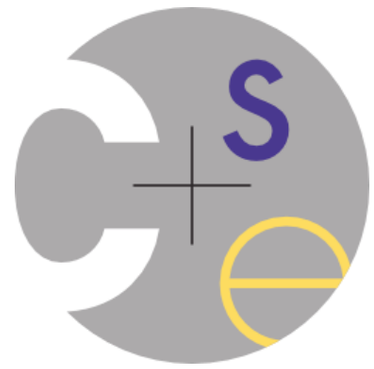
CSE 333

Lecture 1 – Course Overview

John Zahorjan

Department of Computer Science & Engineering

University of Washington



Welcome!

Today's goals:

- **introductions**
- *course syllabus*
- *understanding C*
- *quick C refresher*

Introductions

Course Staff (cse333-staff@cs)

John Zahorjan (Instructor)
Meghan Cowan (TA)
Renshu Gu (TA)
Kevin Kang (TA)
Soumya Vasisht (TA)

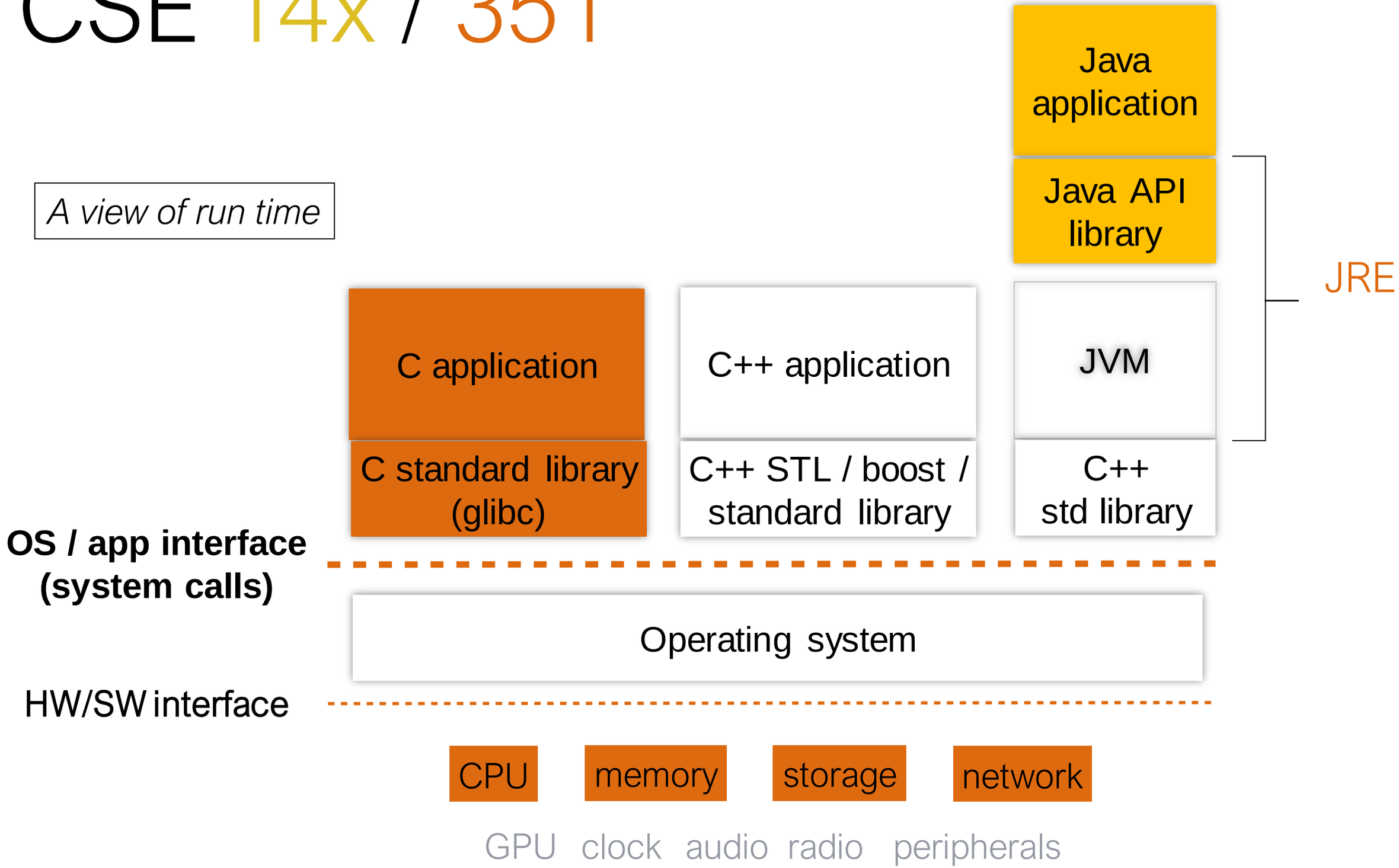
Welcome!

Today's goals:

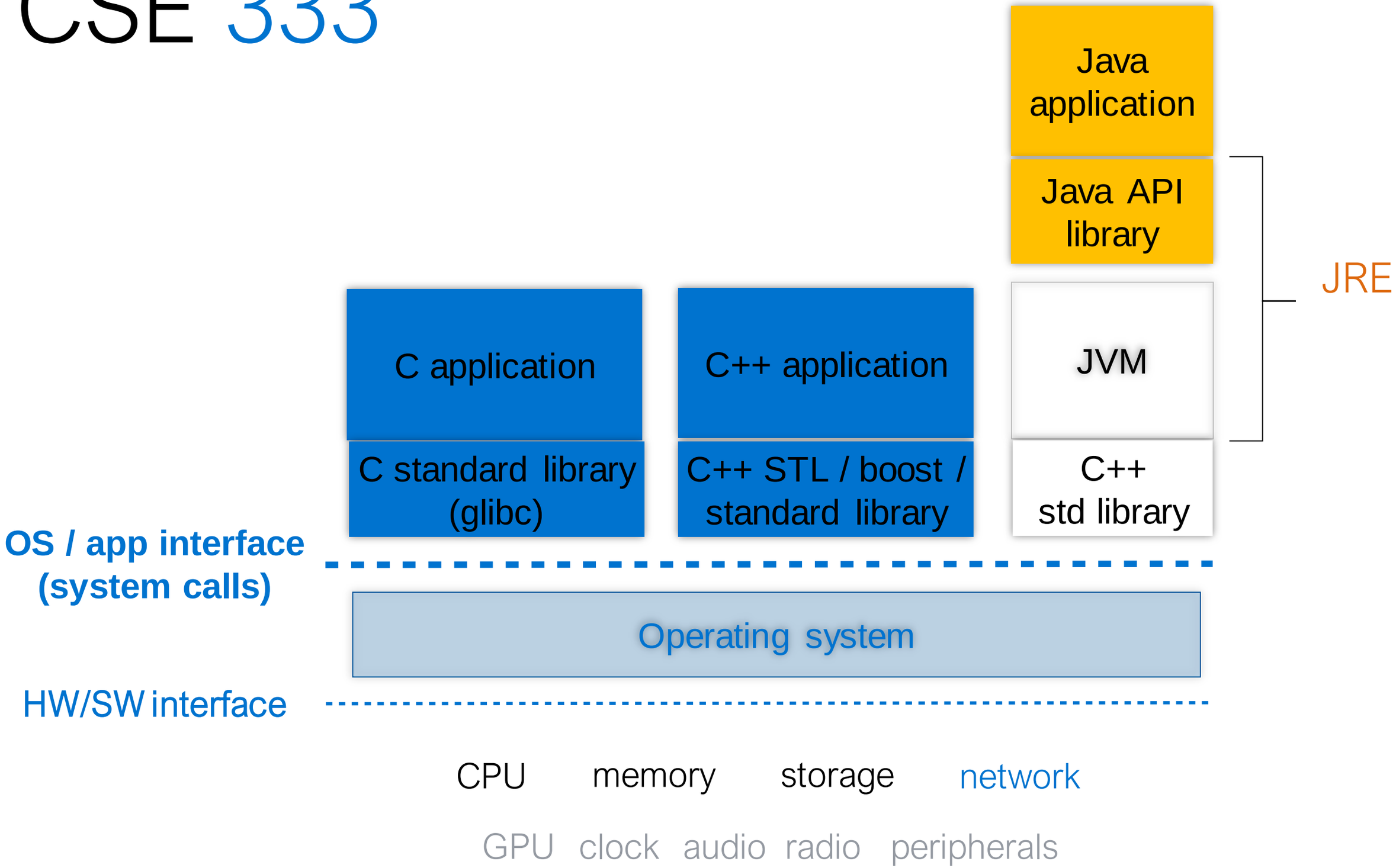
- *introductions*
- **course syllabus**
- *understanding C*
- *quick C refresher*

CSE 14x / 351

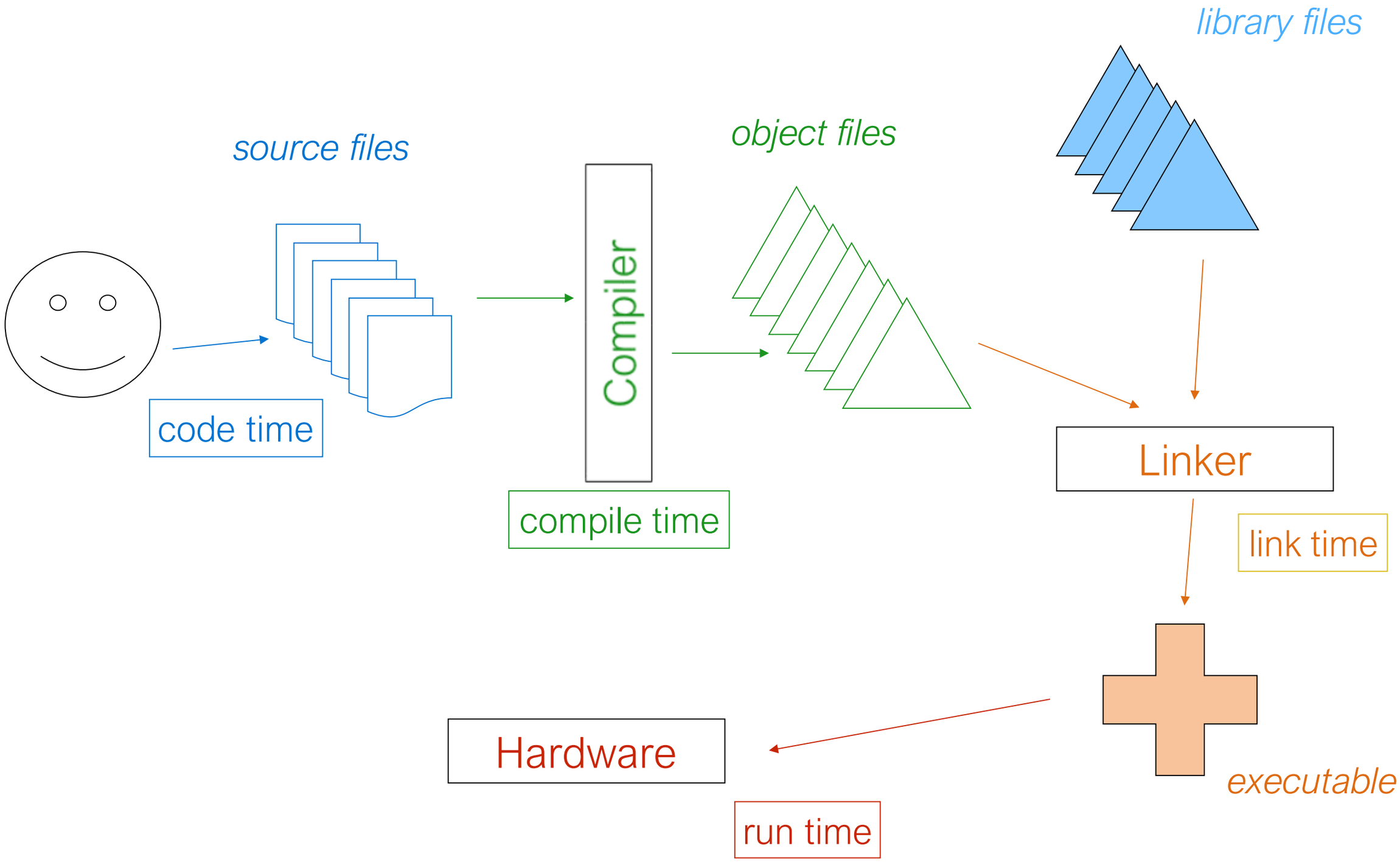
A view of run time



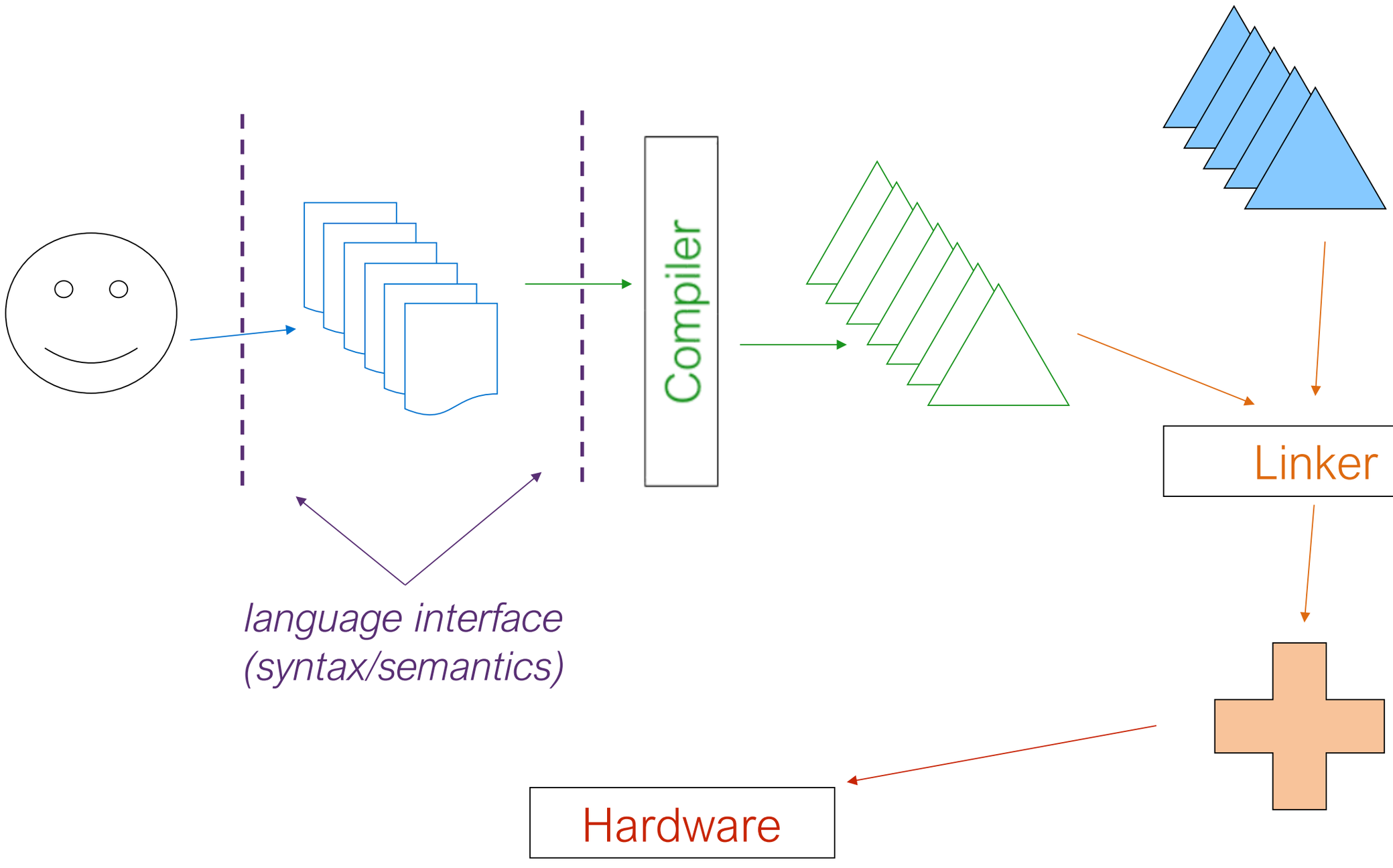
CSE 333


Java application
Java API library
JVM
C++ std library
JRE
C application
C++ application
C standard library (glibc)
C++ STL / boost / standard library
OS / app interface (system calls)
Operating system
HW/SW interface
CPU
memory
storage
network
GPU
clock
audio
radio
peripherals

Another viewpoint



What is “the language”?



The language interface

The language interface affects two primary things:

- How easy/hard it is for the programmer to implement the application
- How easy/hard it is for the compiler to translate the source file into efficient executable code

This is vastly simplifying, but to some extent the differences among languages are attributable to making different trade-offs between those two goals.

Java

- Designed to be “safe”
  - Helps the programmer avoid/find bugs
  - Helps the user trust the code
- Designed to be portable
  - The language completely defines what every legal statement means
  - An architecture-dependent JVM insulates the compiled code from the vagaries of the hardware it runs on
- Not designed to be fast…
  - Over time, speed became important, and more sophisticated compilation techniques were developed

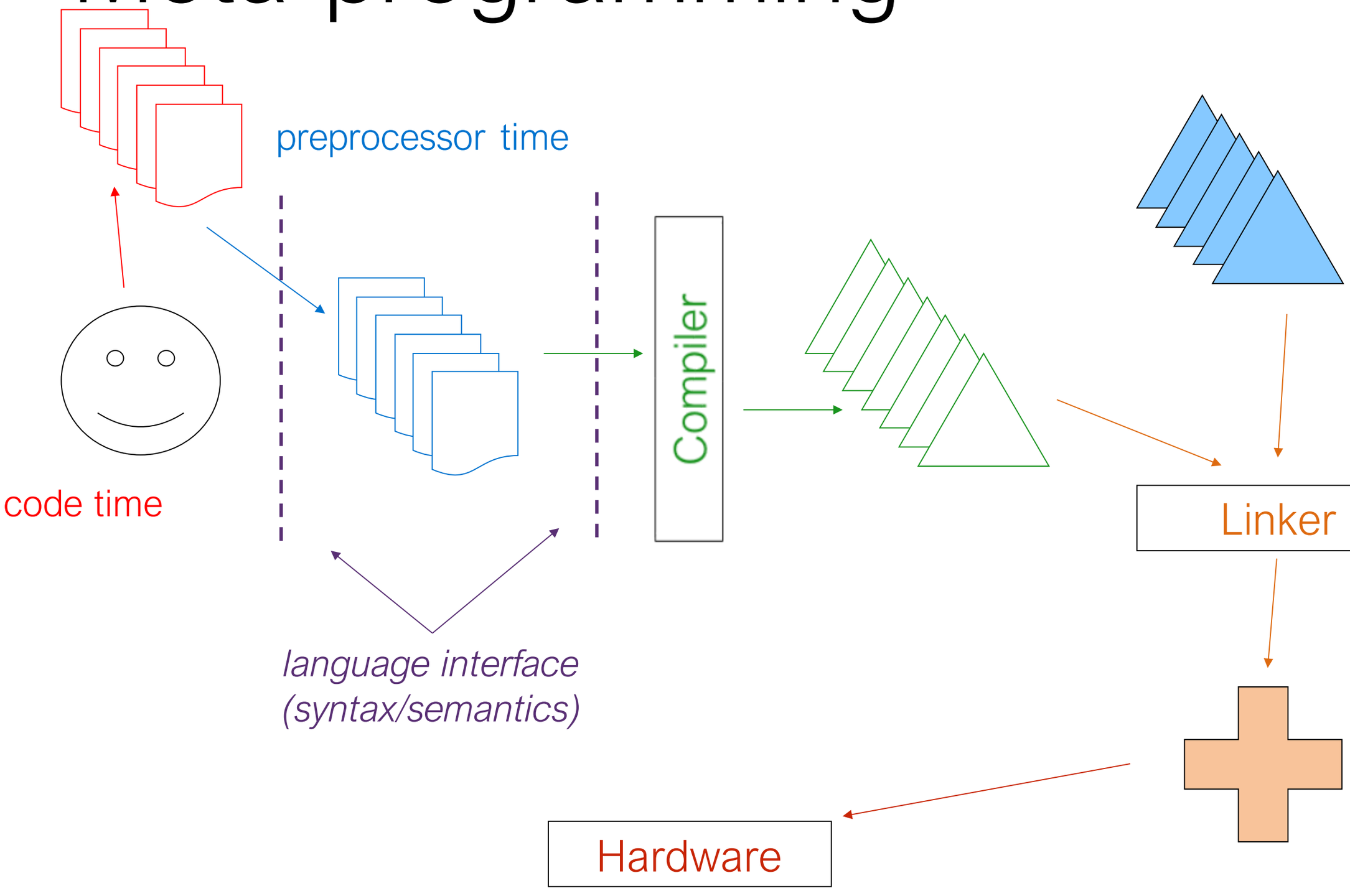
C

- "More expressive than assembly language"…
  - That is, it isn't trying to let the programmer express complicated ideas in only a few lines of code
- Implement anything hard as a library function, not part of the language
  - Often relies on "idioms" that make it possible for the right thing to happen
- Wants to preserve the same access to hardware resources that is available with assembler
  - memory, cpu, OS interface/abstractions

C (cont.)

- Possible to achieve small, fast executables
  - C doesn't get in the way of writing a small, fast program
  - It's design assumes it won't offer much help, either
- C "is portable"
  - portable if you recompile for each target system, maybe
- C introduces meta-programming
  - Writing code that generates code that is then compiled
  - #include, #define, etc.

Meta-programming



C++

- Originally designed to be a more expressive C
  - Classes
  - Initially compiled into C…
- A much more extensive runtime library
  - Collections, algorithms, IO, …
- As features are added, additional features are "required" (e.g., for runtime efficiency of executables)

CSE 333 This Quarter

"Systems Programming"

- Not apps, but code that facilitates other code
  - Must strike a balance between expressiveness and speed
- Sometimes close to the hardware
- Often concurrent
- Often asynchronous
- Often worried about failure modes

What you will be doing

Attending lectures and sections

- lectures: ~30 of them, MWF here
- sections: 10 of them, Thur., see time schedule - rooms might change
- quizzes
- Take notes!!!! Don't expect everything to be on the web

Doing programming projects

- Distributed Candy Crush

Doing programming exercises

- "one per lecture," due before the next lecture begins
- coarse-grained grading (0,1,2,3)

Midterm and a final exam

Deadlines & Conduct

Deadlines

- We'll have due dates. We have some late policy.
- For projects: you're the expert on you and what's best for you.
- A few days is just a few days; a week or more is something we'll notice
- For exercises: these present a considerable grading challenge, and so we'd like them to be on time.

Conduct

- You're the expert on you. Do what helps you learn.
- If unsure about whether something is allowed or not, ask.
- No one learns anything by simply copying code or not contributing to their team

Course web/calendar

Linked off of the course web page

- master schedule for the class (still needs midterm date)
- links to:
  - lecture slides
  - code discussed in lectures
  - assignments, exercises (including due dates)
  - optional "self-exercise" solutions
  - various C/C++/Linux/git/CSE resources

Communicating with us

Office hours: plan is to have something five days/week

Discussion board: stay in touch outside of class

Mailing list for announcements

You're already on it

Email: cse333-staff@cs.washington.edu

Welcome!

Today's goals:

- *introductions*
- *course syllabus*
- **understanding C**
- *quick C refresher*

Understanding C (and Java)

Java: "No"

- execution must be "safe" so…
- every legal statement has a well defined meaning and..
- I'll compile only legal statements and
- I'll check the rules even at runtime if I have to

C: "Sure!"

- It's *possible* for you to write correct programs and…
  if you do I'll compile correct code but…
  if you don't, I'll just do something
- If I put a feature in the language, the compiler has to implement it. If I just define a convention for programmers to follow, then when they do, great (and when they don't, it's their bug, not mine)

C Simplifications (simplified)

Java	C
types	bits
type checking (data)	“does it integer?”
type checking (methods)	“you’re the boss”
arrays	pointers
strings	null terminated ~~arrays~~ pointers
exceptions	return codes

Types vs. Bits (source code)

Which lines compile and which don't?

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
  long int  myInt = 3;
  char      string[10] = "My string";

  printf("(1) %s\n",  string + myInt);
  printf("(2) %ld\n", string + myInt);
  printf("(3) %p\n",  string + myInt);

  string[0] = 90.625;
  printf("(4) %s\n", string);

  myInt += "four";
  printf("(5) %ld\n", myInt);

  printf("(6) 0x%x\n", main);
  printf("(7) 0x%x\n", *(int*)main);

  return EXIT_SUCCESS;
}
```

Types vs. Bits (compilation)

```
$ gcc -std=c11 -Wall -g typesVsBits.c
typesVsBits.c: In function ‘main’:
typesVsBits.c:7:3: warning: implicit declaration of function ‘printf’ [-Wimplicit-
function-declaration]
   printf("(1) %s\n",  string + myInt);
   ^
typesVsBits.c:7:3: warning: incompatible implicit declaration of built-in function
‘printf’
typesVsBits.c:7:3: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
typesVsBits.c:8:10: warning: format ‘%ld’ expects argument of type ‘long int’, but
argument 2 has type ‘char *’ [-Wformat=]
   printf("(2) %ld\n", string + myInt);
          ^
typesVsBits.c:14:9: warning: assignment makes integer from pointer without a cast
[-Wint-conversion]
   myInt += "four";
         ^
typesVsBits.c:17:10: warning: format ‘%x’ expects argument of type ‘unsigned int’,
but argument 2 has type ‘int (*)(int,  char **)’ [-Wformat=]
   printf("(6) 0x%x\n", main);
          ^
```

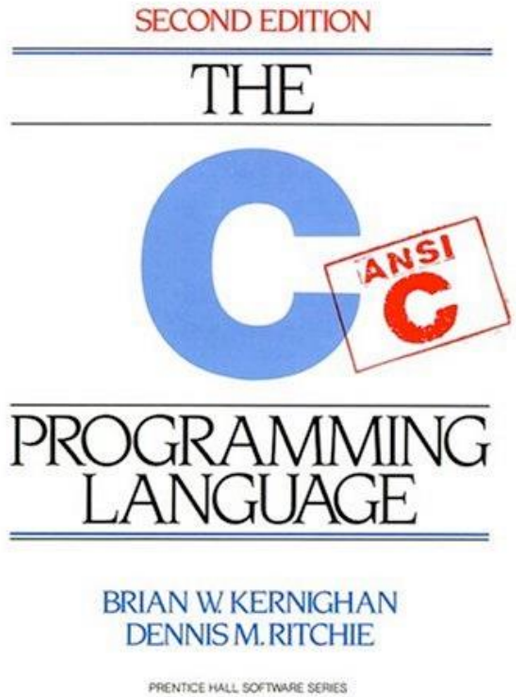
Types vs. Bits (execution)

```
$ ./a.out

(1) string

(2) 140731083028675

(3) 0x7ffe823678c3

(4) Zy string

(5) 4196184

(6) 0x400596

(7) 0xe5894855
```

The Linker

- C compiles one file at a time
  - It doesn't matter how many files you specify on the command line
  - It doesn't matter how many files your app has
- When C compiles a call to method foo(), it has no way to tell if foo() exists (unless it's in the same file as the call)
  - At compile time, should we care if it exists or not?
- When you create the executable, you care if foo() exists (static linking)
  - The linker "links" the instructions calling a method to the method's code
- The linker is more general than C
  - So, don't expect it to do much C-specific enforcement

C



Created in 1972 by Dennis Ritchie

- designed for creating system software
- portable across machine architectures
- most recently updated in 1999 (C99) and 2011 (C11)

Characteristics

- low-level, smaller standard library than Java
- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly

Welcome!

Today's goals:

- *introductions*
- *course syllabus*
- *understanding C*
- ***quick C refresher***

C v. Java

Things that are the same as Java

- much of the syntax for statements, control structures, function calls
- types (sort of): `int, double, char, long, float`
- type-casting syntax: `float x = (float) 5 / 3;`
- expressions, operators, precedence
  `+ - * / % ++ -- = += -= *= /= %= < <= == != > >= && || !`
- scope (local scope is within a set of `{ }` braces)
- comments: `/* comment */` `// comment`

Primitive types in C

integer types

- char, int

floating point

- float, double

modifiers

- short [int]
- long [int, double]
- signed [char, int]
- unsigned [char, int]

typical sizes

type	bytes (32 bit)	bytes (64 bit)	32 bit range	printf
char	1	1	[0, 255]	%c
short int	2	2	[-32768,32767]	%hd
unsigned short int	2	2	[0, 65535]	%hu
int	4	4	[-214748648, 2147483647]	%d
unsigned int	4	4	[0, 4294967295]	%u
long int	4	8	[-2147483648, 2147483647]	%ld
long long int	8	8	[-9223372036854775808, 9223372036854775807]	%lld
float	4	4	approx [10^{-38}, 10^{38}]	%f
double	8	8	approx [10^{-308}, 10^{308}]	%lf
long double	12	16	approx [10^{-4932}, 10^{4932}]	%Lf
pointer	4	8	[0, 4294967295]	%p

C99 extended integer types

Solves the conundrum of “how big is a long int?”

```
#include <stdint.h>

void foo(void) {
  int8_t  w;     // exactly 8 bits, signed
  int16_t x;     // exactly 16 bits, signed
  int32_t y;     // exactly 32 bits, signed
  int64_t z;     // exactly 64 bits, signed

  uint8_t a;     // exactly 8 bits, unsigned
  ...etc.
}
```

Similar to Java…

- Variables / scope
  - C99/C11: don't have to declare at start of a function or block
  - need not be initialized before use *(gcc -Wall will warn)*

varscope.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int x, y = 5;   // note x is uninitialized!
  long z = x+y;

  printf("z is '%ld'\n", z); // what's printed?
  {
    int y = 10;
    printf("y is '%d'\n", y);
  }
  int w = 20;  // ok in c99
  printf("y is '%d', w is '%d'\n", y, w);
  return 0;
}
```

Similar to Java…

const

- a qualifier that indicates the variable's value cannot change
- compiler will issue an **error** if you try to violate this
- why is this qualifier useful?

consty.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  const double MAX_GPA = 4.0;

  printf("MAX_GPA: %g\n", MAX_GPA);
  MAX_GPA = 5.0;  // illegal!
  return 0;
}
```

Similar to Java…

for loops

- C99/C11: can declare variables in the loop header

if/else, while, and do/while loops

- any type can be used; 0 means **false**, everything else **true**
- C99/C11: **bool** type supported, with #include <stdbool.h>

loopy.c

```
int i;

for (i = 0; i < 100; i++) {
  if (i % 10 == 0) {
    printf("i: %d\n", i);
  }
}
```

Similar to Java...

pointy.c

parameters / return value

- C always passes arguments by value
- C always returns by value
- “pointers”
  - lets you "pass by reference"
  - more on these soon
  - least intuitive part of C
  - very dangerous part of C

```
void add_pbv(int c) {
  c += 10;
  printf("pbv c: %d\n", c);
}

void add_pbr(int *c) {
  *c += 10;
  printf("pbr *c: %d\n", *c);
}

int main(int argc, char **argv) {
  int x = 1;

  printf("x: %d\n", x);

  add_pbv(x);
  printf("x: %d\n", x);

  add_pbr(&x);
  printf("x: %d\n", x);

  return 0;
}
```

Very different than Java

arrays

- just a bare, contiguous block of memory of the correct size
- an array of 10 ints requires 10 x 4 bytes = 40 bytes of memory (or maybe 10 x 8 bytes = 80 bytes)

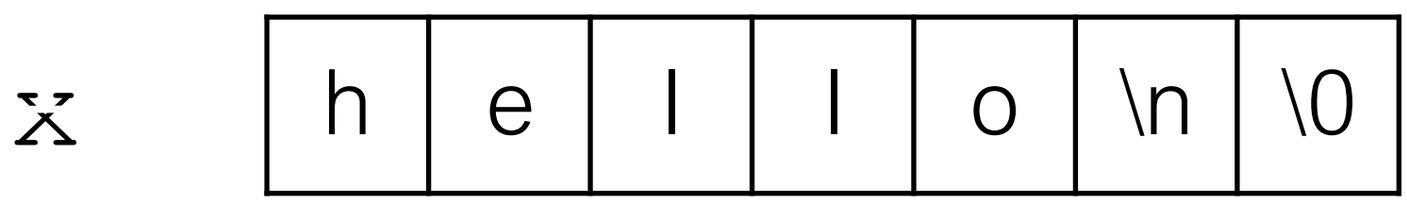
arrays have no methods, do not know their own length, don't even know they're arrays (except maybe at compile time)

- C doesn't stop you from overstepping the end of an array!!
- many, many security bugs come from this

Very different than Java

strings

- array of char
- terminated by the NULL character ‘\0’
- are not objects, have no methods; string.h has helpful utilities




```
char *x = "hello\n";
```

Very different than Java

errors and exceptions

- C has no exceptions (no try / catch)
- errors are typically indicated by integer error codes returned from functions
- makes error handling ugly and inelegant

crashes

- if you do something bad, you'll end up spraying bytes around memory, hopefully causing a "segmentation fault" and crash

objects

- C doesn't provide any
- You can adopt an object-oriented-like style, using struct
- C++ takes that approach and makes it work…

Very different than Java

memory management

- **you** must worry about this; there is no garbage collector
- local variables are allocated on the stack
  - freed when you return from the function
- global and static variables are allocated in a data segment
  - allocated before execution starts and freed execution ends
- you can allocate memory in the heap segment using malloc( )
  - you must free malloc'ed memory with free( )
  - failing to free is a leak, double-freeing is an error (hopefully crash)

Very different than Java

Libraries you can count on

- C has very few compared to most other languages
  - no built-in trees, hash tables, linked lists, sort , etc.
- you have to write many things on your own
  - particularly data structures
  - error prone, tedious, hard to build efficiently and portably
- this is one of the reasons C is a much less productive language than Java, C++, python, or others

For next time

Exercise 00 is due *before* class:

- look on the calendar or homework page for the link

Resign yourself to using the discussion board

- Get it to alert you of new messages

HW0 (project 0) will come out on Friday