Systems Architecture

2. Basics of C programming

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

- In this lecture, we continue learning the foundations of the C programming language
- In particular, we study the following:
  - Functions
  - Operators
  - Control flow
  - Arrays
  - Strings
  - Structured data (structs and unions)
  - Top-down design

Table of contents

2. Functions

- The code written in a C program is divided into **functions**
  - Although similar to Java methods, C functions are not embedded in classes
  - A C function is defined by a name, parameter(s), and result type

```
result_type name(param1_type param1, ..., paramN_type paramN ) {
    /*
     * Function body
     */
    return <value>;
}
```

```
void name(param1_type param1, ..., paramN_type paramN ) {
    /*
     * Function body
     */
}
```

If the function is declared as **void**, it does not return any value

2. Functions

- Example of a very basic function:

```c
#include <stdio.h>

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

int main() {
    int i = 1;
    int j = 2;

    printf("%d + %d = %d\n", i, j, sum(i, j));
    return 0;
}
```

```
1 + 2 = 3
```

2. Functions

- The order matters in C: the function declaration needs to be add before the first call of the function

```c
#include <stdio.h>

int main() {
    int i = 1;
    int j = 2;

    printf("%d + %d = %d\n", i, j, sum(i, j));
    return 0;
}

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

```
functions_2.c: In function ‘main’:
functions_2.c:7:36: warning: implicit declaration of function ‘sum’ [-Wimplicit-function-declaration]
    7 |     printf("%d + %d = %d\n", i, j, sum(i, j));
      |                                    ^~~
1 + 2 = 3
```

2. Functions

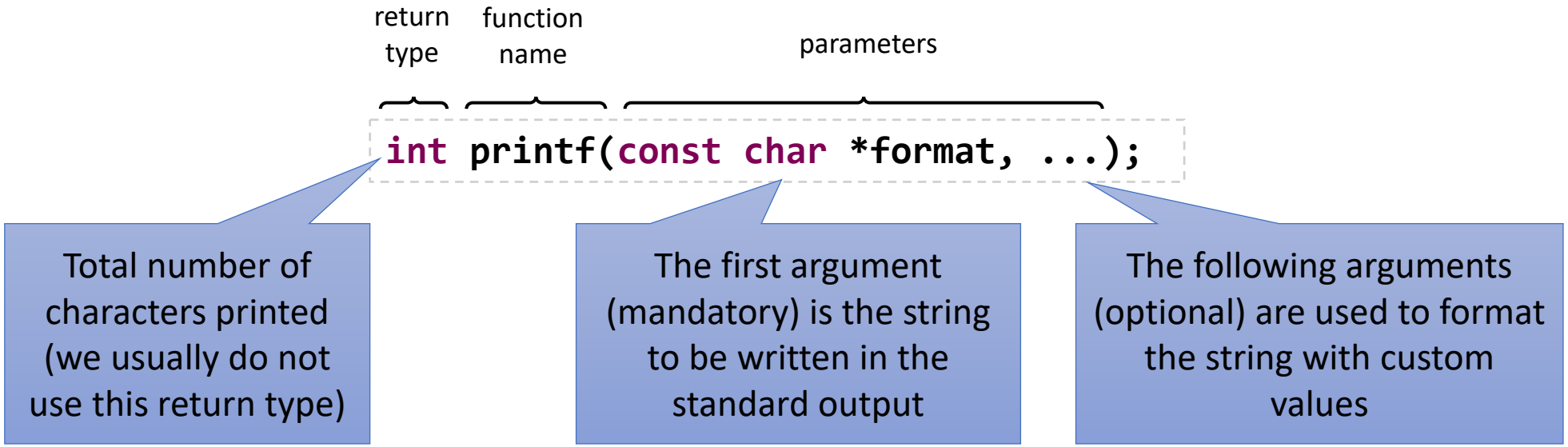
- A common practice is to put the function declaration above **`main()`** and the function definition below **`main()`**

The *prototype* of a function is the declaration of its function, parameters, and return type

```c
#include <stdio.h>

int sum(int a, int b); // Function declaration

int main() {
    int i = 1;
    int j = 2;

    printf("%d + %d = %d\n", i, j, sum(i, j));
    return 0;
}

int sum(int a, int b) { // Function definition
    return a + b;
}
```

2. Functions - Variadic

- Variadic functions are functions (also called *varargs* functions) can take a variable number of arguments
- The declaration of a variadic function uses an ellipsis (`...`) as the last parameter
- For example, **`printf`** is a variadic function, and its prototype is as follows:



2. Functions - Variadic

- The following example illustrates how a variadic function (`printf`, in this case) can be invoked
- The examples repository contains another example to create a custom variadic function

```c
#include <stdio.h>

int main() {
    int number = 10;
    char character = 'a';

    printf("Using printf\n");
    printf("Number is %d\n", number);
    printf("Number is %d and character is %c\n", number, character);

    return 0;
}
```

```
Using printf
Number is 10
Number is 10 and character is a
```

Table of contents

3. Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical functions
- The C language provides the following types of operators:
  - Arithmetic: to perform basic actions on actions on numbers
  - Logical: to evaluate boolean expressions
  - Relational: to evaluate the relationship between two arguments
  - Bitwise: to perform bit-by-bit operation
  - Assignment: to assign a new value to a variable
  - Miscellaneous: other operators

3. Operators - Arithmetic

Operator	Description	Example
+	Addition (adds two operands)	1 + 1
-	Subtraction (subtracts second operand from the first)	2 - 2
*	Multiplication (multiplies both operands)	3 * 4
/	Division (divides numerator by de-numerator)	5.0 / 4.0
%	Modulus (remainder of after an integer division)	5 % 3
++	Increment (increases the integer value by one)	a++
--	Decrement (decreases the integer value by one)	b--

```c
#include <stdio.h>

int main() {
    int sum = 1 + 1;
    int subtraction = 2 - 2;
    int multiplication = 3 * 4;
    float division = 5.0 / 4.0;
    int module = 5 % 3;

    printf("Sum: %d\n", sum);
    printf("Subtraction: %d\n", subtraction);
    printf("Multiplication: %d\n", multiplication);
    printf("Division: %.2f\n", division);
    printf("Module: %d\n", module);
    printf("Increment: %d\n", ++sum);
    printf("Decrement: %d\n", --sum);

    return 0;
}
```

```
Sum: 2
Subtraction: 0
Multiplication: 12
Division: 1.25
Module: 2
Increment: 3
Decrement: 2
```

3. Operators - Logical

- C does not have a basic type for boolean values, and instead, it uses integers for logic operations
  - `0` means false
  - Different than `0` means true

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true	a && b
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true	a || b
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!a

3. Operators - Logical

```c
#include <stdio.h>

int main() {
    int a = 0;
    int b = 10;

    if (a && b) {
        printf("First condition is true\n");
    } else {
        printf("First condition is not true\n");
    }

    if (a || b) {
        printf("Second condition is true\n");
    }

    if (!a) {
        printf("Third condition is true\n");
    }

    return 0;
}
```

```
First condition is not true
Second condition is true
Third condition is true
```

3. Operators - Logical

- The library **stdbool.h** defines the type bool and the constants true and false (but internally, it uses 0 and 1)

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

int main() {
    bool t = true;
    bool f = false;

    if (t) {
        printf("True: %d\n", t);
    }

    if (!f) {
        printf("False: %d\n", f);
    }

    return 0;
}
```

```
True: 1
False: 0
```

3. Operators - Logical

- Alternatively, we can use custom macros for TRUE and FALSE:

```c
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int main() {
    if (TRUE) {
        printf("True: %d\n", TRUE);
    }

    if (!FALSE) {
        printf("False: %d\n", FALSE);
    }

    return 0;
}
```

```
True: 1
False: 0
```

3. Operators - Logical

- Also, we can use a custom ***boolean*** type:

```c
#include <stdio.h>

typedef enum {false = 0, true} boolean;

int main() {
    boolean t = true;
    boolean f = false;

    if (t) {
        printf("True: %d\n", t);
    }

    if (!f) {
        printf("False: %d\n", f);
    }

    return 0;
}
```

```
True: 1
False: 0
```

3. Operators - Relational

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true	a == b
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true	a != b
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true	a > b
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true	a < b
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true	a >= b
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true	a <= b

3. Operators - Relational


```c
#include <stdio.h>

int main() {
    int a = 0;
    int b = 10;

    printf("%d > %d : %d\n", a, b, a > b);
    printf("%d < %d : %d\n", a, b, a < b);
    printf("%d == %d : %d\n", a, b, a == b);
    printf("%d != %d : %d\n", a, b, a != b);

    return 0;
}
```

```
0 > 10 : 0
0 < 10 : 1
0 == 10 : 0
0 != 10 : 1
```

3. Operators - Bitwise

Operator	Description	Example
&	Binary AND (copies a bit to the result if it exists in both operands)	a & b
|	Binary OR (copies a bit if it exists in either operand)	a | b
^	Binary XOR (copies the bit if it is set in one operand but not both)	a ^ b
~	Binary one's complement (flipping' bits)	a ~ b
>>	Binary left shift operator (the left operands value is moved left by the number of bits specified by the right operand)	a >> b
<<	Binary right shift operator (the left operands value is moved right by the number of bits specified by the right operand)	a << b

3. Operators - Bitwise


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

void print_bin(unsigned char byte) {
    int i = CHAR_BIT; // Number of bits in a byte, i.e., 8
    while (i--) {
        putchar('0' + ((byte >> i) & 1));
    }
}

int main() {
    int a = 201, b = 11;

    printf("a\t");
    print_bin(a);
    printf("\n");

    printf("b\t");
    print_bin(b);
    printf("\n");

    printf("a&b\t");
    print_bin(a & b);
    printf("\n");

    return 0;
}
```

```
a       11001001
b       00001011
a&b     00001001
```

3. Operators - Assignment

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	`C <<= 2 is same as C = C << 2`
>>=	Right shift AND assignment operator	`C >>= 2 is same as C = C >> 2`
&=	Bitwise AND assignment operator	`C &= 2 is same as C = C & 2`
^=	Bitwise exclusive OR and assignment operator	`C ^= 2 is same as C = C ^ 2`
|=	Bitwise inclusive OR and assignment operator	`C |= 2 is same as C = C | 2`

3. Operators - Miscellaneous

Operator	Description	Example
`sizeof()`	Returns the storage size (in bytes) of a variable or type	`sizeof(int)`
&	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
? :	Ternary operator (to run one code when the condition is true and another code when the condition is false)	(a > b) ? c : d

```c
#include <stdio.h>

int main() {
    int age = 18;
    printf("You age is %d. ", age);
    (age >= 18) ? printf("You can vote.\n") : printf("You cannot vote.\n");

    int canvote = (age >= 18) ? 1 : 0;
    printf("canvote=%d\n", canvote);

    return 0;
}
```

```
You age is 18. You can vote.
canvote=1
```

Table of contents

4. Control flow

- In computer science, **control flow** is the order in which the individual statements of an imperative program are executed or evaluated
- In C programming, there are two types of control flow statements:
  - **Branching**, which is deciding what actions to take
    - `if-else`
    - `switch`
  - **Looping**, which is deciding how many times to take a certain action
    - `while`
    - `do-while`
    - `for`

4. Control flow - `if-else`

- It takes an expression in parenthesis and an statement or block of statements
- if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped (and the **`else`** block, is present, is evaluated)

```
if (expression) {
    ...
}
else {
    ...
}
```

Remember that C uses integers for logic operations:

- `0` means false
- Different than `0` means true

4. Control flow - `switch`

- The switch statement is used to perform different actions based on different conditions. It is like a nested if-else statement
  - The value of the expression is compared with the values of each case
  - If there is a match, the associated block of code is executed
  - The **`break`** statement breaks out of the switch block and stops the execution
  - The **`default`** (optional) specifies some code to run if there is no case match

```
switch (expression) {
    case value1:
    ...
    break;
    case value2:
    ...
    break;
    ...
    default:
    ...
    break;
}
```

The expression should be an integer or enumerated

4. Control flow - `switch`

The function `scanf()` is used to take input from the user

```c
#include <stdio.h>

int main() {
    int month;

    printf("Enter month number(1-12): ");
    scanf("%d", &month);

    switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        printf("31 days\n");
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        printf("30 days\n");
        break;
    case 2:
        printf("28/29 days\n");
        break;

    default:
        printf("Invalid month\n");
    }

    return 0;
}
```

```
Enter month number(1-12): 1
31 days
```

4. Control flow - `while` and `do-while`

- The while loop loops through a block of code as long as a specified condition is true:

```
while (expression) {
    ...
}
```

- The do-while loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true:

```
do {
    ...
} while (expression);
```

4. Control flow - for

- The for loop iterates a number of times:

```
for (init_expr; cond_expr; update_expr) {
    ...
}
```

```c
#include <stdio.h>

int main() {
    int i;

    for (i = 0; i < 5; i++) { // iterate i from 0 to 4
        printf("%d\n", i);
    }
}
```

```
0
1
2
3
4
```

4. Control flow - `break` and `continue`

- The **`break`** statement is used to jump out of a loop:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 4) {
            break;
        }
        printf("%d\n", i);
    }
}
```

```
0
1
2
3
```

4. Control flow - `break` and `continue`

- The **`continue`** statement breaks one iteration in the loop and continues with the next iteration:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 4) {
            continue;
        }
        printf("%d\n", i);
    }
}
```

```
0
1
2
3
5
6
7
8
9
```

Table of contents

5. Arrays

- An **array** is a collection of the data with the same type and stored at contiguous memory locations
  - We use square brackets **[ ]** to create arrays
  - We use an index number to access the array elements
  - The size of an array is fixed once it is declared

```c
#include <stdio.h>

int main() {
    int array_1[5]; // declaration
    array_1[0] = 100;
    array_1[1] = 200;
    printf("The value of the position 0 in array_1 is %d\n", array_1[0]);
    printf("The value of the position 1 in array_1 is %d\n", array_1[1]);

    int array_2[] = { 25, 50, 75, 100 }; // initialization
    printf("The value of the position 0 in array_2 is %d\n", array_2[0]);
    printf("The value of the position 3 in array_2 is %d\n", array_2[3]);

    return 0;
}
```

```
The value of the position 0 in array_1 is 100
The value of the position 1 in array_1 is 200
The value of the position 0 in array_2 is 25
The value of the position 3 in array_2 is 100
```

5. Arrays

- We usually use a **for** loop to iterate an array, for instance:

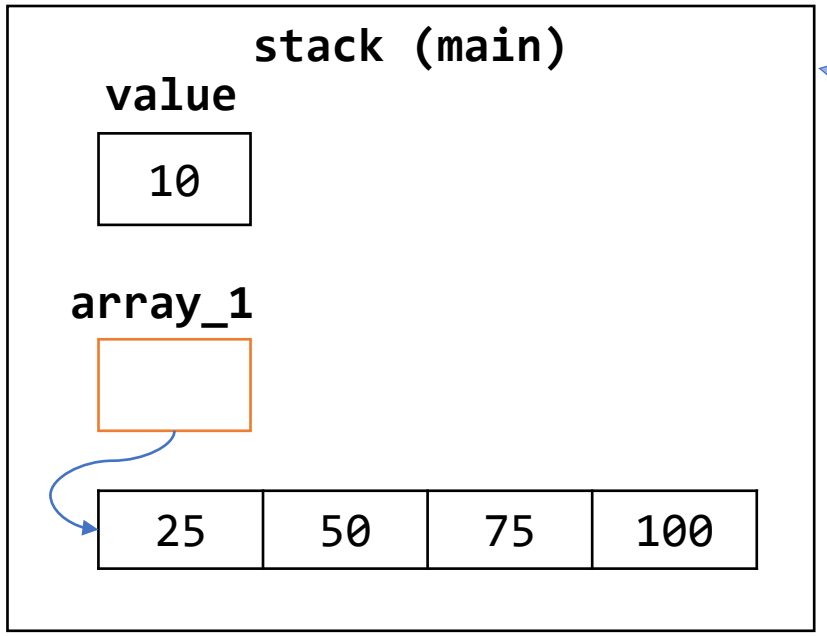
```c
#include <stdio.h>
#define SIZE 4

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

    for (int i = 0; i < SIZE; i++) {
        printf("The value of the position %d in array is %d\n", i, array[i]);
    }

    return 0;
}
```

```
The value of the position 0 in array is 25
The value of the position 1 in array is 50
The value of the position 2 in array is 75
The value of the position 3 in array is 100
```

5. Arrays

- Internally, an array in C is a **pointer**
  - A pointer is a variable that stores the memory address of another variable as its value
  - An array is a pointer that contains the memory address to the 0^{th} element of the array

```c
int main() {
    int value = 10;
    int array_1[] = { 25, 50, 75, 100 };

    // ...
}
```



The **stack** is the memory segment which stores local variables in runtime

5. Arrays

- Internally, arrays are constant pointers, which means that they can be initialized (i.e., set value in the declaration) but they cannot be assigned of another array

```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
      |             ^
```

5. Arrays

- To fix the previous error, the **`memcpy`** function can be used
  - **`memcpy`** copies **`size`** characters from memory area **`source`** to memory area **`dest`**

```
void *memcpy(void *dest, const void *source, size_t size);
```

Pointer to destination, i.e., ***dest**

Pointer to the destination where the content is to be copied

Pointer to the source of data to be copied

Number of bytes to be copied

5. Arrays

```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
```

The operator **sizeof** returns the total number of bytes of **array_1**

Table of contents

6. Strings

- In programming, a string is a sequence of characters
- Unlike many other programming languages, C does not have a native type to create string variables
- Instead, we use the char type to create an **array of characters** to handle strings in C
- We can use double quotes (" ") to initialize strings in C (called ***strings literals***)

```c
#include <stdio.h>

int main() {
    char greetings[] = "Hello";
    printf("%s\n", greetings);
    return 0;
}
```

```
Hello
```

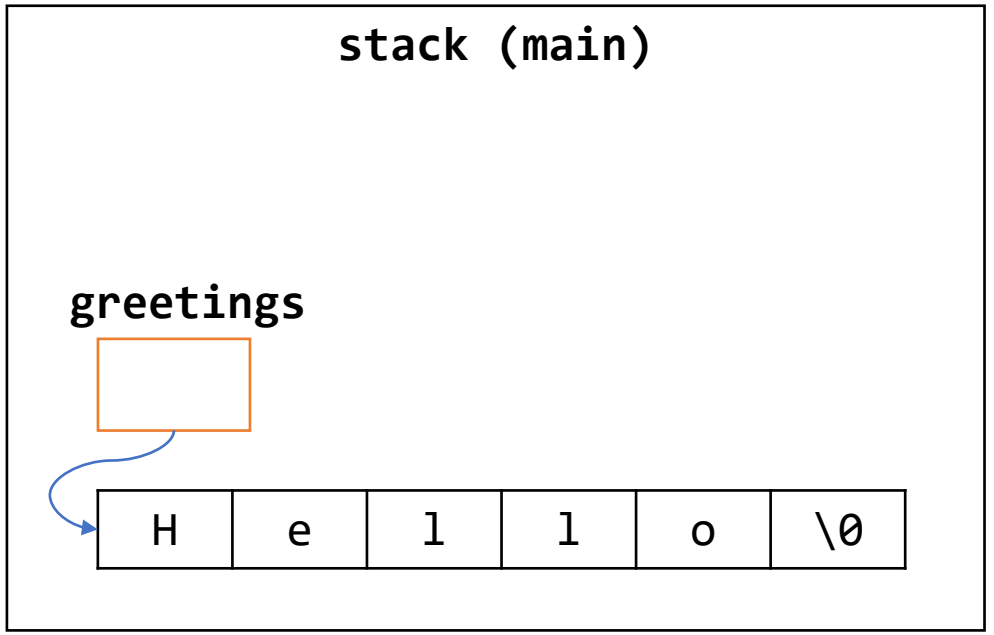
6. Strings

- Internally, each string in C (i.e., an array of characters) ends in an special character known as the *null terminating character*
  - We represent the null terminating character in C as `'\0'` (equivalent to `0` in decimal)

```c
int main() {
    char greetings[] = "Hello";
    // ...

    return 0;
}
```

```c
int main() {
    char greetings[] = { 'H', 'e', 'l', 'l', 'o', '\0' };

    // ...

    return 0;
}
```

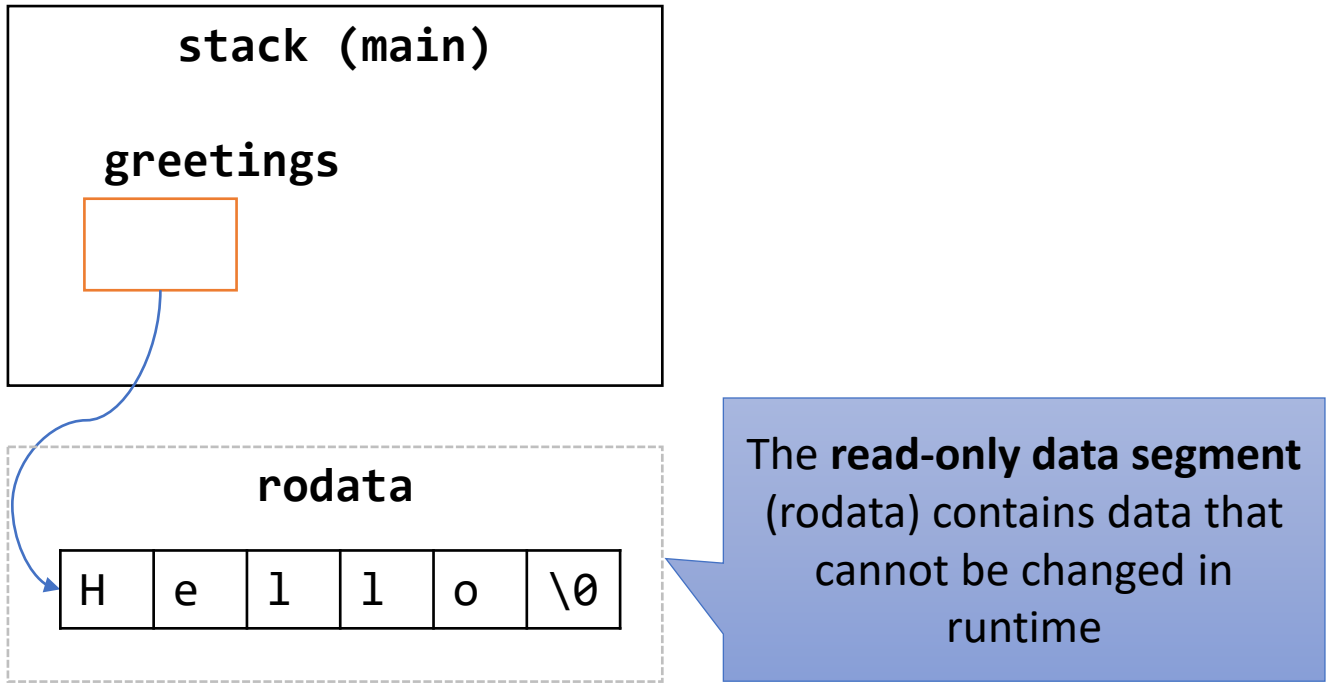
An equivalent way to define an string would be using an array of characters, using single quotes (' ') instead of double quotes (" "), but this way is much less readable, and so, it is not recommended



6. Strings

- Since arrays are internally pointers, we can also use the operator * to declare strings in C
  - But unlike when defining string literals with [ ], the strings declared using * are **immutable** (i.e., we cannot change its value in runtime) since they are stored in the read-only data segment

```c
int main() {
    char *greetings = "Hello";
    // ...
    return 0;
}
```



6. Strings

- To manipulate strings, we can use the functions defined in the standard library **<string.h>**
- Some of the most relevant functions defined in this library are:

Prototype	Description
`int strcmp(const char *str1, const char *str2)`	Compares two strings character by character. If the strings are equal, the function returns `0`
`char *strcpy(char *dest, const char *source);`	Copies the string pointed by the source (2nd argument) to the destination (1st argument)
`size_t strlen(const char *str)`	Calculates the length of a given string
`char *strcat(char *dest, const char *source);`	Concatenates the destination string (2nd argument) and the source string (1st argument), and the result is stored in the destination string
`char *strtok(char *str, const char *delim);`	Breaks an string (1st argument) into a series of tokens using some delimiter (2nd argument)
`void *memset(void *str, int character, size_t size);`	Copies some character (2nd argument) to the first number of characters (3rd argument) of a string (1st argument)

6. Strings - Comparison

- Basic types can be compared using the == operator in C. For instance, characters:

```c
#include <stdio.h>

int main() {
    char character1 = 'a';
    char character2 = 'a';

    if (character1 == character2) {
        printf("'%c' and '%c' are EQUAL\n", character1, character2);
    }

    return 0;
}
```

```
'a' and 'a' are EQUAL
```

6. Strings - Comparison


```c
#include <stdio.h>

int main() {
    char str1[] = "hello";
    char str2[] = "hello";

    if (str1 == str2) {
        printf("\"%s\" and \"%s\" are EQUAL\n", str1, str2);
    }

    return 0;
}
```

Can we use the comparator operator to compare strings as well?

6. Strings - Comparison

- We must use **`strcmp`** for comparing strings in C:

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

int main() {
    char str1[] = "hello";
    char str2[] = "hello";

    // strcmp returns 0 when both strings are equal
    if (strcmp(str1, str2) == 0) {
        printf("\"%s\" and \"%s\" are EQUAL\n", str1, str2);
    }

    return 0;
}
```

```
"hello" and "hello" are EQUAL
```

6. Strings - Assignment

- String can be initialized (i.e., we can assign a value of an string variable when it is declared)
- But we cannot do an string assignment once it is declared:

```c
#include <stdio.h>

#define SIZE 80

int main() {
    char greetings[SIZE];
    greetings = "Hello";

    printf("%s\n", greetings);

    return 0;
}
```

```
error_string.c: In function 'main':
error_string.c:7:15: error: assignment to expression with array type
    7 |     greetings = "Hello";
      |               ^
```

6. Strings - Assignment

- To do string assignment, we use the function `strcpy`:

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

#define SIZE 80

int main() {
    char greetings[SIZE];
    strcpy(greetings, "Hello");

    printf("%s\n", greetings);

    return 0;
}
```

```
Hello
```

6. Strings - Length

- We can try to use the operator **`sizeof`** to calculate the length of an string, for instance:

```c
#include <stdio.h>

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

    printf("The size of the greetings string is %ld\n", size);

    return 0;
}
```

What is the value of the variable `size` in this example?

6. Strings - Length

- To calculate the length of an string in C (without counting the null-terminating character), we use `strlen`:

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

int main() {
    char str[] = "hello";
    size_t str_length = strlen(str);
    size_t str_size = sizeof(str) / sizeof(char);

    printf("The string \"%s\" has a length of %ld characters\n", str,
           str_length);
    printf("The string \"%s\" is stored in an array of %ld positions\n", str,
           str_size);
    return 0;
}
```

```
The string "hello" has a length of 5 characters
The string "hello" is stored in an array of 6 positions
```

6. Strings - Enumerated types

- As we know, enumerated types (**enum**) are a useful way to define a set of named integer constants
- However, C does not natively provide a native way to convert an **enum** value to its corresponding string representation (the name of the enumerator) or to convert a string back to the corresponding **enum** value

```c
typedef enum {
    ON, OFF
} key;


int main() {
    key my_key = ON;
    // TODO 1: how to convert my_key to string?

    char *my_string = "off";
    // TODO 2: how to convert my_string to enum?

    return 0;
}
```

For example. How to convert these enumerated types to string and vice versa?

To achieve this, we have to manually implement these conversions

6. Strings - Enumerated types


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

typedef enum {
    ON, OFF
} key;

typedef struct {
    key key;
    char *str;
} key_converter;

key_converter key_conv_arr[] = { { ON, "on" }, { OFF, "off" } };

key str2key(char *str) {
    for (int i = 0; i < sizeof(key_conv_arr) / sizeof(key_conv_arr[0]); i++) {
        if (strcmp(str, key_conv_arr[i].str) == 0) {
            return key_conv_arr[i].key;
        }
    }
    return OFF; // Default value
}

char* key2str(key key) {
    return key_conv_arr[key].str;
}
```

```c
int main() {
    key my_key = ON;
    // TODO 1: how to convert my_key to string?

    char *my_string = "off";
    // TODO 2: how to convert my_string to enum?

    // Solution 1:
    char *my_key_as_string = key2str(my_key);
    printf("1. Original enum: %d -- string value: %s\n", my_key,
           my_key_as_string);

    // Solution 2:
    key my_string_as_enum = str2key(my_string);
    printf("2. Original string: %s -- enum value: %d\n", my_string,
           my_string_as_enum);

    return 0;
}
```

```
1. Original enum: 0 -- string value: on
2. Original string: off -- enum value: 1
```

Table of contents

7. Structured data - Structs

- Structures (also called *structs*) are a way to group several related variables into the same variable
  - Unlike an array, a structure can contain different data types (**`int`**, **`char`**, etc.)
  - Each variable in the structure is known as a *member* of the structure
  - We use the **`struct`** keyword to create structures. We declare its members inside curly braces
  - We use the dot syntax (`.`) to access the members of a structure

```c
#include <stdio.h>

struct my_struct {
    int num;
    char letter;
};

int main() {
    struct my_struct s1;

    s1.num = 10;
    s1.letter = 'A';

    printf("My number: %d\n", s1.num);
    printf("My letter: %c\n", s1.letter);

    return 0;
}
```

```
My number: 10
My letter: A
```

7. Structured data - Structs

- We can use the keyword **typedef** to declare a type for an structure:

We can assign values to members of a structure variable at declaration time, in a single line in a comma-separated list inside curly braces {}

```c
#include <stdio.h>

typedef struct {
    int num;
    char letter;
} my_structure;

int main() {
    my_structure s1 = { 10, 'A' };

    printf("My number: %d\n", s1.num);
    printf("My letter: %c\n", s1.letter);

    return 0;
}
```

```
My number: 10
My letter: A
```

7. Structured data - Unions

- A union is a user-defined type similar to structs in C except for one key difference: structures allocate enough space to store all their members, whereas unions can only hold one member value at a time
  - We use the **union** keyword to create unions. We declare its members inside curly brackets (braces)
  - Unions provide an efficient way of using the same memory location for multiple-purpose, since all members share the same memory
  - Unions are used to save memory (e.g., in embedded systems) or when only some member is required at a time

```c
#include <stdio.h>

union job {
    float salary;
    int id;
};

int main() {
    union job my_job;

    my_job.salary = 50000.0;

    my_job.id = 55;

    printf("Worker id = %d\n", my_job.id);
    printf("Salary = %.1f\n", my_job.salary);

    return 0;
}
```

When `my_job.id` is assigned a value, `my_job.salary` will no longer hold `50000.0`

```
Worker id = 55
Salary = 0.0
```

7. Structured data - Unions

- The size of the union is based on the size of the largest member of the union

```c
#include <stdio.h>

#define MAX_STR 80

struct data_1 {
    int integer;
    char str[MAX_STR];
};

union data_2 {
    int integer;
    char str[MAX_STR];
};

int main() {
    struct data_1 d1;
    union data_2 d2;

    printf("The size of data_1 is %ld\n", sizeof(d1));
    printf("The size of data_2 is %ld\n", sizeof(d2));

    return 0;
}
```

```
The size of data_1 is 84
The size of data_2 is 80
```

Table of contents

8. Top-down design

- Structured programming languages (such as C) typically uses a design principle called **top-down**
- In the top-down design, the general aspects of a program are broken down into smaller components or functions
  - The development process starts by identifying the high-level functions or main components of the program
  - After establishing the high-level structure, the top-down approach involves progressively refining each part of the system into more specific and detailed sub-components or functions
- The top-down approach encourages modularity by breaking the system into manageable pieces or modules

8. Top-down design


```c
#include <stdio.h>

void display_menu();
void borrow_book();
void return_book();
void view_books();

int main() {
    int choice;
    do {
        display_menu();
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            borrow_book();
            break;
        case 2:
            return_book();
            break;
        case 3:
            view_books();
            break;
        case 4:
            printf("Exiting the library system.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
        }

    } while (choice != 4);

    return 0;
}
```

```c
void display_menu() {
    printf("Library Management System\n");
    printf("1. Borrow Book\n");
    printf("2. Return Book\n");
    printf("3. View Books\n");
    printf("4. Exit\n");
}

void borrow_book() {
    printf("Borrowing a book...\n");
    // Implementation details will be added later
}

void return_book() {
    printf("Returning a book...\n");
    // Implementation details will be added later
}

void view_books() {
    printf("Viewing all books...\n");
    // Implementation details will be added later
}
```

Table of contents

9. Takeaways

- A C function is defined by a name, parameter(s), and result type
- There are different types of operators in C: arithmetic, logical, relational, bitwise, assignment, and miscellaneous
- There is no boolean type in C. Instead, we use `0` for false and different than `0` for true
- There are two types of control flow statements in C: branching (**`if-else`** , **`switch`**) and looping (**`while`** , **`do-while`** , **`for`**)
- Arrays are collection of the data with the same type and stored at contiguous memory
- C does not have a native type to create string variables, instead, arrays of characters are used
- Structures (or structs) allows to group several related variables into the same variable
- Unions are similar to structs although unions can only hold one member value at a time
- The top-down approach in programming is a design methodology where the general aspects of a system are broken down into smaller components or functions