COMPSCI2030 Systems Programming

Basics of Systems Programming

Yehia Elkhatib

Personal Introduction

- Reader ~Associate Professor
- Member of The GLAsgow Systems Section (GLASS)
- Education
  - BSc Alexandria University
  - MSc Lancaster University
  - PhD Lancaster University
- Research: data-driven deployment in complex infrastructures
  - cloud and edge computing
  - system-of-systems composition
  - intent-driven networking

My Lecturing Approach

- I strongly recommend handwritten notes
  - makes knowledge last a bit more, even if you discard the notes
  - helps with auditory processing disorder and ADHD
- There are no stupid questions or answers
- Specifically for SP(GA) part 1
  - Material is top-heavy to get you started quickly on the coursework
  - I would rather go fast then slow down rather than vice versa
  - So feedback is very welcome
  - Question lottery

What is Systems Programming?

- Writing computer system software:
  used as a platform for other software, a layer of abstraction (scaffolding)
  - i.e. the main ‘customer’ is other software, not necessarily users
  - In contrast to application software
- Examples of system software: operating system, drivers, compilers, …
- Usually, system software has specific performance constraints such as:
  - fast execution time
  - low memory consumption
  - low energy usage
- To meet these constraints, systems programming languages allow for a more fine-grained control over the execution of programs

History of Systems Programming

- Until the 1970s
  - System software was generally written in processor-specific assembly languages
    - mostly based on ALGOL 60 concepts
  - very difficult to write and maintain
- 1972
  - Ritchie and Thompson wanted to port UNIX from the DEC PDP-7 to the PDP-11
  - They looked for a portable programming language (tried their B type-less language)
  - Invented C as an imperative language supporting structured programming

Ken Thompson (sitting) and Dennis Ritchie (standing) at a PDP-11 minicomputer

History of Systems Programming

- 1983
  - due to its popularity, C had numerous variants
  - ANSI defined standard versions: C85, C99, C11, …
- early 1980s
  - Bjarne Stroustrup aimed to enrich C with new abstraction mechanisms
  - Inspired by Simula, the first object-oriented language, he creates C++
    - Simula is a superset of ALGOL 60
- 2010s
  - New systems programming languages
  - e.g. Rust (2010) and Swift (2014)

Bjarne Stroustrup

A simple Python program

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

- Q: What value does x hold at the end of the program execution?
  - (not a tricky question)
  - A: 42
- Q: How much memory does Python take to store x?
  - (a tricky question)
  - A: It depends on the Python implementation.
    `sys.getsizeof(x)` gets the answer. On my machine: 24 bytes (* 8 = 192 bits)
- Q: How many instructions does Python execute to compute x+1?
  - (even more tricky)
  - A: I don't know, but many more than just the addition ...
    Python is dynamically typed, so the data type of x could change at any time.
    Every operation tests the data types of the operands to check which instruction to execute.

A lot of uncertainty and lack of control 😕

A simple C program

```c
int main() {
    int x = 41;
    x = x + 1;
}
```

- Q: What value does x have at the end of the program execution?
  - (not a tricky question)
  - A: 42
- Q: How much memory does C take to store x?
  - (not that tricky)
  - A: `sizeof(int)` usually 4 bytes (* 8 = 32 bits)
- Q: How many instructions does C take to compute x+1?
  - (not that tricky)
  - A: 1 add instruction and 2 memory (mov) instructions
  - Note: a useful tool for this is https://godbolt.org/

Certainty allows strong reasoning about the program's performance and execution behaviour

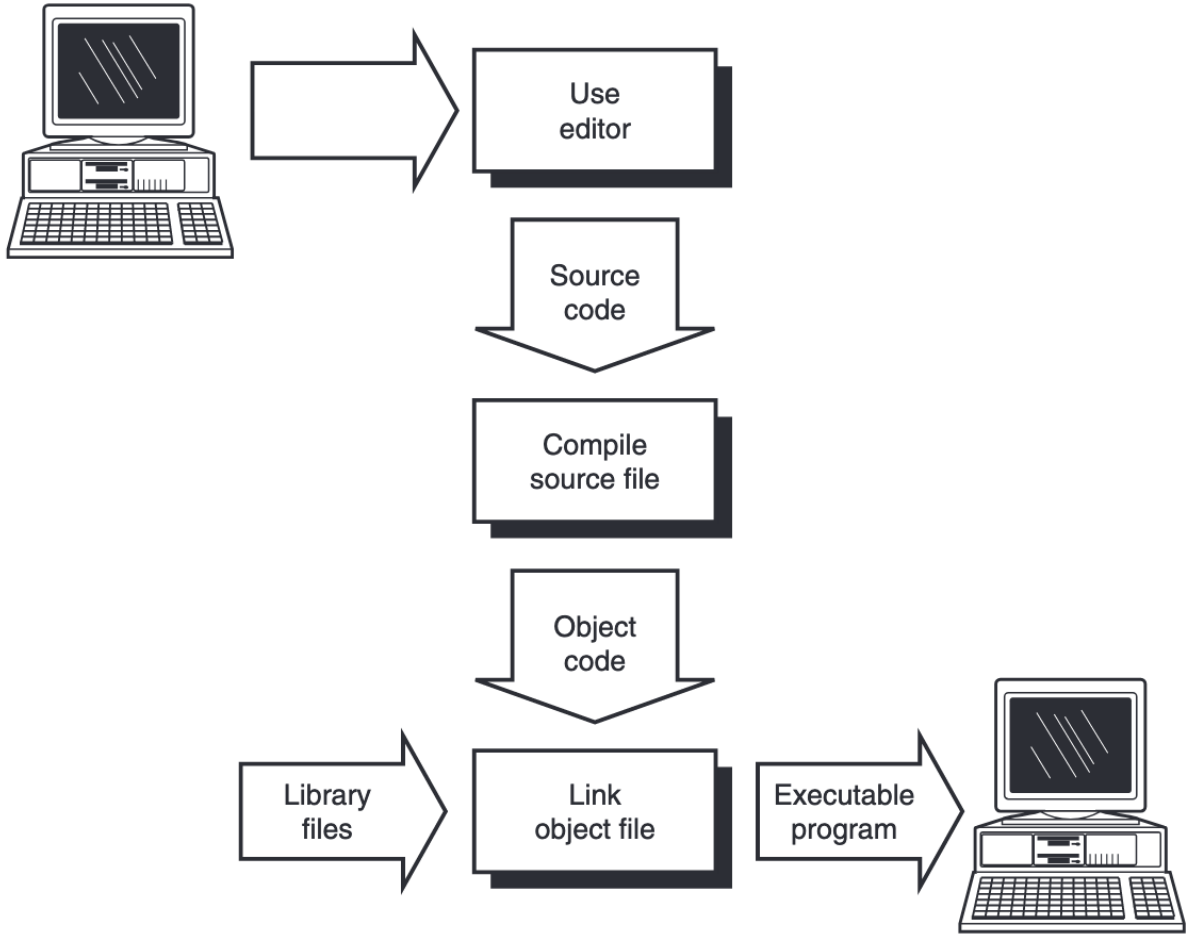
Efficiency and Reliability

Why C?

- Versatile
  - whole OS, compilers, networking, games, word processors, web apps, …
- Portable (thanks to ANSI/ISO standards)
  - can be compiled and run on different platforms with no modification
- Systems with constrained resources
- Small number of keywords (32)
- Modular
  - code reuse through functions and structures
- Builds understanding of resource management
  - important for becoming a better developer

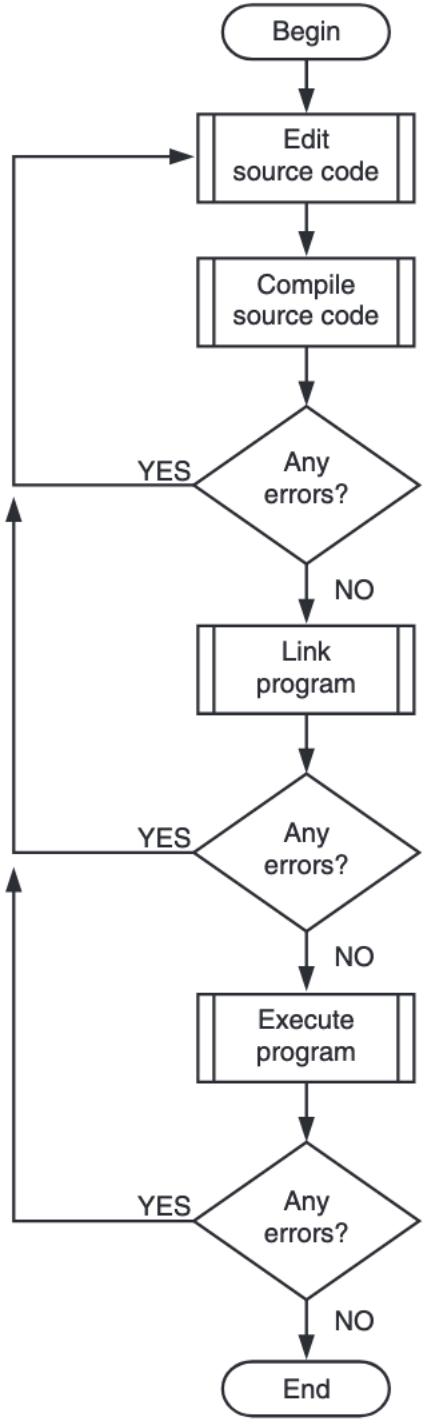
Development Cycle

- Applies to all compiled languages



Development Cycle

- A compiler error indicates that it is not possible to compile your program
  - You must change your program to get it compiling
- A compiler warning indicates an unusual condition that may (and quite often does) indicate a problem
  - Do not ignore warnings
  - You should change the program to: either fix the problem or clarify the code to avoid the warning



Compilers

- You understand your C code (I hope!)
- Your computer does not; it only understands machine language
- A compiler translates from *source code* to *object code*
- There are many C compilers, around 50 or so
  - Borland Turbo C (1987)
  - GNU Compiler Collection, GCC (1987)
  - Tiny C (2005)
  - Clang (2007)

Compiling

```
clang source.c -o executable

./executable
```

- To compile and then execute a C program:
- Translating source code into machine code is a multi-step process:
  1. The preprocessor expands macros
  2. In the compiling stage, the source code is
     a) parsed and turned into an intermediate representation
     b) machine-specific assembly code is generated
     c) machine code is generated in an object file
  3. The linker combines multiple object files into an executable
- We will peek into each of these with examples

```
#include <stdio.h>

#define PI 3.14
```

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
  ret i32 0
}

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

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-mat
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 i
  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)

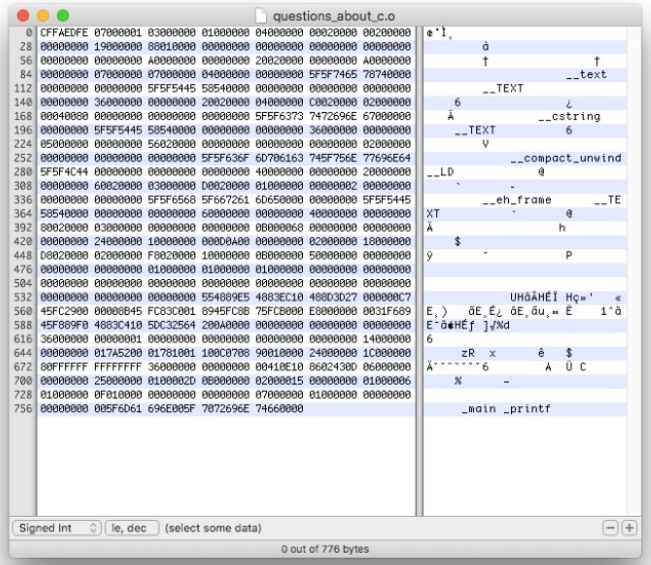
```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl    _main                   ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .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
    .asciz    "%d \n"
.subsections_via_symbols
```

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
    .asciz    "%d \n"
.subsections_via_symbols
```

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

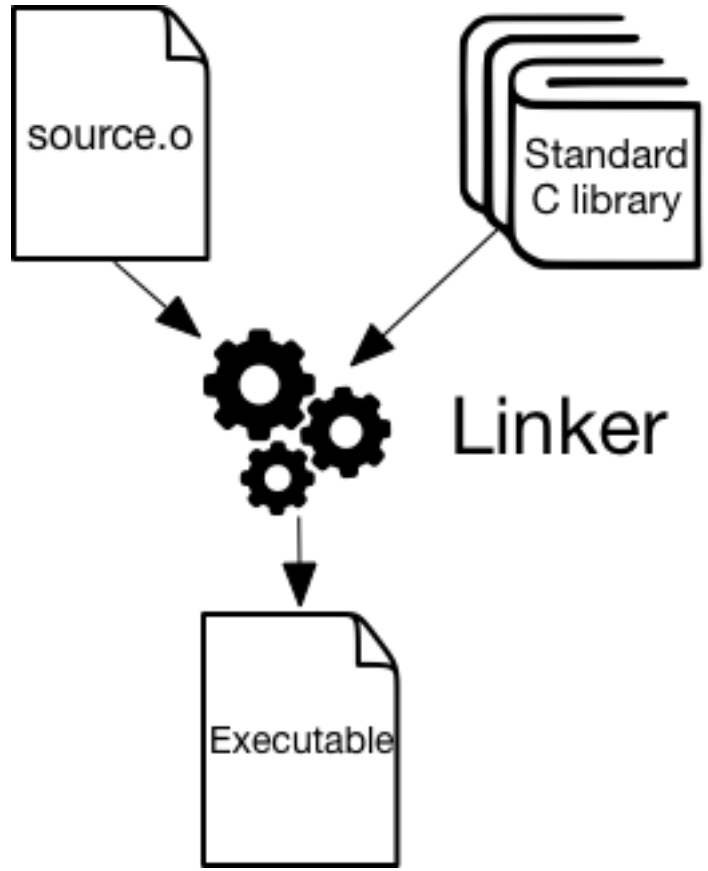


You can generate an object file with machine code using the `-c` flag:

```
clang source.c -c -o source.o
```

3. 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 available
  - e.g. `printf`'s machine code is in the C standard library
- If the machine code for a function cannot be found, the linker complains
  - e.g.

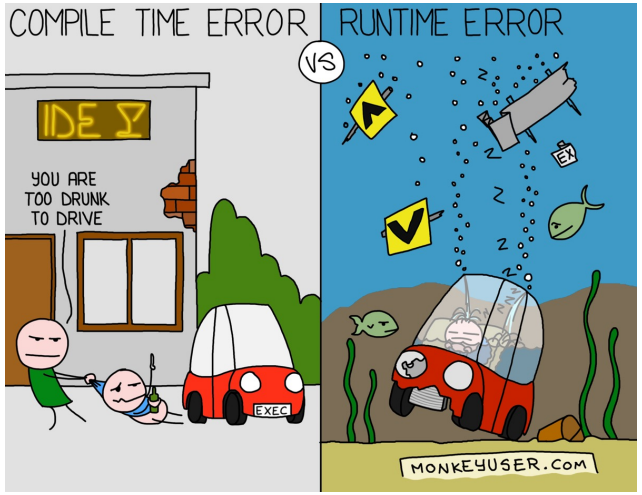


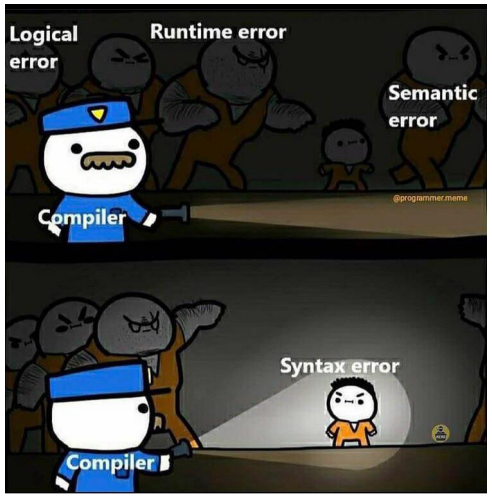

```c
int main() {
    foo();
}
```

```
linking-error.c:2:2: error: implicit declaration of function 'foo' is invalid in C99 [-Werror,-Wimplicit-function-declaration]
        foo();
        ^
1 error generated.
```

The compiler is your friend

- Errors and warnings indicate that there is something **wrong**
- As you mature as a developer, you will wish that your compiler throws more errors rather than get runtime ones!





Shell Detour

- Basic file exploring
  - `ls, pwd, mkdir, cd, cat, head, tail, cp, mv, rm, touch`
- Editing text files
- man

The `main()` Function

```
int main() {
```

- The only one required for *any* C program
- It denotes the entry point to the program
  - there can only be one and exactly one
- It returns an int to signify the exit code
  - `0` = normal execution and termination, i.e. at last statement in main
  - Non Zero Exit Code = abnormal termination
  - if no `return` statement, an implicit `return 0` is executed

Basic output with `printf`

- Defined in `stdio.h`, it allows the formatted printing of values
- The first argument is the *format string* (using special characters)
- The second argument onwards are the values to be printed
- The number and order of format strings and values have to match

Special Characters	Explanation	Argument Type
%c	Single character	`char`
%s	Character string	`String: (char *)`
%d	signed integer in decimal representation	`int`
%l or %ld or %li	Long	`long`
%f	floating-point number in decimal representation	`float double`

Full list at: https://en.cppreference.com/w/c/io/fprintf

Common questions at this point

- *But… I want to use an IDE*
  - Feel free to, but we will not support you as this is not an aim of the course
  - There are real benefits in developing code through command line tools
    - Understand the compile, link, execute process
    - Sometimes the only way to develop, e.g. for small devices or headless servers
  - Can cause other issues (e.g. huge cache files) that we cannot help with
- *How can I install clang on [platform]?*
  - Guide for common platforms in the labsheet
  - Follow instructions relevant to your platform at https://clang.llvm.org/get_started.html