Lecture 2
From Java to C

COMP26120

Giles Reger

Have you picked up the handouts at the back? Do you have a bit of paper and pen?

October 2018
Aim and Learning Outcomes

The aim of this lecture is to:

Highlight the key differences between Java and C and challenge your mental model of how programs run to allow you to ask the right questions when learning C

Learning Outcomes

By the end of this lecture you will be able to:

1. Recall major differences between Java and C
2. Explain how C programs are compiled and run
3. Sketch the big picture of what happens in the computer (e.g. in memory) when running a program
4. Write a C program performing simple input/output
Learning a new language

We assume that you are competent Java programmers.

We introduce C as a second language. We don’t cover all of C.

How do you learn a new programming language?
Learning a new language

We assume that you are competent Java programmers.

We introduce C as a second language. We don’t cover all of C.

How do you learn a new programming language?

- What’s the same? Usually if, then, else, while, for etc
- What’s the paradigm?
- What’s the tool ecosystem?
- Where do I go to find more? Standard reference/libraries?
- Try writing some code!
Spot the Difference

You should have picked up two bits of source code:

```
SalaryAnalysis.java and SalaryAnalysis.c
```

In pairs or threes:

1. mark the differences between the two
2. write a list of the concepts that these differences relate to
You should have picked up two bits of source code:

```
SalaryAnalysis.java   and   SalaryAnalysis.c
```

In pairs or threes:

1. mark the differences between the two
2. write a list of the concepts that these differences relate to

What did we find?
In this lecture I briefly cover:

- Comparative History
- From Source to Execution
- Dealing with Memory (the big one)
- Input/Output
- Coping without Classes
- Some Gotchas

I am not teaching you how to program in C. I am pointing out the things you should be aware of when learning C after learning Java.
Comparative History

History of C
- 1970s: BCLP to B to C
- 1983: C++ emerges
- 1989: ANSI/ISO Standard (C89)
- 1999: ISO Standard (C99)
- 2011: ISO Standard (C11) makes lots of changes
- 2018: ISO Standard (C18) makes very few changes

History of Java
- 1991: Project started
- 1996: Sun released Java 1.0
- 1997: Sun gave up on standardising the language
- 2004: Java 5 added generics
- 2006/7: Java went open-source
- 2014: Java 8 added lambdas
- 2017: Java 9 added G1
This is one of the first stumbling blocks when going from Java to C

In Java things are warm and fluffy, whereas C is a bit spiky. In Java you just need to run javac then java and it ‘works’ and (importantly) if something goes wrong you (usually) get a nice error message and a stack trace.

...but in C there’s these headers and link errors and SEGFAULTS!

So what’s the difference?
Name the Thing
Name the Thing

Diagram:

1. **Write**
   - **C Code**
   - **Compiler + Linker**
   - **Machine Code**

2. **Run**
   - **Python Code**
   - **Interpreter**
   - **Machine Code**

3. **Java Code**
   - **Compiler**
   - **Byte Code**
   - **Virtual Machine**
   - **Machine Code**
What does a Compiler (and Linker) do?

At a (very) high level

(Preprocessing)

Parsing to produce an AST and then intermediate code

Optimisation

Machine code generation

Putting together (linking) the bits of machine code (including libraries)

Try disassembling your executables and relate it back to your code
Also see https://godbolt.org/ for exploring compilers
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
**name.h**

```c
// File containing my name
#define NAME "Giles"
```

**hello.h**

```c
// Declare hello functions
void sayHello(char*);
```

**command line**

```bash
gcc hello.c -o hello
```

**hello.c**

```c
#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
```
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
C Source: Anatomy of a C program

name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
name.h

// File containing my name
#define NAME 'Giles'

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char*);

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
C Source: Anatomy of a C program

```c
name.h

// File containing my name
#define NAME "Giles"

hello.h

// Declare hello functions
void sayHello(char*);

command line

gcc hello.c -o hello

hello.c

#include <stdio.h>
#include "name.h"
#include "hello.h"

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
```
// File containing my name
#define NAME "Giles"

// Declare hello functions
void sayHello(char*);

// Prints my name
int main()
{
    sayHello(NAME);
    return 0;
}

void sayHello(char* name)
{
    printf("Hello %s!\n", name);
}
A bit more on the preprocessor

Show output of preprocessor

```
gcc -E hello.c
```

Compile-time macro definitions

```
gcc -DWorld hello.c -o hello
```

```c
#include <stdio.h>
#include "hello.h"

// Prints my name
int main()
{
    #ifdef WORLD
        sayHello("World");
    #elif defined(NAME)
        sayHello(NAME);
    #else
        sayHello("Nobody");
    #endif
    return 0;
}
...```
Optional but very useful Exercise

Edit the previous files to say some more interesting things. Use your programming knowledge from Java to see, for example, how iteration, conditions, and recursion work in C.

Process for submitting and marking (do this with Ex0 to see how it works)

- Complete task in the correct directory
- Run submit on school machine
- Go to https://marking.cs.manchester.ac.uk/
- Log in with your University details
- Click on exercise to see feedback and request marking
Dealing with Memory

C has explicit memory management

You will explore this a lot more in the next few lectures but I try and lay the groundwork for this here

The labs will help you explore these ideas. It is important you understand them. Use Valgrind

Exercise: A Cross-Section of Running a Program

Try drawing a cross-section (e.g. across multiple physical/conceptual layers) of what happens when running a program.
Low-level memory

Remember: Indirect Addressing from COMP15111

In COMP15111 you met the ADR and LDR ARM instructions for indirect addressing. The register storing an address can be seen as a pointer to that address. You also saw address arithmetic e.g. calculating new addresses from old ones.

Remember: Data Representation from COMP15111

In COMP15111 you discussed different concepts about how data is represented in memory e.g. endianness and alignment. Recall that data types tell us how much memory we need for different data items.

Both these topics are relevant for C programming.
class Thing{
    Thing otherThing;
    public static void main(String[] args){ makeThings(5); }
    public static void makeThings(int number){
        Thing thing = new Thing();
        Thing lastThing = thing;
        while(number-- > 0){ lastThing.otherThing = new Thing(); }
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}

What happens under the hood when we

- Call makeThings
- Call new Thing()
- Evaluate lastThing.otherThing
- Call System.out.println(thing)
- Return from makeThings
class Thing{
    Thing otherThing;
    public static void main(String[] args){ makeThings(5); }
    public static void makeThings(int number){
        Thing thing = new Thing();
        Thing lastThing = thing;
        while(number-- > 0){ lastThing.otherThing = new Thing(); }
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}

What happens under the hood when we

- Call makeThings  Stack frames and local variables
- Call new Thing()
- Evaluate lastThing.otherThing
- Call System.out.println(thing)
- Return from makeThings
class Thing{
    Thing otherThing;
    public static void main(String[] args){ makeThings(5); }
    public static void makeThings(int number){
        Thing thing = new Thing();
        Thing lastThing = thing;
        while(number-- > 0){ lastThing.otherThing = new Thing(); }
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}

What happens under the hood when we

- Call makeThings  Stack frames and local variables
- Call new Thing()  Allocate on heap, store address, object header
- Evaluate lastThing.otherThing
- Call System.out.println(thing)
- Return from makeThings
```java
class Thing {
    Thing otherThing;

    public static void main(String[] args) {
        makeThings(5);
    }

    public static void makeThings(int number) {
        Thing thing = new Thing();
        Thing lastThing = thing;
        while (number-- > 0) {
            lastThing.otherThing = new Thing();
        }
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}
```

What happens *under the hood* when we

- Call `makeThings`  
  Stack frames and local variables

- Call `new Thing()`  
  Allocate on heap, store address, object header

- Evaluate `lastThing.otherThing`  
  `putfield` takes `objectref`

- Call `System.out.println(thing)`

- Return from `makeThings`
class Thing{
    Thing otherThing;
    public static void main(String[] args){ makeThings(5); }
    public static void makeThings(int number){
        Thing thing = new Thing();
        Thing lastThing = thing;
        while(number-- > 0){ lastThing.otherThing = new Thing(); } 
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}

What happens under the hood when we

- Call makeThings  
  Stack frames and local variables
- Call new Thing()  
  Allocate on heap, store address, object header
- Evaluate lastThing.otherThing  
  putfield takes objectref
- Call System.out.println(thing)  
  call-by-value
- Return from makeThings
High-level memory… Java

class Thing{
    Thing otherThing;
    public static void main(String[] args){ makeThings(5); }
    public static void makeThings(int number){
        Thing thing = new Thing();
        Thing lastThing = thing;
        while(number-- > 0){ lastThing.otherThing = new Thing(); }
        lastThing.otherThing = thing;
        System.out.println(thing);
    }
}

What happens under the hood when we

- Call makeThings  Stack frames and local variables
- Call new Thing()  Allocate on heap, store address, object header
- Evaluate lastThing.otherThing  putfield takes objectref
- Call System.out.println(thing)  call-by-value
- Return from makeThings  reachability-based garbage collection
What can we do in C?

In C we can see memory and talk about it very explicitly

Refer to the addresses of things, store those in variables, and access them

```c
int a = 10; int b = 20;
int *ptr = &a;
*ptr = b;
```

Without any explanation... guess what happens?
Given our previous mental model, where does a live?

We can allocate bits of memory and use them

```c
int *thing = malloc(3 * sizeof(int));
// do stuff
free(thing)
```

Function pointers!
What can we do in C?

In C we can see memory and talk about it very explicitly

Refer to the addresses of things, store those in variables, and access them

```c
int a = 10; int b = 20;
int *ptr = &a;
*ptr = b;
```

Without any explanation... guess what happens?
Given our previous mental model, where does a live?

We can allocate bits of memory and use them

```c
int *thing = malloc(3 * sizeof(int));
// do stuff
free(thing)
```

Function pointers!

Giles Reger
What can we do in C?

In C we can see memory and talk about it very explicitly

Refer to the addresses of things, store those in variables, and access them

```c
int a = 10; int b = 20;
int *ptr = &a;
*ptr = b;
```

Without any explanation... guess what happens?

**Given our previous mental model, where does a live?**

We can allocate bits of memory and use them

```c
int *thing = malloc(3*sizeof(int));
// do stuff
free(thing)
```

**Function pointers!**
What can we do in C?

In C we can see memory and talk about it very explicitly

Refer to the addresses of things, store those in variables, and access them

```c
int a = 10; int b = 20;
int *ptr = &a;
*ptr = b;
```

Without any explanation... guess what happens?
Given our previous mental model, where does a live?

We can allocate bits of memory and use them

```c
int *thing = malloc(3*sizeof(int));
// do stuff
free(thing)
```

Function pointers!

Giles Reger
Lecture 2
October 2018 18 / 27
What can we do in C?

In C we can see memory and talk about it very explicitly

Refer to the addresses of things, store those in variables, and access them

```c
int a = 10; int b = 20;
int *ptr = &a;
*ptr = b;
```

Without any explanation... guess what happens?
Given our previous mental model, where does a live?

We can allocate bits of memory and use them

```c
int *thing = malloc(3*sizeof(int));
// do stuff
free(thing)
```

Function pointers!
What is a SEGFAULT?

What would in Java when doing something like this?

```java
int main(void)
{
    int a[10];
    for (int i = 0; i < 20; i++){
        a[i] = i;
    }
    return 0;
}
```

In C we get

giles$ ./seg
Segmentation fault: 11

You will see this kind of thing (a lot)

Google Segmentation fault and find out what it means
Important: There are no arrays or strings in C

Arrays are syntactic sugar for pointers e.g. we have a pointer to the start of the array and we can use pointer arithmetic to access elements

\[ a[i] \equiv * (a + i) \]

Creating an array gives a pointer to a continuous bit of memory

Strings are null-terminated arrays of characters – we need the null terminator to know when the string is finished.

```c
#include <stdio.h>
void main()
{
    char* string = "Hello World";
    printf("%d,%d\n", string[0], string[11]);
}
```
Many similar ideas to Java but

- See ‘no such thing as strings’
- More low-level functions for input/output
  - Character: putchar, getchar
  - Line: gets, puts
  - Formatted: printf, scanf - further reading needed
- Concept of streams (sequences of bytes of data) more apparent
  - Familiar predefined streams (stdin, stdout, stderr)
  - Some functions use these (e.g. getchar) others use an explicit stream
```c
#define LINE 20
#include <stdio.h>
int main() {
    long l; double d;
    puts("Enter an integer and a floating point number.");
    scanf("%ld %lf", &l, &d);

    puts("Type some text.");
    int ch; char line[LINE+1]; int len = 0;
    while ((ch = getchar()) != '\n'){
        line[len++] = ch;
        if (len == LINE){
            line[len] = 0; puts(line); len = 0;
        }
    }
    return 0;
}
```
A major difference between C and Java is the lack of classes.
Coping without Classes

A major difference between C and Java is the lack of classes.

What do classes give us?

- Encapsulation of data
- Encapsulation of functionality (co-located with data)
- Separation of concerns (e.g. data hiding)
- Object composition
- Subtype polymorphism
A major difference between C and Java is the lack of classes.

What do classes give us?

- Encapsulation of data
- Encapsulation of functionality (co-located with data)
- Separation of concerns (e.g. data hiding)
- Object composition
- Subtype polymorphism

What does C have instead? structs
Coping without Classes

A major difference between C and Java is the lack of classes.

What do classes give us?

- Encapsulation of data
- Encapsulation of functionality (co-located with data)
- Separation of concerns (e.g. data hiding)
- Object composition
- Subtype polymorphism

What does C have instead? structs
Structured data can be defined using the `struct` keyword. For example:

```c
struct A { char x; char y; int z; } sA;

typedef struct { char x; int z; char y; } B;
```

This defines a structure `A` with fields `x`, `y`, and `z`, and a typedef for another structure `B` with the same fields.

The `main` function then initializes these structures and prints out some values:

```c
int main() {
    sA.x = 'a';
    struct A sC = sA;
    struct { int a, b; } sD = {1,2};

    B *sE = malloc(sizeof(B));
    printf("%c %d %d \n", sC.x, sD.b, sE->z);

    printf("%d %d \n", sizeof(sA), sizeof(B));
}
```

Key features of structs include:

- Continuous memory (modulo packing)
- Access variables using `.`, if local and `->`, if pointer
- Can tag or name to reuse same structure again
- No inheritance, no local functions
- See `bit fields` for memory hacking
There are a few areas where C is different from Java. If in doubt, look it up. Here are some obvious ones (for memory things see later lectures):

- No boolean type, use int
- Implicit type conversions
- Difference between * and &
- Pass by value, need to pass by reference explicitly
- x++ vs ++x
- No automatic garbage collection
- No bounds checking (ArrayIndexOutOfBoundsException) of arrays
Further reading:

- Online Standard C book (Plauger and Brodie)
- The C Programming Language (Kernighan and Ritchie)
- C: A Reference Manual (Harbison and Steele)
- Expert C Programming (van der Linden)