```
404b30:       77 ae                   ja     404ae0 <.omp_outlined..8+0x1c0>
404b32:       c5 f9 2e 04 25 48 60    vucomisd xmm0,QWORD PTR ds:0x406048
404b39:       40 00
404b3b:       75 02                   jne    404b3f <.omp_outlined..8+0x21f>
404b3d:       7b a1                   jnp    404ae0 <.omp_outlined..8+0x1c0>
404b3f:       c5 fb 11 54 24 28       vmovsd QWORD PTR [rsp+0x28],xmm2
404b45:       c5 fb 11 44 24 20       vmovsd QWORD PTR [rsp+0x20],xmm0
404b4b:       e8 60 d5 ff ff          call   4020b0 <log@plt>
404b50:       c5 fb 59 05 e8 14 00    vmulsd xmm0,xmm0,QWORD PTR [rip+0x14e8]        # 406040 <_IO_stdin_used+0x40>
404b57:       00
404b58:       c5 fb 5e 44 24 20       vdivsd xmm0,xmm0,QWORD PTR [rsp+0x20]
404b5e:       c5 f9 2e 04 25 48 60    vucomisd xmm0,QWORD PTR ds:0x406048
404b65:       40 00
404b67:       72 06                   jb     404b6f <.omp_outlined..8+0x24f>
404b69:       c5 fb 51 c0             vsqrtsd xmm0,xmm0,xmm0
404b6d:       eb 05                   jmp    404b74 <.omp_outlined..8+0x254>
404b6f:       e8 ac d5 ff ff          call   402120 <sqrt@plt>
404b74:       c5 fb 59 4c 24 10       vmulsd xmm1,xmm0,QWORD PTR [rsp+0x10]
404b7a:       c4 c1 7b 11 4e 10       vmovsd QWORD PTR [r14+0x10],xmm1
404b80:       41 c6 46 18 01          mov    BYTE PTR [r14+0x18],0x1
404b85:       c5 fb 59 44 24 28       vmulsd xmm0,xmm0,QWORD PTR [rsp+0x28]
404b8b:       c4 c1 7b 59 46 08       vmulsd xmm0,xmm0,QWORD PTR [r14+0x8]
404b91:       c4 c1 7b 58 06          vaddsd xmm0,xmm0,QWORD PTR [r14]
404b96:       49 8b 07                mov    rax,QWORD PTR [r15]
404b99:       c5 fb 11 04 e8          vmovsd QWORD PTR [rax+rbp*8],xmm0
404b9e:       41 c6 46 18 00          mov    BYTE PTR [r14+0x18],0x0
404ba3:       c4 c1 7b 10 46 10       vmovsd xmm0,QWORD PTR [r14+0x10]
404ba9:       31 c0                   xor    eax,eax
404bab:       e9 3e fe ff ff          jmp    4049ee <.omp_outlined..8+0xce>
404bb0:       bf 68 81 40 00          mov    edi,0x408168
404bb5:       44 89 e6                mov    esi,r12d
404bb8:       e8 c3 d6 ff ff          call   402280 <__kmpc_for_static_fini@plt>
404bbd:       48 83 c4 38             add    rsp,0x38
404bc1:       5b                      pop    rbx
404bc2:       41 5c                   pop    r12
404bc4:       41 5d                   pop    r13
404bc6:       41 5e                   pop    r14
404bc8:       41 5f                   pop    r15
404bca:       5d                      pop    rbp
404bcb:       c3                      ret
404bcc:       48 89 c7                mov    rdi,rax
404bcf:       e8 0c 00 00 00          call   404be0 <__clang_call_terminate>
404bd4:       48 89 c7                mov    rdi,rax
404bd7:       e8 04 00 00 00          call   404be0 <__clang_call_terminate>
404bdc:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
```

x86_64 executable binaries in action

Introduction to executable binaries

Antonin Portelli

Supported by UK EPSRC grants EP/V001329/1, EP/V001191/1, EP/V001477/1, EP/V001248/1, EP/V001140/1

- Memory segmentation
- The stack
- Your best friend the debugger
- Let’s play with stack frames
- The Executable and Linkable Format (ELF)
- Hello world again
- Wrap up

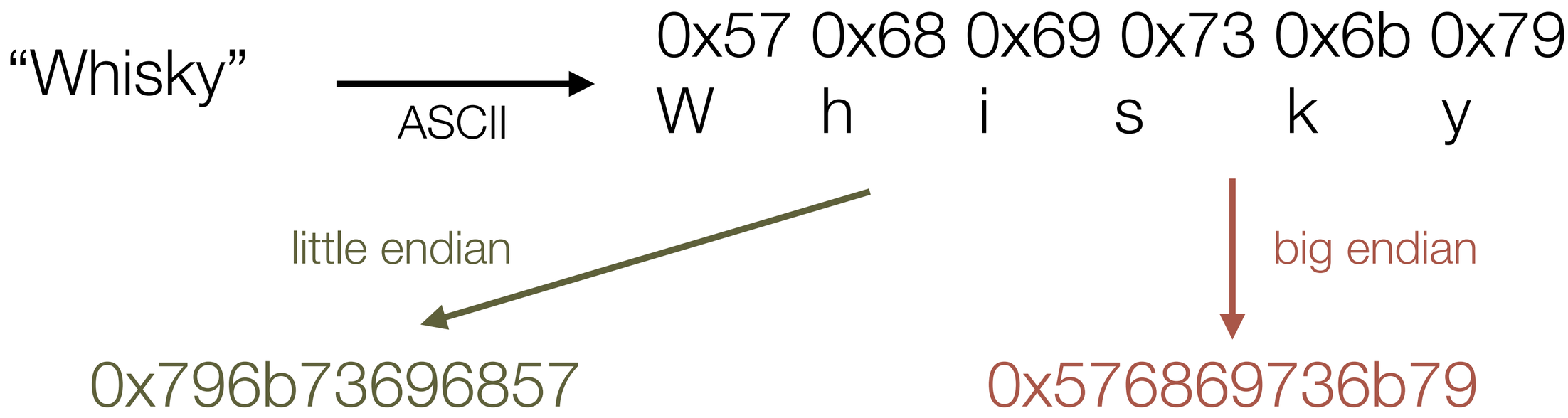
Memory segmentation

Endianness

- **Endianness** defines the order in which bytes are written in memory. Bytes not bits!

- Big endian: weakest bytes first “left to right” (e.g. PowerPC architecture)

- Little endian: strongest bytes first “right to left” (e.g. x86, x86_64 and ARM)



Memory segments

- A program memory is separated in mainly **5 segments**

  From low to high addresses

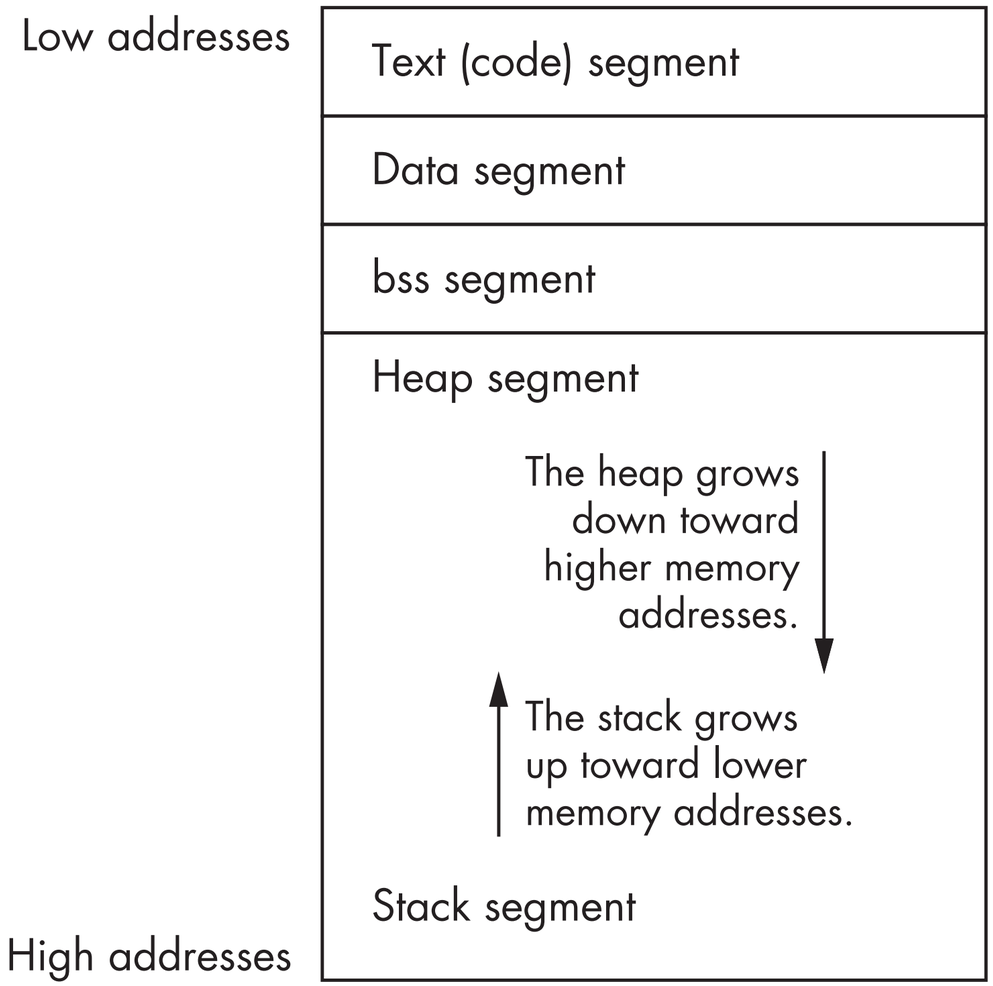
  1- Text segment: program code

  2- Data: initialised static variables

  3- BSS: uninitialised static variables

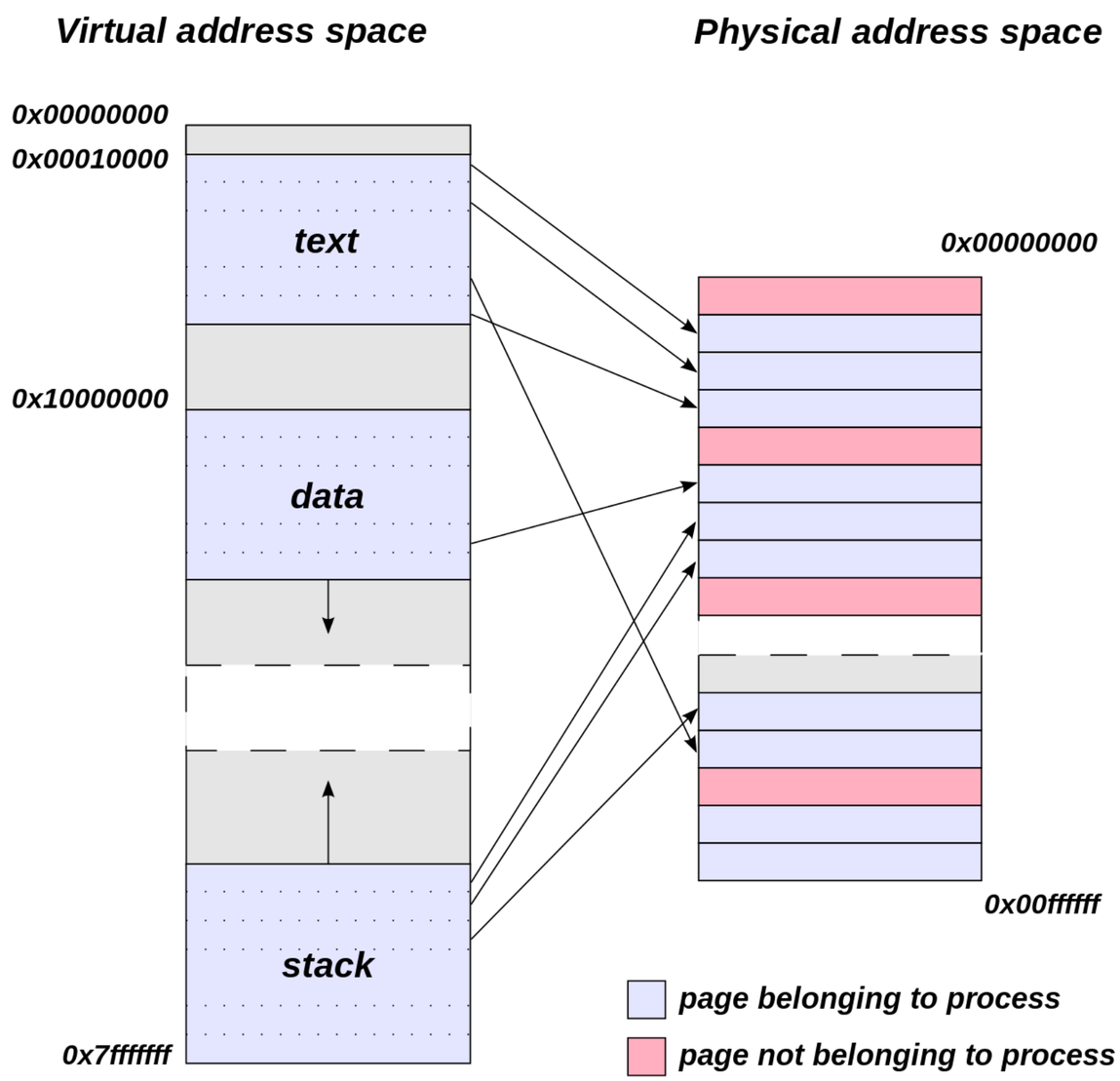
  4- Heap: global variable space

  5- Stack: local “scratch pad”



Virtual memory in modern OSs

- Program address space is **not physical**
- It **may not be contiguous** in physical memory
- **Fragmented** in pages (typically 4 kB)
- Mapping to physical memory (page tables) maintained by the **kernel**



Text (code) segment

- Contains the program code in binary form
- Each instruction is code using an **opcode** and binary representation of the operands
- x86 has **variable size** instructions!

```
$ objdump -d -M intel ./hello-asm

...
48 c7 c0 01 00 00 00    mov    rax,0×1
48 c7 c7 01 00 00 00    mov    rdi,0×1
48 8b 75 d0             mov    rsi,QWORD PTR [rbp-0×30]
48 c7 c2 0d 00 00 00    mov    rdx,0xd
0f 05                   syscall
...
```

- This segment also contains constants

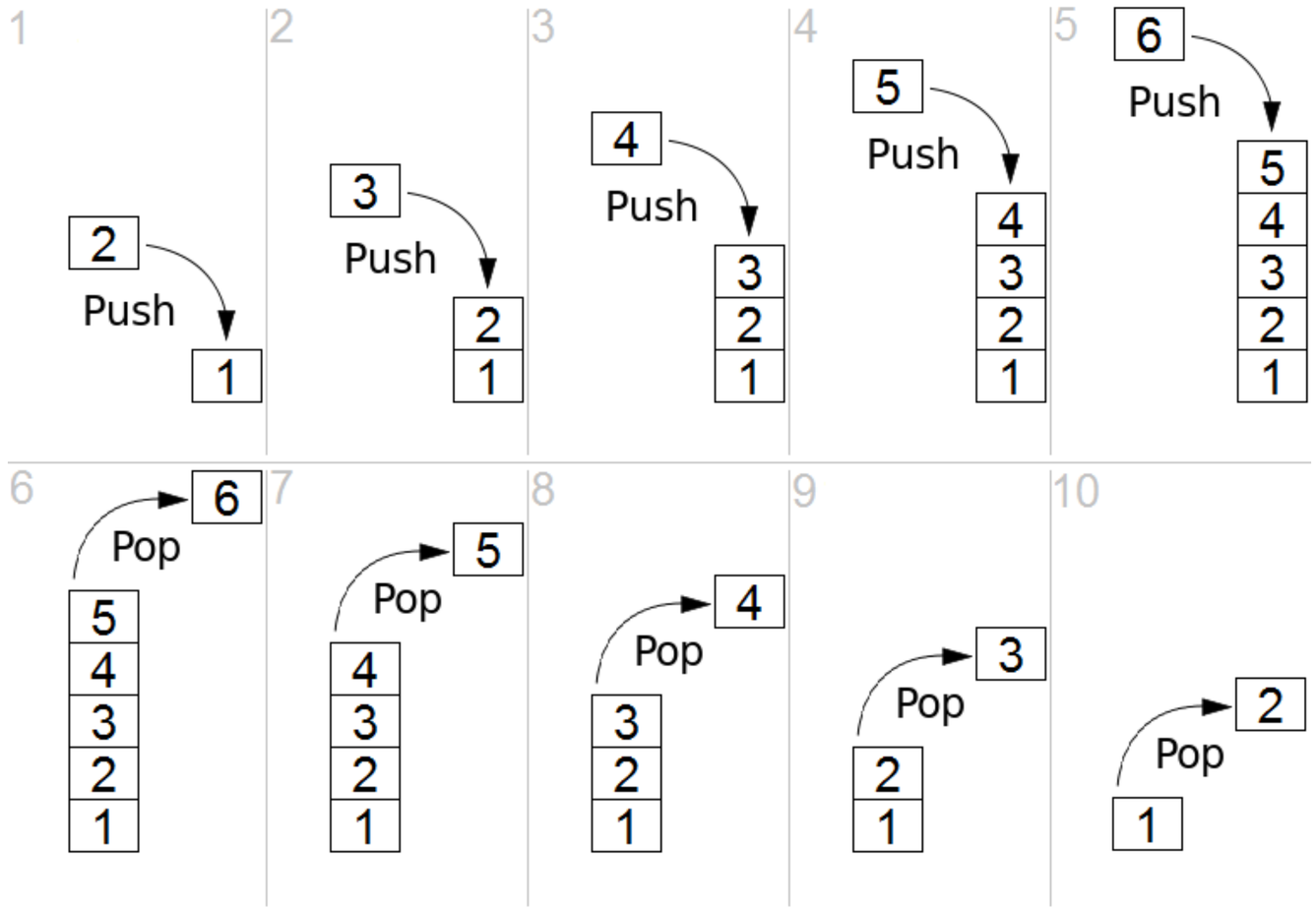
Heap & stack

- Heap: **extensible** segment meant to store persistent data
- The programs **needs to request** space on the heap, and needs to declare when the space isn't needed anymore
- In C one uses `malloc` & `free` to get heap space
  In C++ it is done with the `new` & `delete` operators
- Stack: **"scratch pad"** for the program
- Also extensible, but with a **much simpler structure**
- Allocations are generally managed by the compiler

The stack

Stack generalities

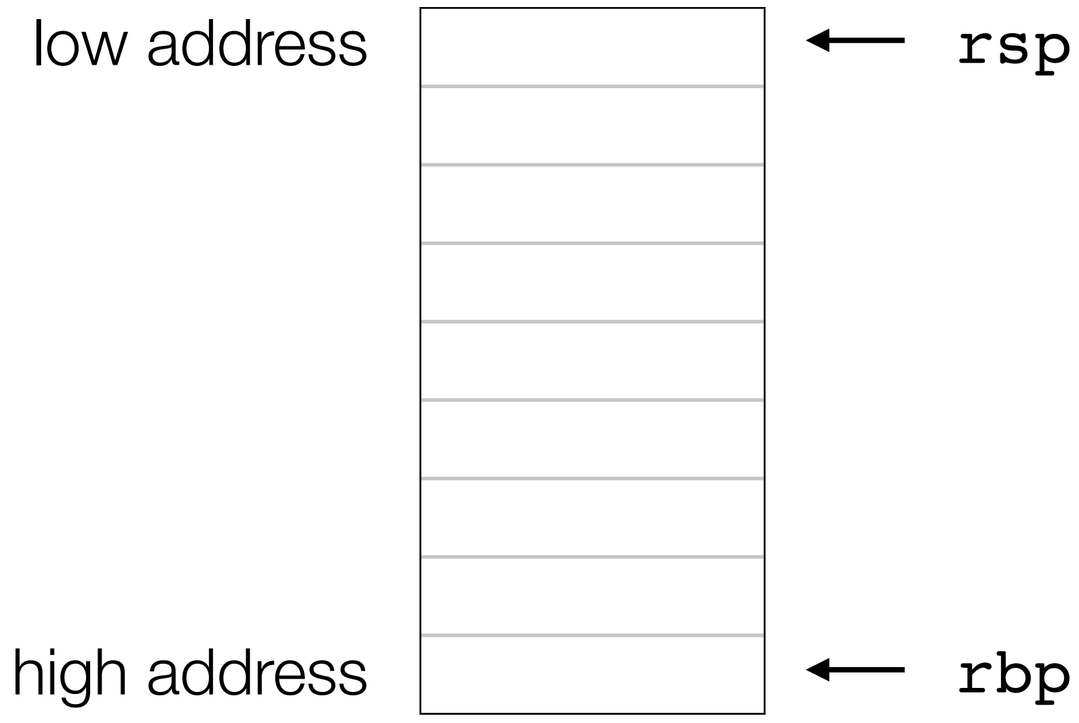
- The stack is a FILO container. **First In Last Out**
- Accessed through **push** & **pop** operations



Stack manipulation in ASM

- Two registers are used to track the stack

  `rbp`: base pointer (bottom of the stack address)
  `rsb`: stack pointer (top of the stack address)



`rbp` ≥ `rsp`

Stack manipulation in ASM

- **`push src`** : add `src` on the stack & update `rsp`

  equivalent to (if `src` is 8B)

```
sub rsp, 8
mov qword ptr [rsp], src
```

- **`pop dest`** : move top element to `dest` & update `rsp`

  equivalent to (if `dest` is 8B)

```
mov dest, qword ptr [rsp]
add rsp, 8
```

Function calls

- Functions can be called with **`call address`**
  Next instruction address (**`rip`**) is pushed on the stack
  Then execution jump to **`address`**

- Functions can return using **`ret`**
  Pop top stack element as an address and jumps to it

- Argument passing is defined though a convention called **Application Binary Interface (ABI)**

- For x86_64 on Linux, **System V AMD64 ABI**
  https://bit.ly/3kwiMxH

System V AMD64 ABI

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
`%rbx`	callee-saved register	Yes
`%rcx`	used to pass 4th integer argument to functions	No
`%rdx`	used to pass 3rd argument to functions; 2nd return register	No
`%rsp`	stack pointer	Yes
`%rbp`	callee-saved register; optionally used as frame pointer	Yes
`%rsi`	used to pass 2nd argument to functions	No
`%rdi`	used to pass 1st argument to functions	No
`%r8`	used to pass 5th argument to functions	No
`%r9`	used to pass 6th argument to functions	No
`%r10`	temporary register, used for passing a function's static chain pointer	No
`%r11`	temporary register	No
`%r12-r14`	callee-saved registers	Yes
`%r15`	callee-saved register; optionally used as GOT base pointer	Yes
`%xmm0–%xmm1`	used to pass and return floating point arguments	No
`%xmm2–%xmm7`	used to pass floating point arguments	No

More complex arguments (e.g. structs) passed through the stack

Stack frames

- **Local stack space during function call**

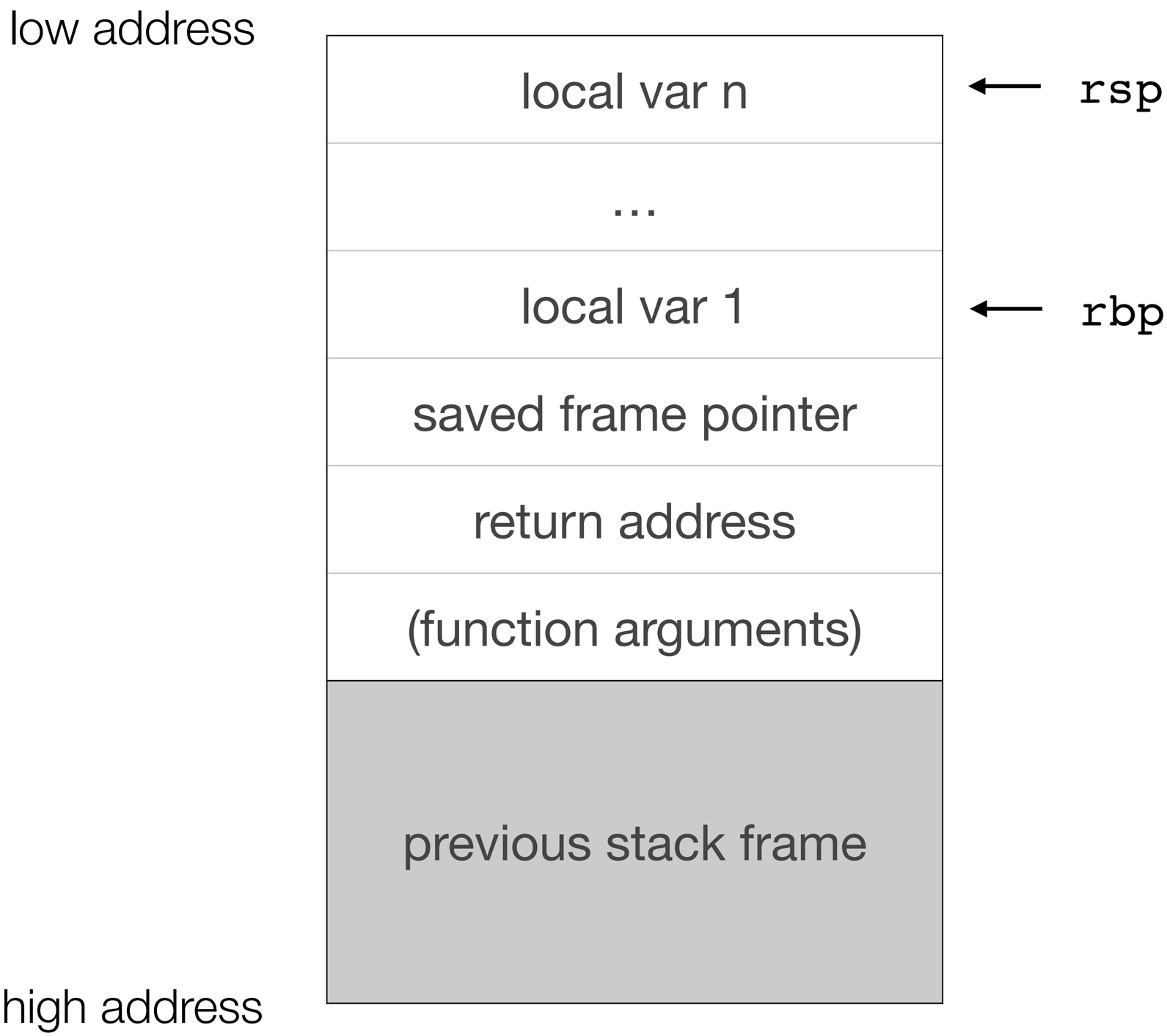
- When the function starts:

  1) the return address is pushed (`call` does that)
  2) the previous base pointer rbp is pushed for backup
  3) the base pointer is set to the current position rsp
  4) rsp is decreased (stack grows) the size of local variables
  5) space for local variables is between rbp and rsp

- This set of data is the **stack frame**

- If a function is called within the function, a new stack frame is created on top of the current one, etc…

Stack frame visualisation



Stack frame ASM implementation

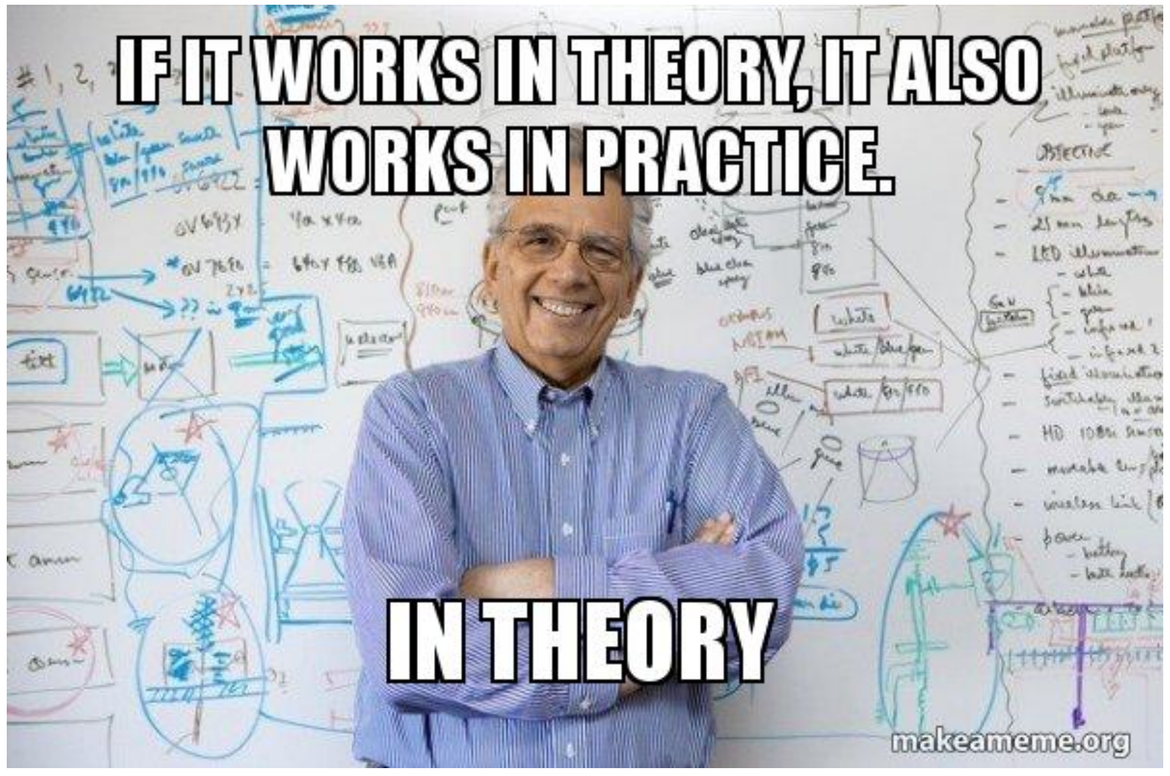
```c
void function(void)
{
    int a = 1, b = 2;

    printf("%d\n", a+b);

    return;
}
```

```asm
function: # @function
  ; save previous frame pointer
  push rbp
  mov rbp, rsp
  ; allocate 16B on the stack
  sub rsp, 16
  ; push a on the stack
  mov dword ptr [rbp - 4], 1
  ; push b on the stack
  mov dword ptr [rbp - 8], 2

  ; ... printf call ...

  ; remove 16B from the stack
  add rsp, 16
  ; restore saved base pointer
  pop rbp
  ; return (will pop return address)
  ret
```

https://godbolt.org/z/nPE7sK

Remember…

- **These are all conventions! You do what you want!**
- But this is what compilers are going to generate respecting the System V AMD64 ABI
- Nothing replaces practice for these things…



Your best friend the debugger

Generalities on debuggers

- Debuggers allow you to step line-by-line (or instruction-by-instruction) through a program execution

- They generally offer possibilities to disassemble the code and investigate memory and registers

- **They are invaluable tools to understand runtime issues when testing a program**

- Most common is probably GDB, here I will use **LLDB** from the LLVM project.

  LLDB vs. GDB syntax: https://bit.ly/3q7dme4

Starting LLDB

- Compiling C/C++ code with the `-g` option allow debuggers to be aware of the source code

```
$ clang -g hello.c -o hello
$ ./hello
Hello world
$ lldb hello
(lldb) target create "hello"
Current executable set to '/mnt/hgfs/binary-hack/lectures/1-x86_64/hello' (x86_64).
(lldb) l
   5    {
   6      printf("Hello world\n");
   7
   8      return EXIT_SUCCESS;
   9    }
   10
(lldb) r
Process 61856 launched: '/mnt/hgfs/binary-hack/lectures/1-x86_64/hello' (x86_64)
Hello world
Process 61856 exited with status = 0 (0×00000000)
```

Breakpoints

- Breakpoints are locations (code or binary) where the execution will be paused

```
(lldb) b 6
Breakpoint 1: where = hello`main + 15 at hello.c:6:3, address = 0×000000000040113f
(lldb) r
Process 61989 launched: '/mnt/hgfs/binary-hack/lectures/1-x86_64/hello' (x86_64)
Process 61989 stopped
* thread #1, name = 'hello', stop reason = breakpoint 1.1
    frame #0: 0×000000000040113f hello`main at hello.c:6:3
   3
   4    int main(void)
   5    {
→ 6      printf("Hello world\n");
   7
   8      return EXIT_SUCCESS;
   9    }
(lldb) c
Process 61989 resuming
Hello world
Process 61989 exited with status = 0 (0×00000000)
```

Breakpoints

- Breakpoints are locations (code or binary) where the execution

binary breakpoint

```
(lldb) d -n main
hello`main:
0×401130 <+0>:  push   rbp
0×401131 <+1>:  mov    rbp, rsp
0×401134 <+4>:  sub    rsp, 0×10
0×401138 <+8>:  mov    dword ptr [rbp - 0×4], 0×0
0×40113f <+15>: movabs rdi, 0×402004
0×401149 <+25>: mov    al, 0×0
0×40114b <+27>: call   0×401030                  ; symbol stub for: printf
0×401150 <+32>: xor    ecx, ecx
0×401152 <+34>: mov    dword ptr [rbp - 0×8], eax
0×401155 <+37>: mov    eax, ecx
0×401157 <+39>: add    rsp, 0×10
0×40115b <+43>: pop    rbp
0×40115c <+44>: ret

(lldb) b 0×40114b
Breakpoint 2: where = hello`main + 27 at hello.c:6:3, address = 0×000000000040114b
(lldb) r
Process 62132 launched: '/mnt/hgfs/binary-hack/lectures/1-x86_64/hello' (x86_64)
Process 62132 stopped
* thread #1, name = 'hello', stop reason = breakpoint 1.1
    frame #0: 0×000000000040113f hello`main at hello.c:6:3
   3
   4    int main(void)
   5    {
→  6      printf("Hello world\n");
   7
   8      return EXIT_SUCCESS;
   9    }
```

Backtraces

- LLDB can provide the stack of calls that led to the current code position

```
$ lldb functions
(lldb) target create "functions"
Current executable set to '/mnt/hgfs/binary-hack/lectures/2-elf/functions' (x86_64).
(lldb) l
   20   {
   21     printf("%f\n", function2(2.));
   22
   23     return EXIT_SUCCESS;
   24   }
(lldb) b function1
Breakpoint 1: where = functions`function1 + 10 at functions.c:7:11, address = 0×000000000040113a
(lldb) r
Process 107292 launched: '/mnt/hgfs/binary-hack/lectures/2-elf/functions' (x86_64)
Process 107292 stopped
* thread #1, name = 'functions', stop reason = breakpoint 1.1
    frame #0: 0×000000000040113a functions`function1(a=2, b=2) at functions.c:7:11
   4
   5    int function1(int a, int b)
   6    {
→ 7      int c = a + b;
   8
   9      return c;
   10   }
(lldb) bt
* thread #1, name = 'functions', stop reason = breakpoint 1.1
  * frame #0: 0×000000000040113a functions`function1(a=2, b=2) at functions.c:7:11
    frame #1: 0×000000000040116c functions`function2(a=2) at functions.c:14:19
    frame #2: 0×00000000004011ac functions`main at functions.c:21:18
    frame #3: 0×00007ffff7de60b3 libc.so.6`__libc_start_main + 243
    frame #4: 0×000000000040106e functions`_start + 46
```

Investigating registers and memory

- Registers and memory can be investigated with a large choice of formats

```
(lldb) re r
General Purpose Registers:
       rax = 0x0000000000401190  functions`main at functions.c:20
       rbx = 0x00000000004011d0  functions`__libc_csu_init
       rcx = 0x00000000004011d0  functions`__libc_csu_init
       rdx = 0x00007fffffffdfe8
       rdi = 0x0000000000000002
       rsi = 0x0000000000000002
       rbp = 0x00007fffffffdea0
       rsp = 0x00007fffffffdea0
        r8 = 0x0000000000000000
        r9 = 0x00007ffff7fe0d50  ld-2.31.so`___lldb_unnamed_symbol59$$ld-2.31.so
       r10 = 0x0000000000000002
       r11 = 0x0000000000000000
       r12 = 0x0000000000401040  functions`_start
       r13 = 0x00007fffffffdfd0
       r14 = 0x0000000000000000
       r15 = 0x0000000000000000
       rip = 0x000000000040113a  functions`function1 + 10 at functions.c:7:11
    rflags = 0x0000000000000202
        cs = 0x0000000000000033
        fs = 0x0000000000000000
        gs = 0x0000000000000000
        ss = 0x000000000000002b
        ds = 0x0000000000000000
        es = 0x0000000000000000
(lldb) x/x 0x00007fffffffde9c
0x7fffffffde9c: 0x00000002
(lldb) x/d 0x00007fffffffde9c
0x7fffffffde9c: 2
```

- LLDB is actually a great calculator for programming!

LLDB demonstrations

- Complete commented logs

  https://bit.ly/2NFGEmS
  https://bit.ly/3kudPFT

Let’s play with stack frames

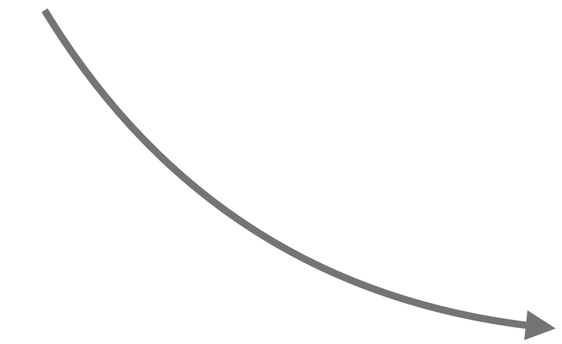
Test program with nested calls

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

int function1(int a, int b)
{
  int c = a + b;

  return c;
}

double function2(double a)
{
  double b = 3.14*function1(a, a);

  return b;
}

int main(void)
{
  printf("%f\n", function2(2.));

  return EXIT_SUCCESS;
}
```

https://godbolt.org/z/T17j3e

Some code analysis first

```c
int function1(int a, int b)
{
  int c = a + b;

  return c;
}
```



```asm
function1:
  push rbp
  mov rbp, rsp
  mov dword ptr [rbp - 4], edi
  mov dword ptr [rbp - 8], esi
  mov eax, dword ptr [rbp - 4]
  add eax, dword ptr [rbp - 8]
  mov dword ptr [rbp - 12], eax
  mov eax, dword ptr [rbp - 12]
  pop rbp
  ret
```

Should be quite clear at this point

Do you see how sub-optimal this is?
(no compiler optimisations)

Some code analysis first

```
1 double function2(double a)
2 {
3   double b = 3.14*function1(a, a);
4
5   return b;
6 }
```

```
1  .LCPI1_0:
2    .quad 4614253070214989087
3  function2:
4    push rbp
5    mov rbp, rsp
6    sub rsp, 16
7    movsd qword ptr [rbp - 8], xmm0
8    cvttsd2si edi, qword ptr [rbp - 8]
9    cvttsd2si esi, qword ptr [rbp - 8]
10   call function1
11   movsd xmm0, qword ptr [rip + .LCPI1_0]
12   cvtsi2sd xmm1, eax
13   mulsd xmm0, xmm1
14   movsd qword ptr [rbp - 16], xmm0
15   movsd xmm0, qword ptr [rbp - 16]
16   add rsp, 16
17   pop rbp
18   ret
```

Notes:

- `cvt*` convert `double` <> `int`
- `rip + .LCPI_0` will be the content at `.LCPI_0` (cf. RIP-relative addressing)
- 4614…. is 3.14 in binary/decimal!
- you can spot some useless statements

Stack frame from debugger

- Complete commented log

  https://bit.ly/3aYM318

Make your own: extracting stake frame info

- Stack frame info can be directly extracted in C

```c
void stackFrameInfo(void)
{
  uint64_t **rbp; // it does not have to be a double pointer
                  // it is just convenient for what follows

  __asm__ volatile
  (
    "movq %%rbp, %0;"
    : "=r"(rbp)
  );

  // here rbp is the base pointer of stackFrameInfo
  printf("==== STACK FRAME INFO ====\n");
  // 8B after is the return address of stackFrameInfo
  // i.e. the caller of stackFrameInfo
  printf("Called from %p\n", *(rbp + 1));
  // we actually want the base pointer of the caller...
  // remember, this is the first stack element (at the bottom)
  rbp = (uint64_t **)(*rbp);
  // again, return address is 8B after
  printf("Stack base %p -- return address %p\n", rbp, *(rbp + 1));
  printf("==========================\n");
}
```

Make your own: extracting stake frame info

- Let’s insert it in all functions of the previous example

```
$ ./stackframes
==== STACK FRAME INFO ====
Called from 0×401254
Stack base 0×7ffc25cac000 -- return address 0×7fb9062a70b3
==========================
==== STACK FRAME INFO ====
Called from 0×4011dc
Stack base 0×7ffc25cabfc0 -- return address 0×40120c
==========================
==== STACK FRAME INFO ====
Called from 0×401226
Stack base 0×7ffc25cabfe0 -- return address 0×401261
==========================
12.560000
```

- `addr2line` can convert addresses into code positions

```
$ addr2line -fp -e ./stackframes -a 0×401254 0×4011dc 0×401226
0×0000000000401254: main at /mnt/hgfs/binary-hack/lectures/2-elf/stackframes.c:49
0×00000000004011dc: function1 at /mnt/hgfs/binary-hack/lectures/2-elf/stackframes.c:35
0×0000000000401226: function2 at /mnt/hgfs/binary-hack/lectures/2-elf/stackframes.c:44
```

Make your own: backtrace

- You can walk the stack frame and get a backtrace

```c
void backtrace(void)
{
  uint64_t     **rbp;
  unsigned int level = 0;

  __asm__ volatile
  (
    "movq %%rbp, %0;"
    : "=r"(rbp)
  );

  printf("==== BACKTRACE ====\n");
  // keep walking until rbp is NULL
  // no guarantee this is a safe way to do thing
  while (rbp ≠ NULL)
  {
    // get the return address 8B after base pointer
    printf("frame -: %p\n", level, *(rbp + 1));
    // set rbp to be next frame pointer
    rbp = (uint64_t **)(*rbp);
    level++;
  }
  printf("==================\n");
}
```

Make your own: backtrace

- Call `backtrace` from `function1`

```
$ ./backtrace
==== BACKTRACE ====
frame  0: 0×4011cc
frame  1: 0×4011fc
frame  2: 0×40123c
frame  3: 0×7f4e7d29c0b3
==================
12.560000
$ addr2line -fp -e ./backtrace -a 0×4011cc 0×4011fc 0×40123c 0×7f4e7d29c0b3
0×00000000004011cc: function1 at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:36
0×00000000004011fc: function2 at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:41
0×000000000040123c: main at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:48
0×00007f4e7d29c0b3: ?? ??:0
```

- Why the last address is so different?
  What are the `??  ??` ?
  Why is it random between executions?
  I'll explain later

Frame pointer optimisation

- Actually **you don't need to save `rbp`**…
- You can keep **everything relative to `rsp`** and reduce `rsp` when needed to make the stack bigger
- This allow one more **free register** and **reduce code size**
- GCC/clang do that with `-fomit-frame-pointer`, it is included in `-O1` and higher
- Of course this is going to break the code before!

Frame pointer optimisation

```
$ clang -g -fomit-frame-pointer -o stackframes ./stackframes.c
antonin@ubuntu /mnt/hgfs/binary-hack/lectures/2-elf $ ./stackframes
==== STACK FRAME INFO ====
[1]    89547 segmentation fault (core dumped)  ./stackframes
$ clang -g -fomit-frame-pointer -o backtrace backtrace.c
antonin@ubuntu /mnt/hgfs/binary-hack/lectures/2-elf $ ./backtrace
==== BACKTRACE ====
==================
12.560000
```

- There is a POSIX API for backtraces… https://bit.ly/3r3NEYZ but I hope you learned something!

- And… (from the man page above)

```
*  Omission of the frame pointers (as implied by any of gcc(1)'s
   nonzero optimization levels) may cause these assumptions to be
   violated.
```

- So in general:

  Mind your optimisation level while debugging

The Executable and Linkable Format (ELF)

Remaining open questions

- How does the OS know how to start a program?
- How are jumps and calls handled if the program is divided across several executable binaries (i.e. libraries)?

Executable binary format

- When an executable is generated, its compiled code is wrapped up with all sorts of data explain the kernel how to run the program

- This data is generated by the **linker**

- On Linux, the adopted format is the **Executable and Linkable Format (ELF)**

```
$ file ./hello
./hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=113ce67758fa247a837becd3bac8a314e8c6bcf5, for GNU/Linux 3.2.0, with debug_info, not stripped
```

- This is **completely OS-dependent**, although ELF is common across a number of platforms (macOS/iOS uses Mach-O, Windows uses PE)

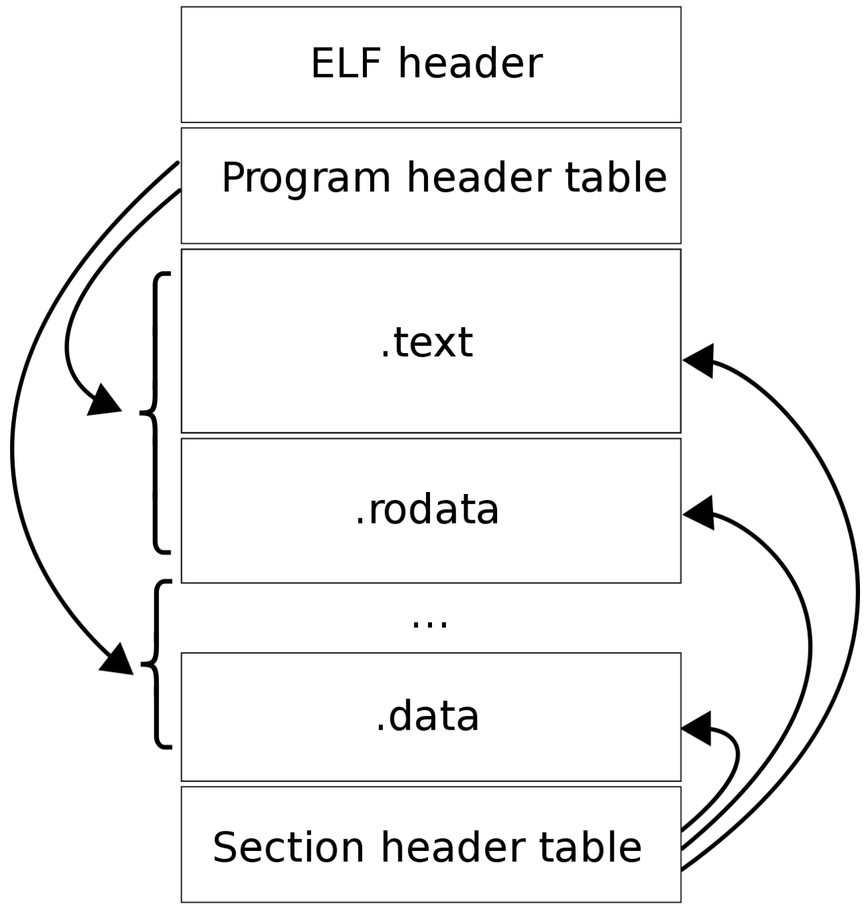
Quick description of ELF

1. A **ELF header** containing global properties of the binary

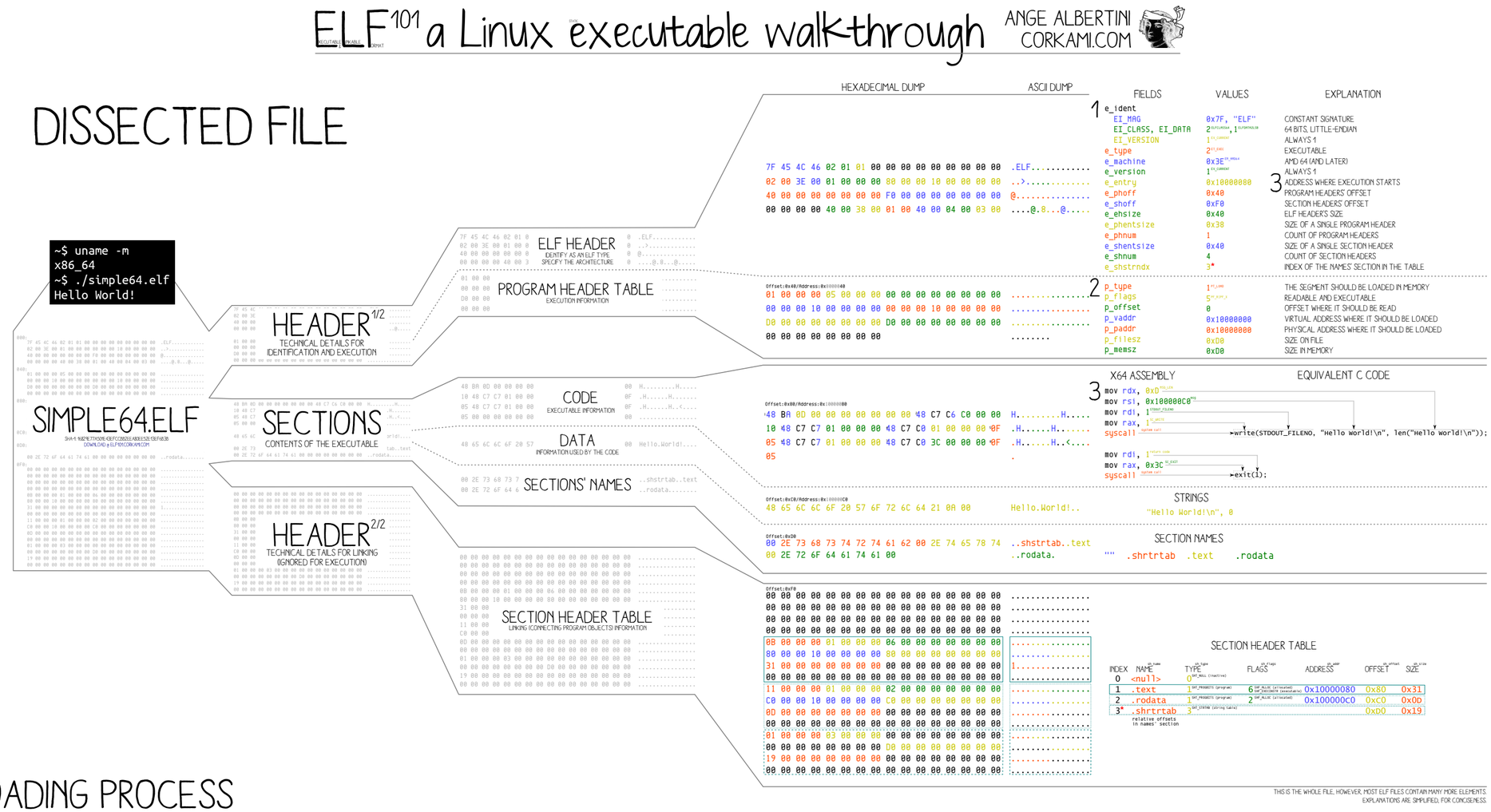
2. A **program table** describing the memory segments to create

3. **Sections** containing the content of the binary (code, data, …)

4. A **section header table** describing how to link against the binary



ELF in one chart



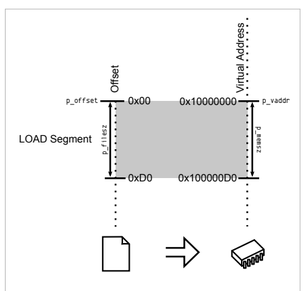
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE RAX REGISTER
- CALLING INSTRUCTION SYSCALL

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S.L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

https://bit.ly/37U3RbM

Binary tools

- The **`readelf`** command reads the ELF information
- The **`objdump`** command can read the content of the binary and disassemble code
- The **`nm`** command gives the symbols table (functions, constants, we'll see that in a min)
- The **`strings`** command gives the constant strings

Linking

- Remember, **function calls are just jumps**

- The linker implements in the binary all the necessary tools to find the addresses of functions when called

- ```
  $ clang ./hello.c -o ./hello
  ```

  actually does 2 things: compiling & linking

- Use `-c` to just compile, the result is an **object file**

  ```
  $ clang -c ./hello.c -o ./hello.o
  $ file ./hello.o
  ./hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
  ```

  the linker assemble all object files into a properly formed ELF executable binary
  ```

Linking

- The default linker on Linux is GNU LD
- **Calling the linker by hand is tricky**… there is a lot of information to provide on how to link to system libraries
- Using the compiler as a linker automatises that part

```
$ clang hello.o -o hello
$ ./hello
Hello world
```

- Use `-v` to know what the linking command was, here it is

```
/usr/bin/ld -z relro --hash-style=gnu --build-id --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.
so.2 -o ./hello /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crt1.o /usr/bin/../lib/gcc/x86_64-li
nux-gnu/9/../../../x86_64-linux-gnu/crti.o /usr/bin/../lib/gcc/x86_64-linux-gnu/9/crtbegin.o -L/usr/bin/../lib/gcc/x8
6_64-linux-gnu/9 -L/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu -L/usr/bin/../lib/gcc/x86_64-linu
x-gnu/9/../../../../lib64 -L/lib/x86_64-linux-gnu -L/lib/../lib64 -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib64 -L/
usr/lib/x86_64-linux-gnu/../../lib64 -L/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../.. -L/usr/lib/llvm-10/bin/../lib
 -L/lib -L/usr/lib /tmp/hello-a175e2.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as
-needed /usr/bin/../lib/gcc/x86_64-linux-gnu/9/crtend.o /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-
gnu/crtn.o
```

!?

Static linking

- During **static linking**, the binary code of the external function is **relocated and copied** inside the final binary

- Pro: the executable binary is **independent** from the library and therefore **more portable**

- Con: if the library is updated, all statically linked executables **need to be relinked**

- Con: statically linked executables are **larger**, and **redundant** information is stored

- Linux static libraries have extension `.a` and can be assembled from objects using `ar`

Dynamic linking

- Linking is done at runtime: when the program starts, the linked code is loaded in the program memory space

- The system and the executable contain informations on how to find the dynamically linked libraries at runtime (**dynamic links**)

- Pro: **smaller** executables

- Pro: if the library is updated, nothing needs to be done to dynamically linked programs (assuming same interface)

- Con: executables are **less portable** as they will always need the external libraries when loaded

Dynamic linking environment

- When dynamic linking, only references to the libraries files are generated in the binary, code is only linked at runtime

- You can see the dynamic link resolution using `ldd`

```
$ ldd latan-sample-read
        linux-vdso.so.1 (0×00007ffe04767000)
        libLatAnalyze.so.0 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libLatAnalyze.so.0 (0×00007fe1dbb70000)
        libstdc++.so.6 ⇒ /lib/x86_64-linux-gnu/libstdc++.so.6 (0×00007fe1db97a000)
        libm.so.6 ⇒ /lib/x86_64-linux-gnu/libm.so.6 (0×00007fe1db82b000)
        libgcc_s.so.1 ⇒ /lib/x86_64-linux-gnu/libgcc_s.so.1 (0×00007fe1db810000)
        libc.so.6 ⇒ /lib/x86_64-linux-gnu/libc.so.6 (0×00007fe1db61e000)
        libgsl.so.25 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libgsl.so.25 (0×00007fe1db33f000)
        libgslcblas.so.0 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libgslcblas.so.0 (0×00007fe1db2fb000)
        libnlopt.so.0 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libnlopt.so.0 (0×00007fe1db258000)
        libhdf5_cpp.so.103 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libhdf5_cpp.so.103 (0×00007fe1db1e5000)
        libhdf5.so.103 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libhdf5.so.103 (0×00007fe1dae01000)
        libMinuit2.so.0 ⇒ /home/antonin/Desktop/Plateau/dependencies/prefix/lib/libMinuit2.so.0 (0×00007fe1dad59000)
        /lib64/ld-linux-x86-64.so.2 (0×00007fe1dbd39000)
        libz.so.1 ⇒ /lib/x86_64-linux-gnu/libz.so.1 (0×00007fe1dad3b000)
```

- Path resolution sequence depends on the environment and information in the binary https://bit.ly/3bojyZJ

Linking troubleshooting checklist

```
error while loading shared libraries: libhdf5.so.10: cannot open shared object file: No such file or directory
```

```
/usr/bin/ld: undefined reference to `H5T_NATIVE_INT32_g'
/usr/bin/ld: undefined reference to `H5Pcreate'
/usr/bin/ld: undefined reference to `H5Tcopy'
/usr/bin/ld: undefined reference to `H5T_NATIVE_UINT32_g'
/usr/bin/ld: undefined reference to `H5Sget_simple_extent_dims'
/usr/bin/ld: undefined reference to `H5Eset_auto2'
/usr/bin/ld: undefined reference to `H5Sget_simple_extent_ndims'
/usr/bin/ld: undefined reference to `H5Aget_type'
/usr/bin/ld: undefined reference to `H5Fcreate'
/usr/bin/ld: undefined reference to `H5Sclose'
/usr/bin/ld: undefined reference to `H5T_NATIVE_SCHAR_g'
/usr/bin/ld: undefined reference to `H5Aopen_name'
/usr/bin/ld: undefined reference to `H5Gunlink'
/usr/bin/ld: undefined reference to `H5Lget_val'
/usr/bin/ld: undefined reference to `H5Tget_native_type'
/usr/bin/ld: undefined reference to `H5Acreate2'
/usr/bin/ld: undefined reference to `H5Aclose'
/usr/bin/ld: undefined reference to `H5Pset_deflate'
/usr/bin/ld: undefined reference to `H5T_C_S1_g'
/usr/bin/ld: undefined reference to `H5Tset_precision'
/usr/bin/ld: undefined reference to `H5Dopen2'
/usr/bin/ld: undefined reference to `H5Dget_type'
/usr/bin/ld: undefined reference to `H5P_CLS_LINK_ACCESS_ID_g'
/usr/bin/ld: undefined reference to `H5T_NATIVE_FLOAT_g'
/usr/bin/ld: undefined reference to `H5P_CLS_DATASET_CREATE_ID_g'
/usr/bin/ld: undefined reference to `H5Lunpack_elink_val'
/usr/bin/ld: undefined reference to `H5Dwrite'
/usr/bin/ld: undefined reference to `H5Lcreate_external'
/usr/bin/ld: undefined reference to `H5Screate_simple'
/usr/bin/ld: undefined reference to `H5Aread'
/usr/bin/ld: undefined reference to `H5check_version'
/usr/bin/ld: undefined reference to `H5Iget_name'
/usr/bin/ld: undefined reference to `H5Lis_registered'
/usr/bin/ld: undefined reference to `H5Fget_obj_ids'
/usr/bin/ld: undefined reference to `H5Gcreate2'
/usr/bin/ld: undefined reference to `H5Giterate'
/usr/bin/ld: undefined reference to `H5Tget_size'
/usr/bin/ld: undefined reference to `H5Lexists'
```

More about the linking process

- If a function is not implemented in the code (e.g. just declared in a header), it is compiled as an **undefined symbol**

```
$ nm ./hello.o
0000000000000000 T main
                 U printf
```

- When an executable is linked **all undefined symbols must be resolved** user linker options (e.g. `-l` & `-L` flags)

- This is a **recursive process**, libraries themselves might have undefined symbols

- This is true for both static & dynamic linking

C++ name mangling

- Symbol names are compiler-generated, C/C++ standards do not impose any convention

- However, C symbols are generally `_function` and C++ symbols are **mangled** according to **ABI standards**

- `c++filt` can be used to de-mangle C++ names

```
$ cat mangling.cpp
#include
std::string function(std::string s) {return "hello " + s;}
$ clang++ -c mangling.cpp -o mangling.o
$ nm mangling.o
0000000000000000 r GCC_except_table1
                 U __gxx_personality_v0
                 U strlen
                 U _Unwind_Resume
0000000000000000 T _Z8functionNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
                 U _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv
0000000000000000 W _ZNSt11char_traitsIcE6lengthEPKc
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE6appendEPKcm
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE6appendERKS4_
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE7reserveEm
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev
0000000000000000 W _ZStplIcSt11char_traitsIcESaIcEENSt7__cxx1112basic_stringIT_T0_T1_EEPKS5_RKS8_
$ echo _Z8functionNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE | c++filt
function(std::__cxx11::basic_string, std::allocator >)
```

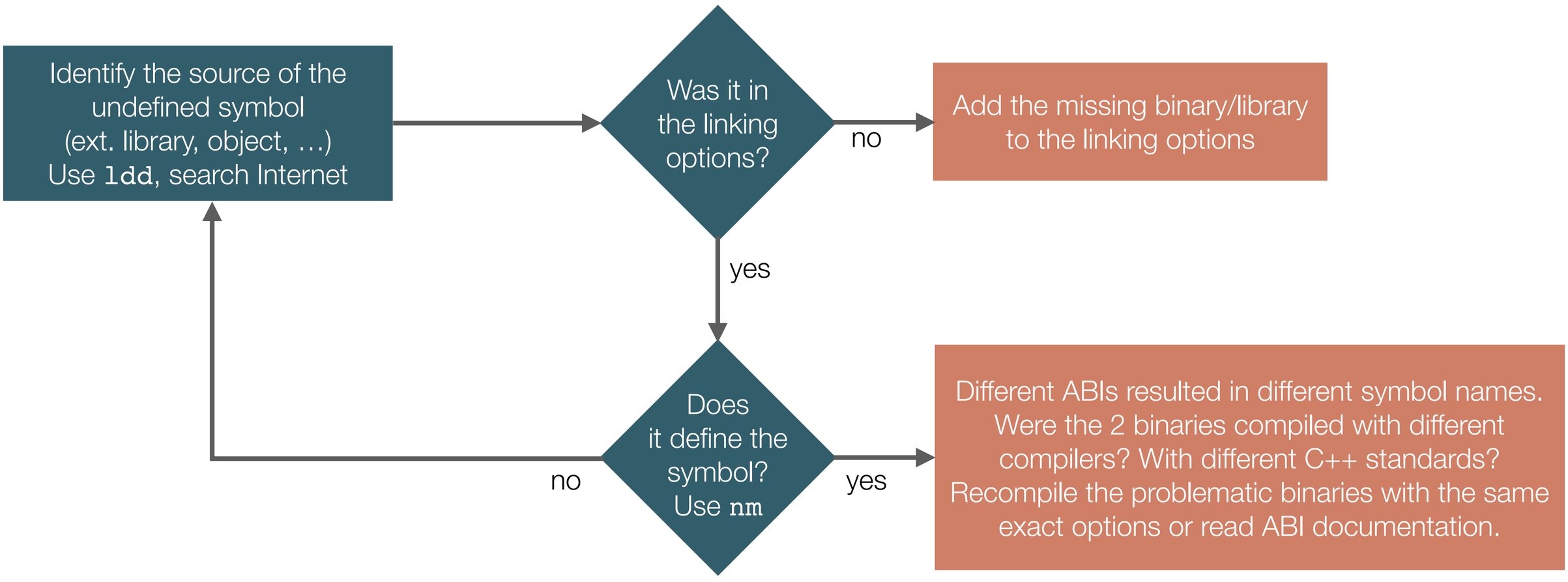
C++ name mangling

- Symbol nam ++ standards do not impo
- However, C on and C++ symbols are **ndards**
- `c++filt` c mes


```
$ cat ma
#include
std::str
$ clang-
$ nm mai
0000000
                 U strlen
                 U _Unwind_Resume
0000000000000000 T _Z8functionNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
                 U _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv
0000000000000000 W _ZNSt11char_traitsIcE6lengthEPKc
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE6appendEPKcm
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE6appendERKS4_
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE7reserveEm
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev
0000000000000000 W _ZStplIcSt11char_traitsIcESaIcEENSt7__cxx1112basic_stringIT_T0_T1_EEPKS5_RKS8_
$ echo _Z8functionNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE | c++filt
function(std::__cxx11::basic_string, std::allocator >)
```

‘undefined reference to’ error resolution

- Both static & dynamic, a symbol is still undefined after all linking options have been considered



- Rinse and repeat until it links

‘cannot open shared object file’ error resolution

- The binary was compiled with a dynamic link, but the linked library cannot be found at runtime.
  Use `ldd`, you’ll likely see ‘not found’ on one of the links.

- Have you messed with `rpath` or `runpath` linker options? If yes double-check that

- You can append the problematic library path to the environment variable `LD_LIBRARY_PATH`

- Very useful: `LD_DEBUG=libs ldd` shows the whole path resolution process

- Useful post for more info https://bit.ly/3axmcwQ

Loose ends

Make the “Hello world” shell code

- Position-independent ASM

```
1  section .text
2    global _start
3
4    _start:
5      jmp msg ; jump & call trick
6    print:
7      mov rax, 1
8      mov rdi, 1
9      pop rsi
10     mov rdx, 12
11     syscall
12     mov rax, 60
13     xor edi, edi
14     syscall
15   msg:
16     call print
17     db "Hello world",0xa
```

Make the “Hello world”

- Position-independent AS

```
1  section .t
2    global _
3
4    _start:
5      jmp ms
6    print:
7      mov rax, 1
8      mov rdi, 1
9      pop rsi
10     mov rdx, 12
11     syscall
12     mov rax, 60
13     xor edi, edi
14     syscall
15   msg:
16     call print
17     db "Hello world",0xa
```

- `call` will push the address of the next instruction on the stack
- here is it the address of the string!
- it can be popped later in `print`
- `print` actually never returns

You can write machine code directly in C!

“shellcode”

```c
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>

const unsigned char code [] =
  "\xeb\x1b\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x5e\xba\x0c"
  "\x00\x00\x00\x0f\x05\xb8\x3c\x00\x00\x00\x31\xff\x0f\x05\xe8"
  "\xe0\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";

int main(int argc, char **argv)
{
  // It used to be easier to break computers...
  // the commented code will likely end up in SIGSEGV on modern OSs
  // for security data is tagged non-executable
  ////////////////////////////////////////////////////////////////
  // void (*hello)(void);
  //
  // hello = (void(*)(void))code;
  // hello();
  ////////////////////////////////////////////////////////////////

  char *buf;
  int  prot  = PROT_READ | PROT_WRITE | PROT_EXEC;
  int  flags = MAP_PRIVATE | MAP_ANONYMOUS;

  buf = mmap(0, sizeof(code), prot, flags, -1, 0);
  memcpy(buf, code, sizeof(code));

  ((void (*)(void))buf)();

  return EXIT_SUCCESS;
}
```

You can write machine code directly in C!

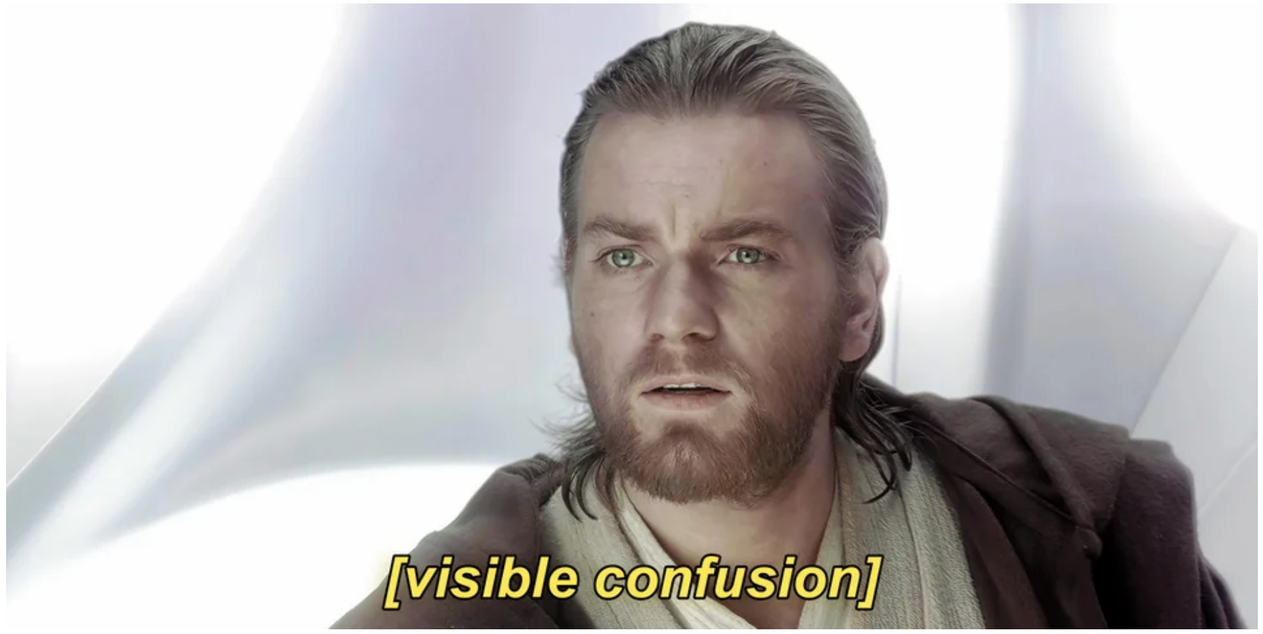
- `mmap` create a new memory region with executable permissions
- The shellcode is then copied in this new region
- Look at `/proc/<pid>/maps` before and after mmap is called, you will see the new region being created!
- In general, be curious about `/proc/<pid>`, it contains all the information about the program memory space
- Log from debugger investigation https://bit.ly/304b9VP

```
21  char *buf;
22  int  prot  = PROT_READ | PROT_WRITE | PROT_EXEC;
23  int  flags = MAP_PRIVATE | MAP_ANONYMOUS;
24
25  buf = mmap(0, sizeof(code), prot, flags, -1, 0);
26  memcpy(buf, code, sizeof(code));
27
28  ((void (*)(void))buf)();
29
30  return EXIT_SUCCESS;
31 }
32
```

Making the “Hello world” shell code

- Assemble and transform into a binary string

```
nasm -f elf64 hello-pi.asm -o hello-pi.o              # assemble
ld hello-pi.o -o hello-pi                             # link
objcopy -O binary hello-pi hello-pi.bin               # extract binary section
hexdump -v -e '"\\""x" 1/1 "%02x" ""' hello-pi.bin # convert to C string
```



- Probably better after next lecture

Making the “Hello world” shell code

- Assemble and transform into a binary string

```
nasm -f elf64 hello-pi.asm -o hello-pi.o              # assemble
ld hello-pi.o -o hello-pi                             # link
objcopy -O binary hello-pi hello-pi.bin               # extract binary section
hexdump -v -e '"\\""x" 1/1 "%02x" ""' hello-pi.bin # convert to C string
```

- **`objcopy`** extract the loadable part of an ELF binary (i.e. code & constants)
- Use **`objdump`** on the result to see!
- The **`hexdump`** command convert bytes e.g. 0xa3 to the notation **`\xa3`** used in C strings

- Probably better after next lecture

Make your own: backtrace

- Call `backtrace` from `function1`

```
$ ./backtrace
==== BACKTRACE ====
frame  0: 0×4011cc
frame  1: 0×4011fc
frame  2: 0×40123c
frame  3: 0×7f4e7d29c0b3
==================
12.560000
$ addr2line -fp -e ./backtrace -a 0×4011cc 0×4011fc 0×40123c 0×7f4e7d29c0b3
0×00000000004011cc: function1 at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:36
0×00000000004011fc: function2 at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:41
0×000000000040123c: main at /mnt/hgfs/binary-hack/lectures/2-elf/backtrace.c:48
0×00007f4e7d29c0b3: ?? ??:0
```

- Why the last address is so different?
  What are the `??  ??` ?
  Why is it random between executions?
  I'll explain later

Make your own: backtrace

- Call `backtrace` from `function1`
- The mysterious address was a live memory address where dynamically linked code was loaded (it was the system loader calling **`main`**)
- This address can't be resolved after the program terminated, hence the ???
- It is randomised for security reasons, that's called Address Space Layout Randomization (ASLR)
- ASLR makes buffer overflow exploits much harder to create
- I'll explain later

Wrap up

Summary

- The program memory space is split is different segments with different purposes
- The stack stores temporary data related to local variables
- Debuggers are invaluable development tools which allow to have full visibility & control of the program's execution
- The ELF binary format split code and data in sections which are mapped onto the segments during execution
- The ELF format allow linking, i.e. the modular creation of an executable image from different binaries

Next lecture

- More about the debugger, this time from an actual debugging point of view

- The horrors of memory bugs and some strategies to try to systematically deal with them

- Code profiling and optimisation strategy

- The roofline performance model

- Practical profiling and performance measurement for two simple linear algebra routines