inclusion “in mid-code”; we disagree with this and will code nested classes “literally.”
- Also IMHO, destructor should be virtual whenever a class might be subclassed, even if no virtual methods are present. This is wider than what the text says on p200. (NB: The new C++ sealed keyword, which is like Java final, seems not to exist for g++ on timberlake yet.)
Deeper, Deep, and Deepest Copy

- Deepest copy clones all non-const fields in the swath of an object.
- The “next-hop” copy-constructor clones all pointer fields, e.g.:

```cpp
class Foo {
    Bar x;
    const Haw c;
    const Haw& d;
    Delta* dp;

public:
    Foo(...) : ... { ... }
    Foo(const Foo& other)
        : x(other.x)
        , c(other.c) //OK to *initialize* a constant, copies c?
        , d(other.d) //definitely does not copy d
        , dp(new Delta(*(other.dp))) //invokes Delta copy ctor!
    { }
```
Parameter Passing and Copying

- `int meth(Foo arg)` Value parameter, *copies* passed-in Foo obj.
- `int meth(const Foo arg)` Constant value parameter, guarantees arg can’t be assigned or mutated in body of Foo, but copies obj in the call (*unless* optimization settings intervene?).
- `int meth(const Foo& arg)` Constant reference parameter, same as above but guarantees that obj is *not* copied.
- `int meth(Foo& arg)` Reference parameter, avoids copying obj, and allows the body of meth to modify the *original* of obj.
  - Some authorities hold that only `void` methods should have reference parameters, as was the rule in the programming language *Ada* used by the US DoD in the 1980s and 1990s.
- `int meth(Foo* arg)` Pointer parameter, copies only the pointer, and allows body of meth to modify the original obj.
- `int meth(const Foo* arg)` Pointer to constant data, similar effect to a const reference but with pointer syntax inside the body.
“Next-Hop” Assignment Operator

Continuing the same class Foo...

```cpp
Foo& operator=(const Foo& rhs) {
    x = rhs.x;
    //! c = rhs.c; d = rhs.d; //cannot *assign* to const
dp = new Delta(*(rhs.dp)); //again invokes Delta c-ctor
return *this; //allows chained assignments
} //such as obj1 = obj2 = obj3;
```

- Note that the constant-reference parameter in both the copy-ctor and `operator=` averts premature copying of the argument object.
- The above will produce deepest copy if all objects in the swath do this, (again excepting double-indirection).
- But should they?
Managed Copy by LinkedList

Back inside our templated LinkedList<I> class:

LinkedList<I> (const LinkedList<I>& other)
    : head(other.head ? new Cell<I>(*(other.head)) : NULL)
{
    Cell<I>* curr = other.head->next;  // current cell to copy
    Cell<I>* target = this->head;     // INV: copied up to target
    while (curr != NULL) {
        target->next = new Cell<I>(*curr);    // use Cell copy ctor
        // target->next = new Cell<I>(curr->data, NULL);  // also OK
        target = target->next;
        curr = curr->next;
    }
}

LinkedList<I>& operator=(const LinkedList<I>& other) {
    head = (other.head ? new Cell<I>(*(other.head)) : NULL);
    [repeat above body!---?] [what about deleting old Cells??]
}
As the text notes on p250, maintaining sizeable duplicate code for `operator=` is yucky.

The text code for `operator=` invokes the copy-constructor to create a new list, swaps it with `this`, and finally deletes the old self.

Another idea is to “factor” the common while-loop code into a separate private method—but that still leaves the task of destructing the old cells linked from `head`.

Because we did not have `Cell “Copy Thy Neighbor,”` and because the `data` field of `Cell` is a value, the default `Cell` copy ctor is fine. If it had `I* data`, then we would have to define a different `Cell` copy ctor too.

Also note the assumption that the client for `I` can copy the `data`. 
When are the “Big Three” Needed?

- Basically when a class *allocates* a pointer to (non-const) data.
- An override of `operator=` is also needed whenever a class has a “member const” field...
- …unless you want to forbid assignments altogether—since any attempt to use the default `operator=` will generate a compile error on the attempt to assign a constant field.
  - Example of member const: a `maxSize` limit that is tailored for an object at construction, rather than set for the class as a whole.
  - A `static const` field is fixed for the whole class, and not copied by the default `operator=`, so no problem.
  - The function-objects in `RealFn.h` have all-`const` fields, including `const Foo* const` pointers. Hence no assignments allowed.
- To forbid cloning, one can *disable* the copy ctor and `operator=` by declaring them `private`.
  - The `iostream` library does this with streams.
  - But if the client for `I` in `Cell<I>` does this, screenfuls of template errors—if you’re lucky!
When They’re Not Needed—“Value Classes”

- If any pointer fields point to data that the class does not “own” or “manage,” then no responsibility to copy or delete it.
- If all other fields are value declarations, we have a “value class.”
- A “value class” can have a simple constructor that initializes each field, and the default “Big Three” are fine for it.
- Example: a typical iterator class. E.g. `FlexArray<T>::iterator` (Fall 2010) can have the constructor (assuming its fields are called `myFlex, whichNode, localIndex`):

```cpp
iterator(FlexArray<T>* myFlex, //ref to parent container
         Node<T>* whichNode,   //ctor itself is private,
         size_t localIndex)    //called by public begin()
    : myFlex(myFlex), whichNode(whichNode),  //end(), rbegin()
      localIndex(localIndex) { }
```
What Templates Mean

Suppose Item is a client type for Foo<\textit{I}>

- Formally Foo is a compile-time function that takes a type parameter and returns a class, here Foo<Item>.
- So read it as “Foo-of-Item,” just like we read “\textit{f}(\textit{x})” as “\textit{f}-of-\textit{x}.”
- Thus generally called \textit{parametric} polymorphism.
- Not Foo “Has-A” Item (certainly not “Is-A” either way); maybe one can say Foo<Item> is Foo “Serving” Item.
  - UML diagram (text p776) shows tandem with Foo bigger.
- For a container class like vector, the reading “vector-of-Item” or “vector-serving-Item” is especially apt.
- Template classes can have more than 1 parameter, and parameters can also be objects as in ordinary function parameters, e.g.
  template <typename \textit{I}, int \textit{maxSize}> class Stack { ...
  - Creates separate Stack<\textit{Item},\textit{s}> classes for each size \textit{s}. (\textit{Compare} passing \textit{maxSize} as a constructor argument.) Solves “member const” problem but bloats code! Dilemma meatier with function-objects...
  - Functions and methods can be templated individually...
Kinds of Containers

- **Container**: a class that manages a collection of items.
  - One-at-a-time access (e.g. stack, queue, heap)
  - Sequential, can “go inside,” rewind, re-read (list)
  - Random-Access (array)
  - Key-access (dictionary, hash-table?)

- Sorted or Unsorted?

Modern focus is not on the classical name of the data-structure, but the kind of *access/iteration* it allows, and what *asymptotic performance* guarantees it offers.
Iterators for Containers

- **An iterator** is a “Pointer Object.”
- Most important: can pass iterators rather than whole containers to methods.
- C++ STL syntax based on Array Pointers. Given `Foo arr[n];`—
  - `Foo* p = arr;` begin p on the first element `arr[0]`
  - `p++` or `++p` move p to the next element—compiler knows the memory-size m of a Foo object and converts this to `p += m;`
  - `x = *p` return current element
  - `x = *p++` return current elt. and move on
  - `*p++ = x;` can assign unless p is `const Foo*`
- **Iterator Classes** overload these (and maybe other) operators. Are typically *nested* inside templated containers.
More Array Pointers and Iterator Kinds

- **Random-Access Iterators** also emulate the following features of array pointers:
  - `p += k` advance `k` places, compiled as `p += m*k`; Similar for `p -= k`; and `p--, --p` etc.
  - `Foo* end = p + n`; *past-end* of size-`n` array
  - `p[k]` is same as `p + k`, while `arr[k]` is same as `*(p+k)`.
  - If a pointer `q` is already on that cell, fetching `*q` is quicker than `arr[k]` which involves arithmetic.
  - Can compare `p < q`, `p <= q`, `p > q`, `p >= q`, as well as `p == q`, `p != q`.
  - All iterators can of course assign `p = q;` to each other, but only RAI can be init from any cell `k`.

- **bidirectional_iterator** adds only `p--` and `--p` to `forward_iterator`, plus creation by `.rbegin()`, `.rend()`.
Using Iterators

Assuming `vector<int> vec` of size `n`:

```cpp
for (int i = 0; i < n; i++) { sum += vec[i]; }
```

becomes

```cpp
for (vector<int>::const_iterator it = vec.begin();
     it != vec.end(); it++) {
    sum += *it;
}
```

which really translates, for `int arr[n]`:

```cpp
const int* pastEnd = arr + n;
for (const int* arrp = arr; arrp != pastEnd; arrp++) {
    sum += *arrp;
}
```

The natural-looking indexing code is slowest, while the bulky iterator code is nearly as fast as the pointer code. (Demo: `templatesorts.cpp`)
In Java and C#, with C++ to follow in 201x?, the notion of (forward/reverse/?) iteration is being brought into the basic language syntax, e.g:

```java
foreach (int item: arr) {
    sum += item;
}
```

This has “fewer moving parts” than a regular for-loop, and avoids explicit reference to a (const) iterator—though a container class must still implement `Iterable` to use this syntax.
The following while-loop

```cpp
vector<int>::const_iterator it = vec.begin();
while (it != vec.end()) {
    sum += *it++;
}
```

quite literally translates Java

```java
while (vec.hasNext()) {
    sum += vec.next(); //side-effect of advancing
}
```

Hence the *it++ idiom is traditional.
Iterators In Motion

For any Container\(<I>\) that supports these operations, with the
following STL syntax:

cont.begin(): iterator on first element
cont.end(): iterator one past last elt.
I *itr: data item pointed at
*itr = item;: can assign to location, except...
const I *itr: if itr is a const_iterator.
itr->meth(...): invoke meth(...) on data item
itr++: move itr to next cell forward
itr--: ...or backward, if cont allows.

With operator-- one can employ cont.rbegin() which returns an
iterator on the last elt., and cont.rend() which is before the first elt.
STL Iterator Class Hierarchy Categories

- A random_access_iterator (RAI, RI in text)
  - is-a? [Library is more complicated than this!]
- bidirectional_iterator, which
- is-a? forward_iterator and also
- is-a? reverse_iterator
- Each kind is-a? basic iterator, which actually breaks down into read-only and write-only, before the ultimate base which can only do (pointer-)assignment and comparison by ==, ! =.
- Since iterators are themselves value-objects, one cannot use the base class to refer to them—this would cause Info Loss!
- Instead each container creates a nested class Container<I>::iterator by extending, type-aliasing (via typedef), or just imitating the appropriate one of the above STL library classes.
  - A declaration with a template variable before :: needs the keyword typename in front—text, p281. Needed for return types too.
Delegation vs. Inheritance

A class $\text{Foo}$ is said to wrap a class $\text{Bar}$ if:

- “most” of a $\text{Foo}$ object consists of a $\text{Bar}$ object $\text{bar}$, and
- “most” of the $\text{Foo}$ methods get “most” of their functionality by calling method(s) on that $\text{bar}$.

**Example.** Rather than extend a $\text{LinkedList}$ class, the text’s $\text{OrderedList}$ wraps $\text{std::list}$. It could have a field $\text{std::list* const theList}$ held by a constant pointer, or use value syntax as in the text. Then rather than inherit a method like $\text{LinkedList::size()}$ or $\text{std::list::size()}$, it codes a method $\text{size()}$ whose body simply delegates to the enclosed list:

```cpp
size_t size() const {
    return theList->size();
}
```
Delegation vs. Inheritance II

This may look like a waste of code and (run-)time, but:

- An optimizing compiler, helped along by `const`-correctness, can often spare you the overhead of the “extra” method call.
- Whereas inheritance, especially with virtual methods, requires an extra class-table lookup.
- If `Bar` uses outdated syntax, the `Foo` wrapper can supply a conforming interface. (“Adapter” Pattern)
- Inheritance can hurt modularity; wrapping can improve it.

The text’s `Ordered_List<_List<I>>::iterator` class delegates to the corresponding methods of the `std::list::iterator`.

As we’ve observed, C++ templates can assume the argument implements certain methods, without a Java-like `interface` specifying them. (If some are missing—and a client tries to use one—a link-time error results.) This is like “Duck Typing,” but compile-time/static instead of run-time/dynamic. Templates go well with delegation and the former, and produce efficient code (per demos).