TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

IMT Grand Est

UNIVERSITÉ DE LORRAINE

C

Introduction to C Programming

2022-2023

Gérald Oster <gerald.oster@telecomnancy.eu>

https://arche.univ-lorraine.fr/course/view.php?id=55190

Course Organization

- 3 × dedicated lectures [11/1, ~~6/2 and 13/3~~]
- 5 × 4-hours labs dedicated labs [16/1, 23/1, 6/2, 27/2 and 13/3]
- 1 × TD [7/2]
- 4 × 4-hours labs on data-structures (SD) will use C programming [27/3, 11/4, 2/5, 9/5]

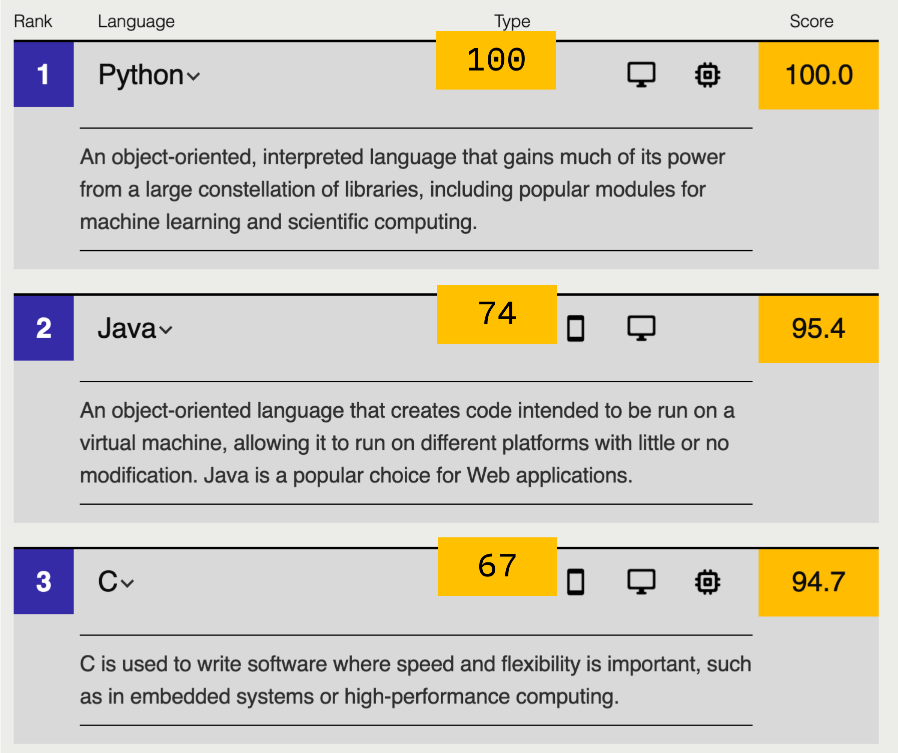
- **Significant amount of personal work**
- Evaluations (continuous assessment):
  - dedicated tests
  - evaluated 4-hours lab
  - exam

$$\text{Final Mark} = \frac{\mathrm{T} + 2\times\mathrm{L} + 2\times\mathrm{E}}{5}$$

Course Objectives

- Familiarize yourself with C programming language
- Understand how a program is run by a computer
- Learn and apply some best development practices
- Understand memory management with pointers
- Learn to how to use some development tools (compiler, debugger, make, …)

IEEE Top Programming Language 2022



Rank	Language	Type	Score
1	Python⌄	100 🖥 ⚙	100.0
	An object-oriented, interpreted language that gains much of its power from a large constellation of libraries, including popular modules for machine learning and scientific computing.		
2	Java⌄	74 📱 🖥	95.4
	An object-oriented language that creates code intended to be run on a virtual machine, allowing it to run on different platforms with little or no modification. Java is a popular choice for Web applications.		
3	C⌄	67 📱 🖥 ⚙	94.7
	C is used to write software where speed and flexibility is important, such as in embedded systems or high-performance computing.		

https://spectrum.ieee.org/top-programming-languages/

Resources

- ***The C Programming Language Book***, Brian W. Kernighan and Dennis M. Ritchie, 2nd Edition (1988)
- ***Programmation en Langage C***, Anne Canteaut (2008) https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/
- ***Modern C***, Jens Gustedt (2019) https://gustedt.gitlabpages.inria.fr/modern-c/
- *Effective C : An Introduction to Professional C Programming*, Robert C. Seacord (2020)
- *21st Century C*, Ben Klemens, 2nd Edition (2014)

Features of C Programming Language

- C is an imperative (procedural) programming language
- C is a compiled programming language
- Fast and Efficient
- A correct C program is portable between different platforms
- Mid-level programming language
- Statically type
- Dynamic memory management
- And many more…

History

- Designed and build by Dennis Ritchie at Bell Labs (AT&T) in 1972
- Used to build UNIX operating system and tools by Dennis Ritchie and Ken Thompson around 1972-1973

Year	C Standard
1972	Birth
1978	K&R C
1989/1990	ANSI C and ISO C
1999	C99
2011	C11
2017	C17
TBD	C2x

Timeline of C language development

First C Program

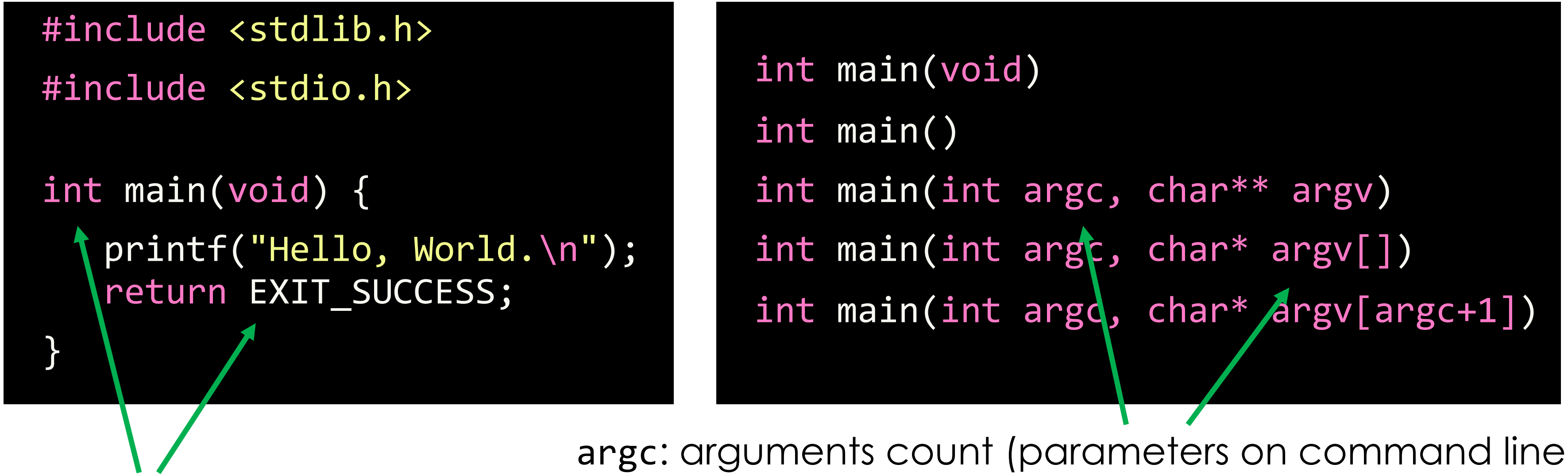
```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
   printf("Hello, World.\n");
   return EXIT_SUCCESS;
}
```

```
$ clang -std=c99 -Wall helloworld.c -o helloworld
$ ./helloworld
Hello, World.
```

C Language: Basics

- Main function (program entry point)

Other allowed definitions:



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

int main(void) {
   printf("Hello, World.\n");
   return EXIT_SUCCESS;
}
```

```c
int main(void)
int main()
int main(int argc, char** argv)
int main(int argc, char* argv[])
int main(int argc, char* argv[argc+1])
```

`argc`: arguments count (parameters on command line)
`argv`: array of « strings » - `argv[0]` == run command

return code = exit status
Do you recall what you have seen in **COLD course** ?
`EXIT_SUCCESS` defined in `stdlib.h`

C Language: Data Types

- Data types:

```
[signed] char
[unsigned|signed] short
[unsigned|signed] int
[unsigned|signed] long
[unsigned|signed] long long
float
double
long double
```

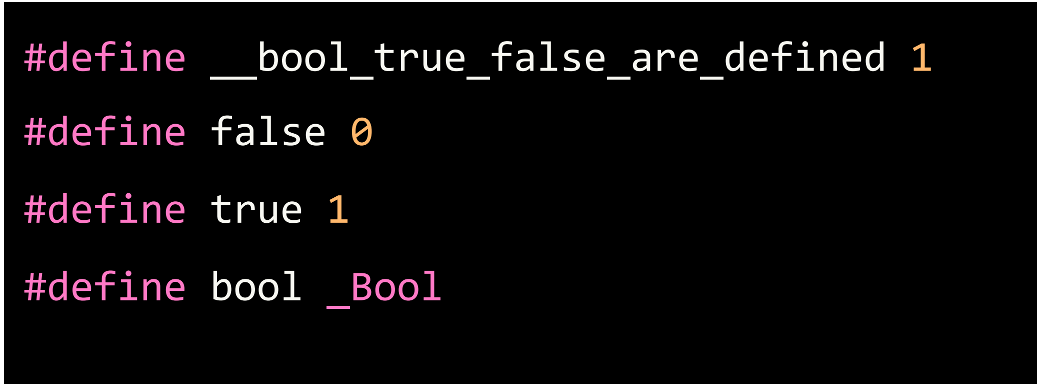
- Some **narrow** types are promoted (to `signed int`) before doing arithmetic
- There sizes in memory differ and their interval.

  If you want to be sure of their size, include `stdint.h` and use `int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t`

Take-away Message: Do not try to do **premature optimization!**

C Language: Booleans

- Please note, there is no Boolean datatype
- In C, `0` represents logical false, any other value (non null) is interpreted as logical true
- one may include `stdbool.h` which defines 4 macros:



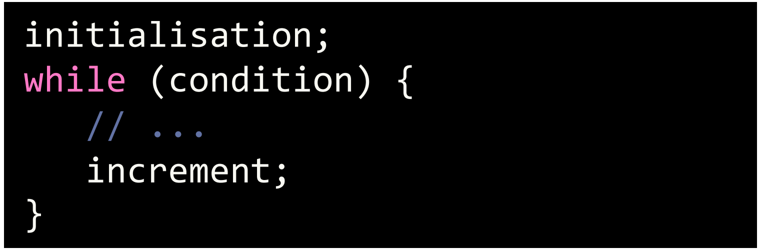
```
#define __bool_true_false_are_defined 1

#define false 0

#define true 1

#define bool _Bool
```

C Language: Structures of Control

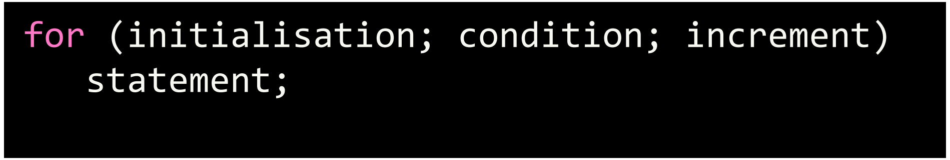
```
if (some_condition)
    statement
else if (some_other_condition)
    statement
else
    statement
```

```
switch (expression) {
    case constant_expression1:
        statement;
        [[ break; ]]
    case constant_expression2:
        statement;
        [[ break; ]]
    ...
    default:
        statement;
        ...
}
```

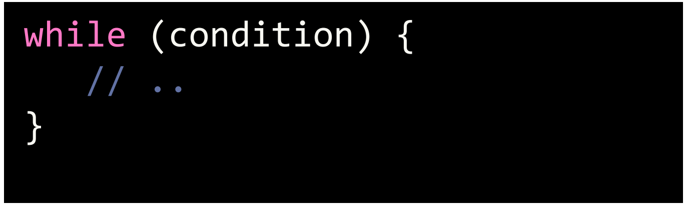
C Language: Structures of Control /2



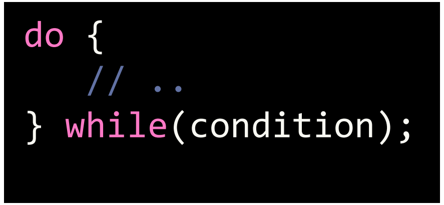
```c
initialisation;
while (condition) {
    // ...
    increment;
}
```



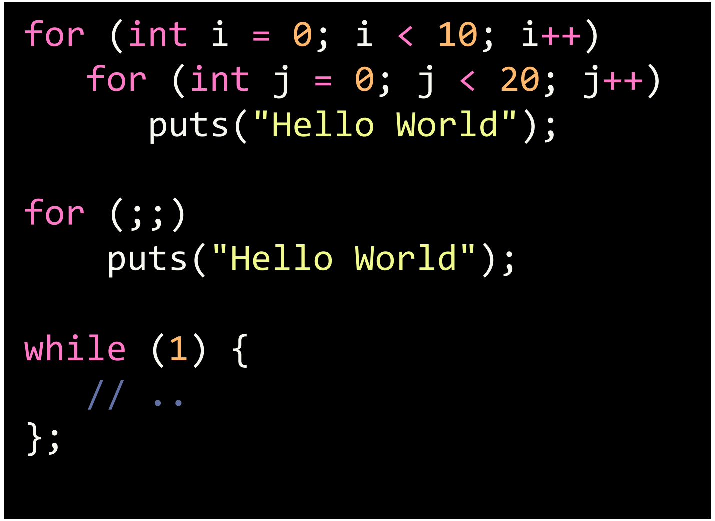
```c
for (initialisation; condition; increment)
    statement;
```



```c
while (condition) {
    // ..
}
```



```c
do {
    // ..
} while(condition);
```



```c
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 20; j++)
        puts("Hello World");

for (;;)
     puts("Hello World");

while (1) {
    // ..
};
```

C Language: Arrays

- Elements are store contiguously in memory
- Array indices start at 0
- Declaration:

```
TypeName array_name[size];
```

```
int values[10];
char message[8];
```

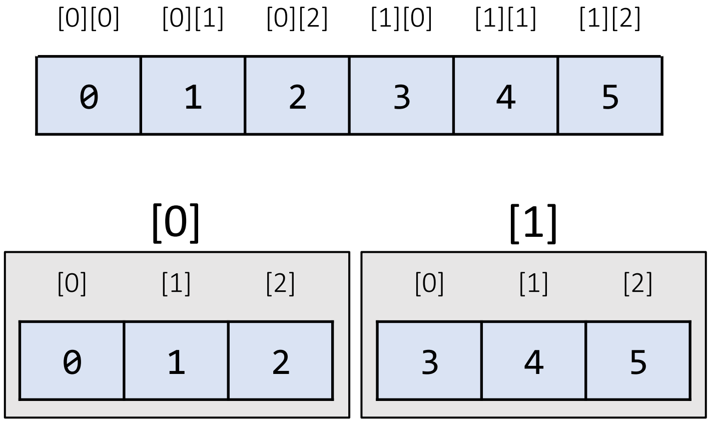
- Initialization:

```
int values[5] = { 0 }; // zero-initialization

int tab[8] = { 1, 2, 3 }; // initialize 3 first elements
                             to the provided values,
                             0 for other values
int tab[] = { 1, 2, 3, 4 }; // size is computed at compilation
```

C Language: Multi-Dimensional Arrays

```
int tab[2][3] = { { 0, 1, 2 }, { 3, 4, 5} };
```

Memory Representations



[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]
0	1	2	3	4	5

[0]

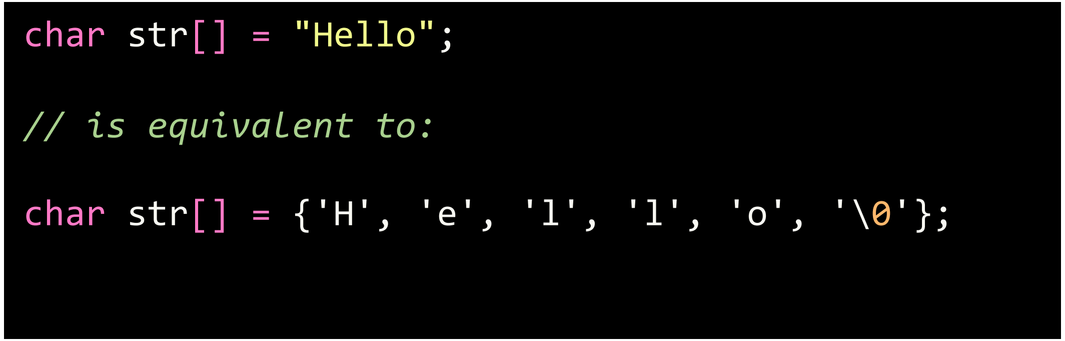
[0]	[1]	[2]
0	1	2

[1]

[0]	[1]	[2]
3	4	5

C Language: Characters Strings

- There is no string **datatypes** in C
- String are represented as array of characters (`char`) terminated with `nul` character (‘\0’)



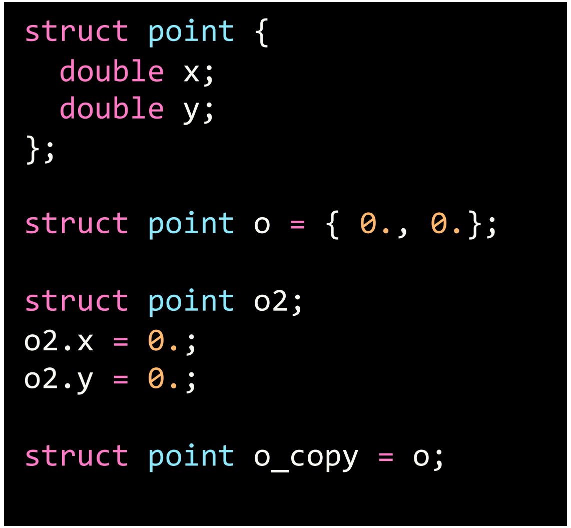
```
char str[] = "Hello";

// is equivalent to:

char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Memory Representation

[0]	[1]	[2]	[3]	[4]	[5]
H	e	l	l	o	\0

Structures

- **User defined Type** as a **composition of finite** (typed) fields in a **contiguous** block of memory



```
struct point {
  double x;
  double y;
};

struct point o = { 0., 0.};

struct point o2;
o2.x = 0.;
o2.y = 0.;

struct point o_copy = o;
```

Definition of a new data structure named **`struct point`** composed of 2 fields.

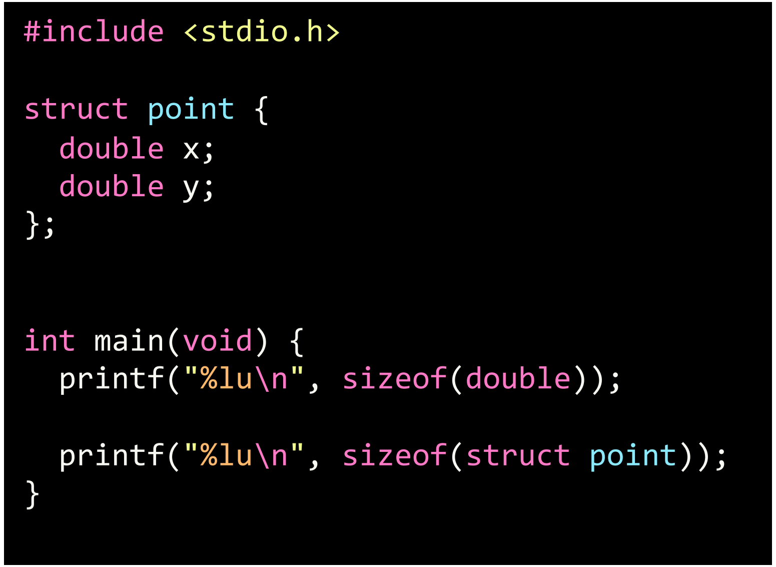
Initialization of a variable whose type is **`struct point`**

Another kind of initialization (field by field)

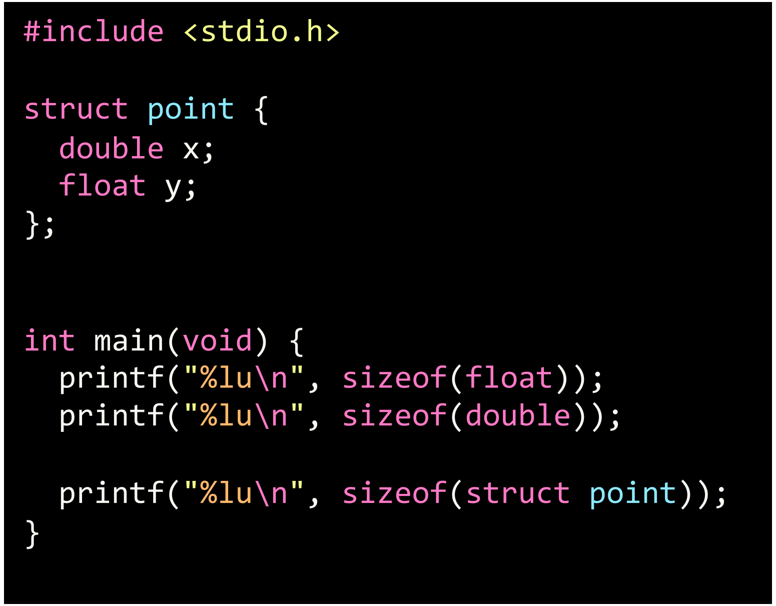
= is used to copy the value
therefore, **o** and **o_copy** are two distinct structures

Structures

- Use **`sizeof()`** to get the size of a data structure



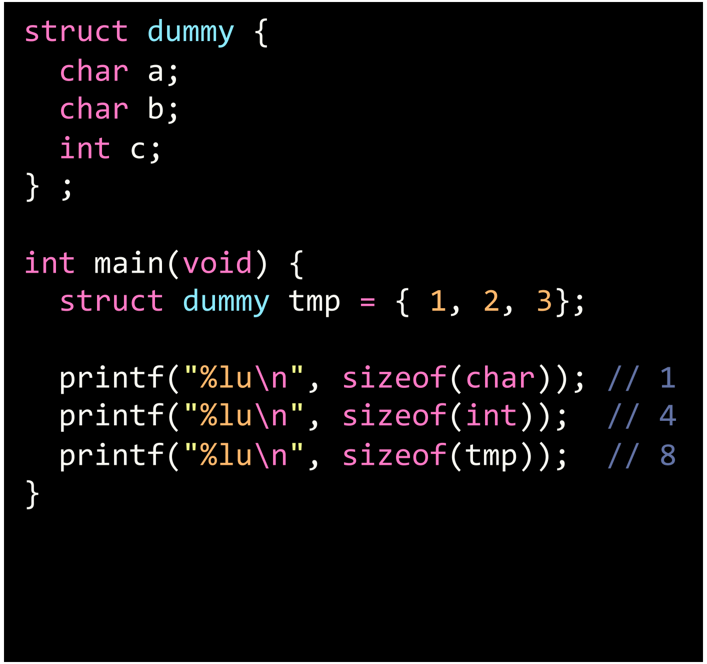
```c
#include <stdio.h>

struct point {
  double x;
  double y;
};


int main(void) {
  printf("%lu\n", sizeof(double));

  printf("%lu\n", sizeof(struct point));
}
```



```c
#include <stdio.h>

struct point {
  double x;
  float y;
};


int main(void) {
  printf("%lu\n", sizeof(float));
  printf("%lu\n", sizeof(double));

  printf("%lu\n", sizeof(struct point));
}
```

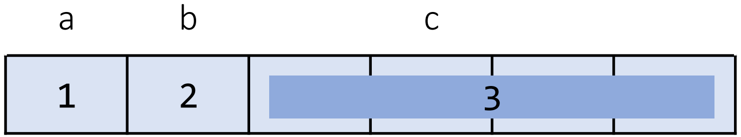
Beware of **structure padding**!

Structures Padding

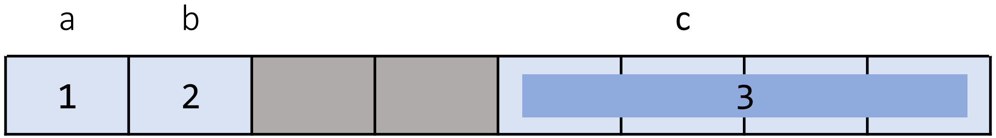


```
struct dummy {
  char a;
  char b;
  int c;
} ;

int main(void) {
  struct dummy tmp = { 1, 2, 3};

  printf("%lu\n", sizeof(char)); // 1
  printf("%lu\n", sizeof(int));  // 4
  printf("%lu\n", sizeof(tmp));  // 8
}
```

expected:



got:



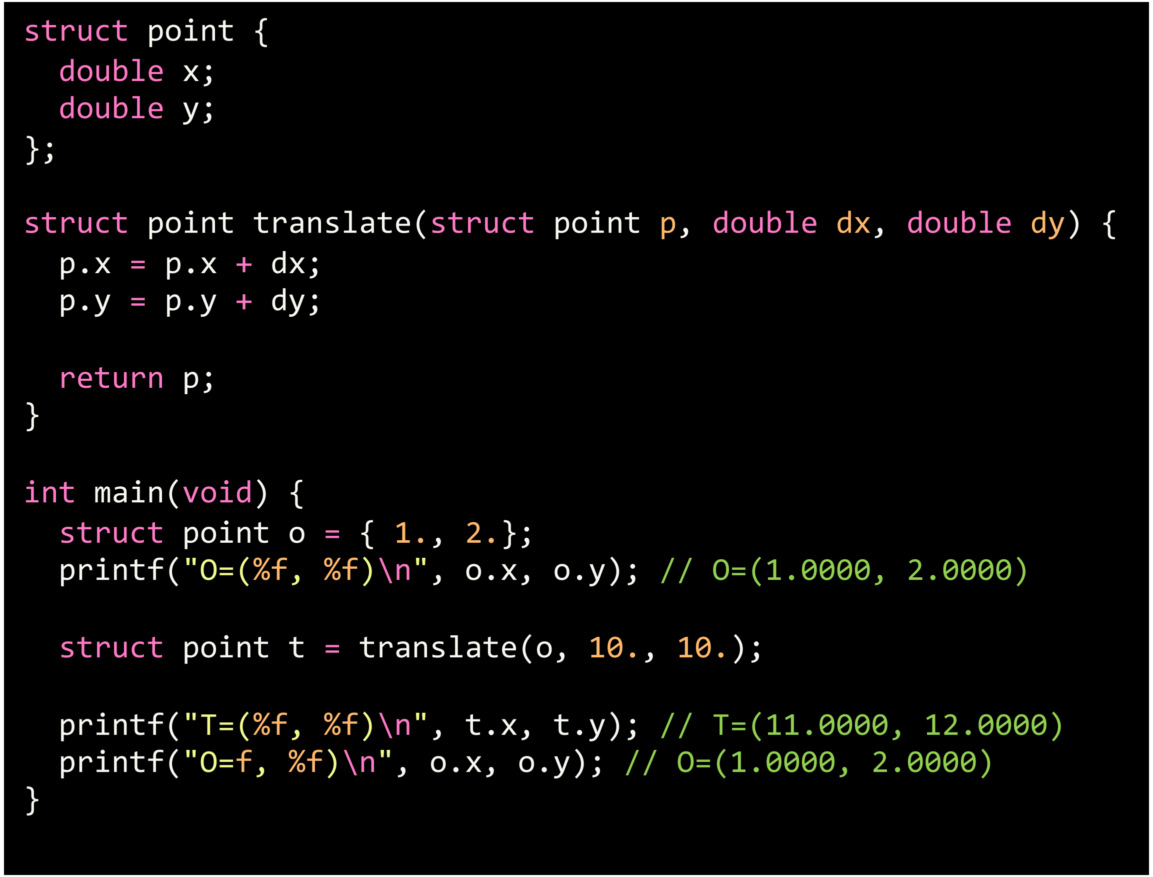
On my architecture

What about `#pragma pack(1)`?

Structures as Parameters/Return value

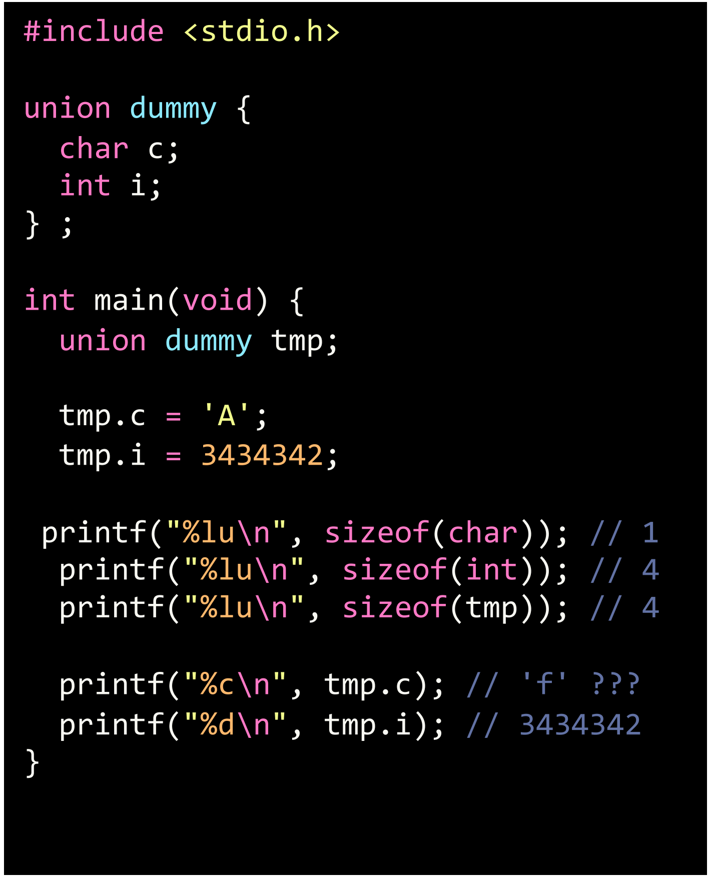
Passing by value:

- means that a copy of the data is made
- and stored by way of the name of the parameter.
- Any changes to the parameter have NO affect on data in the calling function.



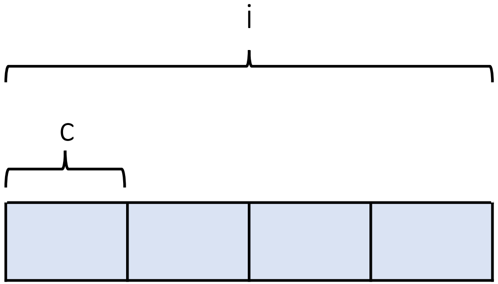
```c
struct point {
  double x;
  double y;
};

struct point translate(struct point p, double dx, double dy) {
  p.x = p.x + dx;
  p.y = p.y + dy;

  return p;
}

int main(void) {
  struct point o = { 1., 2.};
  printf("O=(%f, %f)\n", o.x, o.y); // O=(1.0000, 2.0000)

  struct point t = translate(o, 10., 10.);

  printf("T=(%f, %f)\n", t.x, t.y); // T=(11.0000, 12.0000)
  printf("O=f, %f)\n", o.x, o.y); // O=(1.0000, 2.0000)
}
```

Unions



```c
#include <stdio.h>

union dummy {
  char c;
  int i;
} ;

int main(void) {
  union dummy tmp;

  tmp.c = 'A';
  tmp.i = 3434342;

 printf("%lu\n", sizeof(char)); // 1
  printf("%lu\n", sizeof(int)); // 4
  printf("%lu\n", sizeof(tmp)); // 4

  printf("%c\n", tmp.c); // 'f' ???
  printf("%d\n", tmp.i); // 3434342
}
```

- Fields/members **share** the same memory block



- Size of a union is the size of the largest member

Bit fields, enums, unions

```c
struct my_bit_fields {
  unsigned int flag : 1;
  unsigned int value : 31;
};
```

- Bit fields: explicitly defined how many bits are dedicated to a field.
- The entire struct as the size of an **`int`**

```c
enum Color {
  Red,
  Blue,
  Green
};


enum Color my_color;
my_color = Red;

enum AnotherColor {
  Orange = 4,
  Yellow = 12,
  Green = 4
};
```

- User defined types to assign names to integer constants (names are easier to handle/remember in program)
- Enums vs Macros
  #1 enums can be declared in local scope
  #2 automatically initialized/managed by the compiler
- Enums can be manually initialized
  A **single value** can be mapped to **several names**

Type Alias

- Syntax: **`typedef existing_data_type new_data_type ;`**

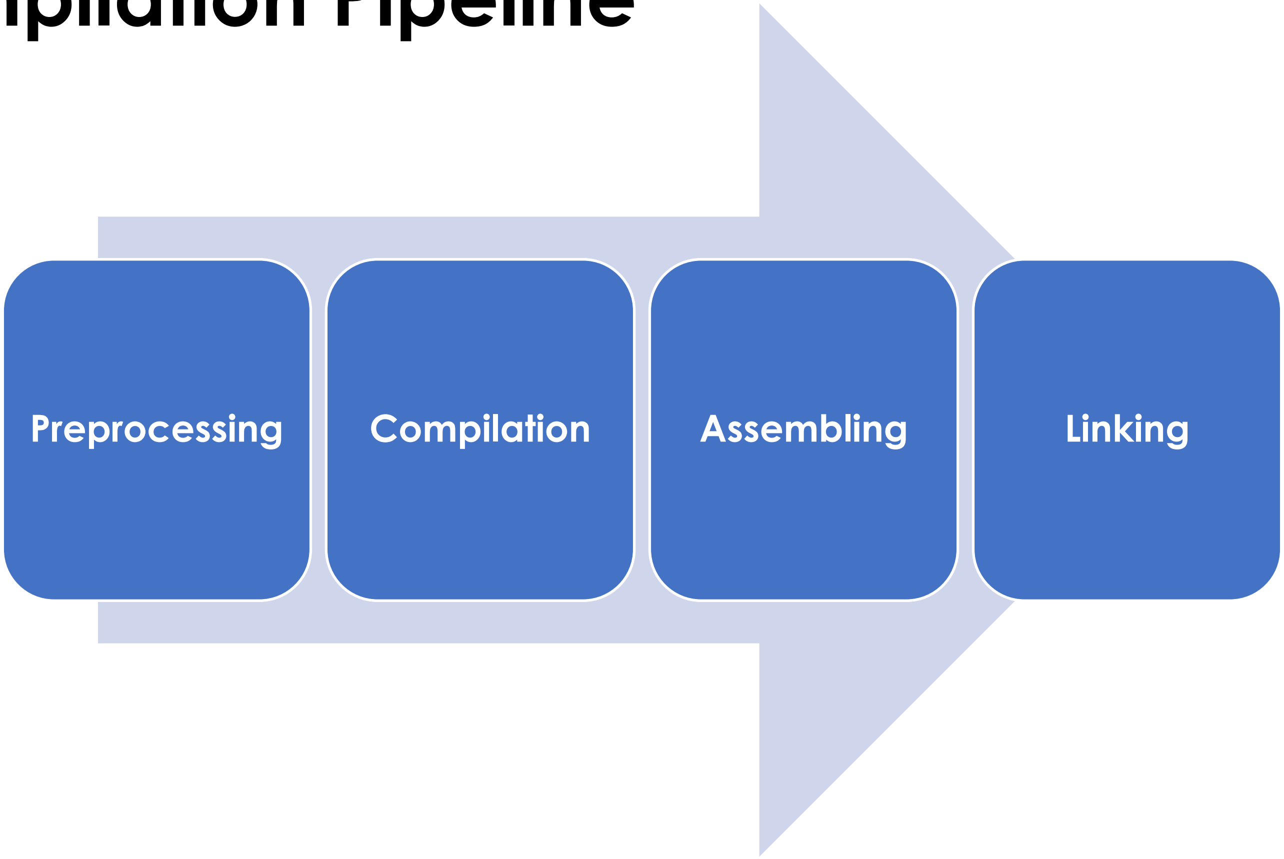
```c
struct _vector
{
  unsigned int values_count;
  int *values;
};

typedef struct _vector vector_t;
```

```c
typedef struct _vector {
  unsigned int values_count;
  int *values;
} vector_t;
```

```c
typedef struct {
  unsigned int values_count;
  int *values;
} vector_t;
```

*Do not care about the * symbol, we will come back later on it* 😱😱😱

Compilation Pipeline



Preprocessor Step

```
#include <stdlib.h>

#include <stdio.h>


int main(void) {

   printf("Hello, World.\n");
   return EXIT_SUCCESS;

}
```

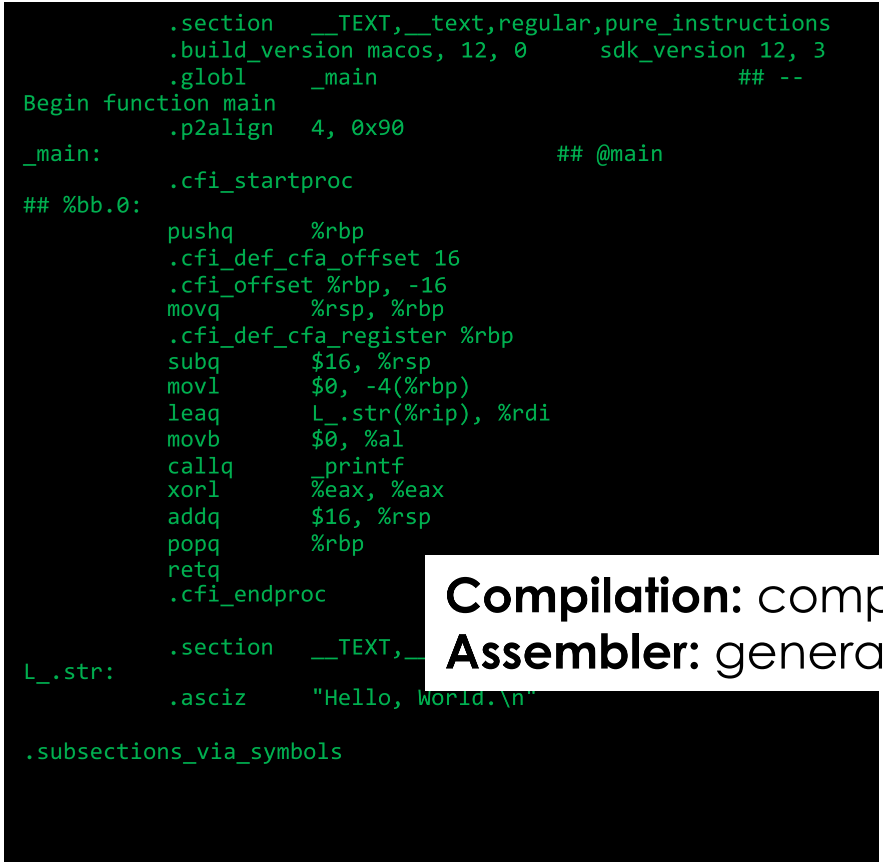
Preprocessing: remove comments, include (header) files, expand macro, etc.

```
$ clang -E helloworld.c -o helloworld.i
```

```
…
int printf(const char * restrict, ...) __attribute__((__format__ (__printf__, 1, 2)));
…
int main(void) {

   printf("Hello, World.\n");
   return 0;

}
```

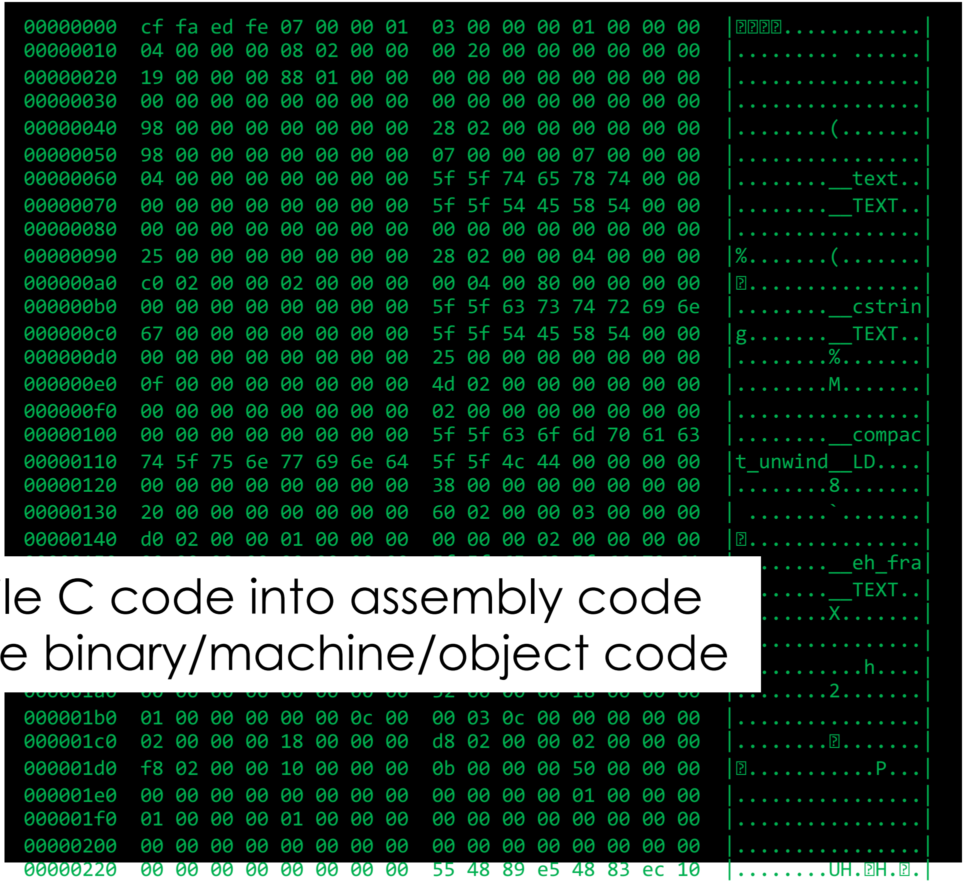
Compilation Step / Assembler Step

```
$ clang -S helloworld.c
```



```
            .section    __TEXT,__text,regular,pure_instructions
            .build_version macos, 12, 0     sdk_version 12, 3
            .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
            movl        $0, -4(%rbp)
            leaq        L_.str(%rip), %rdi
            movb        $0, %al
            callq       _printf
            xorl        %eax, %eax
            addq        $16, %rsp
            popq        %rbp
            retq
            .cfi_endproc

            .section    __TEXT,__
L_.str:
            .asciz      "Hello, World.\n"

.subsections_via_symbols
```

```
$ clang -C helloworld.c
```



```
00000000  cf fa ed fe 07 00 00 01  03 00 00 00 01 00 00 00  |����............|
00000010  04 00 00 00 08 02 00 00  00 20 00 00 00 00 00 00  |.........  ......|
00000020  19 00 00 00 88 01 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  98 00 00 00 00 00 00 00  28 02 00 00 00 00 00 00  |........(.......|
00000050  98 00 00 00 00 00 00 00  07 00 00 00 07 00 00 00  |................|
00000060  04 00 00 00 00 00 00 00  5f 5f 74 65 78 74 00 00  |........__text..|
00000070  00 00 00 00 00 00 00 00  5f 5f 54 45 58 54 00 00  |........__TEXT..|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000090  25 00 00 00 00 00 00 00  28 02 00 00 04 00 00 00  |%.......(.......|
000000a0  c0 02 00 00 02 00 00 00  00 04 00 80 00 00 00 00  |�...............|
000000b0  00 00 00 00 00 00 00 00  5f 5f 63 73 74 72 69 6e  |........__cstrin|
000000c0  67 00 00 00 00 00 00 00  5f 5f 54 45 58 54 00 00  |g.......__TEXT..|
000000d0  00 00 00 00 00 00 00 00  25 00 00 00 00 00 00 00  |........%.......|
000000e0  0f 00 00 00 00 00 00 00  4d 02 00 00 00 00 00 00  |........M.......|
000000f0  00 00 00 00 00 00 00 00  02 00 00 00 00 00 00 00  |................|
00000100  00 00 00 00 00 00 00 00  5f 5f 63 6f 6d 70 61 63  |........__compac|
00000110  74 5f 75 6e 77 69 6e 64  5f 5f 4c 44 00 00 00 00  |t_unwind__LD....|
00000120  00 00 00 00 00 00 00 00  38 00 00 00 00 00 00 00  |........8.......|
00000130  20 00 00 00 00 00 00 00  60 02 00 00 03 00 00 00  | .......`.......|
00000140  d0 02 00 00 01 00 00 00  00 00 00 02 00 00 00 00  |�...............|
                                                              .......__eh_fra|
                                                              .......__TEXT..|
                                                              .......X.......|
                                                              ...............|
                                                              ...........h....|
                                                            |........2.......|
000001b0  01 00 00 00 00 00 0c 00  00 03 0c 00 00 00 00 00  |................|
000001c0  02 00 00 00 18 00 00 00  d8 02 00 00 02 00 00 00  |........�.......|
000001d0  f8 02 00 00 10 00 00 00  0b 00 00 00 50 00 00 00  |�...........P...|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
000001f0  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000220  00 00 00 00 00 00 00 00  55 48 89 e5 48 83 ec 10  |........UH.�H.�.|
```

Compilation: compile C code into assembly code

Assembler: generate binary/machine/object code

Linking Step

- `$ clang helloworld.o -o helloworld`

Linking: merge/assemble binary code, function locations are resolved

- Static or dynamic linking with shared libraries also

Compilation Pipeline

```
$ clang -E helloworld.c -o helloworld.i
```

`helloworld.i` – Produced by **Preprocessor step**

```
$ clang -S helloworld.i -o helloworld.s
```

`helloworld.s` – Produced by **Compiler**

```
$ clang -c helloworld.s -o helloworld.o
```

`helloworld.o` – Produced by **Assembler**

```
$ clang helloworld.o -o helloworld
```

`helloworld.{|out|exe}` (Linux/macOS/Windows)
Executable file – Produced by **Linking step**

More on this:
https://hackthedeveloper.com/c-program-compilation-process/

Preprocessing Instructions

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

File inclusion

```c
#define PI 3.141
#define Answer_to_the_ultimate_question_of_Life 42

#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
```

Macros

```c
#ifndef __SEEN__
#define __SEEN__
…
#else
…
#endif /* __SEEN__ */
```

Conditional compilation

Modular Programming / Headers

- Split your code in several `.c` files
- For each `.c` file write a corresponding (header) `.h` file
- In the `.h` file, write the function profiles (usually also the structure definitions) you want “to export”
- Include the `.h` file in your `.c` files that use these functions
- Protect your `.h` file from multiple inclusion using `#ifndef` (or `#pragma once`) preprocessing instruction

Modular Programming /2

functions.c

```c
#include "functions.h"

int multiply(int x, int y) {
  return x * y;
}
```

functions.h

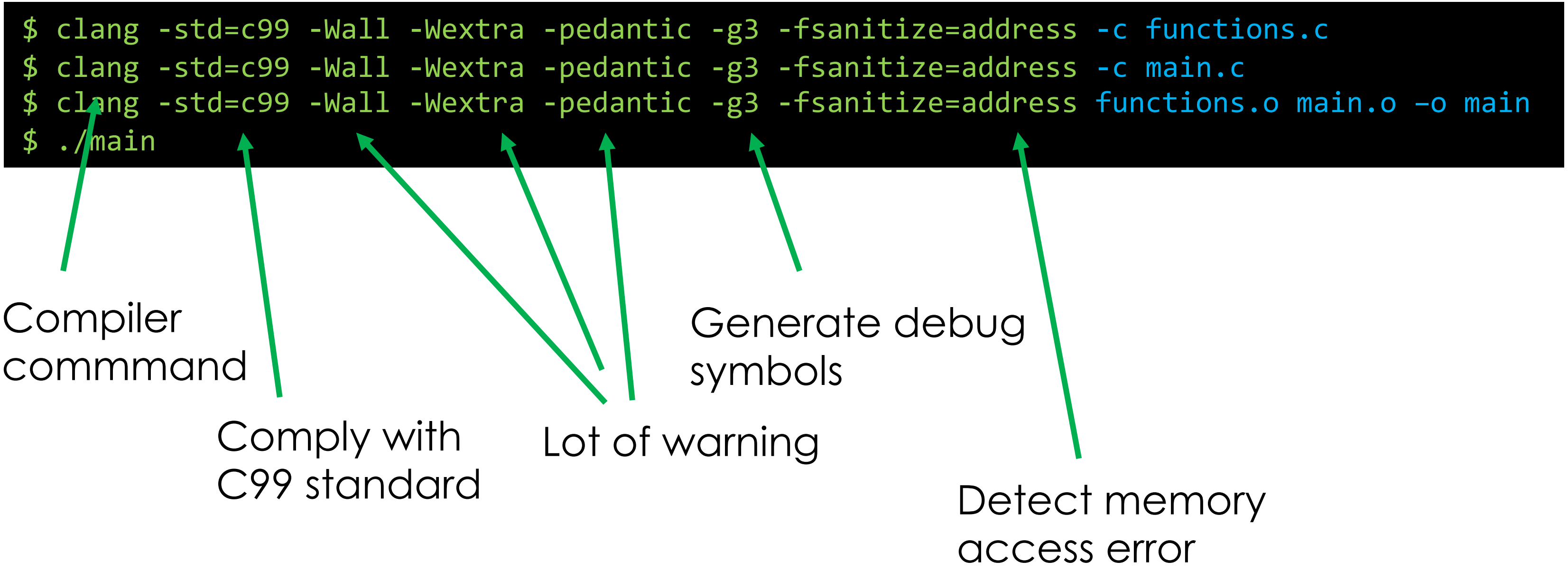
```c

#ifndef __FUNCTIONS_H__
#define __FUNCTIONS_H__

int multiply(int x, int y);

#endif /* __FUNCTIONS_H__ */
```

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "functions.h"

int main(void) {
  int a = 7;
  int b = 6;

  int r = multiply(a, b);
  printf("%d x %d = %d \n", a, b, r);

  return EXIT_SUCCESS;
}
```

```
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c functions.c
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c main.c
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address functions.o main.o –o main
$ ./main
```

Some Compilation Parameters



```
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c functions.c
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c main.c
$ clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address functions.o main.o –o main
$ ./main
```

Compiler commmand

Comply with C99 standard

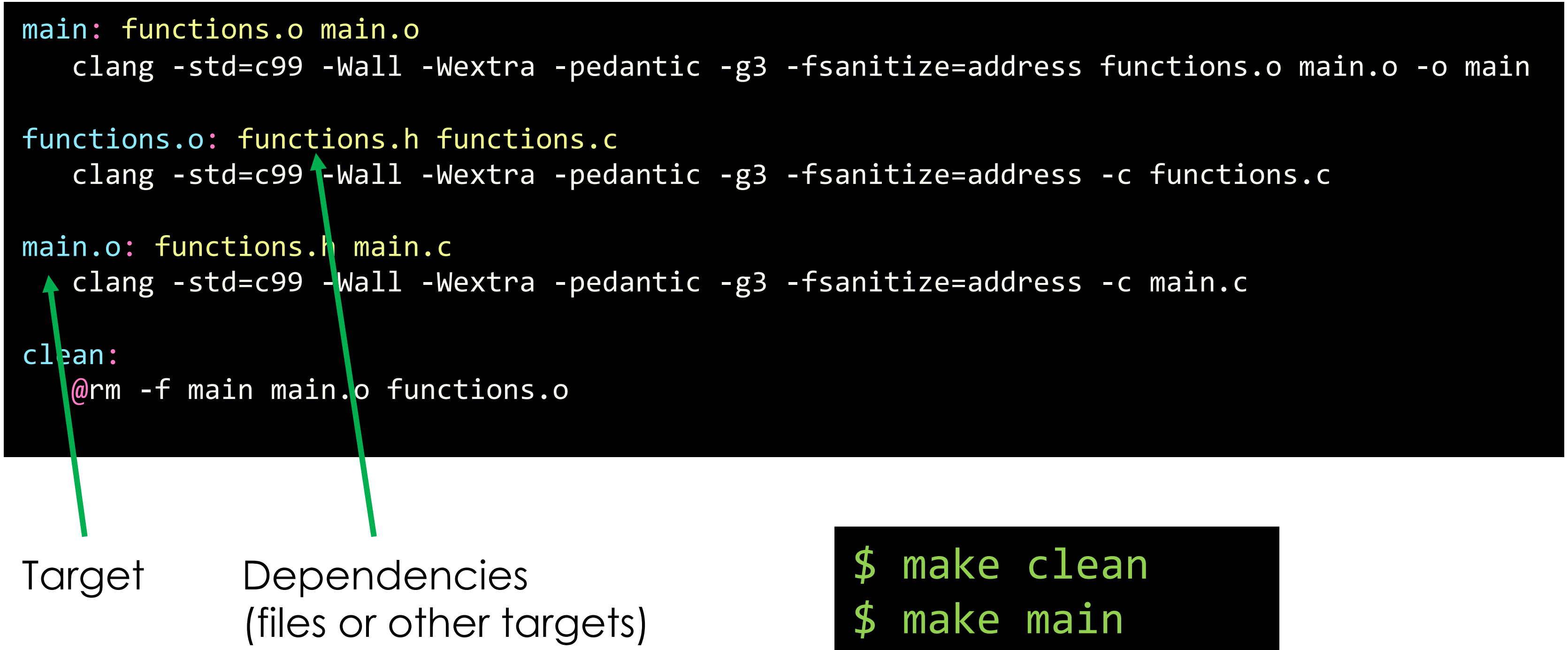
Lot of warning

Generate debug symbols

Detect memory access error

Makefile (in one slide!)

makefile



```
main: functions.o main.o
	clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address functions.o main.o -o main

functions.o: functions.h functions.c
	clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c functions.c

main.o: functions.h main.c
	clang -std=c99 -Wall -Wextra -pedantic -g3 -fsanitize=address -c main.c

clean:
	@rm -f main main.o functions.o
```

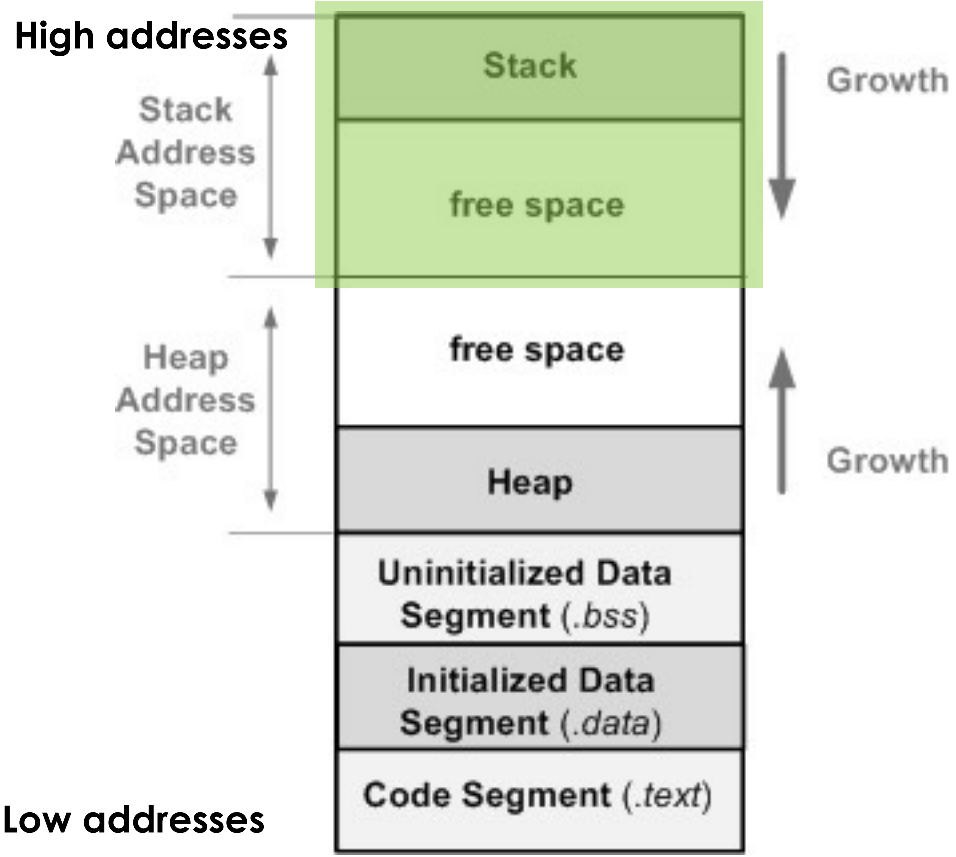
Target

Dependencies
(files or other targets)

```
$ make clean
$ make main
$ make
```

Break (short)

Memory Layout



Code Segment (.code): executable program code (instructions) and variable with const qualifier

Initialized Data Segment (.data): global variables (static / extern) and local static variables that have been initialized with initial values different than zero

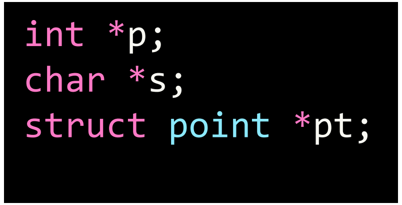
Uninitialized Data Segment (.bss): global variables that are initialized to zero or with no initialization

Heap: Segment of memory that provides dynamic memory allocation

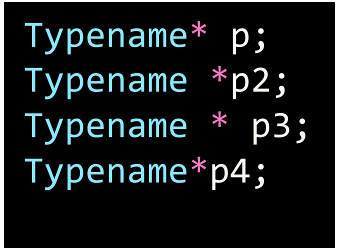
Stack: Segment of memory for local variables, function parameters, return values, etc. Managed as LIFO structure.

Pointers

- A pointer is a variable that stores a (valid or non-valid) memory address
- The size of a pointer depends on the architecture
  - 64 bits -> address of 8 bytes ; 32 bits -> address of 4 bits
- Syntax:



```c
int *p;
char *s;
struct point *pt;
```



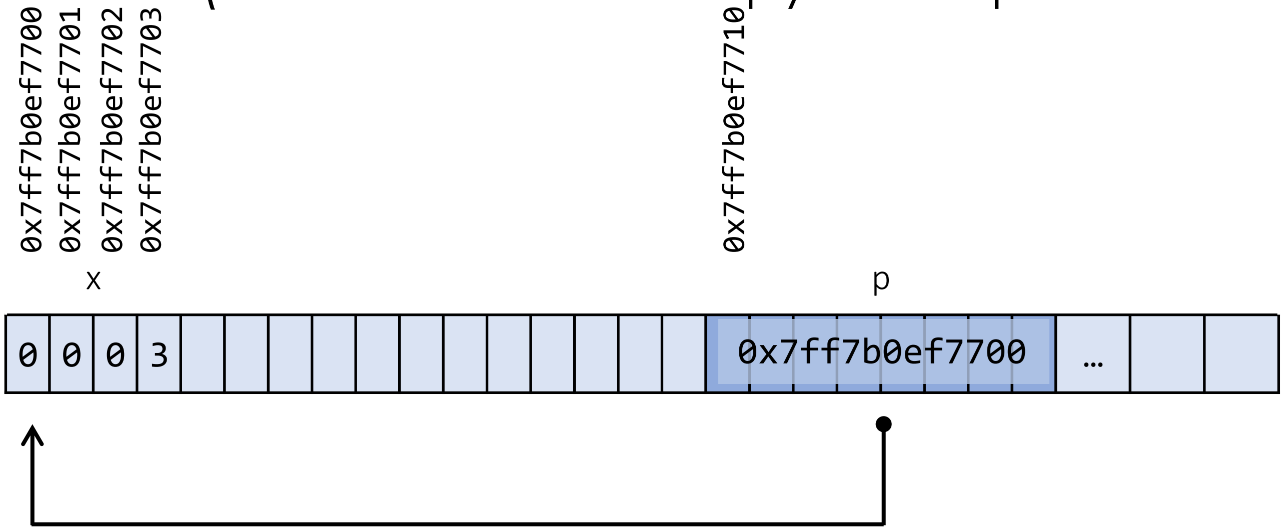
```c
Typename* p;
Typename *p2;
Typename * p3;
Typename*p4;
```

All these definitions are equivalent

Pointers

& : address of operator
***** : dereferencing of a pointer – value located at the address
= : copy the address (does not make a copy of the pointed value)

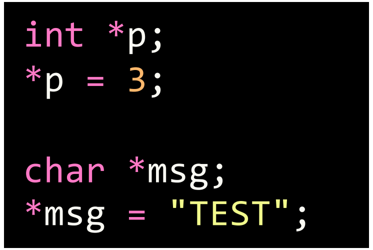
```
int x = 3;
int *p;

p = &x;


int z = *p;
int *ptr;

ptr = p;
```



Be ware: **&t = t,** if **t** is an array (so for **int t[10];** we have **t == &t == &t[0]**)

Pointers

- The following code is wrong!
- Do you know why?

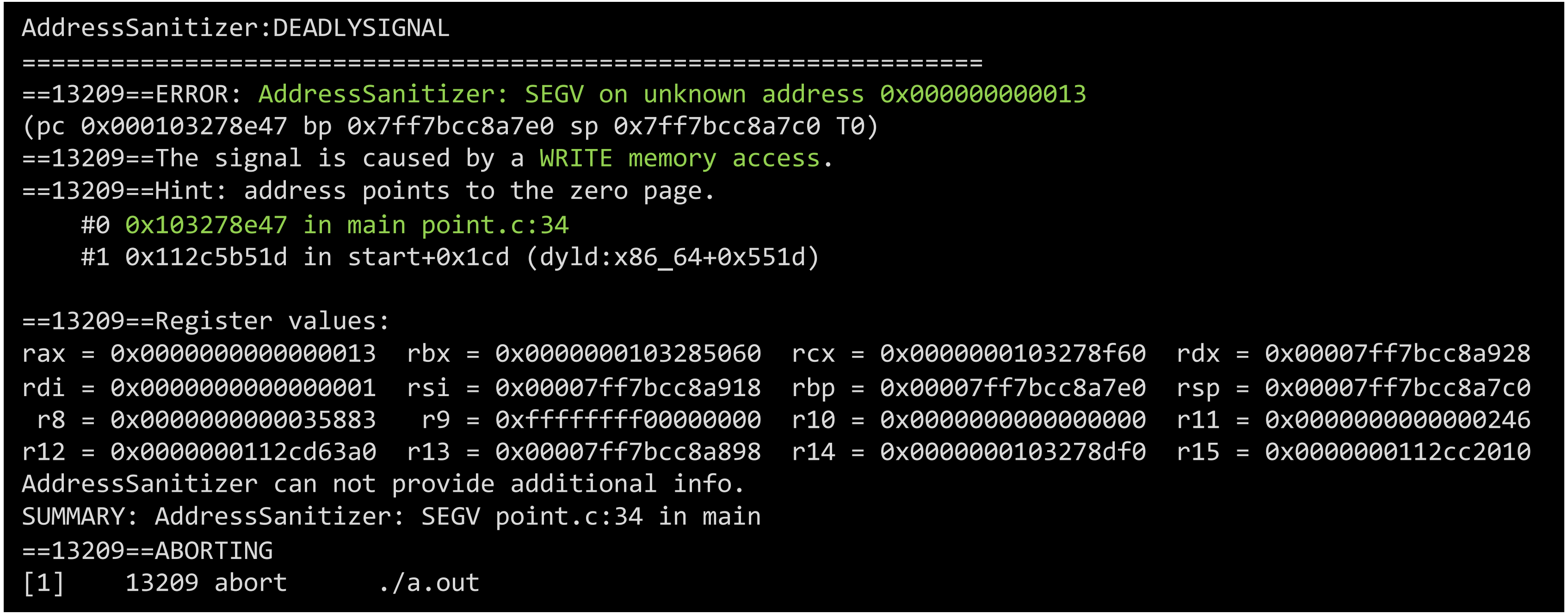


```c
int *p;
*p = 3;

char *msg;
*msg = "TEST";
```

```
$ ./a.out
[1]    13125 segmentation fault  ./a.out
```

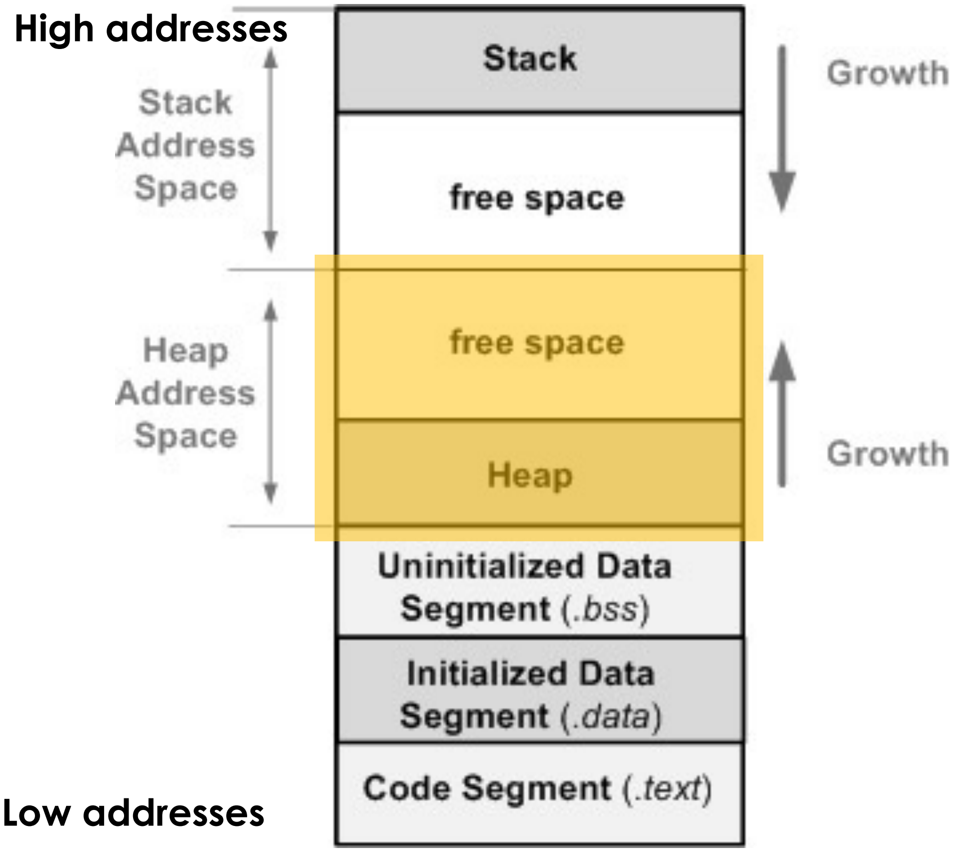
Pointers

- Or compiled with AddressSanitizer (**`-fsanitize=address`**)



```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==13209==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000013
(pc 0x000103278e47 bp 0x7ff7bcc8a7e0 sp 0x7ff7bcc8a7c0 T0)
==13209==The signal is caused by a WRITE memory access.
==13209==Hint: address points to the zero page.
    #0 0x103278e47 in main point.c:34
    #1 0x112c5b51d in start+0x1cd (dyld:x86_64+0x551d)

==13209==Register values:
rax = 0x0000000000000013  rbx = 0x0000000103285060  rcx = 0x0000000103278f60  rdx = 0x00007ff7bcc8a928
rdi = 0x0000000000000001  rsi = 0x00007ff7bcc8a918  rbp = 0x00007ff7bcc8a7e0  rsp = 0x00007ff7bcc8a7c0
 r8 = 0x0000000000035883   r9 = 0xffffffff00000000  r10 = 0x0000000000000000  r11 = 0x0000000000000246
r12 = 0x0000000112cd63a0  r13 = 0x00007ff7bcc8a898  r14 = 0x0000000103278df0  r15 = 0x0000000112cc2010
AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV point.c:34 in main
==13209==ABORTING
[1]    13209 abort      ./a.out
```

Memory Layout



Code Segment (.code): executable program code (instructions) and variable with const qualifier

Initialized Data Segment (.data): global variables (static / extern) and local static variables that have been initialized with initial values different than zero

Uninitialized Data Segment (.bss): global variables that are initialized to zero or with no initialization

Heap: Segment of memory that provides dynamic memory allocation

Stack: Segment of memory for local variables, function parameters, return values, etc. Managed as LIFO structure.

Dynamic Memory Allocation

- **`void * malloc(size_t size)`**: to allocate memory on the **heap**
- **`void free(void *ptr)`**: to free up previously allocated memory
- **`void * calloc(size_t count, size_t size)`**: same as **`malloc()`**, but memory block is filled with 0
- **`void * realloc(void *ptr, size_t size)`**: to change the size (or reallocate) a previously allocated block of memory

```c
char *msg = malloc(4 * sizeof(char));

msg[0] = 'A';
msg[1] = 'B';
msg[2] = 'C';
msg[3] = '\0';
printf("%s\n", msg);

free(msg); // do not forget
```

```c
char *msg = calloc(4, sizeof(char));

msg[0] = 'A';
msg[1] = 'B';
msg[2] = 'C’;
printf("%s\n", msg);

free(msg); // do not forget
```

Pointers Arithmetic

- As pointers contain numeric values (memory addresses), one can make computation with the following arithmetic operators: **+**, **-**, **++** and **--**.
- With **ptr** of type **Typename**, the expression **ptr = ptr+step** adds **step*sizeof(Typename)** to **ptr**

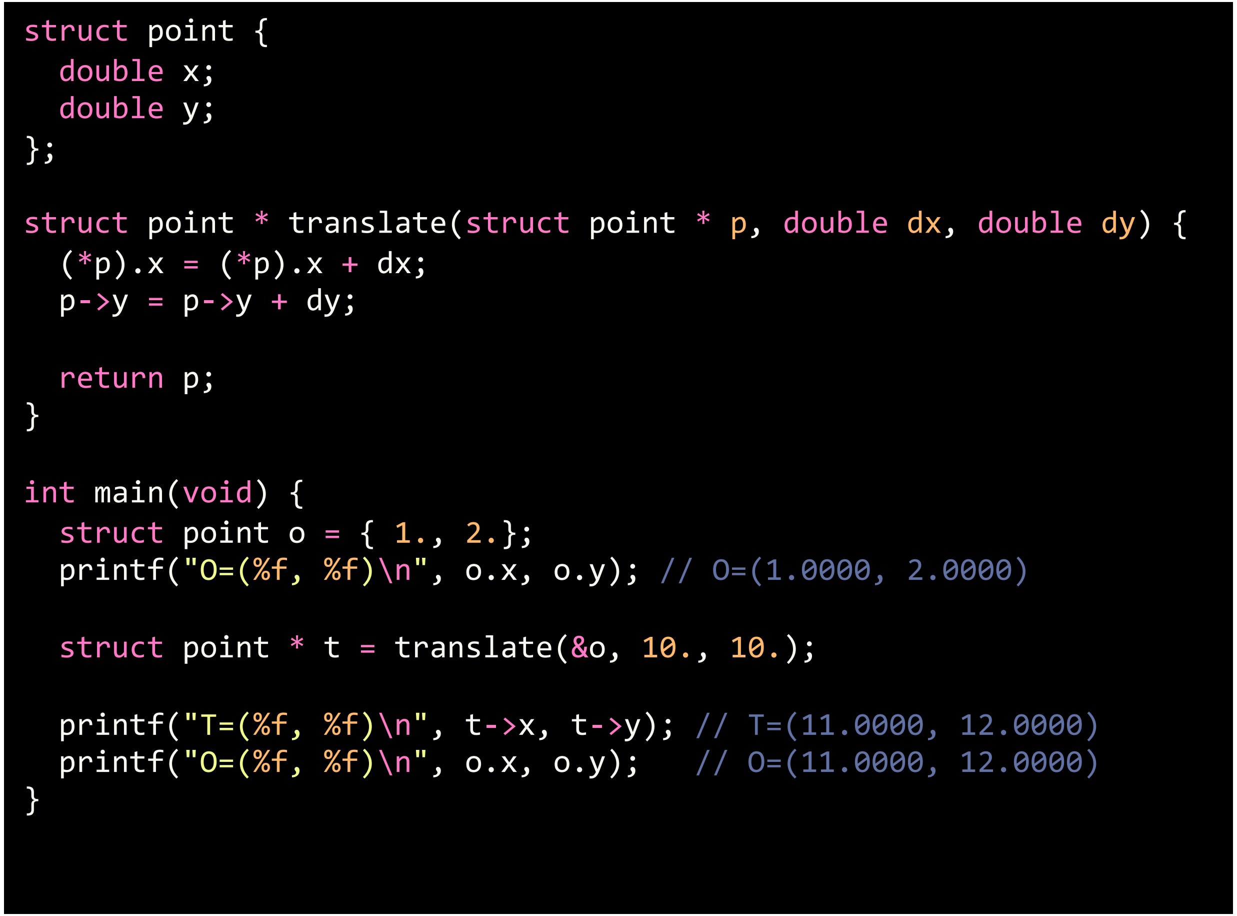
```
int tab[5] = { 0, 1, 2, 3, 4 } ;
int *ptr = tab;
ptr = ptr + 2;

printf("%d\n", *ptr); // prints 2
```

Structures as Parameters/Return value /2

Passing by reference:

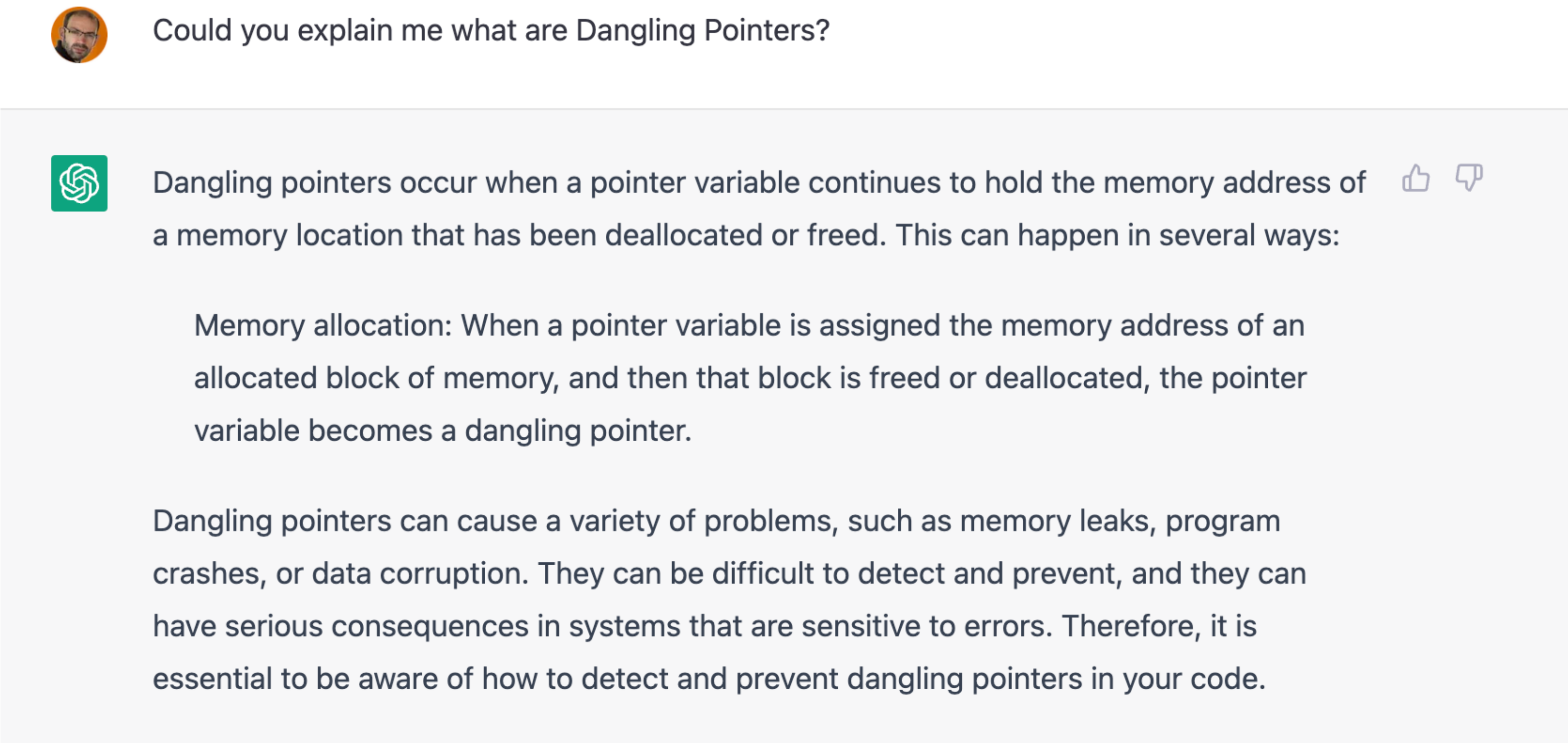
- A reference parameter « refers » to the original data in the calling function
- Any changes to the parameter affect the original data in the calling function.

- Note: arrays are always passed by references



```c
struct point {
  double x;
  double y;
};

struct point * translate(struct point * p, double dx, double dy) {
  (*p).x = (*p).x + dx;
  p->y = p->y + dy;

  return p;
}

int main(void) {
  struct point o = { 1., 2.};
  printf("O=(%f, %f)\n", o.x, o.y); // O=(1.0000, 2.0000)

  struct point * t = translate(&o, 10., 10.);

  printf("T=(%f, %f)\n", t->x, t->y); // T=(11.0000, 12.0000)
  printf("O=(%f, %f)\n", o.x, o.y);   // O=(11.0000, 12.0000)
}
```

Dangling Pointers



Could you explain me what are Dangling Pointers?

Dangling pointers occur when a pointer variable continues to hold the memory address of a memory location that has been deallocated or freed. This can happen in several ways:

> Memory allocation: When a pointer variable is assigned the memory address of an allocated block of memory, and then that block is freed or deallocated, the pointer variable becomes a dangling pointer.

Dangling pointers can cause a variety of problems, such as memory leaks, program crashes, or data corruption. They can be difficult to detect and prevent, and they can have serious consequences in systems that are sensitive to errors. Therefore, it is essential to be aware of how to detect and prevent dangling pointers in your code.

Pointers on arrays vs Arrays of pointers

```
int *ptr1[10]; // array of 10 pointers to int value
int* ptr2[10]; // same as above

int (*ptr3)[10]; // pointer to an array: int[10]
```

Pointers to functions

- Side note: one can make pointer to a function

```c
int add(int a, int b) {
  return a + b;
}

int main(void) {
  int (*ptr)(int, int);

  ptr = &add;

  int r = ptr(2, 3);
  printf("%d\n", r);
}
```

Additional References

- **C Programming** YouTube

  https://www.youtube.com/playlist?list=PLBlnK6fEyqRggZZgYpPMUxdY1CYkZtARR

- **Pointers in C/C++** YouTube

  https://www.youtube.com/playlist?list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_

That's all Folks