COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2023-January 2024

UNIVERSITÀ DEGLI STUDI DI PADOVA

Getting started - continue

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Recap basic sintax rules and some new ones

- Curly Braces for Code Blocks
- Semicolons at the End of Statements
- Case Sensitivity
- Symbolic Constants
- Arithmetic operators
- Equality operators
- If-then-else
- Loops
  - for
  - while
  - do-while
- Data Types
- Arguments - Call by Value

Curly braces

C uses curly braces { and } to define code blocks or compound statements. Code blocks group together multiple statements and are used with control structures like loops (for, while, do-while) and conditional statements (if, else, switch). It's crucial to enclose the statements within curly braces properly. This ensures that the compiler understands the scope of the code block. For example:

```
while (condition) {
    // Code block
    statement1;
    statement2;
}
```

Using curly braces correctly is essential for code clarity and to prevent logical errors in your C programs.

Semicolons at the End of Statements

Another critical syntax rule in C is the use of semicolons at the end of statements.

In C, each statement must be terminated with a semicolon (;).

Statements are the building blocks of a C program and include things like variable declarations, assignments, function calls, and control structures.

Failing to add a semicolon at the end of a statement will result in a syntax error and prevent your code from compiling.

Case Sensitivity

In C, one of the fundamental syntax rules is case sensitivity.
This means that C distinguishes between uppercase and lowercase letters.
For example, variables and function names in C are case-sensitive.
This means that ***myVariable*** and ***myvariable*** are treated as two different identifiers in C.
It's essential to use the correct case when referring to variables, functions, and other identifiers to avoid errors in your code.

Symbolic constants

Symbolic constants, also known as named constants or macros, are an essential feature in C that allow you to define meaningful names for values that do not change throughout your program. Instead of using literal values directly in your code, you can assign them to symbolic constants, which makes your code more readable, maintainable, and easier to update.

Symbolic constants are typically defined using the **#define** preprocessor directive, and they are often written in uppercase letters to distinguish them from variables.

Symbolic constants

Here's the basic syntax for defining a symbolic constant:

#define CONSTANT_NAME constant_value

For example, defining a symbolic constant for the value of pi:

#define PI 3.14159265359

Once defined, you can use the symbolic constant PI throughout your code wherever you need the value of pi, like this:

```
double radius = 5.0;
double circumference = 2 * PI * radius;
```

Advantages of symbolic Constants

Readability: using meaningful names like PI instead of raw numbers makes your code more self-explanatory and easier for others to understand.

Maintainability: if you need to change the value of a constant, you only have to update it in one place (the #define statement) rather than searching for and changing every occurrence of the literal value in your code.

Error Prevention: symbolic constants help prevent typos and errors caused by accidentally changing the value of a constant during maintenance.

Consistency: by using symbolic constants, you ensure that the same value is consistently used throughout your program, reducing the risk of inconsistencies and bugs.

Constants vs. Variables

It's important to note that symbolic constants, once defined, cannot be changed or modified during program execution. They remain constant throughout the program's lifetime. In contrast, variables can hold changing values. Symbolic constants are particularly useful for values that should not change, such as mathematical constants, configuration values, and flags.

An example in the next slide where MAX_SCORE is used as a symbolic constant to represent the maximum possible score, making the code more readable and maintainable.
Symbolic constants are a valuable tool in C programming for improving code quality, maintainability, and reliability. They help create code that is easier to understand, update, and debug.

Constants vs. Variables

Università degli Studi di Padova

```
#include <stdio.h>
#define MAX_SCORE 100
int main() {
    int studentScore = 85;

    if (studentScore > MAX_SCORE) {
        printf("Error: Invalid score. Score should be less than or equal to %d\n", MAX_SCORE);
    } else {
        printf("Student score: %d\n", studentScore);
    }

    return 0;
}
```

Arithmetic operators

Arithmetic operators in C are used to perform various mathematical calculations on numeric operands. These operators allow you to carry out fundamental arithmetic operations like addition, subtraction, multiplication, division, and more. Here are the primary arithmetic operators in C:

Addition (+): The addition operator is used to add two values together.
Example:

int sum = 5 + 3; // sum is assigned the value 8

Subtraction (-): The subtraction operator is used to subtract the right operand from the left operand.
Example:

int difference = 10 - 4; // difference is assigned the value 6

Arithmetic operators

Multiplication (*): The multiplication operator is used to multiply two values.

Example:

*int product = 6 * 7; // product is assigned the value 42*

Division (/): The division operator is used to divide the left operand by the right operand. If both operands are integers, the result is truncated to an integer (the decimal part is discarded).

Example:

int quotient = 15 / 4; // quotient is assigned the value 3

Modulus (%): The modulus operator calculates the remainder when the left operand is divided by the right operand. It's often used to check for divisibility and to work with cyclical patterns.

Example:

int remainder = 15 % 4; // remainder is assigned the value 3

Arithmetic operators

Increment (++) and **Decrement (--)**: These operators are used to increase or decrease the value of a variable by 1, respectively.

Example:

```
int num = 5;
num++; // num is incremented to 6
num--; // num is decremented back to 5
```

Compound Assignment Operators: C also provides compound assignment operators that combine an arithmetic operation with assignment in a single step. For example, += is equivalent to x = x + y, where x and y are variables.

Example:

```
int x = 10;
int y = 5;
x += y; // Equivalent to x = x + y, so x becomes 15
```

Arithmetic operators

Arithmetic operators are fundamental in C and are used extensively in mathematical calculations, expressions, and algorithms. They are essential for performing tasks ranging from simple arithmetic to complex mathematical computations in C programs.

Some exercises about arithmetic operators

Equality operators

In the C programming language, equality operators are used to compare two values and determine whether they are equal or not. These operators are essential for making decisions and controlling the flow of your program based on conditions. C provides two main equality operators: the equality operator (==) and the inequality operator (!=). Here's an explanation of how these operators work

The **equality operator (==)** is used to check if two values are equal.

It returns a Boolean result, either true (1) if the values are equal or false (0) if they are not. It is commonly used in conditional statements, such as if and while, to compare values.

It is important to note that == is a comparison operator, not an assignment operator. Using = for comparison will result in a syntax error.

Equality operators


```
int a = 5;
int b = 7;

if (a == b) {
    printf("a is equal to b.\n");
} else {
    printf("a is not equal to b.\n");
}
```

In this example, the == operator checks if a is equal to b and prints the appropriate message.

Equality operators

The **inequality operator (!=)** is used to check if two values are not equal.
It returns true (1) if the values are not equal and false (0) if they are equal.
Like the equality operator, it is used in conditional statements to test for inequality between variables or expressions.

```
int x = 10;
int y = 20;

if (x != y) {
    printf("x is not equal to y.\n");
} else {
    printf("x is equal to y.\n");
}
```

Equality operators

In the previous example, the != operator checks if x is not equal to y and prints the corresponding message.

Equality operators are fundamental for writing conditional logic in C. They allow you to make decisions based on whether values meet certain conditions, making your programs more versatile and adaptable. These operators can be used not only with numeric types but also with other data types, such as characters and pointers, to determine equality or inequality.

Equality operators

In the C programming language, the greater-than (>) and less-than (<) operators are used to compare two values and determine their relative order. These operators are essential for making decisions and controlling program flow based on conditions. Here's an explanation of how the greater-than and less-than operators work:

The **greater-than operator (>)** is used to check if the value on the left is greater than the value on the right.
It returns a Boolean result, either true (1) if the left value is greater than the right value or false (0) if it's not.
This operator is often used in conditional statements to compare values.

Equality operators

UNIVERSITÀ DEGLI STUDI DI PADOVA

```
int a = 5;
int b = 3;

if (a > b) {
    printf("a is greater than b.\n");
} else {
    printf("a is not greater than b.\n");
}
```

In this example, the > operator checks if a is greater than b and prints the appropriate message.

Equality operators

The **less-than operator (<)** is used to check if the value on the left is less than the value on the right.
Like the greater-than operator, it returns true (1) if the left value is less than the right value or false (0) if it's not.
It is commonly used in conditional statements to compare values.

```
int x = 10;
int y = 20;

if (x < y) {
    printf("x is less than y.\n");
} else {
    printf("x is not less than y.\n");
}
```

Equality operators

Greater-Than or Equal To Operator (>=) and Less-Than or Equal To Operator (<=):

In addition to the greater-than and less-than operators, C also provides "greater-than or equal to" (>=) and "less-than or equal to" (<=) operators.
The >= operator checks if the value on the left is greater than or equal to the value on the right.
The <= operator checks if the value on the left is less than or equal to the value on the right.
These operators are used when you want to include equality as a valid condition.

Equality operators

UNIVERSITÀ DEGLI STUDI DI PADOVA

```
int p = 8;
int q = 8;
if (p >= q) {
    printf("p is greater than or equal to q.\n");
} else {
    printf("p is less than q.\n");
}
```

Greater-than and less-than operators are fundamental for writing conditional logic in C. They allow you to make decisions based on the relative order of values, making your programs more flexible and capable of handling different scenarios. These operators can be used with various data types, including numeric types and characters, to determine the relative order of values.

if-then-else

In the C programming language, the "if-then-else" statement is a fundamental control structure used to make decisions in your code. It allows you to execute different blocks of code based on a condition's evaluation. Here's an explanation of how the "if-then-else" statement works in C:

if Statement: The basic form of the "if" statement consists of the keyword "if" followed by a condition enclosed in parentheses. The condition is an expression that results in a boolean value (true or false). If the condition is true, the code block enclosed in curly braces immediately following the "if" statement is executed. If the condition is false, the code block is skipped.

```
if (condition) {
    // Code to execute if the condition is true
}
```

if-then-else

else Statement: The "else" statement is used in conjunction with the "if" statement to specify a block of code to execute if the condition in the "if" statement is false. You can use "else" immediately after an "if" block, and its code block will be executed if the "if" condition is false.

```
if (condition) {
    // Code to execute if the condition is true
} else {
    // Code to execute if the condition is false
}
```

if-then-else

else if Statement: In many cases, you need to check multiple conditions in sequence. You can use "else if" statements to evaluate additional conditions after the initial "if" condition. The code block associated with the first true condition encountered will be executed, and subsequent "else if" conditions are not evaluated.

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if neither condition1 nor condition2 is true
}
```

if-then-else

Here's a simple example to illustrate the use of "if-then-else" in C:

```
#include <stdio.h>
int main() {
    int number = 5;
    if (number > 0) {
        printf("The number is positive.\n");
    } else if (number < 0) {
        printf("The number is negative.\n");
    } else {
        printf("The number is zero.\n");
    }
    return 0;
}
```

Loops in C programming

Loops are essential control structures in the C programming language allowing you to execute a block of code repeatedly and to create repetitive tasks and automate processes in your programs. They are used when you need to perform a specific task or set of tasks multiple times without writing the same code over and over. They are crucial for handling various programming challenges and scenarios.

Loops are powerful tools in C programming, and they are used extensively for tasks like iterating through arrays, processing data, and implementing algorithms. Understanding when and how to use each type of loop is essential for writing efficient and effective C programs.

C provides three main types of loops: for, while, and do-while.

for loop

The **for** loop is used when you know in advance how many times you want to execute a block of code. It consists of three parts:

- Initialization: Setting an initial value.
- Condition: A condition that determines whether to continue looping.
- Iteration: Updating the loop control variable.

Here's the basic structure of a **for** loop:

```
for (initialization; condition; iteration) {
    // Code to be repeated
}
```

For example, to print numbers from 1 to 5:

```
for (int i = 1; i <= 5; i++) {
    printf("%d\n", i);
}
```

while loop

The **while** loop is used when you want to execute a block of code as long as a condition is true. It checks the condition before entering the loop, which means the code inside the loop may never run if the condition is false from the start. Here's the basic structure of a **while** loop:

```
while (condition) {
    // Code to be repeated
}
```

For example, to print all powers of 2 until you get over a thousand:

```
int result = 1;
while (result < 1000) {
    printf("power: %d\n", result);
    result = 2*result;
}
```

do-while loop

The **do-while** loop is similar to the while loop, but it guarantees that the code inside the loop is executed at least once because it checks the condition after executing the code block. Here's the basic structure of a **do-while** loop:

```
do {
    // Code to be repeated
} while (condition);
```

For example, to print all powers of 2 until you get over a thousand:

```
int result = 1;
do {
    printf("power: %d\n", result);
    result *= 2;
}
while (result < 1000);
```

Loop Control Statements

In C, you can control loops using statements like **break** (to exit a loop prematurely).

In the example, we use a for loop to find the first even number in a given range. Once we find the first even number, we use the break statement to exit the loop early.

```
#include <stdio.h>
int main() {
    int start = 1;
    int end = 10;
    for (int i = start; i <= end; i++) {
        if (i % 2 == 0) {
            printf("Found the first even number: %d\n", i);
            break; // Exit the loop as soon as an even number is found
        }
    }
    return 0;
}
```

Loop Control Statements

In C, you can control loops using statements like **continue** (to skip the rest of the current iteration and continue with the next).
In the example, we use a for loop to print all numbers in a given range except for those that are divisible by 3. We use the continue statement to skip the printing of numbers that meet the specified condition.

```
#include <stdio.h>
int main() {
    int start = 1;
    int end = 10;
    for (int i = start; i <= end; i++) {
        if (i % 3 == 0) {
            continue; // Skip this iteration if the number is divisible by 3
        }
        printf("%d\n", i);
    }
    return 0;
}
```