CO658 bonus material:
C for Java programmers

Stephen Kell

s.r.kell@kent.ac.uk

What is C?

C is a programming language invented by Dennis Ritchie in 1972

... as part of work on the Unix operating system at Bell Laboratories

At the time: a fairly general-purpose language, with a systems bias

Nowadays: primarily a language for systems programming and other low-*ish*-level tasks

One of two main ancestors of Java
(the other: Smalltalk)

C is still in very wide use today

Hello, world!

```
#include <stdio.h>

int main(void)
{
   printf(“Hello, world!\n”);
   return 0;
}
```

Initial observations:

- it looks a bit like Java...

- but simpler: no classes?!

  - here just a *top-level function*

    ... ≅ static method

- *#include* similar to “**import**"

- lots of other differences...

Why learn C?

“It’s a state of mind.”

- It forces you to learn concepts that will improve your understanding of CS.

”It’s a lingua franca.”

- The language of system-builders and language-implementers is drawn from the vocabulary of C.

“It’s useful.”

- Some tasks are most easily achieved in C.
  Some tasks are impossible except in C or a few other systems-y languages

“It’s fun!”

- I enjoy programming in C. Not *everyone* does....

A more interesting bit of code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
     struct bst *left;
     int num;
     struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
     struct bst *new_node = calloc(1,
        sizeof (struct bst));
     new_node->num = n;
     if (!node) { // empty tree
          assert(!root);
          root = new_node;
     } else if (n < node->num && !node->left) {
       node->left = new_node;
     } else if (n >=node->num && !node->right){
          node->right = new_node;
     } else if (n < node->num) {
          bst_insert(n, node->left);
     } else {
          assert(n >= node->num);
          bst_insert(n, node->right);
     }
}
```

A more interesting bit of code

```c
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
     struct bst *left;
     int num;
     struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
     struct bst *new_node = calloc(1,
        sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
         assert(!root);
         root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
         node->right = new_node;
    } else if (n < node->num) {
         bst_insert(n, node->left);
    } else {
         assert(n >= node->num);
         bst_insert(n, node->right);
    }
}
```

```java
import java.util.Scanner;
class SingletonTree {
public static class Node {
   Node left;
   int num;
   Node right;
}
private static Node root;
public static Node getInstance() { return root; }
static void insert(int n, Node node) {
   Node new_node = new Node();

   new_node.num = n;
   if (node == null) {
        assert root != null;
        root = new_node;
   } else if (n < node.num && null == node.left) {
        node.left = new_node;
   } else if (n >=node.num && null==node.right){
        node.right = new_node;
   } else if (n < node.num) {
        insert(n, node.left);
   } else {
        assert n >= node.num;
        insert(n, node.right);
   }
}
```

A more interesting bit of code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
        sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

- no classes, but can use **struct**s to group data

- no classes, but *the file is a module* (**static** is like Java ‘private’, not ‘static’!)

A more interesting bit of code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
       sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

- indirection (reference) is <u>explicit</u>, using *pointers:*

`T*`	is a pointer to type `T`
`*p`	means “follow `p`”
`p->f`	means `(*p).f`
`&n`	means “get me a pointer to `n`”

- i.e. no ‘value types’ vs ‘reference types’
  - ... can point to any type `T`
  - ... can *nest* any `T` in struct `S`

- more implicit conversions (here pointer to boolean)

A more interesting bit of code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
       sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

- no **new**, no garbage collector

- where possible *avoid* heap allocation, in favour of stack or global/static

- else use library functions like `calloc()` and `malloc()` to allocate heap memory

- ... and must use `free()` when finished with an allocation!

- find the bug in the code to the left

How is it *not* like Java?

It’s simpler... much simpler. (But not easier.)

- Java keywords with no *built-in* equivalent in C:

`abstract`	`implements`	`package`	`synchronized`
`assert`	`import`	`protected`	`this`
`catch`	`interface`	`public`	`throw`
`extends`	`native`	`strictfp`	`throws`
`finally`	`new`	`super`	`transient`

The standard library is also much smaller

- no string type! use character arrays
- no collections... `hsearch()`, `bsearch()`, `tsearch()`
- more reliance on third-party libraries...
  + in practice: “keep it simple”, “roll your own”...

How is it *not* like Java?

It’s much less friendly.

- C does not try to protect you from yourself
  - bad array index? no IndexOutOfBoundsException
  - bad cast? no ClassCastException
  - null pointer? no NullPointerException
  - divide by zero? no ArithmeticException
  - forgot to initialize a local variable? it takes a spurious value (or worse...)
  - `free()` the same object twice? Or use a pointer `p` after doing `free(p)`? Anything could happen!

How is it *not* like Java?

What *does* happen in any of these error conditions?

- Language spec says: anything could happen
  - “it could make demons fly from your nose”
  - this is called “undefined behaviour”
    (Note: there is no `try`/`catch`/`throw` feature)
- Each implementation *may* document some more specific behaviour
  - ... but in practice they usually don’t
- Why so? Complex question...
  - keep the *language* portable, incl. to tiny devices, low-level programming tasks, ...

How is it not like Java?

What *does* happen in any of these error conditions?

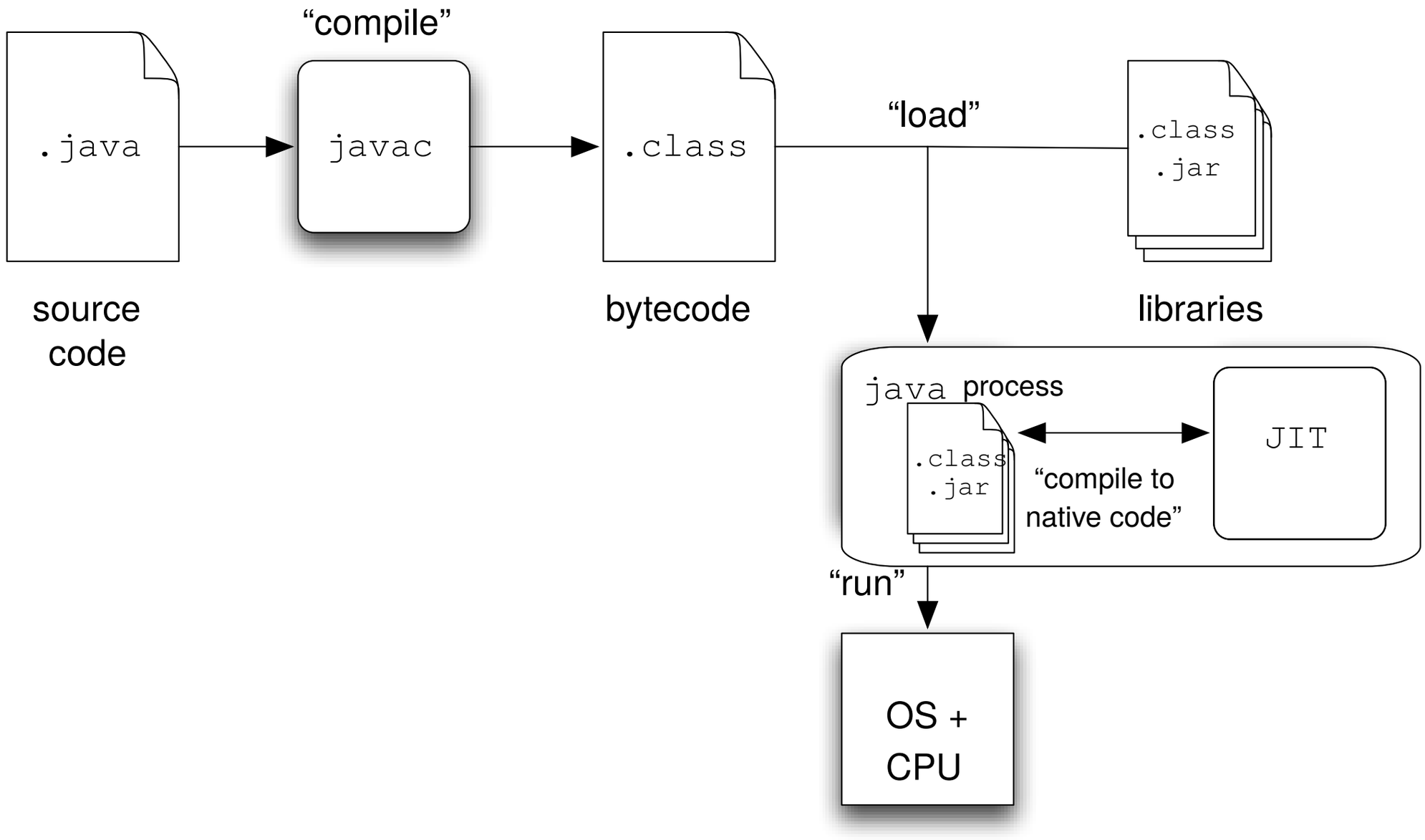
- In practice: maybe your program carries on...
  - e.g. uninitialized local? carry on with bogus value...
  - array out of bounds? access some neighbouring memory as if it was part of your array....
- ... or it gets halted by the operating system
  - “Segmentation fault”, “Floating point exception”
- ... or it does something truly inexplicable
  - if you have compiler optimizations turned on
    - which you normally do!

How can we mitigate all this unfriendliness?

C programmers mitigate this using:

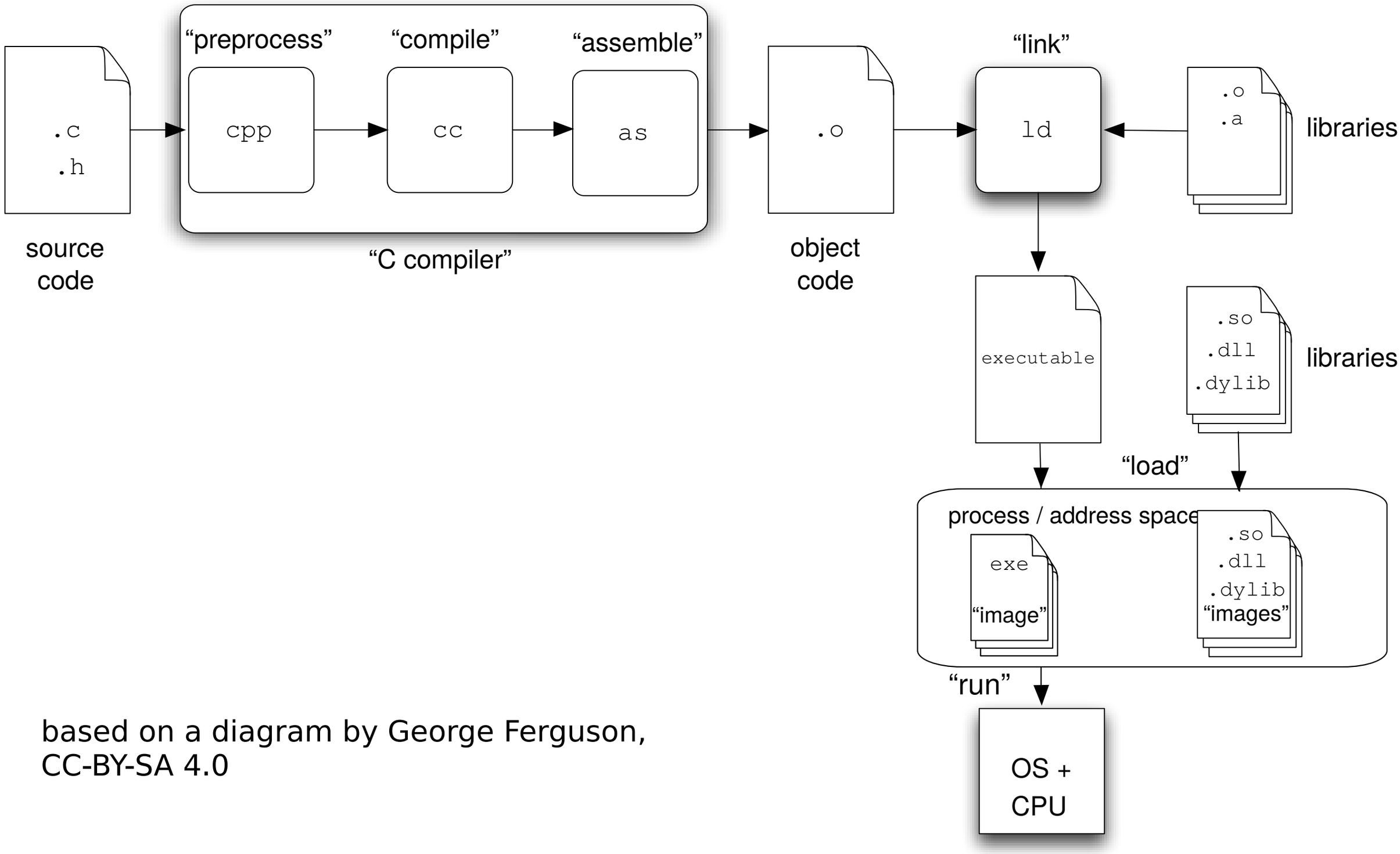
- an abundance of caution
  - testing, defensive programming, ...
  - compiler warnings
- dynamic analysis tools, such as
  - Memcheck (part of Valgrind)
  - AddressSanitizer
  - DrMemory
- (sometimes) static analysis tools
  - lint, Coverity, ...

Comparing the toolchain



based on a diagram by George Ferguson,
CC-BY-SA 4.0

Comparing the toolchain



based on a diagram by George Ferguson,
CC-BY-SA 4.0

A quick demo of compiling and running our tree code

The real reason C is different

In Java, *memory* is private to the Java program

- a heap of ‘Java objects’
  - ... ‘managed’ by the Java Virtual Machine
  - no explicit link to the host machine
    - Operating system? hardware? other-language code?
      These do not appear as Java objects

In C, *memory* is a shared space: the *address space*

The real reason C is different

In C, *memory* is a shared space: the *address space*

- may contain other stuff besides your C program!
  - assembly code, other-language code, stuff from the OS, memory-mapped files/devices, ...
  - *you can get a pointer to state your program didn't create*
  - systems programming is communicative programming

The real reason C is different

It’s not just about sharing memory...

C compilers inhabit a *shared toolchain*

- compiler, *assembler, linker, loader*

Lots of interplay between these, e.g.

- inline assembly
- linking C, assembly and other-language code together
- assembler directives as source annotations
- link-time wrapping or overriding of definitions
- ...

So C is “closer to the machine”, right?

In practice, yes. But...

It’s just a custom!

We *could* implement “the C language” in the self-contained, hermetic style of Java...

... it just wouldn’t be useful in the ways people are accustomed to using C.

... e.g. to talk to the OS, to devices

... e.g. to glue other bits of code together

(Conversely: could we implement Java less hermetically? Yes, although...
... both performance and garbage collection become tricky.)

A common misconception

”If you can get a pointer on it, you can access it”

- **No!** It’s <u>not</u> the case that “anything goes” in C.
  - There are detailed rules very similar to Java
  - e.g. array access: you must stay within array bounds
  - it’s incidental that compilers needn’t catch errors...
  - ... but they also needn’t give usable behaviour!
  - “anything could happen” -- so **it’s a bug** if it does
  - cf. in Java you can rely on the execption & catch it
    - C compiler optimisations may assume in-boundsness
    - if not now, then the next version of the compiler....
- We’ll talk more about arrays and pointers later

A word about safety

C does not try to protect you from yourself

- bad array index? no IndexOutOfBoundsException
- bad cast? no ClassCastException
- ...
- anything could happen!

Languages that guarantee the absence of "anything could happen" scenarios are called *safe*.

- Java is (mostly) a safe language; C is not

- it is possible to make a safe *implementation* of C

- ... but slower, so generally not done (ask me)

A word about “type-*X*”

PL nerds like to talk about “type safety”. And “strongly typed”, “type soundness”, etc. etc.

These terms all mean different things in different contexts. It’s usually a bad idea to use them.

E.g. “strongly typed” might mean a language/system that:

- offers an expressive language of data types, or
- doesn’t permit many implicit conversions, or
- is *safe* (a dynamic property), or
- is *sound*, i.e. it has compile-time checking that guarantees absence of certain errors (a static property)

Java is all of these (er... sort of).
Different languages satisfy different subsets...
... to varying extents! Try to say something precise.

References and next lecture

George Ferguson's "C for Java Programmers"
at University of Rochester
https://www.cs.rochester.edu/u/ferguson/csc/c/tutorial/

- Mostly very good; a bit wacky about some of the background, but I've covered that above
- Has exercises! Well worth attempting

Kernighan & Ritchie. "The C Programming Language." Addison-Wesley

- probably still the best book about C

Next lecture (2 of 2!): pointers, arrays; preprocessing; more data structures; other things...