Systems Architecture

5. Pointers in C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

1. Introduction

- As we know, C is general purpose programming language often classified as **high-level** programming language, since it provides abstractions of the details of the computer using elements close to a natural language
- Nevertheless, C is sometimes called a *middle-level* programming language, since it also provides low-level characteristics (related to memory management)
- This low-level memory management is done through **pointers**, which is one the most powerful features of the C, but at the same time, it can be error-prone
  - The incorrect manipulation of memory in C programs leads to the well-known "**segmentation fault**" problem

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

2. What are pointers?

- Each variable in a C program is stored in the main memory of the computer executing it
- A **pointer** in C is a variable that stores a memory address
- There are two operators in C to handle pointers:

Operator	Description	Example
&	***Reference*** operator (to get the memory address of a variable)	&b
*	***Dereference*** operator (to declare pointer or get the value of a given pointer)	*b

2. What are pointers?

- Pointers, like any other variable in C, need to be declared using a given type and variable name
- The difference from regular variables is that we use the operator ***** before the variable name:

```
type *pointer_name;
```

- For instance:

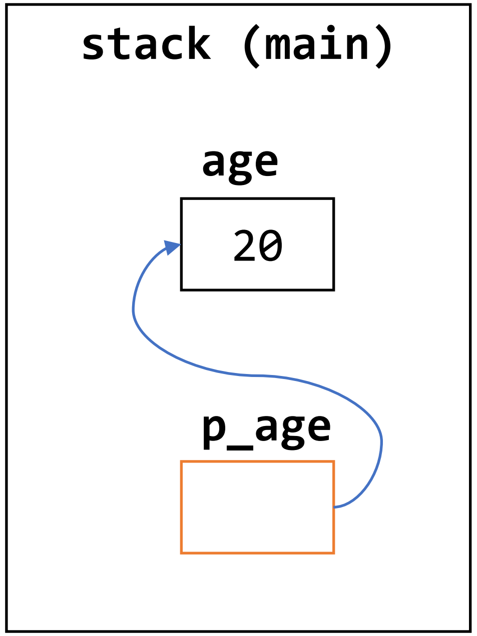
```
int *ip;    // pointer to an integer
double *dp; // pointer to a double
float *fp;  // pointer to a float
char *cp;   // pointer to a character
```

2. What are pointers?

- Consider the following example:

```
int main() {
    int age = 20;
    int *p_age = &age;

    // ...

    return 0;
}
```

This C program defines a regular integer variable (**int**) and a pointer to integer (**int***)



This box represents the memory handled in the scope of the **main** function

Variable	Memory address	Content
	...	
a	0x7ffe66dd68ac	20
	...	
p_age	0x7ffe8e2379aa	0x7ffe66dd68ac
	...	

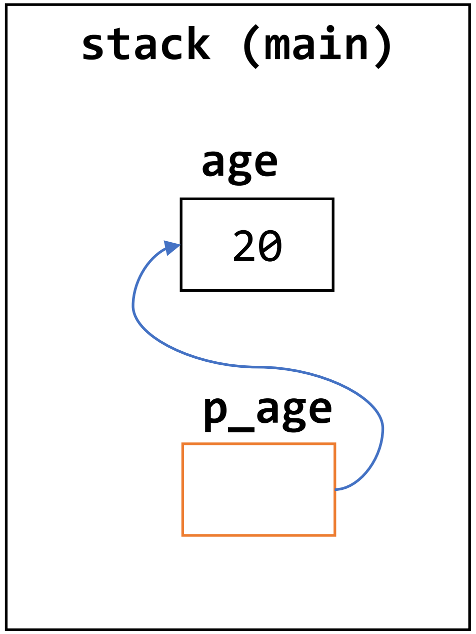
This table represents the physical memory of the computer running the C program

2. What are pointers?

- The following example illustrates a very basic declaration an usage of a pointer variable:

```c
#include <stdio.h>

int main() {
    int age = 20;
    int *p_age = &age;

    printf("The value of the variable age is %d\n", age);
    printf("The memory address in which age is stored is %p\n", p_age);
    printf("The value pointed by p_age is %d\n", *p_age);
    return 0;
}
```

```
The value of the variable age is 20
The memory address in which age is stored is 0x7ffe66dd68ac
The value pointed by p_age is 20
```



2. What are pointers?

```c
#include <stdio.h>

int main() {
    int age = 20;
    int *p_age = &age;

    printf("The value of the variable age is %d\n", age);
    printf("The memory address in which age is stored is %p\n", p_age);
    printf("The value pointed by p_age is %d\n", *p_age);

    age = 40;

    printf("The value of the variable age is %d\n", age);
    printf("The value pointed by p_age is %d\n", *p_age);

    return 0;
}
```

What is the value of *p_age in this example?

Table of contents

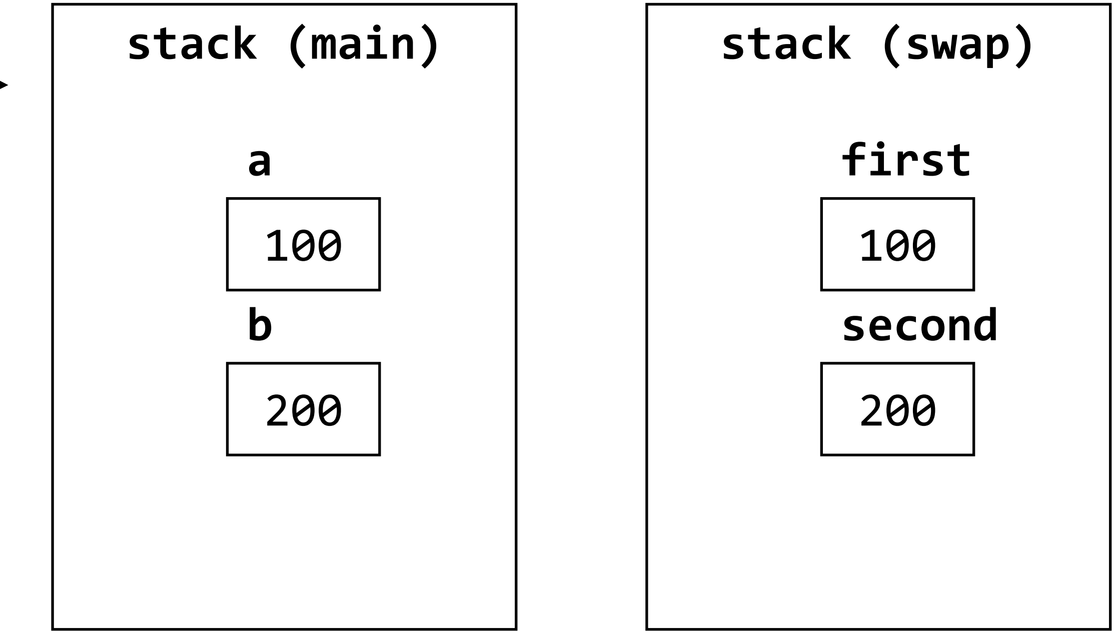
1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

3. Passing arguments by value and by reference

- We when invoke a function in C, and their arguments are not pointers, we say that we are passing arguments **by value.** Passing arguments by value implies:
  - Values of caller parameters are copied to the function
  - Changes made inside functions are not reflected in caller parameters
- On the other hand, if the arguments of a function are pointers, we say that the arguments are passed **by reference**. This implies:
  - Both the caller and functional parameters refer to the same location
  - Changes made inside the function are reflected in caller parameters
- To illustrate it, consider the *swap* function, which is a simple function that exchanges the values of two variables

3. Passing arguments by value and by reference

```
#include <stdio.h>

void swap(int first, int second) {
    int tmp;

    tmp = first;
    first = second;
    second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(a, b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```

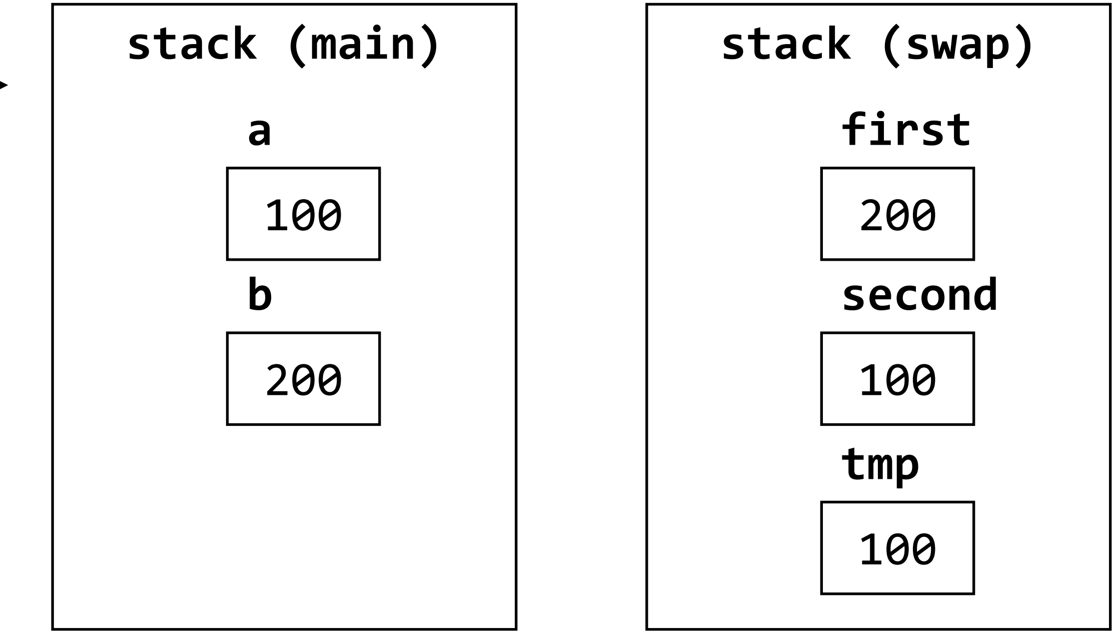


```
Before swap:
         a=100
         b=200
```

This is an example of a function (swap) using pass by value

3. Passing arguments by value and by reference

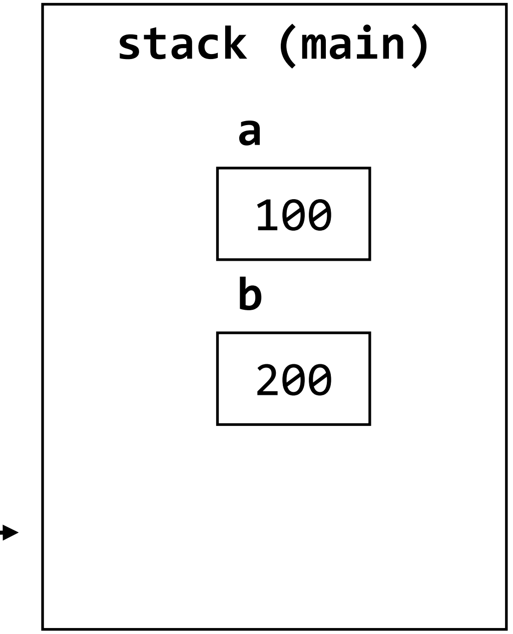
```c
#include <stdio.h>

void swap(int first, int second) {
    int tmp;

    tmp = first;
    first = second;
    second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(a, b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```



```
Before swap:
        a=100
        b=200
```

3. Passing arguments by value and by reference

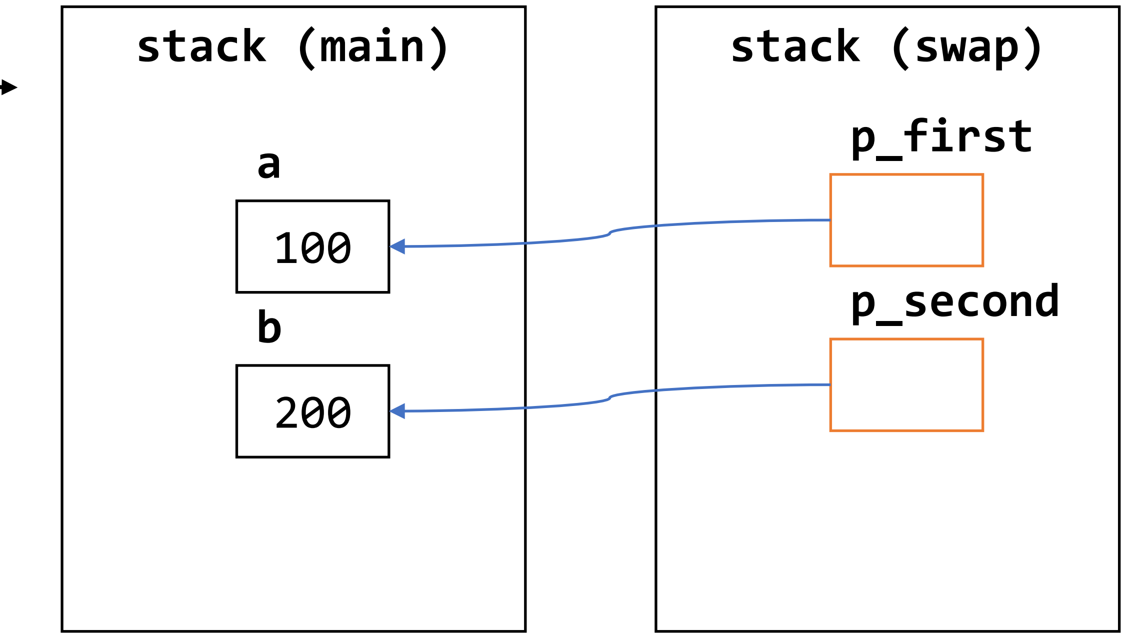
```c
#include <stdio.h>

void swap(int first, int second) {
    int tmp;

    tmp = first;
    first = second;
    second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(a, b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```



```
Before swap:
         a=100
         b=200
After swap:
         a=100
         b=200
```

3. Passing arguments by value and by reference

```
#include <stdio.h>

void swap(int *p_first, int *p_second) {
    int tmp;

    tmp = *p_first;
    *p_first = *p_second;
    *p_second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(&a, &b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```

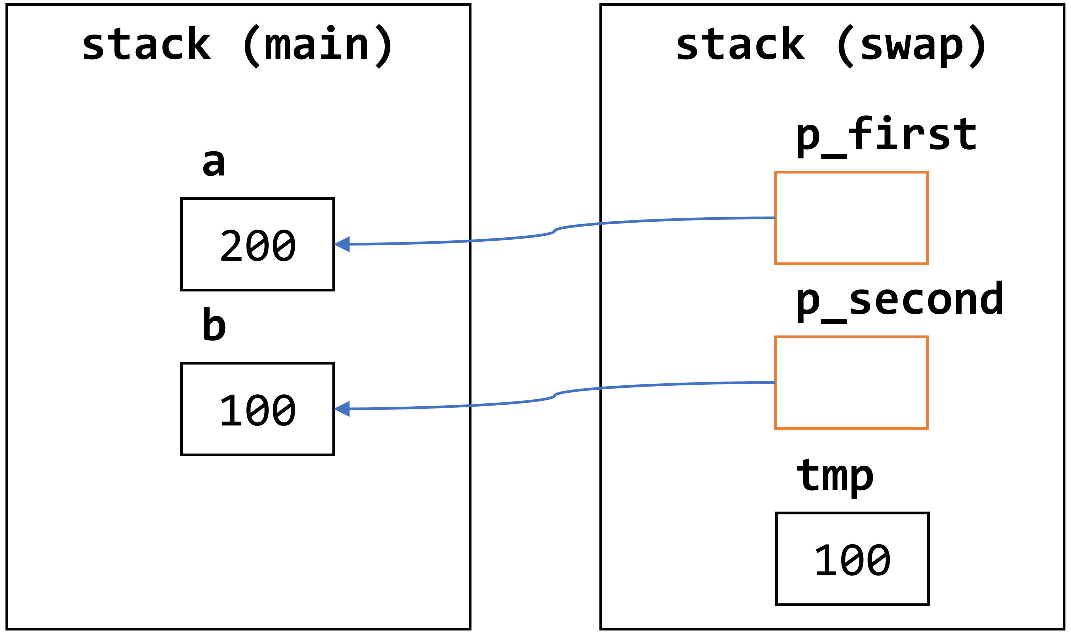


```
Before swap:
        a=100
        b=200
```

This is an example of a function (swap) using pass by reference

3. Passing arguments by value and by reference

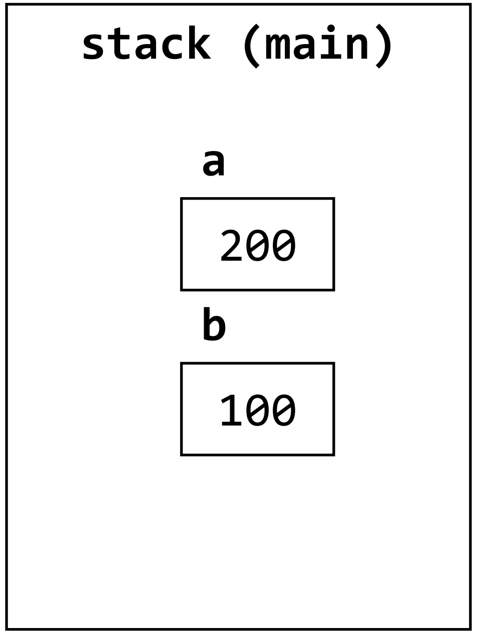
```c
#include <stdio.h>

void swap(int *p_first, int *p_second) {
    int tmp;

    tmp = *p_first;
    *p_first = *p_second;
    *p_second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(&a, &b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```

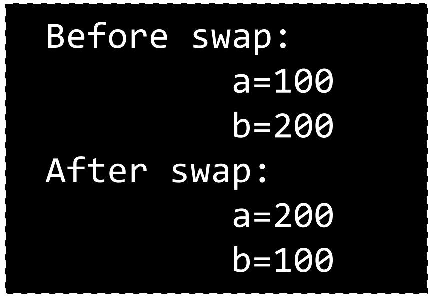


```
Before swap:
         a=100
         b=200
```

3. Passing arguments by value and by reference

```c
#include <stdio.h>

void swap(int *p_first, int *p_second) {
    int tmp;

    tmp = *p_first;
    *p_first = *p_second;
    *p_second = tmp;
}

int main() {
    int a = 100;
    int b = 200;

    puts("Before swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    swap(&a, &b);

    puts("After swap:");
    printf("\t a=%d \n", a);
    printf("\t b=%d \n", b);

    return 0;
}
```





```
Before swap:
         a=100
         b=200
After swap:
         a=200
         b=100
```

Arguments passed by reference are sometimes called “output arguments”, since the function can modify the value stored at that address, which will be reflected in the original variable

3. Passing arguments by value and by reference

- The **scanf** function works using output arguments (i.e., passed by reference):

```
int scanf(const char *format, ...);
```

The *varargs* parameters in **scanf** need to be pointers, because the changes made inside the function **scanf** are reflected in caller parameters

- For this reason, when we invoke **scanf** for basic types (e.g. **char**, **int**, etc.), we need to use the reference operator (&)

```
int i;
char str[40];

scanf("%d", &i);
scanf("%s", str);
```

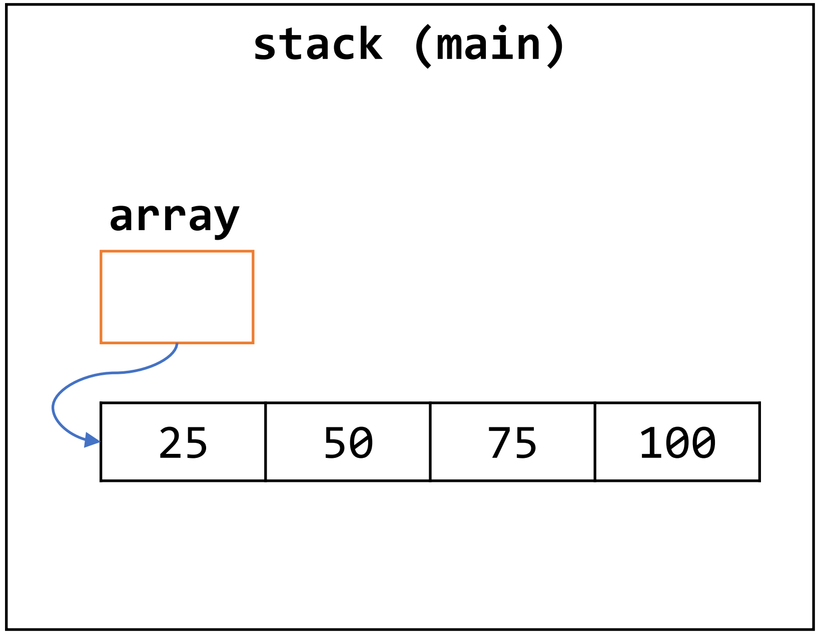
Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

4. Pointers and arrays

- We already know that arrays are collections of data with the same type and stored in contiguous memory
- Arrays behave similarly to pointers, since internally, an array variable is a constant pointer pointing to the first element of the array

```
int main() {
    int array[] = { 25, 50, 75, 100 };

    // ...
}
```



4. Pointers and arrays

- This fact has relevant implications. For example consider the following program:

```c
#include <stdio.h>
#define SIZE 4

void double_array(int array[], int size) {
    for (int i = 0; i < size; i++) {
        array[i] *= 2;
    }
}

int main() {
    int array[SIZE] = { 25, 50, 75, 100 };

    double_array(array, SIZE);

    printf("%d\n", array[0]);

    return 0;
}
```

What can we see in the standard output when this program is executed?

4. Pointers and arrays

- A pointer in c is a memory address, which is a numeric value
- We can perform basic arithmetic operations (i.e., **addition** and **subtraction**) on pointers

```c
#include <stdio.h>

int main() {
    int array[] = { 25, 50, 75, 100 };

    int *a = array;     // initial address
    int *b = array + 1; // initial address + (sizeof(int) * 1)
    int *c = array + 2; // initial address + (sizeof(int) * 2)
    int *d = array + 3; // initial address + (sizeof(int) * 3)

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

    return 0;
}
```

```
*a=25
*b=50
*c=75
*d=100
```

4. Pointers and arrays

- One key difference between array and pointers is the size of the memory required
  - When arrays are created, a fixed size of the memory is allocated
  - That size is unknown when using pointers
  - We can check this difference by invoking the operator **sizeof**

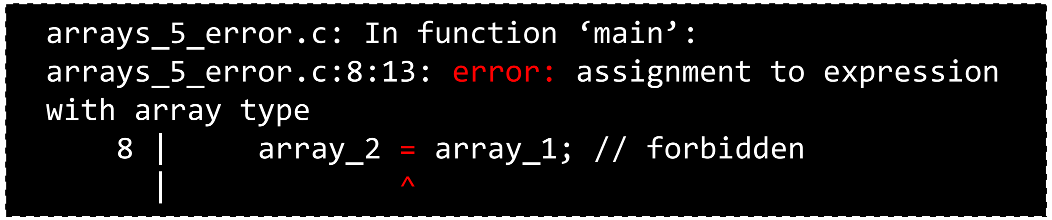
```c
#include <stdio.h>

int main() {
    int array[8];
    int *pointer = array;

    unsigned int s1 = sizeof(array);
    unsigned int s2 = sizeof(pointer);

    printf("s1=%d\n", s1);
    printf("s2=%d\n", s2);

    return 0;
}
```

```
s1=32
s2=8
```

4. Pointers and arrays

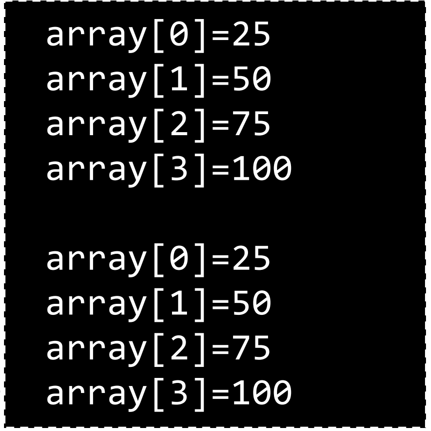
- Also, remember that array variables cannot be assigned of another variable (we use the memcpy function instead):

```c
#include <stdio.h>
#define SIZE 4

int main() {
    int array_1[SIZE] = { 25, 50, 75, 100 };
    int array_2[SIZE];

    array_2 = array_1; // forbidden

    return 0;
}
```



```
arrays_5_error.c: In function ‘main’:
arrays_5_error.c:8:13: error: assignment to expression
with array type
    8 |     array_2 = array_1; // forbidden
      |             ^
```

```c
#include <stdio.h>
#include <string.h>
#define SIZE 4

void display_array(int array[], int size) {
    for (int i = 0; i < size; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int array_1[SIZE] = { 25, 50, 75, 100 };
    int array_2[SIZE];

    memcpy(array_2, array_1, sizeof(array_1));

    display_array(array_1, SIZE);
    display_array(array_2, SIZE);

    return 0;
}
```



```
array[0]=25
array[1]=50
array[2]=75
array[3]=100

array[0]=25
array[1]=50
array[2]=75
array[3]=100
```

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

5. Pointers and strings

- A string in C is an array of characters terminated by a null character `'\0'`
- Declaration examples:

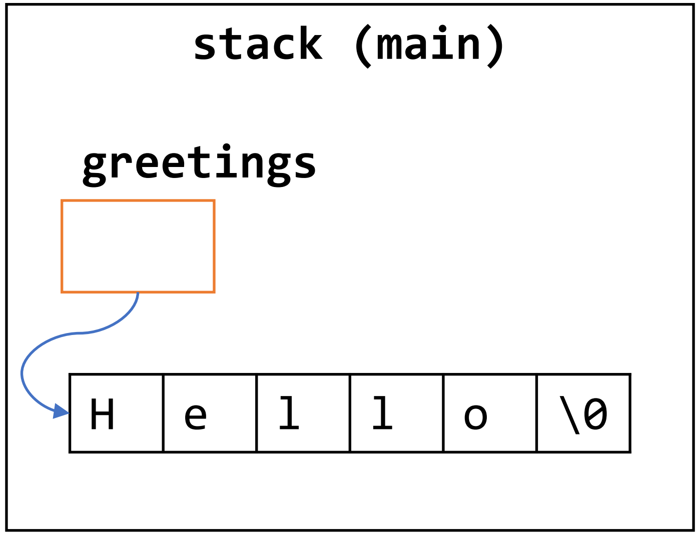
```
char greetings[] = "Hello"; // Array of characters
```

```
char *greetings = "Hello"; // String literal
```

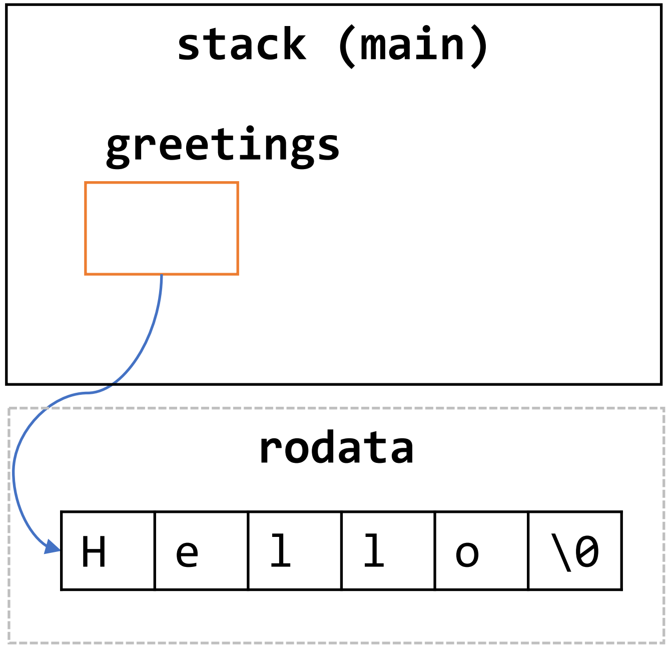
- Internally, these characters are stored contiguously in memory, accessed and manipulated via **pointers**
- The name of a char array acts as a pointer to its first character
  - `*(str + i)` is equivalent to `str[i]`

5. Pointers and strings

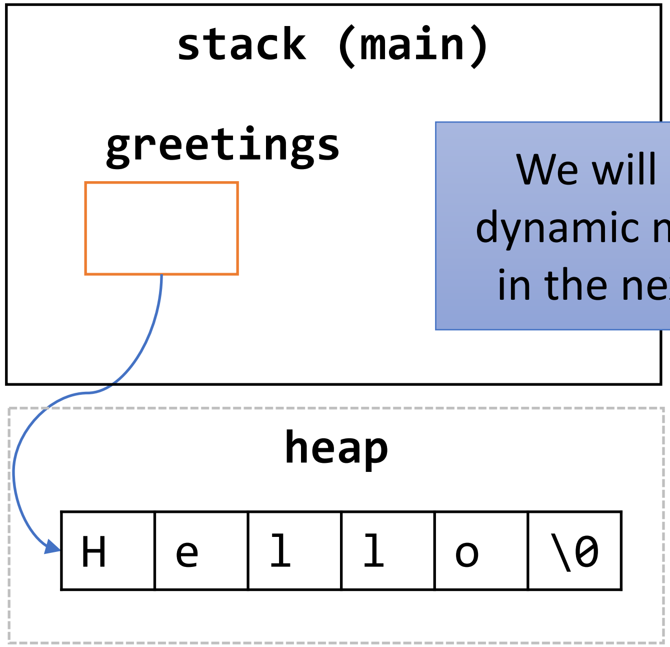
```c
#include <stdio.h>

int main() {
    char greetings[] = "Hello";
    printf("%s\n", greetings);

    return 0;
}
```



```c
#include <stdio.h>

int main() {
    char *greetings = "Hello";
    printf("%s\n", greetings);

    return 0;
}
```



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

int main() {
    char *greetings = strdup("Hello");
    printf("%s\n", greetings);
    free(greetings);

    return 0;
}
```



We will study dynamic memory in the next unit

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

Fork me on GitHub

6. Pointers and structs

- We can use the arrow operator (->) for accessing members of an structure using pointers

```c
#include <stdio.h>

#define MAX_STR 80

typedef struct Person {
    char name[MAX_STR];
    int age;
} Person;

int main() {
    Person person = { "Alice", 25 };
    Person *pointer = &person;

    printf("Name: %s -- Age: %d\n", (*pointer).name, (*pointer).age);
    printf("Name: %s -- Age: %d\n", pointer->name, pointer->age);

    return 0;
}
```

```
Name: Alice -- Age: 25
Name: Alice -- Age: 25
```

(*pointer).name and pointer->name are equivalent, although the use of the arrow operator is more readable

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

Fork me on GitHub

7. Function pointers

- A function pointer in C is a variable that stores the address of a function
  - This allows for dynamic function calls, where the function to be executed is determined at runtime

```c
#include <stdio.h>

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

int main() {
    int (*func_ptr)(int, int); // Declare a function pointer
    func_ptr = add; // Assign the address of 'add' function to the pointer

    int result = func_ptr(5, 3); // Call the function using the pointer
    printf("Result: %d\n", result); // Output will be 8

    return 0;
}
```

See a more complex example in the GitHub repository

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

8. NULL pointer

- Several programming languages (such as C, Java, JavaScript or Python, among others) has the concept of *null* as a special marker indicating that something has no value
- In C, **`NULL`** is a special reserved pointer value that does not point to any valid data object
- We can think in **`NULL`** in C like a memory address (i.e., a pointer) with all its bits put to zero
  - In a 64 bits computer: **`NULL`** = `0x0000000000000000` (i.e., `0` in decimal)
  - Therefore, **`NULL`** is interpreted as false in expressions
- Some of the most common use cases for **`NULL`** are
  - To initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet
  - To check for a null pointer before accessing any pointer variable

Fork me on GitHub

8. NULL pointer

- The following example illustrates a very basic usage of a **NULL** pointer:

```c
#include <stdio.h>

int main() {
    int *pointer = NULL;

    /*
     This equivalent to:
     if (pointer == 0)
     if (!pointer)
     */
    if (pointer == NULL) {
        printf("Pointer is NULL\n");
    }

    return 0;
}
```

```
Pointer is NULL
```

8. NULL pointer

- The following example show how null and non-null references are displayed:

```c
#include <stdio.h>

int main() {
    int *null_pointer = NULL;
    char *my_string = "Hello";

    printf("The address of null_pointer is %p\n", null_pointer);
    printf("The address of my_string is %p\n", my_string);

    return 0;
}
```

```
The address of null_pointer is (nil)
The address of my_string is 0x55c366f79008
```

Table of contents

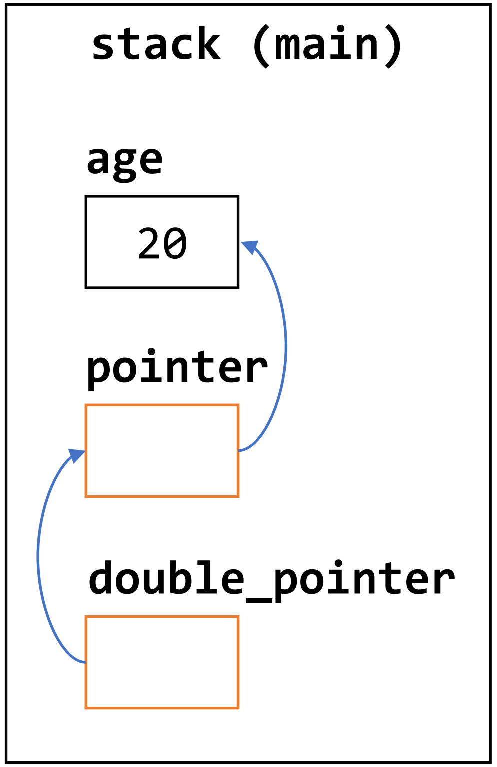
1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. **Double pointers**
10. Program arguments
11. Takeaways

9. Double pointers

- A **double pointer** (also known as *pointer to a pointer*) is a form of multiple indirection, i.e., a chain of pointers
  - We use two stars (******) to declare a double pointer

```c
#include <stdio.h>

int main() {
    int age = 20;
    int *pointer = &age;
    int **double_pointer = &pointer;

    printf("The value of the variable age is %d\n", age);
    printf("The value pointed by *pointer is %d\n", *pointer);
    printf("The value pointed by **double_pointer is %d\n", **double_pointer);

    return 0;
}
```

```
The value of the variable age is 20
The value pointed by *pointer is 20
The value pointed by **double_pointer is 20
```



Fork me on GitHub

9. Double pointers

- Here are some common scenarios where double pointers are used:
  - To implement two-dimension arrays (e.g., a arrays of strings)
  - To modify a pointer outside its scope, e.g. for memory allocation (we will see this in the next unit)

```c
#include <stdio.h>

int main() {
    char *words[2];
    words[0] = "hello";
    words[1] = "world";

    printf("words[0]=%s\n", words[0]);
    printf("words[1]=%s\n", words[1]);

    return 0;
}
```

```
words[0]=hello
words[1]=world
```

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

10. Program arguments

- We already know that the entry point of any C program is the main function

```c
int main() {

    // ...

}
```

- We also know that shell commands can be invoked with a list of arguments (after the command name)

```
$ command <arg1> <arg2>
```

But, how can we handle arguments passed from the command line in our C programs?

10. Program arguments

- To pass command line arguments, we typically define **`main()`** with two arguments:
  - **`argc`** (*argument count*): it is an integer value that stores the number of command-line arguments passed by the user including the name of the program
  - **`argv`** (*argument vector*): it is array of character pointers listing all the arguments

```c
int main(int argc, char *argv[]) {

    // ...

}
```

```c
int main(int argc, char **argv) {

    // ...

}
```

These two ways to define the main arguments are equivalent

Fork me on GitHub

10. Program arguments

- This program illustrates the use of program arguments:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("This program was called with \"%s\"\n", argv[0]);

    if (argc > 1) {
        for (int i = 1; i < argc; i++) {
            printf("argv[%d] = %s\n", i, argv[i]);
        }
    } else {
        puts("The command had no other arguments");
    }

    return 0;
}
```

```
$ gcc args.c -o my-program
```

```
$ ./my-program
This program was called with "./my-program"
The command had no other arguments
```

```
$ ./my-program 1 hello 2 world
This program was called with "./my-program"
argv[1] = 1
argv[2] = hello
argv[3] = 2
argv[4] = world
```

Table of contents

1. Introduction
2. What are pointers?
3. Passing arguments by value and by reference
4. Pointers and arrays
5. Pointers and strings
6. Pointers and structs
7. Function pointers
8. NULL pointer
9. Double pointers
10. Program arguments
11. Takeaways

11. Takeaways

- A pointer in C is a variable that stores a memory address
- There are two operators in C to handle pointers: & (address-of) and * (pointer declaration and value-of)
- When the arguments of a function are pointers, we say that the arguments are passed by reference. In this case, changes made inside the function are reflected in caller parameters
- Arrays behave similarly to pointers, since internally, an array variable is a constant pointer pointing to the first element of the array
- We use array of characters (or a pointer to char) to handle strings in C
- We can use the arrow operator (`->`) for accessing members of an structure using pointers
- Function pointers provide a way to store the address of a function in a variable
- `NULL` is a special reserved pointer value that does not point to any valid data object
- A double pointer (`**`) is a chain of pointers (e.g. to allocate memory for a pointer outside its scope)
- To pass command line arguments, we define the main function with `argc` (argument count) and `argv` (argument vector)