Final Class / Recap

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Course Catalog

CSE333: Systems Programming

Catalog Description: Includes substantial programming experience in languages that expose machine characteristics and low-level data representation (e.g., C and C++); explicit memory management; interacting with operating-system services; and cache-aware programming. Prerequisite: CSE 351.

Course Catalog goals

- *"substantial programming experience in languages that expose machine characteristics and low-level data representation (e.g., C and C++)"*
  - Homeworks (exercises)
- *"explicit memory management"*
  - Yep
- *"interacting with operating-system services"*
  - files, sockets, read/write
- *"cache-aware programming"*
  - nope

- performance?
- code clarity?
- maintainability?
- portability?
- code correctness?

The Actual Course

- The course’s expectations are quite modest
- It presumes very little programming experience
- It equates having a working homework with meeting the course goals

- At this point you have more programming experience
- Let’s review what was covered with that perspective

The C Programming Language

- C is a bridge that connects higher level languages (at least declarative languages) with the hardware
  - Explaining that is sort of the goal of CSE 351?
- All code, in every language, runs on the same hardware(s)
  - The hardware is capable of computing anything that can be computed
- Why does the language matter?
- Two kinds of performance:
  - Run-time
  - Code/maintenance time

C and Runtime Performance

- Run-time performance
  - C is from an era where people were smarter than compilers
  - The programmer is given a set of tools that translate pretty directly to the hardware/implementation
  - The language and runtime don't try to do anything complicated
    - They try hard to stay out of the way
    - One classic way to do that is to saddle every operation with some feature that isn't needed in some application
      - Example: arrays
      - If C had arrays that knew their own length and checked for out of bounds indexes, every operation on them would be more expensive
        - (If you want that, you can create it for yourself, but C doesn't make you have it if you don't)

C and Code Time Performance

- Pretty terrible, actually
  - For example, no strings, because...
    - There are libraries. What's the downside of using them?
  - No help in dynamic memory allocation and deallocation, because...
    - That would be slow at run time
  - Pointers
    - Great idea. Really fast. Has a reasonable abstraction (alias).
    - Many things that can happen at runtime have "undefined" effect – what they do can't be explained in terms of the C language
    - That makes understanding what your program does hard for you, and it makes automated analysis of your program challenging
- Convincing yourself that the code is correct is difficult

Code Correctness (somewhat language independent)

- Automated testing

- Additional Tools
  - E.g., valgrind (and helgrind)
  - Possibly clint.py...

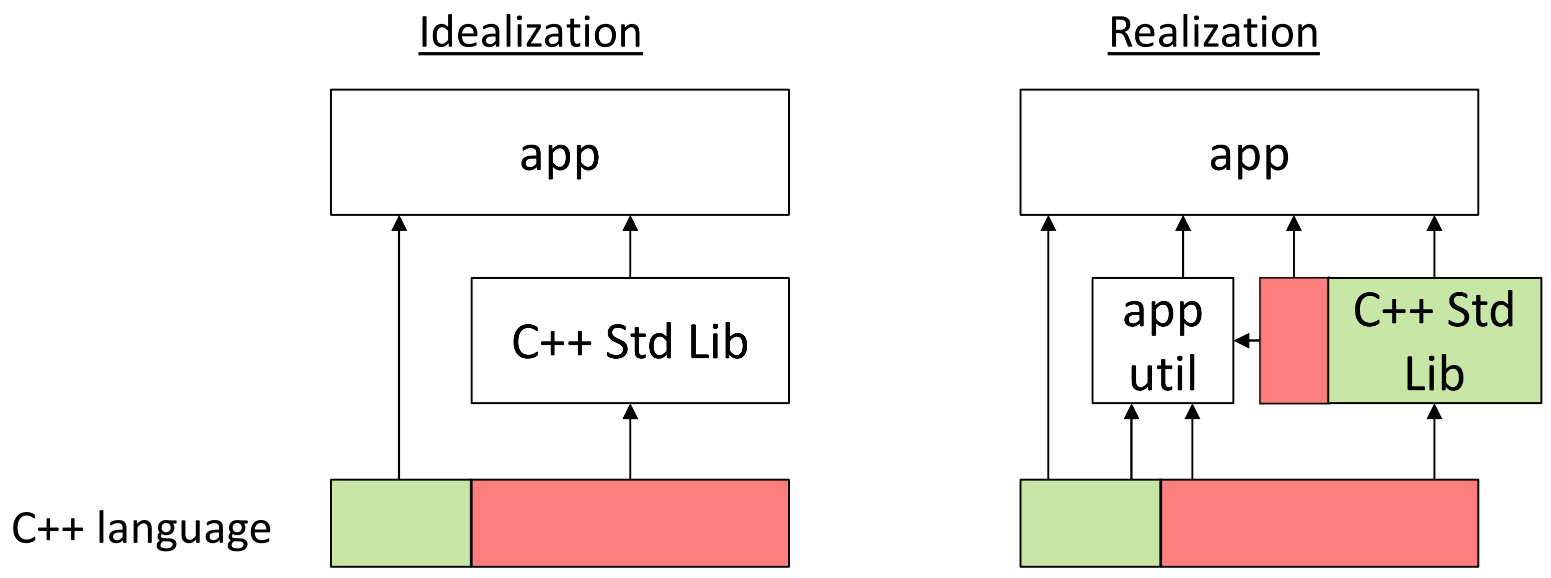
- C vs. Java
  - Potentially much faster execution, once it runs
  - Potentially many more unrecognized serious bugs encountered during production runs.

C Build Phases

- Preprocess
  - Create the C language source by programming in the preprocessor language (e.g., #include, #ifndef, etc.)
- Compile
  - File by file; pretty much precludes whole program analysis by compiler
  - To be able to compile, the compiler needs to know what the OS will be providing
    - e.g., memory layout
- Link
  - Connecting uses of names with definitions of names (binding)

From C to C++

- What happened?
- Greater expressiveness/code time efficiency without sacrificing run time efficiency



C++ Code Time Goals

1. Allow compact expressions of what you mean
   - Support generics
   - Provide some compiler help with type inference, done statically
     - It isn't always possible to do static inference, so the compiler sometimes needs help from the programmer
   - There is a lot going on at compile time to generate the code that will then be turned into machine instructions and run
   - Because so much code is being generated, there is a serious issue in how to reflect errors detected during compilation back to the code the programmer knows about
     - The error is detected in the generated code, which the programmer has never seen

C++ Code Time Goals

2. Allow "every execution" (specialization)
   - Most everything has options and/or can be overridden
   - Familiar syntax with very unfamiliar semantics
   - Most everything is an operator
     - new is an operator (malloc() is a function)
     - "function call" is an operator
   - Operators can be overloaded

3. Allow the programmer to place some restrictions on how their code is used
   - E.g., public vs private, const, deleted constructors
   - I'm going to claim that this is a kind of "meta-programming"

C++ Code Time Goals

4. "Allow" the programmer to communicate annotations to the compiler that allow it to do things it wouldn't otherwise be able to do (e.g., code optimizations)
   - E.g., const methods
   - E.g., move assignment/construction

Static vs. Dynamic Information

- "Static" usually means code time, "dynamic" usually mean run time
- Obviously, you code at code time
- There are things you know statically and things that can't be know except dynamically
  - E.g., if you want to read "a line of input" you can't know how long it will be until run time
    - And even then you can't know how long it will be until you've read it
  - That makes preallocating enough resource impossible
    - Programmers make it "likely" by allocating a lot, and that led to security issues...
  - And that motivates "stream processing" – operating item by item (say, byte by byte) on an unknown number of items

Static vs. Dynamic

- You don’t know statically how big a file you need to read will be
- You don’t know statically how much data will arrive as one request on a network socket
- What to do?
  - Stream process
  - read() in big chunks for performance, but of statically determine size
  - keep reading dynamically until you’ve read the entire thing

Static vs Dynamic

- Whenever you step outside what the language standard defines, you're subject to rules that may depend on where you're running
  - E.g., any system call
    - E.g., what can happen when you try to read a file; what are the possible return values from read()?

- These rules tend to be relatively poorly documented and poorly integrated with the language concepts, compared to the language itself

- Must check for errors, but it's often hard to understand when they can occur and what to do about them

When Should It Happen?

- We start with a simple model of static and dynamic
  - code time vs. run time
- Things are more complicated
- You can try to generalize the idea of static and dynamic when thinking about systems
  - "Static" is before you knew some important value and "dynamic" is when you know it

Examples

- Linking ("Binding")
  - I know the name of method I want to invoke at code time
  - I don't know to address that method in a machine instruction until I know its address in memory

  - I know the web server is called cs.washington.edu from the time I learn of it
  - I don't know it's IP address until I try to contact it

Examples

- Templates
  - I know what the abstract algorithm is at template code time
  - I don’t know what the concrete implementation coding the use of the template

- System Call Return Codes
  - I know the symbolic name EWOULDBLOCK as a possible return from read(), but I don’t know what value that actually has until compile time (system dependent .h file)

There’s More

- There is more than code, compile, link, and run times
- Importantly, there is build time
- Build time includes configuring the code – for example, the set of source files to be used – for
  - some target executable environment
  - to include/exclude certain features
  - to defer choices about what other modules to use to the person building the code rather than the code author
    - E.g., what database system (mysql, oracle, tinydb, mongodb?)

Metaprogramming

- Metaprogramming is (roughly) about writing code that generates code
  - The preprocessor: #ifdef
  - Templates: template<class T> class MyClass { ...};
  - make: Dynamically generates a shell script (usually to build some project)
    - Apache ant, gradle
  - configuration tools
    - gnu automake, cmake

Build Time / Portability

- One way to get portability is to standardize
  - Program to the standard
  - Insist every user's system supports the standard
    - Yeah, right...
- An alternative is to defer configuring your code until you can see the client's system
  - One possibility is build time
  - Another possibility is run time...

- Configuration can be so hard that one approach is to provide virtual machine images
- A more recent/lighter weight approach is to provide containers
  - An abstraction in addition to processes...

Final Comment About Times

- We think “code time, build time, compile time, link time, run time” as clear phases
  - Mostly every phase can be done later than the simple model says
    - If you could do it earlier, you must have all the information needed to do it later
    - Plus, you may have additional information if you do it later that will let you do a better job
  - There is a reason to try to do every phase earlier as well
    - Cheaper
    - Also, you may have information earlier than has been discarded by later
- You can compile at run time, link at run time, detect some kinds of bad pointer uses at compile time, ...
- Try to think of the phases in terms of what they do, not when they occur
  - Every programming system includes these ideas, but when they do things can vary

More Review: Concurrent / Asynchronous

- Computing is concurrent / parallel / asynchronous
  - Users, I/O, processors, networks
- Thread vs. process
  - Thread: shared address space -> fast inter-thread communication
  - Process: isolation from other processes implies certain reliability and error handling semantics
- With “remote” operations usually comes some uncertainty about processing times, and with that...
- Comes high latency operations, and performance issues

Dealing with High Latency Operations

- We've only barely touched on this in this course
- Key questions:
  - How do I start such an operation?
  - How do I know when it's done?
  - Can I do some other work while the operation is going on?

What Did You Learn in this Course?

- For sure, some mechanical things about C and C++ programs and programming

- We'd hope some useful experience with programming, independently of the languages
  - Identifying useful abstractions and translating those into implementations

- A difficult challenge in C++ is to try to raise the level of abstraction probably higher than we're used to
  - It's something to think about, if you're inclined toward that sort of thing

Final Slide

- Our final exam is scheduled for Wed, 2:30-4:20
- We don’t have a final exam
- I’ll be available on Zoom for most of that time to talk about anything you may have thought about relevant to the material in this class and have questions about
  - *There is no final. Asking questions or not has no bearing on your final grade.*

- Going forward...
  - This class didn’t give you much opportunity to design implementations. If you’re interested, pick something relatively simple and implement it. It will quickly become bigger than you expected. (E.g., implement Twitter.)