CSE 333

Lecture 1 - Intro, C refresher

Welcome!

Today's goals:

- **introductions**
- *course syllabus*
- *quick C refresher*
- *why C?*

These are slightly modified versions of slides prepared by Steve Gribble

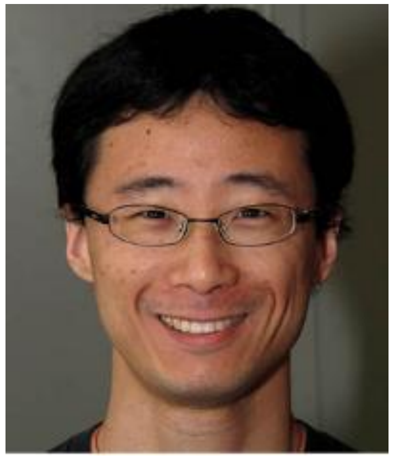
Us

John Zahorjan

James Okada

Renshu Gu

Johnny Yan



Overloading

The overload signup sheet is down here

- come sign up after lecture
- I'll hand the sheet in to the ugrad advisors
- by Monday, they'll let me (and you) know who gets in

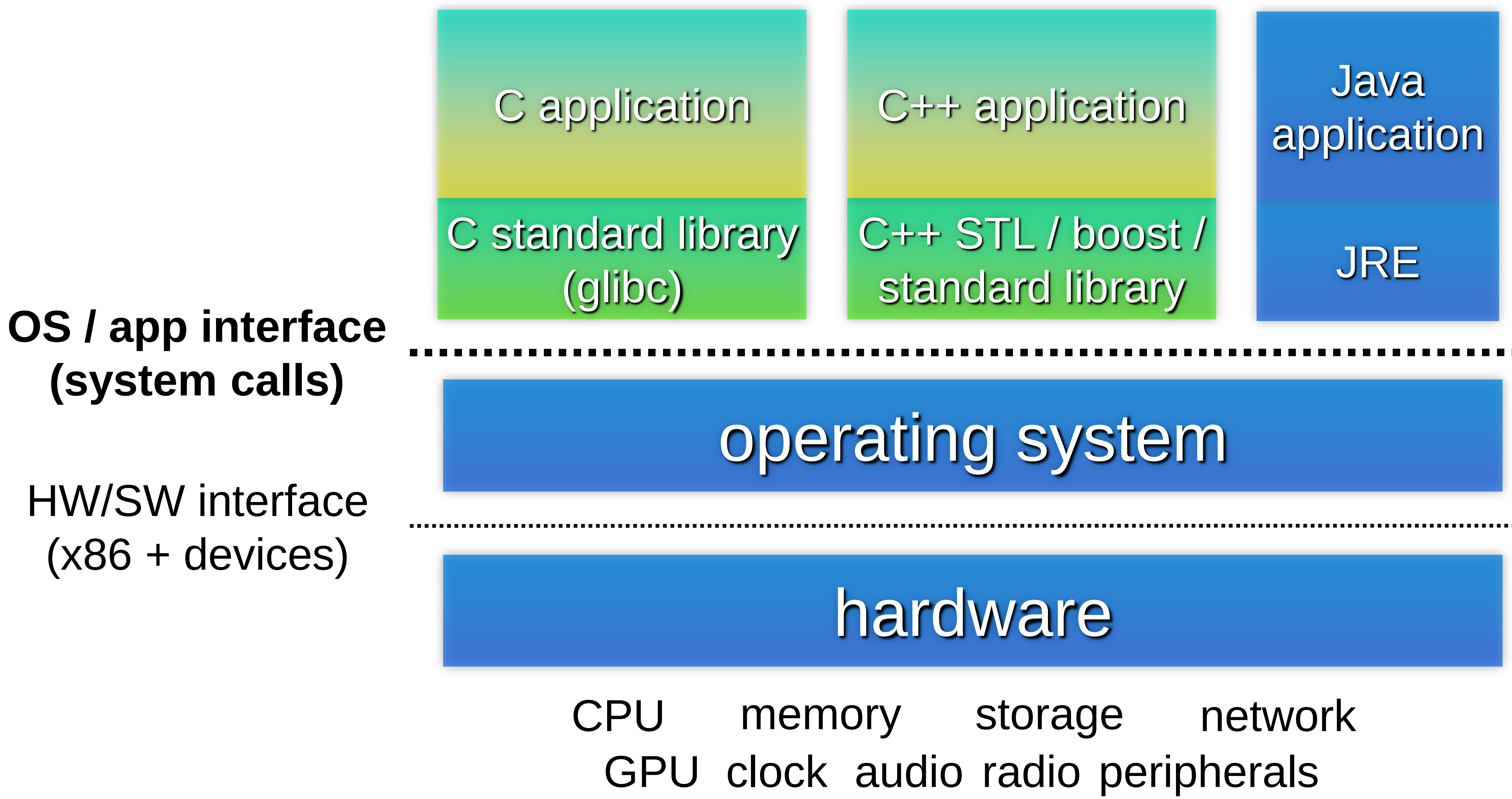
Welcome!

Today's goals:

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

These are slightly modified versions of slides prepared by Steve Gribble

Course map: 100,000 foot view



These are slightly modified versions of slides prepared by Steve Gribble

The Cost of Layering

C

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000

int main(int argc, char** argv) {
  int i, nTrials, trial;
  long int grandSum, sum, data[N];

  nTrials = atoi(argv[1]);

  for ( i=0; i<N; i++) data[i] = random();

  grandSum = 0L;
  for ( trial = 0; trial < nTrials; trial++ ) {
    sum = 0L;
    for ( i=0; i<N; i++ ) sum += data[i];
    grandSum += sum;
  }

  printf("Grand sum = %ld\n", grandSum);

}
```

timingTest.c

Java long

```
import java.util.Random;

class timingTest {
  private static final int N = 1000000;

  public static void main(String[] args) {
    long data[] = new long[N];

    Random random = new Random();
    for (int index=0; index<1000000; index++ )
      data[index] = random.nextInt();

    int nTrials = Integer.parseInt(args[0]);
    long grandSum = 0L;
    for (int trial = 0; trial < nTrials; trial++ ) {
      int sum = 0;
      for ( int i=0; index<N; i++ ) sum += data[i];
      grandSum += sum;
    }

    System.out.println(
        String.format("Grand sum = %d\n", grandSum)
     );

  }
}
```

timingTest.java

Java Long

```
import java.util.Random;

class timingTestLong {
  private static final int N = 1000000;

  public static void main(String[] args) {
    Long data[] = new Long[N];

    Random random = new Random();
    for (Integer index=0; index<N; index++ )
      data[index] = random.nextLong();

    Integer nTrials = Integer.parseInt(args[0]);
    Long grandSum = 0L;
    for (int trial = 0; trial < nTrials; trial++ ) {
      Long sum = 0L;
      for ( Integer i=0; index<N; i++ ) sum += data[i];
      grandSum += sum;
    }

    System.out.println(
      String.format("Grand sum = %d\n", grandSum)
    );
  }
}
```

timingLong.java

The Cost of Layering

Times in seconds

# Trials	C –O0	C –O3	Java long	Java Long
10	.040			
100	.224			
1000	2.100			

The Cost of Layering

Times in seconds

# Trials	C –O0	C –O3	Java long	Java Long
10	.040	.020		
100	.224	.056		
1000	2.100	.480		

The Cost of Layering

Times in seconds

# Trials	C –O0	C –O3	Java long	Java Long
10	.040	.020	.080	
100	.224	.056	.136	
1000	2.100	.480	.648	

The Cost of Layering

Times in seconds

# Trials	C –O0	C –O3	Java long	Java Long
10	.040	.020	.080	.464
100	.224	.056	.136	1.936
1000	2.100	.480	.648	13.473

Systems programming

The programming skills, engineering discipline, and knowledge you need to build a system

programming: C / C++

discipline: testing, debugging, performance analysis

knowledge: long list of interesting topics

- concurrency, OS interfaces and semantics, techniques for consistent data management, distributed systems algorithms, ...
- most important: a deep understanding of the "layer below"

Quiz: is data safely on disk after a "write( )" system call returns?

These are slightly modified versions of slides prepared by Steve Gribble

Discipline?!?

Cultivate good habits, encourage clean code

- coding style conventions
- unit testing, code coverage testing, regression testing
- documentation (code comments, design docs)
- code reviews

Will take you a lifetime to learn

- but oh-so-important, especially for systems code
  - avoid write-once, read-never code

What you will be doing

Attending lectures and sections

- lectures: ~28 of them, MWF in this room
- sections: ~10 of them, Thu (8:30, 9:30, or 10:30)

Doing programming projects

- 5 of them, successively building on each other
- includes C, C++; file system, network, possibly concurrency

Doing programming exercises

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

Exams

One possibility:

- Midterm, Friday, November 1
- Final, Wednesday, December 11

Course calendar

Linked off of the course web page

- master schedule for the class
- will contain links to:
  - lecture slides
  - code discussed in lectures
  - assignments, exercises (including due dates)

Welcome!

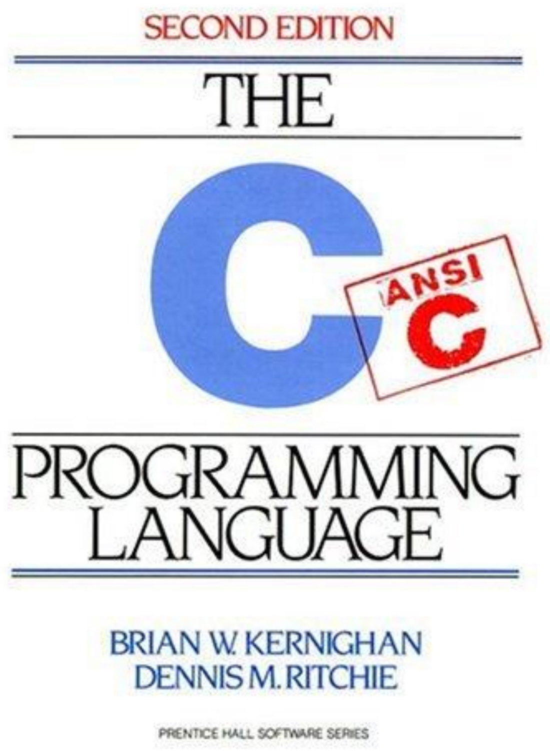
Today's goals:

introductions

course syllabus

quick C refresher

C



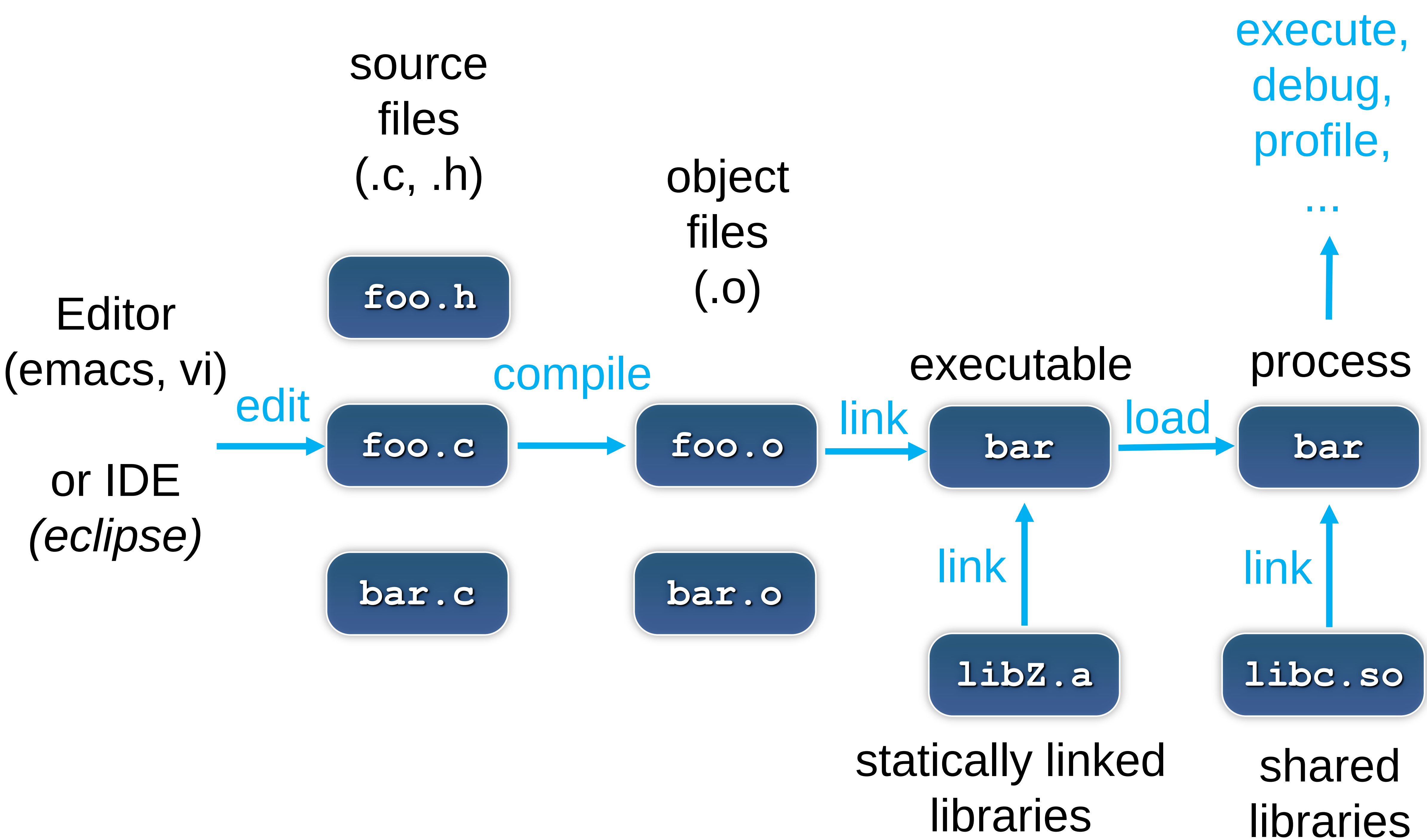
Created in 1972 by Dennis Ritchie

- 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

C workflow



These are slightly modified versions of slides prepared by Steve Gribble

From C to machine code

C source file
(dosum.c)

```
int dosum(int i, int j) {
   return i+j;
}
```

C compiler (gcc -S)

assembly source file
(dosum.s)

```
dosum:
pushl   %ebp
movl    %esp, %ebp
movl    12(%ebp), %eax
addl    8(%ebp), %eax
popl    %ebp
ret
```

assembler (as)

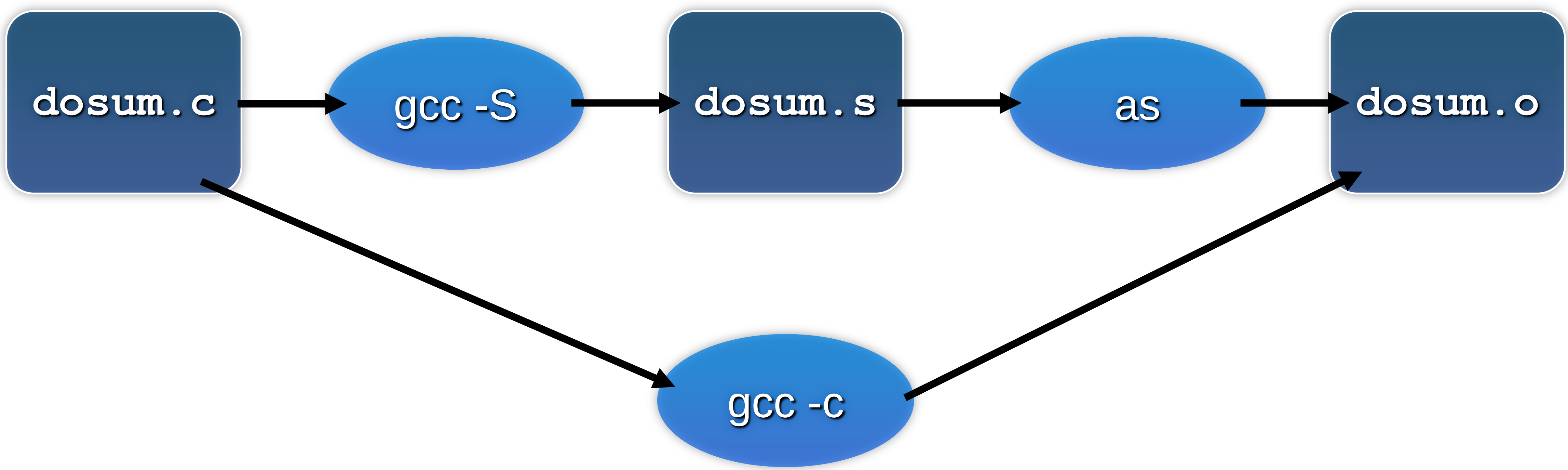
machine code
(dosum.o)

```
80483b0: 55
89 e5 8b 45
0c 03 45 08
   5d c3
```

Skipping assembly language

Most C compilers generate .o files (machine code) directly

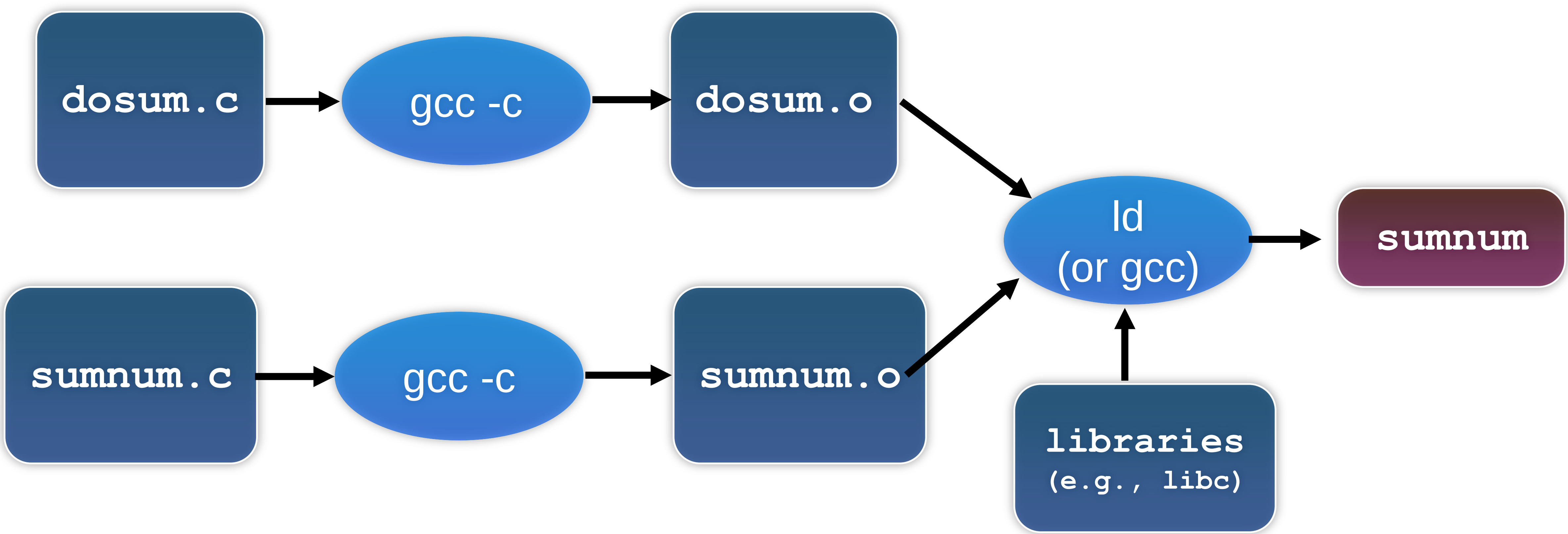
- i.e., without actually saving the readable .s assembly file



Compiling multi-file programs

Multiple object files are ***linked*** to produce an executable

- standard libraries (libc, crt1, ...) are usually also linked in
- a library is just a pre-assembled collection of .o files



Multi-file C programs

C source file
(dosum.c)

```c
int dosum(int i, int j) {
   return i+j;
}
```

C source file
(sumnum.c)

```c
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

Multi-file C programs

C source file
(dosum.c)

```
int dosum(int i, int j) {
   return i+j;
}
```

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

This “prototype” of dosum( ) tells gcc about the types of dosum’s arguments and its return value

dosum( ) is implemented in dosum.c

Multi-file (Multi-compile) C programs

C source file
(dosum.c)

```
int dosum(int i, int j) {
  return i+j;
}
```

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

why do we need this #include?

where is the implementation of printf?

External symbols and linking

sumnum.o, dosum.o are object files

- each contains machine code produced by the compiler
- each might export global symbols (that can be referenced from other files)
- each might use external symbols
  - variables and functions not defined in the associated .c file
  - the compiler needs to know the type of an external symbol (so it can type check), but not the full implementation
- linking resolves these external symbols
  - connects the name to the definition

Let’s dive into C itself

Things that are the same as Java

- syntax for statements, control structures, function calls
- primitive types: 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

Similar to Java...

- variables and scope
  - C99: don’t have to declare at start of a function or block
  - C doesn’t require initialization 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;  // always ok
    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!  Compile time error
  return 0;
}
```

Similar to Java...

for loops

- C99: can declare variables in the loop header

if/else, while, and do/while loops

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

`loopy.c`

```
int i;

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

Similar to Java...

pointy.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;
}
```

parameters / return value

- **C always passes arguments by value**
- “pointers”
  - let you pass thing pointed to by reference
  - more on these soon
  - least intuitive part of C
  - very dangerous part of C

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

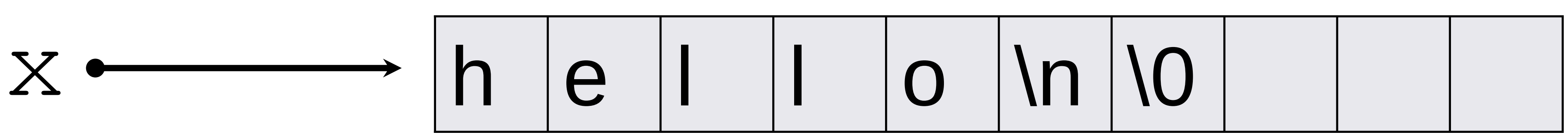
arrays have no methods, do not know their own length

- C doesn’t stop you from overstepping the end of an array!!
  - (Or the beginning of an array!!)
- many, many security bugs come from this

Very different than Java

C doesn’t have strings, it has a string convention

- “strings” are just arrays of char
- Are terminated by the NULL character ‘\0’
- Are not objects, have no methods; string.h has helpful utilities that follow the convention




```
char *x = ”hello\n”;
```

Very different than Java

Errors and exceptions

- C has no exceptions (no try / catch)
- errors are returned as integer error codes 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

- there aren't any; `struct` is closest feature (set of fields)

Very different than Java

Memory management

- **you** must to 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
  - freed only when your program exits
- you can allocate memory in the heap 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 main reasons C is a much less productive language than Java, C++, python, or others

Very different than Java…

Portability

See `sizeofs.c`

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

integer types

- char, int

floating point

- float, double

modifiers

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

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.
}
```

For Friday

Homework 0 is due:

http://www.cs.washington.edu/education/courses/cse333/13au/assignments/hw0/hw0.html

Exercise 0 is due:

http://www.cs.washington.edu/education/courses/cse333/13au/exercises/ex0.html

See you on Friday!