# Reverse Engineering

## Class 0

## Introduction

martin.uy

# Open by default.

# Hello!

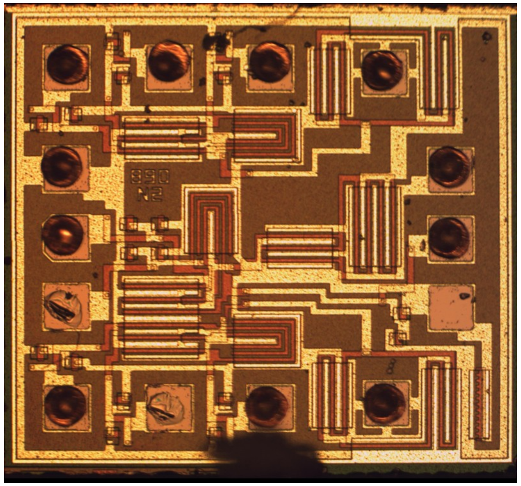
- Name?
- Professional interests?
  - Languages?
  - Technologies?
- Job?
- Free time projects?
- Course expectations?

# Reverse Engineering

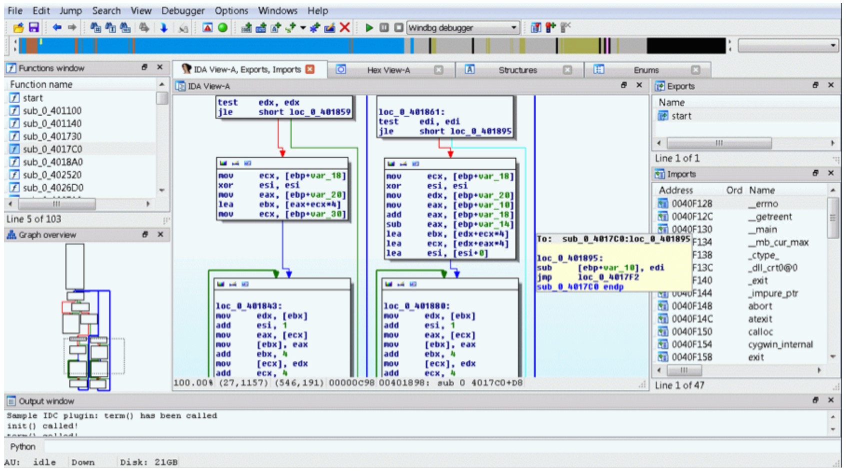
*“study or analyze (a device, as a computer microchip) to **learn** design details, construction and operation, and perhaps to make a copy or an **improved version**” **

* Random House Dictionary, 2017

# Reverse Engineering



Hardware



Function name
start
sub_0_401100
sub_0_401140
sub_0_401730
sub_0_4017C0
sub_0_4018A0
sub_0_402520
sub_0_4026D0

Line 5 of 103

```
test    edx, edx
jle     short loc_0_401859
```

```
mov     ecx, [ebp+var_18]
xor     esi, esi
mov     eax, [ebp+var_20]
lea     ebx, [eax+ecx*4]
mov     ecx, [ebp+var_30]
```

```
loc_0_401843:
mov     edx, [ebx]
add     esi, 1
mov     eax, [ecx]
mov     [ebx], eax
add     ebx, 4
mov     [ecx], edx
add     ecx, 4
```

```
loc_0_401861:
test    edi, edi
jle     short loc_0_401895
```

```
mov     ecx, [ebp+var_18]
xor     esi, esi
mov     edx, [ebp+var_20]
mov     eax, [ebp+var_10]
add     eax, [ebp+var_18]
sub     eax, [ebp+var_14]
lea     ebx, [edx+ecx*4]
lea     ecx, [edx+eax*4]
lea     esi, [esi+0]
```

```
loc_0_401880:
mov     edx, [ebx]
add     esi, 1
mov     eax, [ecx]
mov     [ebx], eax
add     ebx, 4
mov     [ecx], edx
add     ecx, 4
```

```
To:  sub_0_4017C0:loc_0_401895

loc_0_401895:
sub     [ebp+var_10], edi
jmp     loc_0_4017F2
sub_0_4017C0 endp
```

```
100.00% (27,1157) (546,191) 00000C98 00401898: sub 0 4017C0+D8
```

Name
start

Line 1 of 1

| Address | Ord | Name |
|---|---|---|
| 0040F128 | | _errno |
| 0040F12C | | _getreent |
| 0040F130 | | _main |
| F134 | | _mb_cur_max |
| F138 | | _ctype_ |
| F13C | | _dll_crt0@0 |
| F140 | | _exit |
| 0040F144 | | _impure_ptr |
| 0040F148 | | abort |
| 0040F14C | | atexit |
| 0040F150 | | calloc |
| 0040F154 | | cygwin_internal |
| 0040F158 | | exit |

Line 1 of 47

```
Sample IDC plugin: term() has been called
init() called!
```

```
AU:  idle   Down    Disk: 21GB
```

Software

# Jobs

JOB

- Security Consultant
- Malware Analyst
- Security Researcher
- Red Team
- Reverse Engineer
- Exploit Writer

PEACH FUZZER

Google

IOActive

CORE SECURITY

KASPERSKY lab

CYLANCE

# Course goals

- Reverse executable binaries
- Analyze binary malware
- Find vulnerabilities
- Exploit vulnerabilities

- Learn about APIs, ABIs, binary formats, reverse engineering techniques, debugging, systems implementation languages (C/C++), tools and working environments.

# Nice-to-haves

- Knowledge
  - C/C++
  - Operating systems (Windows, Linux)
  - x86 and x86_64 architectures
  - Debuggers
- Soft skills
  - Methodology, systematicity and perseverance
  - Motivation
  - Preparation of suitable working environments
  - Heuristics and intuition

# Course structure

- 1 introductory class
- 10 theoretical and hands-on classes
- 4 project classes (of choice)
  - CTFs / binary crackmes
  - Malware analysis or development
  - Fuzzer development
  - Other idea?

# Course structure (2)

- Important dates
  - Project choice
  - Project deadline
  - Course completion

# Syllabus

- Module 1: Executable binaries (3 classes)
  - ELF, PE, static and dynamic analysis
- Module 2: Malware analysis (2 classes)
  - Development, unpacking and process injection
- Module 3: Bug hunting (2 classes)
  - Fuzzing, binary instrumentation and dynamic analysis

# Syllabus (2)

- Module 4: Binary exploitation (3 classes)
  - Stack overflow, integer overflow, use-after-free and ROP chain

# Materials

- Virtual Box VM (Linux)
  - Brought by the course
- Windows 7 (virtual or physical)
  - Visual Studio Express
  - IDA Pro demo
  - API Monitor
  - CFF Explorer
  - Wireshark

# Communication channels

- Web
  - martin.uy/reverse
    - Updated slides
- Mail

  ✉ m@rtin.uy

# License

- All course materials licensed with **Creative Commons Attribution-ShareAlike International 4.0**
  - creativecommons.org

- Contributions welcomed :-)

# Free software

- Free to use
- Free to study and modify
- Free to distribute
- Free to improve

FSF

FREE SOFTWARE
FOUNDATION

Why joining an open source project?

open source
initiative

fsf.org | opensource.org

# We have cookies!

- Reverse Engineering course
- Free Software Development Group
  - Glibc
  - OpenJDK
- Graduate final projects

# Linux VM Lab Work

Introduction to the course VM
Virtual Box

# Linux VM Lab Work

- Fedora 25 – x86_64
  - 4 GB RAM minimum
  - 100 GB HDD maximum
  - 2+ CPUs recommended
  - Access credentials: user/1234
- Development, deploy and debugging environment
  - Linux kernel
  - Glibc
- See “README_VM” document

# Linux VM Lab Work

- Virtual Machine Manager (qemu)
  - Linux_VM_Lab_Target
    - Fedora 25 (x86_64)
    - IP: 192.168.122.2
    - Access credentials: test/1234
  - Binary translation → slow to run a graphical user interface but enough for command line

# C

- Dennis Ritchie
  - 1941 – 2011
  - Ph.D. Harvard University
  - Unix co-creator (Bell Labs)
  - Turing award 1983
- The C Programming Language
  - Dennis Ritchie & Brian Kernighan
  - 1st edition 1978
  - Recommended reading

# C

THE C PROGRAMMING LANGUAGE

- Standard language
  - ISO/IEC
  - C89, C90, C95, C99, C11
  - Portability (multiple platforms)
  - Components
    - Language (syntax and semantics)
    - Libraries

# C

- Imperative, structured and statically typed language
- General purpose and relatively “low level”
  - Systems implementation
  - Operating systems
  - Compilers
  - Virtual machines (I.e. CPython)
  - “Most of the important code is in C” (*)

(*) Ph.D. Thomas Schwarz

# C

- Simple and easy, yet powerful
- Multi-platform (with some care)
- Compiled to architecture native code (generally)
- No garbage collector: developer has to manage memory (as well as other resources)

(*) Thomas Schwarz

# C

- Structure
  - Headers (.h)
    - Variables declaration, functions and other external data types (from other objects or shared libraries)
  - Implementation (.c)
    - Variables declaration, functions and other object internal data types (encapsulation criteria)
    - Exported variables definition and initialization
    - Exported functions implementation
  - At the end of the day, headers (.h) are just text included in implementation (.c) files

# C

THE C PROGRAMMING LANGUAGE

- Pre-processor macros
  - Text level modification, before compilation

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#define CONST_1 1

/* … */

#endif // HEADER_H
```

# C

THE
C
PROGRAMMING
LANGUAGE

- Some operators (expressions)
  - Arithmetic
    - +, *, /, -, % (binaries) y ++,  - -, (unitary)
  - Booleans
    - && (AND), || (OR), ! (NOT), == (EQ), != (NEQ), >=, <=
  - Bits
    - ^ (XOR), | (OR), ~ (NOT), & (AND), << and >> (shift)
  - Conditional
    - ( condition ) ? true-case : false-case
  - Assignment (=, +=, -=, *=, %=, etc.)

# C

- Some operators (expressions)

```
int a = 0x0;
int b = 0xFFFFFFFF;
a |= (1 << 2);
b &= ~(1 << 2);
```

**What's happening with a?**

**What's happening with b?**

# C

- Some operators (expressions)

```
int a = 0x0;

int b = 0xFFFFFFFF;

a |= (1 << 2);

b &= ~(1 << 2);
```

**a = set a 1 in bit 3 (from the right)**

**b = set a 0 in bit 3 (from the right)**

# C

- Constants
  - Long
    - 1L
  - Unsigned
    - 1U
  - Unsigned long
    - 1UL
  - Float
    - 1.0f, 1e-2
  - Hex
    - 0x1

# C

- Constants
  - Octal
    - 01
  - Characters
    - ‘0’ (ASCII value), ‘\n’, ‘\t’, ‘\0’, ‘\x...’ (# byte), etc.
  - String
    - “abc”
    - What’s the difference between “x” and ‘x’?

# C

- Data types
  - long
  - int
  - short
  - char
  - float / double
  - struct abc {

    …

    }

# C

- Data types
  - void(*)(void) / void*
  - enum abc { … }
  - typedef type_1 type_2

```
typedef struct a {
    int m1;
} a_t;
```

# C

```
struct a {
    int a_1;
};
```

Data aggregation

```
union b {
    int b_1;
    char b_2;
};
```

Size of the larger member. Used in a context that allows to decide what’s the valid variable type for the union.

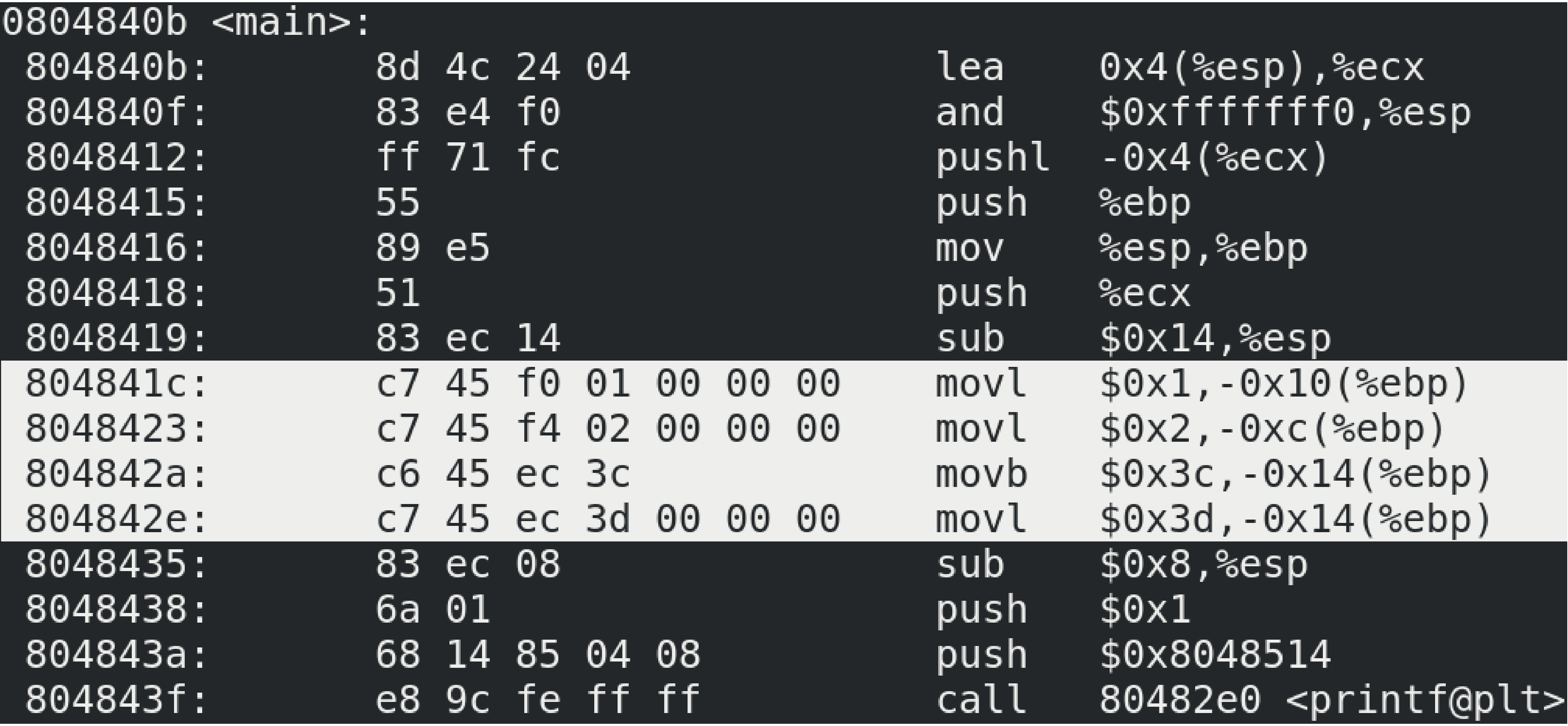
```
enum c {
    c_1 = 0,
};
```

Enum underlying type is decided by the compiler (implementation). Example: int.

# C

```
enum a_e { A = 1, B, C } a;
struct b {
    int a;
    int b;
} b;
union c {
    char d;
    int e;
} c;

b.a = (int)A;
b.b = 2;
c.d = 60;
c.e = 61;
```

# ASM (x86_64)



```
0804840b <main>:
 804840b:	8d 4c 24 04          	lea    0x4(%esp),%ecx
 804840f:	83 e4 f0             	and    $0xfffffff0,%esp
 8048412:	ff 71 fc             	pushl  -0x4(%ecx)
 8048415:	55                   	push   %ebp
 8048416:	89 e5                	mov    %esp,%ebp
 8048418:	51                   	push   %ecx
 8048419:	83 ec 14             	sub    $0x14,%esp
 804841c:	c7 45 f0 01 00 00 00 	movl   $0x1,-0x10(%ebp)
 8048423:	c7 45 f4 02 00 00 00 	movl   $0x2,-0xc(%ebp)
 804842a:	c6 45 ec 3c          	movb   $0x3c,-0x14(%ebp)
 804842e:	c7 45 ec 3d 00 00 00 	movl   $0x3d,-0x14(%ebp)
 8048435:	83 ec 08             	sub    $0x8,%esp
 8048438:	6a 01                	push   $0x1
 804843a:	68 14 85 04 08       	push   $0x8048514
 804843f:	e8 9c fe ff ff       	call   80482e0 <printf@plt>
```

# C

THE C PROGRAMMING LANGUAGE

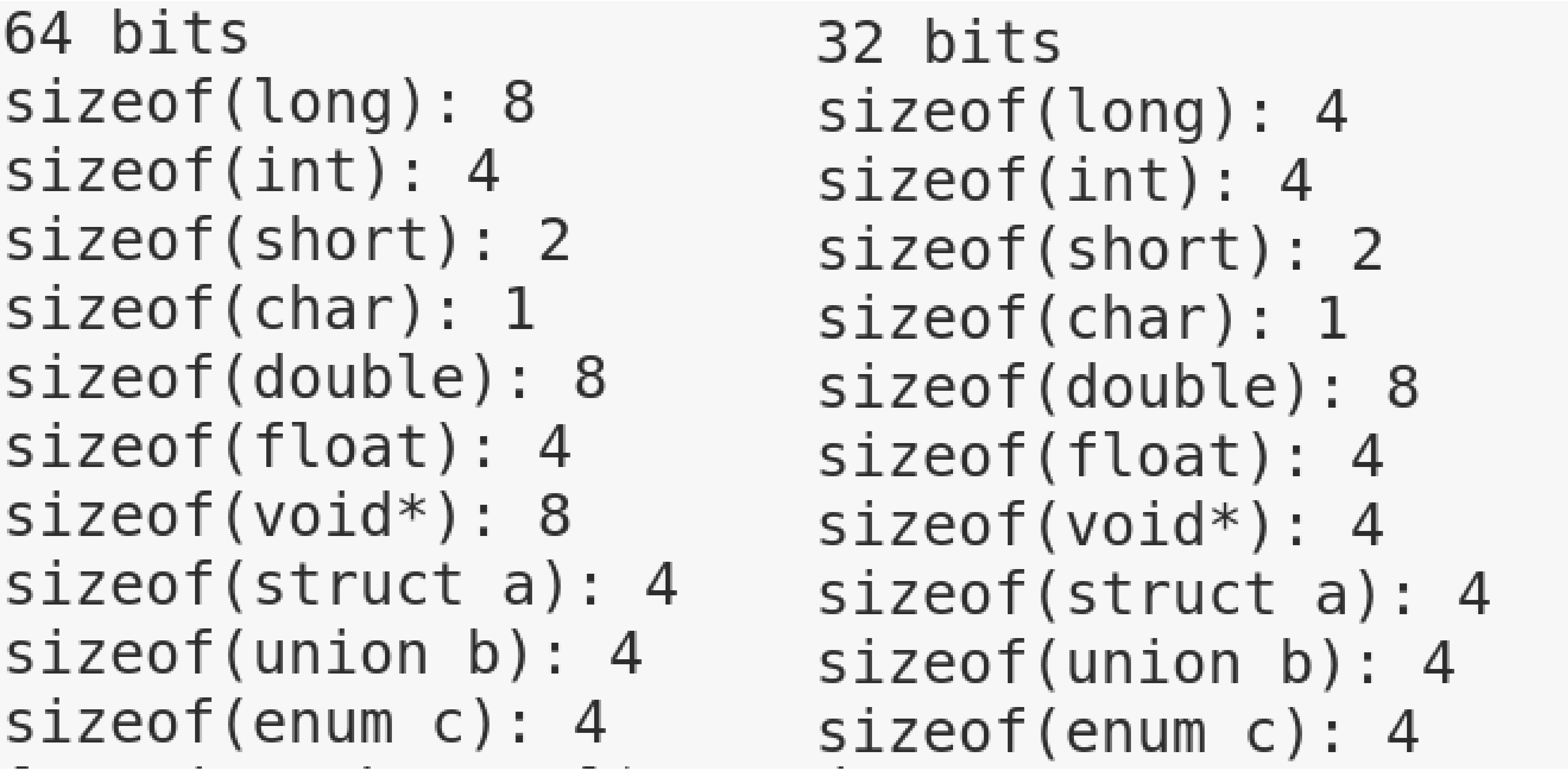
```
printf("sizeof(long): %d\n", sizeof(long));
printf("sizeof(int): %d\n", sizeof(int));
printf("sizeof(short): %d\n", sizeof(short));
printf("sizeof(char): %d\n", sizeof(char));
printf("sizeof(double): %d\n", sizeof(double));
printf("sizeof(float): %d\n", sizeof(float));

printf("sizeof(struct a): %d\n", sizeof(struct a));
printf("sizeof(union b): %d\n", sizeof(union b));
printf("sizeof(enum c): %d\n", sizeof(enum c));
```

Do we have enough information to decide what are the sizes of these data types?

???

# C

THE C PROGRAMMING LANGUAGE



```
64 bits
sizeof(long): 8
sizeof(int): 4
sizeof(short): 2
sizeof(char): 1
sizeof(double): 8
sizeof(float): 4
sizeof(void*): 8
sizeof(struct a): 4
sizeof(union b): 4
sizeof(enum c): 4
```

```
32 bits
sizeof(long): 4
sizeof(int): 4
sizeof(short): 2
sizeof(char): 1
sizeof(double): 8
sizeof(float): 4
sizeof(void*): 4
sizeof(struct a): 4
sizeof(union b): 4
sizeof(enum c): 4
```

# ASM (x86_64)

```
void* d = (void*)-1;

long e = -1L;

int f = -1;

short g = -1;

char h = -1;
```

```
nop
movq    $0xffffffffffffffff,-0x8(%rbp)
nop
movq    $0xffffffffffffffff,-0x10(%rbp)
nop
movl    $0xffffffff,-0x14(%rbp)
nop
movw    $0xffff,-0x16(%rbp)
nop
movb    $0xff,-0x17(%rbp)
```

# C

- Declare (functions and variables)
  - Before usage
  - Specify types (I.e. int a)
- Initialize variables
  - Assign value (I.e. a = 1)
  - Global variables: 0 or NULL by default
  - Locales variables: garbage by default
- It’s possible to declare and initialize variables at the same time (I.e. int a = 1)

# C

- Scope
  - Local (to a function)
  - Object (static)
  - Global
- Flow control structures (if, for, while, do-while, switch, break, goto, return)

# C

- Const correctness

```
const int a = 1;

const int *b = &a;

char *c = "abc";

a = 2; // Is it possible?

*b = 3; // Is it possible?

b = (int*)0x0; // Is it possible?

c[0] = 'b'; // Is it possible?
```

# C

- Const correctness

```
const int a = 1;

const int *b = &a;

char *c = "abc";

a = 2; // Is it possible? ✗

*b = 3; // Is it possible? ✗

b = (int*)0x0; // Is it possible? ✓

c[0] = 'b'; // Is it possible? ✓
```

✓ Compiles ✗ Executes

# C

- Const correctness

```
const int *d = (const int*)0x1;

const int *const e = (const int*)0x1;

int *const f = d; // Is it possible?

int *g = d; // Is it possible?

*e = 2; // Is it possible?

e = (const int*)2; // Is it possible?

*f = 2; // Is it possible?
```

# C

THE C PROGRAMMING LANGUAGE

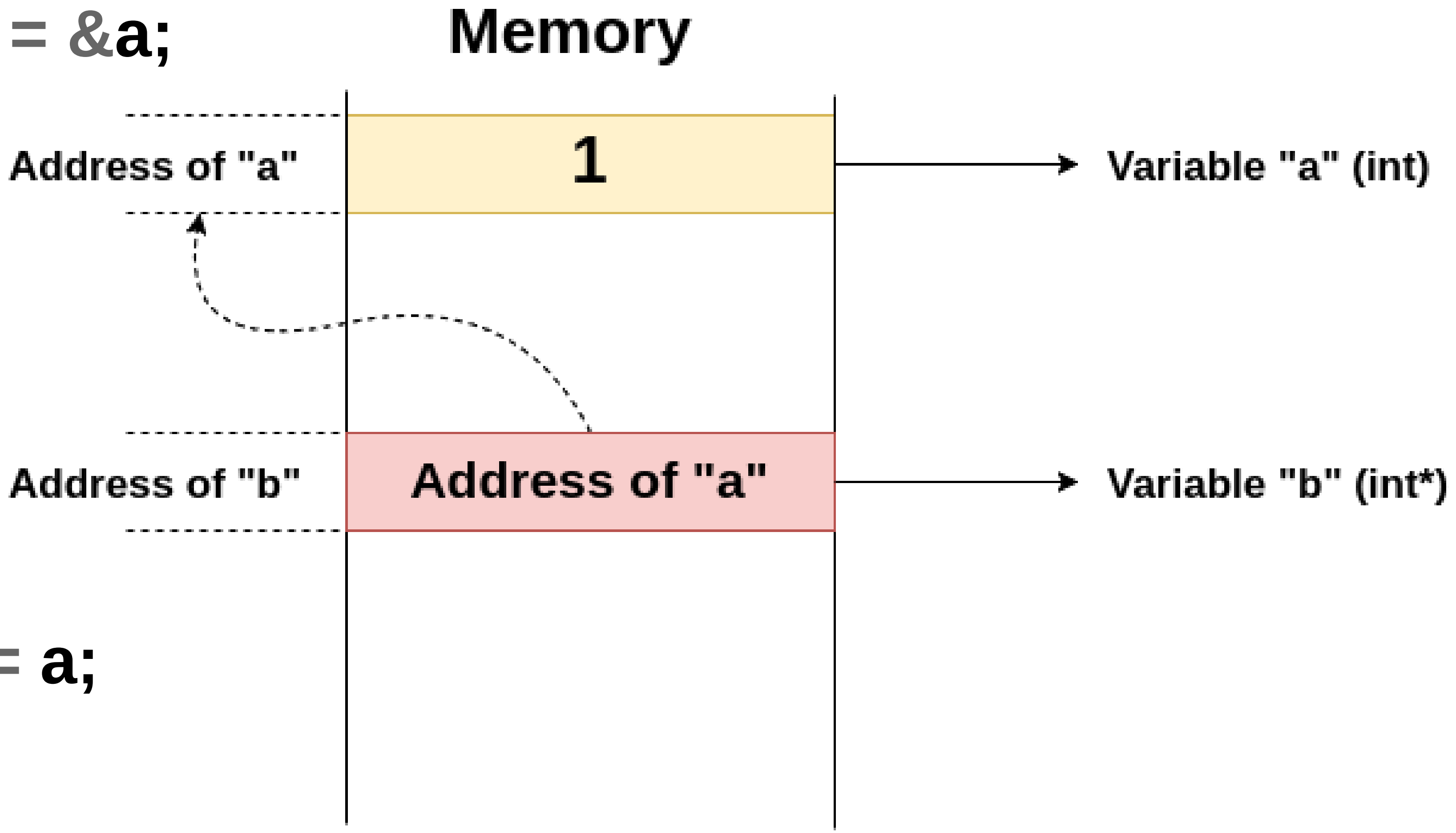
- Const correctness

```
const int *d = (const int*)0x1;

const int *const e = (const int*)0x1;

int *const f = d; // Is it possible? ✓ “const” qualifier is discarded

int *g = d; // Is it possible? ✓ “const” qualifier is discarded

*e = 2; // Is it possible? ✗

e = (const int*)2; // Is it possible? ✗

*f = 2; // Is it possible? ✓ Compiles ✗ Executes
```

# C

- Pointers

```
int a = 1;
int *b = &a;
```



```
*b == a;
```

# C

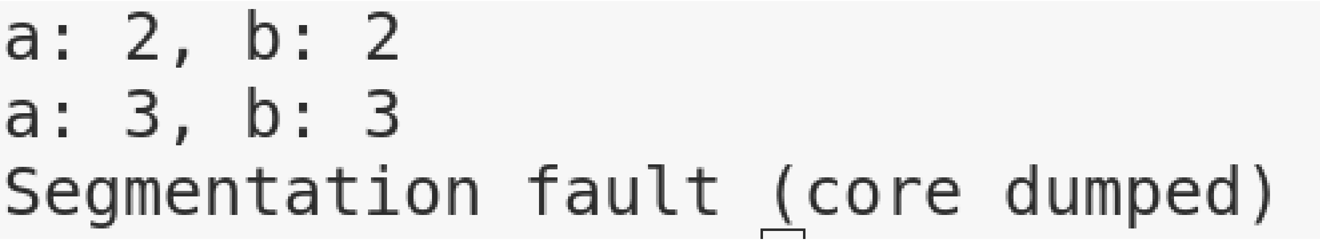
- Pointers

```
int a = 1;
int *b = &a;
a = 2;

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

*b = 3;
printf("a: %d, b: %d\n", a, *b);

b = (int*)0x4;
printf("b: %d\n", *b);
```

# C

- Pointers

```c
int a = 1;
int *b = &a;
a = 2;

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

*b = 3;
printf("a: %d, b: %d\n", a, *b);

b = (int*)0x4;
printf("b: %d\n", *b);
```



```
a: 2, b: 2
a: 3, b: 3
Segmentation fault (core dumped)
```

# C

- Pointers operators

```
struct a {
    int m1;
};

struct a v1;
struct a *v2 = &v1;

v1.m1 = 0;
v2->m1 = 1; // Equivalent to (*v2).m1 = 1;
```

# C

- Pointers arithmetics

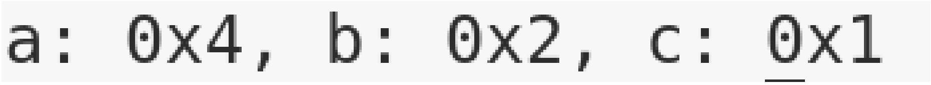
```
int *a = (int*)0x0;
short *b = (short*)0x0;
int *c = (int*)0x0;

a = a + 1;
b = b + 1;
c = (int*)((char*)c + 1);

printf("a: %p, b: %p, c: %p\n", a, b, c);
```

# C

- Pointers arithmetics

```
int *a = (int*)0x0;
short *b = (short*)0x0;
int *c = (int*)0x0;

a = a + 1;
b = b + 1;
c = (int*)((char*)c + 1);

printf("a: %p, b: %p, c: %p\n", a, b, c);
```

a + sizeof(int)



```
a: 0x4, b: 0x2, c: 0x1
```

# C

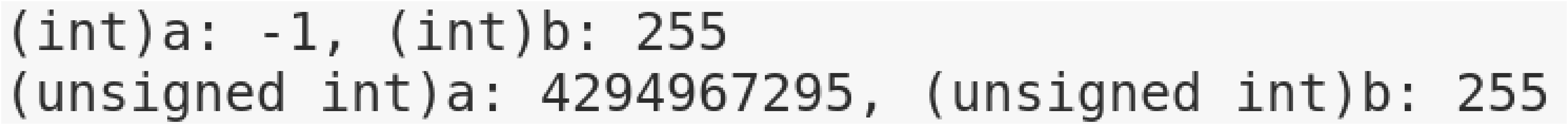
- Casting

```
char a = -1;
unsigned char b = -1;

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

printf("(unsigned int)a: %u, (unsigned int)b: %u\n",
(unsigned int)a, (unsigned int)b);
```

# C

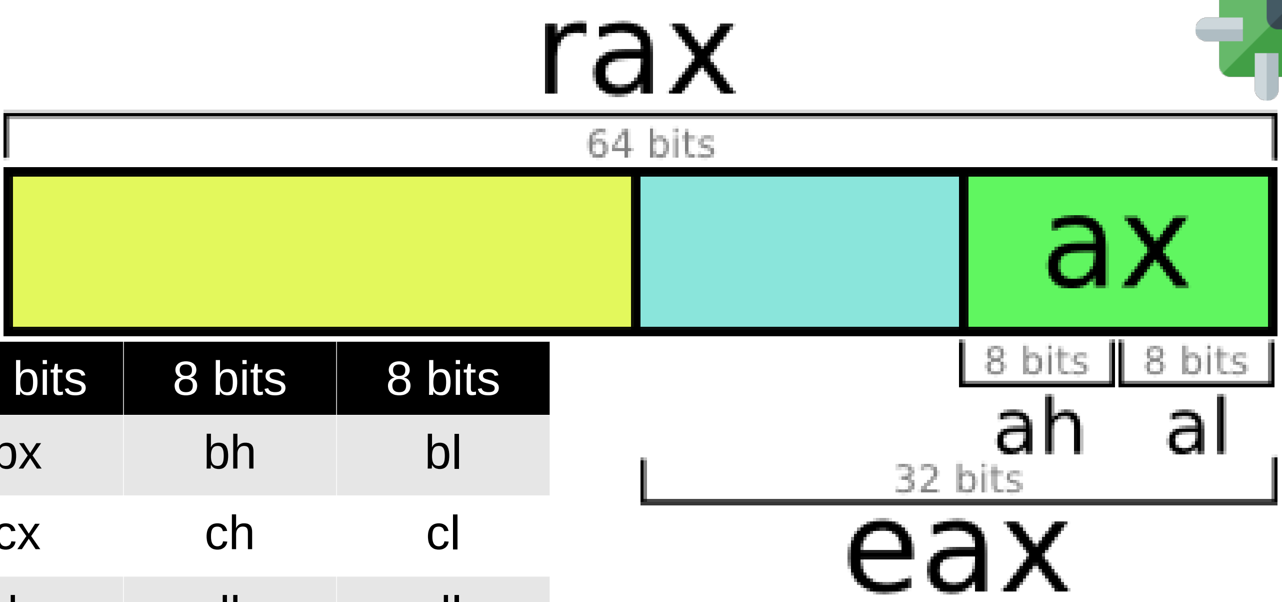
- Casting

```
char a = -1;
unsigned char b = -1;

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

printf("(unsigned int)a: %u, (unsigned int)b: %u\n",
(unsigned int)a, (unsigned int)b);
```



```
(int)a: -1, (int)b: 255
(unsigned int)a: 4294967295, (unsigned int)b: 255
```

# ASM (x86_64)



| 64 bits | 32 bits | 16 bits | 8 bits | 8 bits |
|---|---|---|---|---|
| rbx | ebx | bx | bh | bl |
| rcx | ecx | cx | ch | cl |
| rdx | edx | dx | dh | dl |
| rbp | ebp | bp | - | - |
| rsp | esp | sp | - | - |
| rsi | esi | si | - | - |
| rdi | edi | di | - | - |
| r8 | r8d | r8w | - | r8b |
| ... | ... | ... | ... | ... |

Image from http://nullprogram.com/blog/2015/05/15/

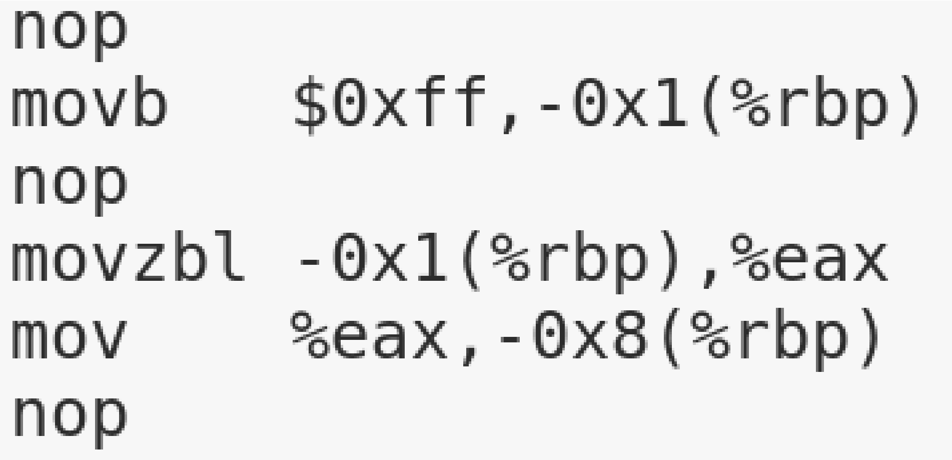
# ASM (x86_64)

```c
char a = -1;

short b = (short)a;

int c = (short)a;

long d = (long)a;

long e = (long)b;

long f = (long)c;
```

```
nop
movb   $0xff,-0x1(%rbp)
nop
movsbw -0x1(%rbp),%ax
mov    %ax,-0x4(%rbp)
nop
movsbl -0x1(%rbp),%eax
mov    %eax,-0x8(%rbp)
nop
movsbq -0x1(%rbp),%rax
mov    %rax,-0x10(%rbp)
nop
movswq -0x4(%rbp),%rax
mov    %rax,-0x18(%rbp)
nop
mov    -0x8(%rbp),%eax
cltq
mov    %rax,-0x20(%rbp)
nop
```

# ASM (x86_64)

```
unsigned char a = 255U;

unsigned int b = (unsigned int)a;

printf("b: %d\n", b);
```



```
nop
movb   $0xff,-0x1(%rbp)
nop
movzbl -0x1(%rbp),%eax
mov    %eax,-0x8(%rbp)
nop
```

```
b: 255
```

# C

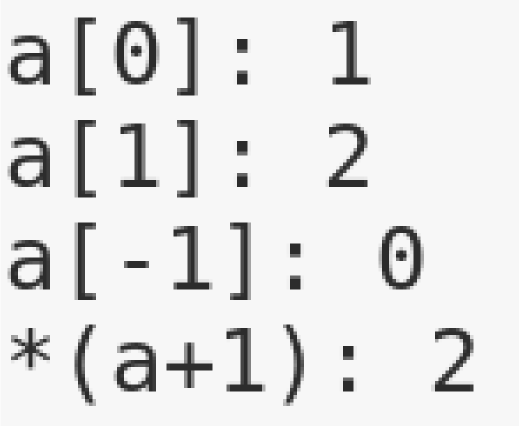
- Arrays

```
int a[2] = {0x1, 0x2};

printf("a[0]: %d\n", a[0]);
printf("a[1]: %d\n", a[1]);
printf("a[-1]: %d\n", a[-1]);
printf("*(a+1): %d\n", *(a+1));
```

# C

- Arrays

```c
int a[2] = {0x1, 0x2};

printf("a[0]: %d\n", a[0]);
printf("a[1]: %d\n", a[1]);
printf("a[-1]: %d\n", a[-1]);
printf("*(a+1): %d\n", *(a+1));
```



```
a[0]: 1
a[1]: 2
a[-1]: 0
*(a+1): 2
```

# C

- Arrays

```
int b[] = {0x1}; // is it possible?

int *c = b; // is it possible?

char *d = "abcde"; // is it possible?

char e[] = "abcde"; // is it possible?

char *f = d; // is it possible?

char g[] = d; // is it possible?
```

# C

- Arrays

`int b[] = {0x1};` ✓

`int *c = b;` ✓

`char *d = "abcde";` ✓

`char e[] = "abcde";` ✓

`char *f = d;` ✓

`char g[] = d;` ✗

# C

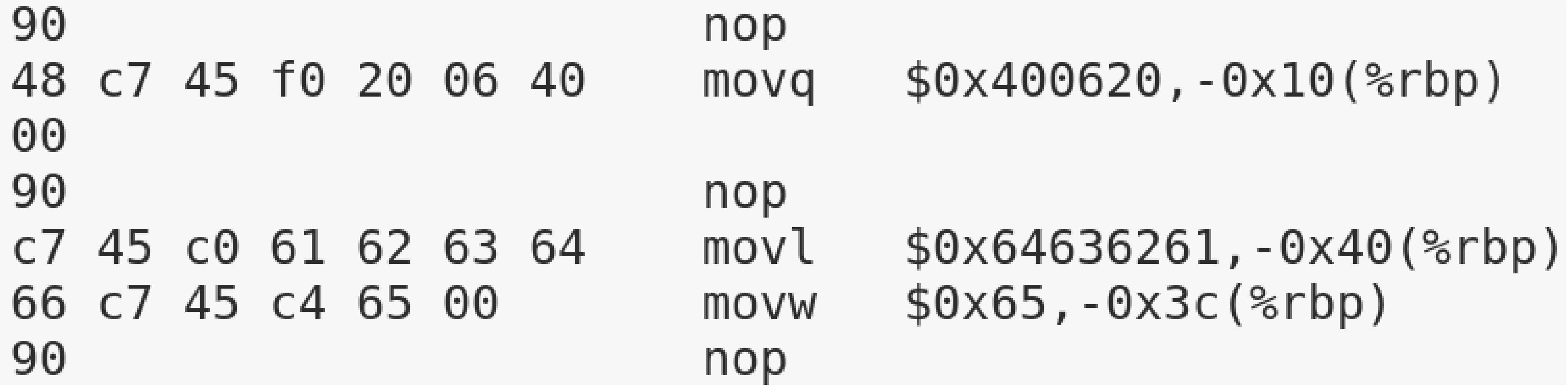
- What’s the difference?

```
char *d = "abcde";

char e[] = "abcde";
```

# ASM (x86_64)

- What’s the difference?

```
char *d = "abcde";

char e[] = "abcde";
```



```
90                        nop
48 c7 45 f0 20 06 40      movq   $0x400620,-0x10(%rbp)
00
90                        nop
c7 45 c0 61 62 63 64      movl   $0x64636261,-0x40(%rbp)
66 c7 45 c4 65 00         movw   $0x65,-0x3c(%rbp)
90                        nop
```

# ASM (x86_64)

- Storage (strings, ints and pointers)

```
char *a = "abc\xEA\x9F\xB9";

int b = 0x01020304;

int *c = &b;
```

# ASM (x86_64)

- Storage (strings, ints and pointers)

```
char *a = "abc\xEA\x9F\xB9";

int b = 0x01020304;

int *c = &b;
```

**UTF-8 encoded string, null terminated**

```
0x4005d0:       0x61    0x62    0x63    0xea    0x9f    0xb9    0x00
(gdb) x/4xb ($rbp - 0x14)
0x7fffffffdd5c: 0x04    0x03    0x02    0x01
(gdb) x/8xb ($rbp - 0x10)
0x7fffffffdd60: 0x5c    0xdd    0xff    0xff    0xff    0x7f    0x00    0x00
```

**Little-endian architecture: “reversed” values in memory**

# C

- Functions call

```
struct a {
    int m1;
};

struct a v1;

f ( &v1 );

void f ( struct a *arg1 ) {
    arg1->m1 = 0;
}
```

Are parameters passed by copy or reference?

# C

- Functions call

```
struct a {
    int m1;
};

struct a v1;

f ( &v1 );

void f ( struct a *arg1 ) {
    arg1->m1 = 0;
}
```

Are parameters passed by copy or reference?

**In C, by copy only** ✓

# C

- Functions call

```
void f1 ( struct a arg1 );

struct a f2 ( void );

void f3 ( char arg1[] );

char[] f4 ( void );

char* f5 ( char* arg1 );
```

**Is it valid?**

# C

- Functions call

```
void f1 ( struct a arg1 );  ✓

struct a f2 ( void );  ✓

void f3 ( char arg1[] );  ✓

char[] f4 ( void );  ✗

char* f5 ( char* arg1 );  ✓
```

# Lab

## Exercise 0.1

- Create a program in user space that prints “hello world” to *stdout*
  - Link to master *glibc*
- Debug *printf* (*glibc*) function
- Debug *sys_write* syscall (kernel)

# Lab

## Exercise 0.2

- Create a bytecodes (Java) interpreter in C that supports the following instruction families:
  - iconst, istore, iload, bipush, iinc, dup, iand, ixor, ior, ineg, irem, idiv, iadd, imul, isub, pop, nop, swap
- The interpreter receives a sequence of hex bytecodes by parameter (argv[1])
- Executable binary name: bytecode_interpreter
- Example: ./bytecode_interpreter 043C053D1B1C60…

# Lab

### Exercise 0.2

- Validate input sequences and return: -1 in case of error, 0 in case of success
  - Valid instructions
  - Stack has to be empty at the end of the execution
  - Do not use uninitialized variables
  - Instructions must have enough operands in stack
  - Stack size <= 100
  - Sequence length <= 200
  - 5 local variables maximum
  - Division by 0 not allowed
  - Other checks?

# Lab

## Exercise 0.2

- Print bytecodes assembly to *stdout* when compiled in “debug” mode (#ifdef DEBUG). I.e.:

```
0: iconst_1
1: istore_1
2: iconst_2
3: istore_2
4: iload_1
5: iload_2
6: iadd
7: istore_3
```

# Lab

## Exercise 0.2

- Print local variables value to *stdout* at the end of execution. Represent with “N” character uninitialized variables. I.e.:

```
0:150,1:90,2:12,3:9,4:N,5:N
```

# Lab

## Exercise 0.2

- Create a script with unit test cases that has both valid and invalid sequences. Call the interpreter and assert in *stdout* both 1) return code and, 2) local variables
- Share unit test cases with your colleagues

# References

- Secure Coding in C and C++
(2nd Edition, 2013) – Robert C. Seacord
- The C Programming Language
  - Dennis Ritchie & Brian Kernighan