Systems Programming

Lecture 1 | 24th of September 2018

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk

Systems Programming (H)

- Lecturer:
  - **Michel Steuwer** | http://michel.steuwer.info
  - Room **M111** in the School of Computing Science
  - Email: michel.steuwer@glasgow.ac.uk

- Schedule:
  - Two-hour lecture each week: **Monday 10 - 12**, Adam Smith Building 1115
  - Labs: **Monday 14 - 17**, Boyd Orr 720

Learning outcomes - Systems Programming (H)

1. Demonstrate competence in **low-level systems programming**
2. **Design, implement and explain the behaviour** of low-level programs written in a systems language
3. Explain the concepts of **stack and heap allocation**, **pointers**, and **memory ownership**, including demonstrating an understanding of issues such as **memory leaks**, **buffer overflows**, **use-after-free**, and **race conditions**
4. Explain how **data structures are represented**, and how this interacts with caching and virtual memory, and to be able to demonstrate the performance impact of such issues
5. Discuss and reason about **concurrency**, **race conditions**, and the **system memory model**
6. **Build** well-structured **concurrent systems programs** of moderate size, using libraries and static analysis tools appropriately.

- In this course we will use `C` and `C++` as they are the two most important systems programming languages

Course Overview and (preliminary) Schedule

Date	Lecture
24/09	Introduction; **No Labs**
01/10	Fundamentals of C and the importance of Types
08/10	Memory and Pointers
15/10	Resource Management and Ownership
22/10	Debugging and Development Tools
29/10	Programming week (**no lecture, but labs**): time to work on your coursework
02/11	**Deadline Coursework 1**
05/11	Introduction to threads and concurrency
12/11	PThreads, thread-safe abstract data type
19/11	Advanced threading
26/11	Guest Lecture
30/11	**Deadline Coursework 2**

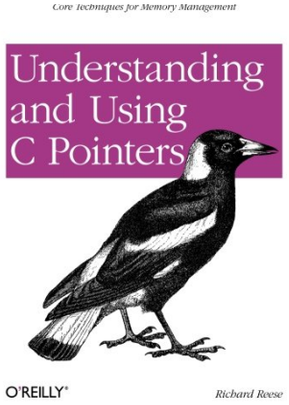
- **Revision Lecture in Semester 2**; **Exam in April/May**

Coursework and Labs

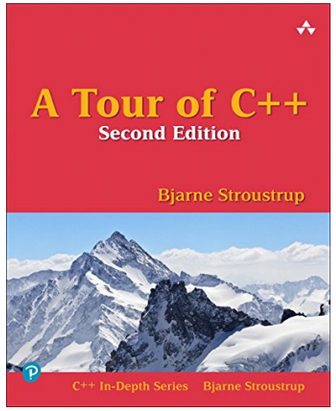
- There will be two pieces of coursework each worth 10% of your mark
- First coursework will be released next week with a deadline on the 2nd November
- Second coursework will be released at the end of October with a deadline on the 30th November
- The exam in April / May is 80% of your mark
- Lab times:
  - Student surnames starting with **A-G: 14:00 - 15:00**
  - Student surnames starting with **H-Mi: 15:00 - 16:00**
  - Student surnames starting with **Mo-Z: 16:00 - 17:00**
- There is a Lab exercise each week for improving your C programming skills
- You can also work on your coursework during the Lab and ask questions

Reading material

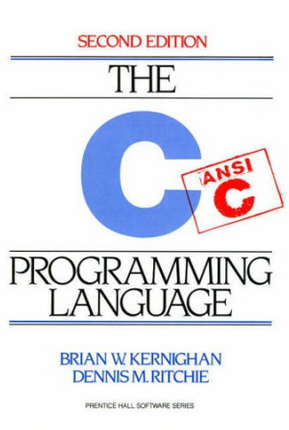
- No book required for this course (programming by yourself is *the key* to understanding)
- Two introductory recommendations (left) and the two definet treatments of C and C++ (right):



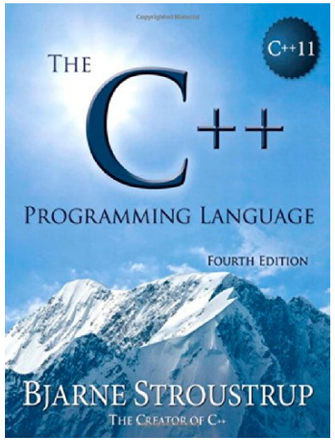
Focus on most challenging part of C: memory management



Good overview of the most important features of C++
Also available for free at isocpp.org/tour



Reference by creators of C
Still good reference today



In-depth (1376 pages) treatment of C++ by the creator of the language

- For basic and syntactic questions use the free C Programming Wikibook and cppreference.com

Course methodology

- This course is new, replacing the prior *Advanced Programming* course
- I do not only want to change the content, but also *how* the course is delivered
- I find teaching programming without actively programming silly!
- Emphasis on ***live coding*** in the lectures instead of reading slides

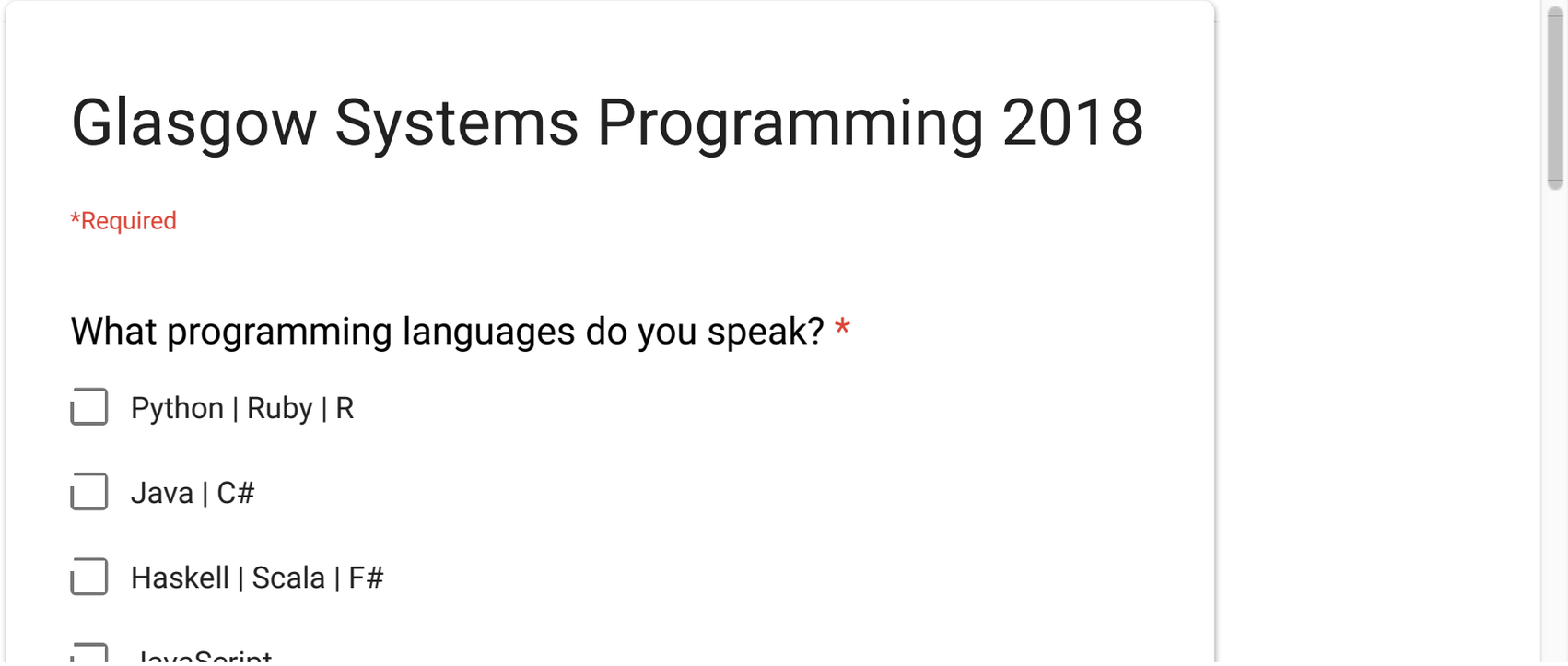
- This is much better with student interactions, so **ask questions!**
  We will find the answer together by trying it out and write programs

Your Feedback and Questions

- Your feedback is important! It will change the material of the course and the way it is delivered
- Use  Padlet to post questions during the lecture
- Use the forum on the course moodle page to ask questions after lectures or regarding the coursework
- Email me *only* with questions regarding personal matters

What languages do you speak? Experience and Expectations

- Please answer at: http://bit.do/GlasgowSP_Languages



Systems Programming and *System Software*?

- *Systems programming* is the activity of writing computer *system software*
- In contrast we call other software *application software*
- Examples of system software:
  operating system, web browser, video games, scientific computing applications and libraries, ...
- System software has (often) particular performance constrains such as:
  fast execution time, low memory consumption, low energy usage, ...
- To achieve these performance constrains systems programming languages allow for a more fine grained **control** over the execution of programs

Teaching of fundamentals not a Language

- You will learn C and a bit of C++ as part of this course, but this is not the main goal
- I want to you to deepen your understanding of fundamental *principles* and *techniques* in *computer systems*
  - **Memory** and **Computation** as fundamental resources of computing
  - **Representation of data structures** in memory and the role of **data types**
  - **Techniques for management** of computational resources
  - Reasoning about **concurrent** systems

Understanding these principles and techniques well will make you a better computer scientist
no matter what area your are interested in

History of Systems Programming Languages - 1

1950s:

Software wasn't distinguished between system and application

A single application always used the entire machine

Early programming languages were invented such as
Fortran, LISP and COBOL

Grace Hopper wrote one of the first *compilers* to automatically turn the human readable program into machine code

Grace Hopper at the UNIVAC I

History of Systems Programming Languages - 2

until the 1970s: System software is written in processor specific assembly languages

1970s: *Dennis Ritchie* and *Ken Thompson* wanted to port UNIX from the `PDP-7` to the `PDP-11`

They looked for a portable programming language and tried a language called `B` but then invented `C` as a portable *imperative* language supporting *structured* programming

PDP-7 minicomputer

Ken Thompson (sitting) and Dennis Ritchie (standing) at

History of Systems Programming Languages - 3

1980s: *Bjarne Stroustrup* aims to enrich `C` with new abstraction mechanisms and creates `C++`

A major influence is the first *object-oriented* programming language `Simula`

Bjarne Stroustrup, the creator of C++

2010s: New systems programming languages appear. `Rust` (2010) and `Swift` (2014) are sucessfull examples which include many *functional* programming language features

Programming Paradigms

- In *imperative* programming computations are sequences of statements that change the program's state

```
x = 41
x = x + 1
```

- *Structured* programming organises programs with subroutines and structured control flow constructes

```
sum = 0
for x in array:
    sum += x
```

- *Object-oriented* programming organises programs into objects containing data and encapsulate behaviour

```
animal = Dog()
animal.makeNoice()
```

- In *functional* programming programs are mathematical functions and avoid explicit change of state

Some questions about a simple Python program

```
x = 41
x = x + 1
```

- What value does x hold at the end of the program execution? (not a tricky question)

Some questions about a simple Python program

```
x = 41
x = x + 1
```

- What value does x hold at the end of the program execution? (not a tricky question)
  - Answer: 42
- How much memory does Python take to store x? (much more tricky question)

Some questions about a simple Python program

```
x = 41
x = x + 1
```

- What value does `x` hold at the end of the program execution? (not a tricky question)
  - Answer: `42`
- How much memory does Python take to store `x`? (much more tricky question)
  - Answer: In depends on the Pythton implementation.
    `sys.getsizeof(x)` gives the answer. On my machine: 28 bytes (= 28 * 8 = 224 bits)
    We can see the defining C code here
- How many instructions does Python execute to compute `x + 1`? (even more tricky question)

Some questions about a simple Python program

```
x = 41
x = x + 1
```

- What value does `x` hold at the end of the program execution? (not a tricky question)
  - Answer: `42`
- How much memory does Python take to store `x`? (much more tricky question)
  - Answer: In depends on the Pythton implementation.
    `sys.getsizeof(x)` gives the answer. On my machine: 28 bytes (= 28 * 8 = 224 bits)
    We can see the defining C code here
- How many instructions does Python execute to compute `x + 1`? (even more tricky question)
  - Answer: I don't know, but many more than just the addition ...
    Python is a dynamically typed language, so the data type of `x` could change at any time.
    Every operation tests the data types of the operants to check which instruction to execute.
    The C implementation for the add operation starts here

Some questions about a simple C program

```c
#include <stdio.h>
int main() {
    int x = 41;
    x = x + 1;
    printf("%d\n", x);
}
```

Online version of the code

Some questions about a simple C program

```c
#include <stdio.h>
int main() {
    int x = 41;
    x = x + 1;
    printf("%d\n", x);
}
```

Online version of the code

- What value does x has at the end of the program execution? (not a tricky question)
  - Answer: 42
- How much memory does C take to store x? (not that tricky any more)
  - Answer: `sizeof(int)` usually 4 bytes (= 4 * 8 = 32 bits)
- How many instructions does C take to compute `x + 1`? (not that tricky any more)
  - Answer: 1 add instruction and 2 memory (`mov`) instructions

This level of control allows strong reasoning about the programs performance and execution behaviour

C by example: Fibonacci

Fibonacci recursively:

```c
int fib(int x, int x1, int x2) {
    if (x == 0) {
        return x2;
    } else {
        return fib(x - 1, x1 + x2, x1);
    }
}

int main() {
    int fib_6 = fib(6, 0, 1);
}
```

Fibonacci iteratively:

```c
int fib(int x) {
  int x1 = 0;
  int x2 = 1;

  while (x > 1) {
    int xtmp = x1 + x2;
    x1 = x2;
    x2 = xtmp;
    x = x - 1;
  }
  return x2;
}

int main() {
    int fib_6 = fib(6);
}
```

Compiling a C program

- To execute a C program we must first *compile* it into machine executable code
- `Java, Haskell` (and many other languages) are also compiled languages
- The *compiler* translates the C source code in multiple steps into machine executable code:
  1. The *preprocessor* expands macros (e.g. `#include <stdio.h>` or `#define PI 3.14`)
  2. In the compiler stage, the source code is a) *parsed* and turned into an *intermediate representation*, b) machine-specific *assembly* code is generated, and, finally, c) *machine code* is generated in an *object file*
  3. The *linker* combines multiple object files into an executable
- In this course we are using the `clang` compiler.

  To compile and then execute a C program run:

```
clang source.c -o program
./program
```

1. Preprocessing

Input program as C source code

```c
#include <stdio.h>

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

Program after preprocessing

```c
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
// ...
int printf(const char * restrict, ...);
// ...

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

You can generate the code after the preprocessor stage with the `-E` flag:

```
clang source.c -E -o source.e
```

2a. Compiler intermediate representation

Program after preprocessing

```c
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
// ...
int printf(const char * restrict, ...);
// ...

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

You can generate the intermediate representation with the `-emit-llvm -S` flags:

```
clang source.c -emit-llvm -S -o source.llvm
```

Program in compiler intermediate representation

```
; ModuleID = 'source.c'
source_filename = "source.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

@.str = private unnamed_addr constant [5 x i8] c"%d \0A\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 41, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = add nsw i32 %2, 1
  store i32 %3, i32* %1, align 4
  %4 = load i32, i32* %1, align 4
  %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str, i32 0
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"Apple LLVM version 10.0.0 (clang-1000.11.45.2)"}
```

2b. Assembly code

Program in compiler intermediate representation

```
; ModuleID = 'source.c'
source_filename = "source.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

@.str = private unnamed_addr constant [5 x i8] c"%d \0A\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 41, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = add nsw i32 %2, 1
  store i32 %3, i32* %1, align 4
  %4 = load i32, i32* %1, align 4
  %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8]
  ret i32 0
}

...
```

You can generate the assembly code with the `-S` flag:

```
clang source.c -S -o source.s
```

Program in machine-specific assembly code (here for x86-64)

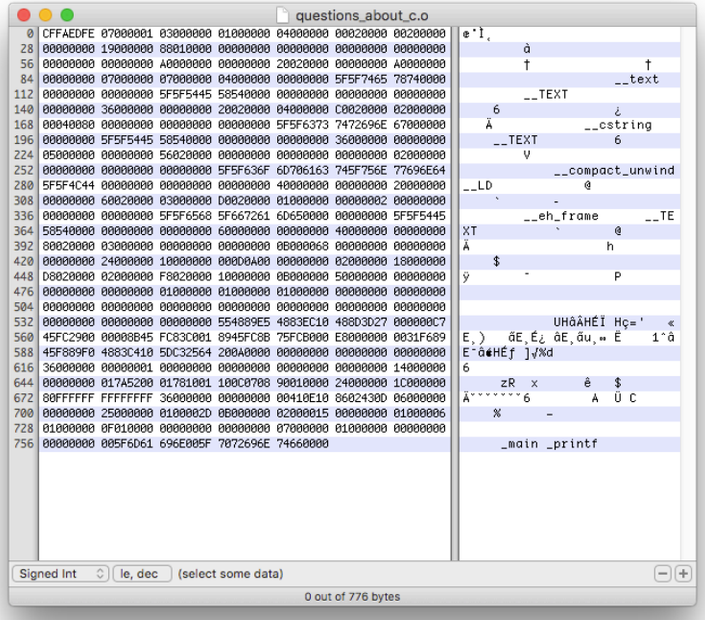
```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl    _main                   ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi
    movl    $41, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %esi
    movb    $0, %al
    callq    _printf
    xorl    %esi, %esi
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %esi, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc                    ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
```

2c. Machine code

Program in machine-specific assembly code (here for x86-64)

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl    _main                     ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi
    movl    $41, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %esi
    movb    $0, %al
    callq    _printf
    xorl    %esi, %esi
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %esi, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc                    ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
```

Program in machine (or *object*) code (here for x86-64)

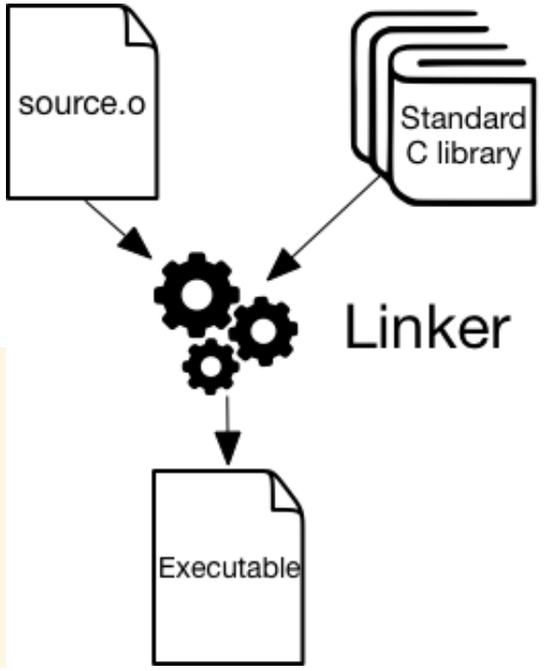


You can generate an object file with machine code using the −c flag:

Linking

- The linker combines one or more object files into a single executable
- The linker checks that all functions called in the program have machine code avaialble (e.g. `printf`'s machine code is in the C standard library)
- If the machine code for a function can not be found the linker complains:

```
Undefined symbols for architecture x86_64:
"_foo", referenced from:
  _main in source.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1
   (use -v to see invocation)
```



Next week: Basics of C

- Basics of C: variables, `structs`, control flow, functions
- What are `data types`, why are they important, and what do they tell us about memory?
- If you know `C` already well you might skip the lecture next week, but do come to the Lab!

Systems Programming

Lecture 1 | 24th of September 2018

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk