From Java to C++

CSE250 Lecture Notes Weeks 1–3

Kenneth W. Regan
University at Buffalo (SUNY)

September 20, 2010

First Day

1. Fill out "Class Survey"—Weeks 1 and 2 recitations...
2. Go over course syllabus:
   (a) Total points system, pre-set curve.
   (b) Exams 50%, + up-to-5% in quizzes.
   (c) Individual and group project(s), problem sets.
   (d) Instructor *may* re-weight by 5%, e.g. drop half of bad prelim while up-weighting good final or homework. Attendance a factor.
3. "Pep-Unpep Talk"—course is unavoidably content-full, hard work. Only CSE C++ course. Like "Orgo" for pre-meds.
4. Hello World—Jello World?

From Java to C++

1. Show `JelloWorld.cpp`—C++ and Java on same screen.
2. C++ and Java code *look* similar, but...
3. ...two major differences in behavior: C++ Values versus java References, and Method Override is Not the Default in C++.
4. To get Java behavior, need pointers and `virtual`.
5. New Syntax: `& * -> :: << >> #`

C++ Type System

Java: primitive *values*, class *references*.

	Value type	`T`
C++:	Pointer type	`T*`
	"Alias" type	`T&`

Prefixing `const` makes a new type. We will prefer pointers to references/aliases, except for employing the type

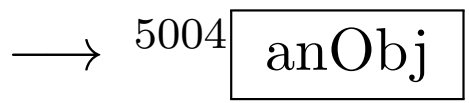
`const T&`

for certain parameters and special constructors. We will use all of:

`const T* xp`	Pointer to constant T
`T* const xp`	Constant pointer to T
`const T* const xp`	Const ptr to const T

What is a Java reference?

In Java, `Foo x = new Foo()` creates *two* items:



This == what you get with the C++ declaration

```
Foo* x = new Foo()
```

In *both* languages, the name `x` is *bound* to an integer object whose *value* is the memory address of an unnamed `Foo` object. Java hides this "pointer value"; C++ doesn't.

C++ References are Just Aliases

C++ references are "disembodied pointers"–they don't have the 5004 part at all.
Main operational difference from Java references: If you do

```
string xhost = "Jello";
string yhost = "Bello";
string& xr = xhost;
string& yr = yhost;
yr = xr;
xr[0] = 'H';
cout << xr << " " << yr << " " << xhost << endl;
```

you get `yr = "Jello"`, whereas with pointers and Java references, `yr = xr;` does not copy the contents of `xr`. One cannot initialize a non-constant reference to a pure value, only to the variable beign aliased. Note that `xhost` is **not** copied, and winds up saying `"Hello"` too.

Pointers and Const

```
Foo y;        //when value, NO (), would look like function
const Foo* xpcd = new Foo();
Foo* const xcp = new Foo();
xpcd = &y;     //OK
xpcd->meth(); //OK *if* meth is const!
//xcp = &y;  //can't reassign Foo* const
xcp->mutate();  //OK
```

And `const Foo* const xcpc = new Foo();` tells the compiler that `xcpc` cannot be re-assigned, and that the anonymous `Foo` object cannot be modified *through* `xcpc` (though it could be modified thru a non-const pointer, a "hole" which some newer OO languages try to fill).

[`HelloConst.cpp` illustrates this with strings.]

C++ `const` and Java `final`

- On primitive types they are equivalent, e.g. `const double PI = 3.14159;`
- A Java `final` reference is equivalent to a C++ constant pointer, but not a pointer to constant data.
- On methods, the meanings are perpendicular:
  - C++ `const` methods *can* be overridden.
  - C++ `const` methods cannot modify the invoking object—at least not its top-level fields. ("Deep Const" is a current research issue.)
  - (A field that is not really part of the "constant state" of the object can be marked `mutable`, to allow `const` methods to change it!)

Why C++?

- C++ compilers are smarter than Java compilers *are allowed to be*.
- C++ gives programmers more control over how objects are laid out:
  - on the stack as values, vs. on the heap via pointers,
  - by user-managed memory,
  - by "packing" and assembly-level tricks.
- C++ allows more-efficient code.
- C++ has more-advanced O-O features (except that it does not yet have Java's template bounds).
- Most working code is in C++, and will be for a long time.

Java to C++:

```
Foo x = new Foo(a,b);
Foo* xp = new Foo(a,b);

x.field, x.method();
xp->field, xp->method();
```

Also legal are `(*xp).field, (*xp).method`

Assignment:

```
x = y;
xp = yp;
```

no change—

—because a Java reference assignment is overtly the same as a C++ pointer assignment.

Translating Java Classes

- Trailing ; after class-closing brace }
- `extends` becomes `: public`
- `public`, `private`, and `protected` denote *regions*, not individual items. Can alternate repeatedly...
- No "default/package scope"—instead write

  `friend class Bar;`

  within class `Foo`, to enable class `Bar` to access internals of class `Foo`. One can also declare a "global function" to be a `friend`—we will reserve this almost exclusively for operators associated to classes such as `operator<<` in example files `LinkArg.{h,cpp}`.
- C++ *namespaces* are like Java packages, and provide a prefix to avoid ambiguity, e.g. `std::cout`
- Between class (or namespace) and field, C++ uses `::` not Java `.`

Translating Within Classes

- Put `virtual` before a method `meth` to get Java behavior—when you call `meth` thru a pointer!
- If `meth` is an *accessor*—i.e., cannot change the object that invokes it—then it should have `const` *after* the method parameters.
- Method code *can* be "inlined" like in Java, but for all but tiny bodies, putting the method header in a `Foo.h` file and the body in a separate `Foo.cpp` file is preferred. (Recitations next week will focus on code files.)
- Special syntax for constructors and "The Big Three"...

Constructors

C++ constructors use special initialization syntax:

```
struct Point {              //struct => top region is public
   double x, y;
   Point(double x, double y) : x(x), y(y) { }
};
```

In Java you would write `this.x = x;` and `this.y = y;` as the first 2 lines inside the {...} of the constructor.

The C++ system attempts to default-construct all fields of a C++ class that are NOT mentioned in the " : " part *before* the opening { is hit—wasting time and sometimes causing havoc! So, use the ":" syntax.

Member Differences

- Can't initialize non-constant fields outside constructor.
- Can use `this->x` akin to `this.x` in Java.
- *Accessor* methods should—must!—have `const` right after their parameters.
- No keyword `abstract`—instead C++ marks abstract member functions (which must be `virtual`) by appending `= 0;` to them—nulling out their function pointers!
- A class is abstract if it has an abstract method.

No Interfaces!?

C++ has unrestricted *multiple inheritance*:

```
class Foo:  public Bar, Com, Delta {...
```

So one can get the effect of a concrete interface by:

```
class Comparable {
 public:
   virtual int compare(const Comparable& rhs) const = 0;
};
```

Note the `const &` parameter idiom, and the abstract method itself being `const`. (No `const` after `&` is needed—const references already can't be on LHS of any assignment.)

C++ Templates Are Real Code

Java `class Foo<T> {...` becomes C++

```
template<typename T> //say class T if T should be a class
class Foo {          //NOT "class Foo<T>" as in Java-->weird errors!
   ...
   Foo(...) ...      //constructors MAY do "Foo<T>", but text doesn'
};

template<typename T> //must repeat for bodies outside class
Foo<T>::Foo(..){..}  //first <T> required, second MIGHT be error!
```

Declarations in the body of `Foo` that use `T` should have `typename` in front.

Major difference from Java: C++ generates separate object code for `Foo<Bar>` and `Foo<Com>` etc. To save code bloat, Java *erases* the template after compile time and generates just `Foo.class`, but you can't tell at runtime whether a client is `Bar` or `Com` or etc.

C++ Templates Are Not Real Code

If you have a template class `Foo`, compiler will let you put declarations in `Foo.h` and bodies in `Foo.cpp`, and even do `CC -c Foo.cpp` to create `Foo.o`.

However, `Foo.o` does not really have object code—instead, when a client file `Bar.cpp` (or `Main.cpp` or etc.) instantiates a `Foo<Bar>` object, the object code for `Foo<Bar>` is embedded into `Bar.o` (**or**, put in a separate templates repository as the Sun CC compiler used to do, or etc.).

Thus `Foo.o` is not "real code," so all template classes are morally headers. *More to the point:*

With `g++` (still) and many other compilers, *linking will fail* if `Foo.cpp` is separate from `Foo.h` and only the latter is included. Hence we will *require* that template bodies be in the same file, either inside or outside the template class braces. (I've read that "outside" can cause another problem, but haven't gotten it with any compiler, and whole previous CSE250 years were not affected.)

No Interface Bounds!

Java has `class Foo<T extends Comparable> {...` to ensure that any valid argument `Bar` for `T` defines `int Bar.compare(Comparable rhs)`.

C++ doesn't do this—a proposal called "Concepts" was dropped from next-update plans in July (2009)!

If a client `Bar` fails to implement `Compare`, a file building a `Foo<Bar>` *may* still compile separately—leaving the fault to be found *at link time*. Our `g++` on `timberlake` seems to be more proactive. *Please try the* `dompare` *typo in the files* `Link*.*` *in* `.../Java2C++/` *on your home systems.*

Later, we will do better by passing a *function object* for `compare` when constructing `Foo`. (Cf. "COMP" at bottom of p300 in text.)

A Java 5 Idiom to Know

```
interface Comparable<T extends Comparable<T>> {
   public int compare(T rhs);
}
class Bar extends Comparable<Bar> {
   public int compare(Bar rhs) { ... }
   ...
}
class Foo<T extends Comparable<T>> {  //Bar OK as T
   ...aMethod(T[] args) {
      ...if (args[i].compare(args[j]) < 0) { ... }
   }...
}  //generic T[] array OK as parameter in Java, can't construct one
```

The difference is that if a different class `Delta` extends `Comparable`, now you get a *compile-time* error if you mistakenly do `bar.compare(delta)`, instead of a *runtime* exception. Java 5 *convulsed* the language to achieve this. C++ templates and linking do this, but less explicitly.

Constructors [slide repeated]

C++ constructors use special initialization syntax:

```
struct Point {              //struct => top region is public
   double x, y;
   Point(double x, double y) : x(x), y(y) { }
};
```

In Java you would write `this.x = x;` and `this.y = y;` as the first 2 lines inside the {...} of the constructor.

The C++ system attempts to default-construct all fields of a C++ class that are NOT mentioned in the " : " part *before* the opening { is hit—wasting time and sometimes causing havoc! So, use the ":" syntax.

No Constructor Forwarding

- Unlike in Java, a C++ constructor may not call another, not even with the initialization syntax (as in C#).
- When a class has many fields, and different constructors must be provided, this forces ugly code duplication. . . at least of the field-initialization syntax.
- Forwarding is proposed for the next C++ revision, but it's being stalled by disagreement over *when* an object can be considered "constructed":
  - when the last (i.e., initially called) constructor exits?
  - when the first constructor exits?
  - when the initially-called constructor finishes its initialization syntax and starts on its body (if any)? (my preference!)

The Big Three

Usually needed only when a class allocates memory, or when the default field-by-field destruction/clone/copy is not the desired behavior...

```
[virtual] ~Foo(...) {...}       //destructor

Foo(const Foo& rhs) {...}       //copy constructor

Foo& operator=(const Foo& rhs) {...} //assignment
```

- Defining a trivial virtual destructor in a base class `Foo`, namely `virtual ~Foo() { }`, allows derived classes to override it. Recommended in general.
- A class can **disable** cloning by declaring the copy constructor and `operator=` to be `private`.
- Prime examples: C++ streams. Hence for *user-created* streams we will initially follow the general use-pointers-for-objects policy.

Implicit Conversions

- A single-argument constructor `Foo(Bar bar)` automatically gets invoked when you write `foo = bar;` ...
- ... UNLESS you prevent this dangerous behavior by putting the keyword `explicit` before the constructor's declaration.
- A class `Foo` can also define an implicit conversion the other way via:

  ```
  operator Bar() const { ...  return bar; }
  ```

- Both kinds apply in assignments and parameter calls, but not stream operators—see `LinkArg.{h,cpp}`.

Summary of Java2C++:

- Much of the line syntax is similar, differences-as-noted.
- File handling is different: {`.h`,`.cpp`}, `#include`, `make`...
  Recitations are emphasizing this and stream I/O.
- C++ has a richer *type system*, and the notion of *value* differs from Java on non-primitive types.
- And there are C++ "references," which I prefer to call "aliases." They inhibit copying, provide assignment and streaming targets, and behave like values when read from.
- Class hierarchies are similar, but the Java default method behavior requires extra work in C++ to get: `virtual`, pointers...
- Templates have similar use to Java generics, but work differently under-the-hood...and give beastly errors...