Why are you here? Why are you taking this class? No wait... many of you are thinking “because its required”. That’s true, but it only pushes back the question: why does the faculty require this course? What should you learn? What do we want you to learn?

*Here are some reasons:*

- To learn C++.
- To learn a 2\textsuperscript{nd} language (and how to learn a second language)
- To bridge the gap between CS200 & CS270.
- To learn what is going on “under the hood”.
- To learn more about Java!
- To learn how to select a language
- To learn about memory management.
- To learn to write efficient and/or real-time programs.
- To learn (through experience) how to write re-usable code
- To learn tools for developing code in unix
- To learn to write, maintain and test large programs

So there is a theme here. This is your 4\textsuperscript{th} and last programming class. After this, you will have classes on specific topics, such as software engineering, graphics, compilers, etc. But you will never again have a course on programming per se. So this class has to turn you in to sophisticated (a.k.a. professional quality) programmers.

Part of being a sophisticated programmer is knowing how to write/test/debug and maintain large programs. After this, you have to be ready to go write a ray tracer from scratch (CS410) or a distributed operating system from scratch (CS455)

Another part is integrating design with implementation. Customers will come to you with tasks, not designs. You have to understand the task (not take the customer at face value), create the designs yourself, and implement it.

Yet another part is writing code that can be re-used in other unanticipated contexts, because its semantically clear, modular, documented, and well-tested. If you always start from scratch, you will not be an efficient or cost-effective programmer.

But the biggest theme is getting you to understand computer systems at many different levels of abstraction. You should be able to program in an abstract level. When necessary, however, you should also be able to understand what the compiler is doing to your code, how your data is organized, where your data is in memory, when your program makes operating system calls, etc.
In other words, we have taught you abstract object oriented programming (CS 160/161/200) in Java, and we have taught you the basics of architecture (CS270), but we have left a large gap in between. This course fills that gap.

In other words, yes, this course will teach you C++, but more importantly it will teach you what goes on “under the hood” in both C++ and Java so as to bridge the gap between CS160/161/200 (OO programming) and CS270 (architecture).

If you haven’t programmed in C++ before, don’t feel intimidated. You don’t have any bad habits to revert to. Most people who “know” C++ have a shallow knowledge of it. Often they are self-taught, or taught by a high school teacher who (no offense) may not have been a computer scientist. The people who enter with C++ experience do not, as group, outperform those who enter without it.

Let’s talk about Chapter 0 of the text. It’s a mixed blessing.

- Good: lots of detailed information about the differences between Java and C++. We will make use of this.
- Bad: the book is snide. It is clear the author thinks Java is “better” than C++. I think this is ridiculous: its like saying salt is better than butter. Both butter and salt are good; often they are good together (e.g. popcorn). But you can’t fry in salt and you can’t preserve with butter. They both have their uses. Use the book’s opinions as an opportunity for critical analysis. Which ones are right? Which ones wrong? When does the author’s language (mis)lead?
- Bad: the book gets the details right, misses the big picture entirely.

Chapter 0 discusses “high-level differences” and “ten reasons to use C++”. It misses the big picture entirely. Let’s look at the ten reasons:

1. C++ is still widely used
2. Templates
3. Operator overloading
4. Standard Template Library
5. Automatic Reclamation of Resources
6. Conditional Compilation
7. Distinctions between Accessor and Mutator
8. Multiple Implementation Inheritance
9. Space Efficiency
10. Private Inheritance

This is a correct (although not complete) list of differences between C++ and Java. Such a list is useful, so that you know what to watch for when switching languages. You know what new features to exploit, and what old features not to count on.

But I don’t think anything above is a reasons to use C++. These are low-level language features. They aren’t broad themes.
Now, imagine you are about to start a new programming project. What language do you do it in?

Don’t pick a language based on detailed language features! That would require designing your program at the level of code, to figure out which features will be good. In multiple languages, before you write any code, just to pick the language. Bad idea.

So the list above is true and useful, but mislabeled. They aren’t reasons to use C++; they are differences between the languages to be aware of.

So let’s go back to the list of “high-level differences”. They are:

1. Compiled vs Interpreted Code
2. Security & Robustness
3. Multithreading
4. API Differences

This is better; it touches on some bigger themes. But its still too detailed. Let’s make a list at the right level for picking a language.

A language, like a program, is the result of many, many decisions. Languages are built to serve specific needs. You don’t want to retrace and evaluate every little decision. To judge whether a language will suit your needs, ask whether it was designed for a similar purpose. (This is why Chapter 0 starts with history. What were the designers of C++ -- or Java – thinking?)

The C language was designed in order to write operating systems. (Prior to C, OSs were written in assembly.) Operating system priorities included:

1. directly reflecting the hardware (for writing device drivers),
2. real-time response
3. machine efficiency

C++ was written to bring the object-oriented paradigm to C. Compatibility with the large body of C/Assembly code was therefore a priority. The goal was to be able to do what C was doing (including operating systems) in an object oriented way.

So here is my list of reasons to use C++:

1. To interface with legacy code (C++, C, Assembly)
2. To directly interact with hardware.
3. To optimize machine efficiency (time & space).
4. For real-time performance.

Note that points #3 and #4 are different. The first is a general call to efficiency. When deciding between C++ and Java, is it worth it to increase the time & difficulty of programming, in order to make the program run faster (maybe 10 times, maybe only 2)? If so, then point #3 applies. The 4th point is stricter: real-time performance is defined as a guarantee that an event will happen within a fixed time. For reasons will discuss more later on (the garbage collector), this is fundamentally impossible in Java.
Side note: Weiss finally figures it out. Look at Section 13.1 (JNI interface). In it, he admits that legacy code, hardware access and efficiency are reasons to program in C++. (He misses real-time, but hey).

What about your other options? What were the goals of the Java designers? They were focused on application programs that run on top of an operating system, and on top of networks with different operating systems.

So here is my list of reasons to use Java:

1. To optimize programmer efficiency
2. For portability
3. For security

In an ideal world, we would like all of the above. But we don’t know how to do this. There are too many contradictions.

- Direct access to hardware vs. Security (If I can directly access your disk... but if I can’t, how do I write a device driver for your disk drive?)
- Directly interacting with hardware vs Portability (What if I access a piece of hardware you don’t own? Big-Endian vs Little-Endian?)
- Machine efficiency vs programmer efficiency (More choices create the possibility for both more efficiency and more errors.)
- Legacy code: native integration vs JNI. C++ integrates seamlessly. Java integrates through JNI and less efficiently. But again, machine vs programmer efficiency...

Now, is Weiss an idiot? No. There is a lot of good stuff in your text; that’s why we use it. But he is weak on the big picture. Let’s look again at his “high-level differences”:

- “Security & Robustness”. Security is absolutely one of the big differences: it’s a goal of Java, not of C++. Robustness I would characterize as part of programmer efficiency (fewer bugs, less debugging), but it is also a security issue (e.g. ability to write out of bounds), so it’s a matter of taste where you put it.
- “Compiled vs Interpreted Code” is correct, but too specific. The broader issue is machine efficiency vs portability. C++ uses compiled code for speed, Java uses interpreted code for portability. This is an example of a speed/portability trade-off.

It’s not the most important example, however. The Java pre-interpreter compilers are getting better. And people may soon build hardware that matches the Java virtual machine to improve machine efficiency.

Other trade-offs: Java insists on bounds-checking all vector/array accesses. This is slow but secure. C++ doesn’t: faster but insecure.

Java uses a garbage collector. This is programmer efficient, but not as machine efficient as well-written programmer-managed memory. It’s a sharp knife however: poorly-designed memory
management in C++ is not only a common cause of bugs, it can even be slower than garbage collection if done badly enough.

C++ wants to know the data type of all objects at compile time (almost); Java relies on run-time type checking. Again, programmer vs machine efficiency.

- Multi-threading as an example of API differences: Java has a very large API, because Java subsumes much of the operating system: graphics, multi-threading, file dependency tracking and event handling are all examples of operating system tasks subsumed by Java. This is necessary for portability, and useful for programmer efficiency. C++ has a much smaller API (STL is about it; Boost is unofficial). Less portable, possibly harder, but arguably more efficient.

Think about it: Interpreted vs compiled. Why is that a reason to pick one language over another? It isn’t. But efficiency and portability are. Reason at the level of goals.

So Weiss’ list is too specific, and also biased. It is worded to favor Java: robustness is good, right? Benefits of C++ (e.g. real-time) are not mentioned.

But it doesn’t mean C++ is better than Java, or vice-versa. Different tools for different tasks. Look at my list of goals, C++ vs. Java. That is how you choose. Which list better suits the goals of your project?

Let’s look too at Weiss’ 10 reasons to use C++. These are real differences, but they should be seen as examples of the goals of the languages:

- The 1st point is C++ legacy code. Good. But it also applies to C and Assembly legacy code. (Note the use of the word “still”. Sorry Mark, but C++ isn’t going away any time soon.)
- Templates are a mechanism that improves machine efficiency through better compile-time typing. (Closest Java analog: generics.) A bit hard to use. Example of machine vs programmer efficiency trade-off.
- Operator overloading. I like this in C++. Java should have it – it’s an example of “protecting” the programmer too much. Java was designed partly as a compendium of C++ best practices for applications programs. Overloading is sometimes a bad idea, so Java bans it. But many times it is a good idea... C++’s sharp knife principle.
- STL is an API implemented in templates. It’s quite elegant and very efficient. More efficient than Java generics, but harder to use.
- Automatic Reclamation of Resources (i.e. destructors). Useful and efficient, but part of the sharp knife principle. In particular, part of the fact that C++ puts the burden of memory management on the programmer.
- Conditional compilation. Very handy for system or machine specific code. Violates portability.
- Accessor/Mutator distinctions. Java folks like this for style reasons, but C++ has it for efficiency.
- Multiple inheritance – is like overloading. Often it is a really bad idea. But occasionally it is an elegant, beautiful and programmer efficient technique. Again, sharp knife – know when to use it, and when not to.
- Space Efficiency. Again, it’s about efficiency.
- Private inheritance. Perhaps sharp knife again, although I personally find this one less useful. I think he just wanted 10 as a number.
There are other, bigger differences he doesn’t mention. I don’t know why:

- **Tree of objects vs forest of objects.** In Java, all objects share a common root; not true in C++. The object tree supports run-time typing in Java, which isn’t emphasized in C++. The forest of trees encourages multiple inheritance, which Java doesn’t allow.
- **Objects as dictate (Java) or option (C++).** Object-oriented programming is a best practice, and Java enforces it. C++ allows it. But there are instances where OO is not the best design; where functional is best. (Functional is the opposite of OO: nothing has state, everything is side-effect free.)
- **Separate compilation.** C++ divides declarations from implementations, putting them in separate files. This allows very big programs to be recompiled quickly (e.g. operating systems). But it is also an opportunity for bugs (mostly caught by the compiler) and increases the number of files and lines of text, so it hurts programmer efficiency.

OK, so which language to use: C++ or Java? Look at your trade-offs: if you need (1) to interface closely with C++, C or assembly legacy code, or (2) you care primarily about machine efficiency or (3) real-time performance or (4) directly controlling hardware, choose C++. If you care about (1) programmer efficiency, (2) portability and (3) security, pick Java.

Look at these goals: there is a reason why you should know both languages, and why neither language is going away any time soon.

- Almost all application programs should be written in Java
- Almost all system programs should be written in C++

Most computer systems have developed layers: a user-application layer written in Java, sitting on top of a systems layer written in C/C++. A computer professional knows both.

Note too that you have more choices than just Java and C++. There are lots of languages. Java is good for generic applications. Special-purpose applications invite still-higher-level languages, e.g. MATLAB (for linear algebra), Prolog (for logical inference), R (for statistical inference), Python (for scripting and rapid prototyping)…

Has this lecture been too abstract? Perhaps. Next week we will roll up our sleeves and start programming. Week 3 we start to accelerate…

But this lecture was important. You need to know why you are learning C++. You need to know how to choose between Java and C++ for any given project. And you will learn more languages are your career progresses; you need to learn how to select languages in general.

More specifically for this class, when there is a difference between C++ and Java, ask why. Usually the reason lies in the motivating applications (operating systems vs networked user apps). Sometimes the difference is just history or convention or an error. Asking why will help you intuitively understand the language, and become a better programmer in both languages.