COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

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
- If-then-else
- Switch-case statement
- Ternary statement
- Goto and labels
- Symbolic Constants
- Arithmetic operators
- Equality operators

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.

bool data type

Programs that follow a linear sequence: Statement 1, Statement 2, etc. until the last statement is executed and the program terminates are very limited in the problems they can solve. Let's see constructs that allow program statements to be optionally executed, depending on the context (input) of the program's execution.

C/C++ supports the non-numeric data type bool, which stands for Boolean, from the name of the British mathematician George Boole. Compared to the numeric types, the bool type is very simple, it can represent only two values: true or false.

In C99 and later, the bool type is defined in the header file <stdbool.h>. You need to include this header to use bool in your code.

bool data type


```
#include <stdbool.h>

bool is_even = true;
```

bool can hold two values:

- true (non-zero, usually 1)
- false (0)

```
bool is_sunny = false;
bool has_data = true;

int is_even = 1;  // true
int is_sunny = 0; // false
```

bool data type

```
#include <iostream>
int main() {
// Declare some Boolean variables
bool a = true, b = false;
std::cout << "a = " << a << ", b = " << b << '\n';
// Mix integers and Booleans
a = 1;
b = 1;
std::cout << "a = " << a << ", b = " << b << '\n';
a = 1725; // Warning issued
b = -19; // Warning issued
std::cout << "a = " << a << ", b = " << b << '\n';
}
```

```
a = 1, b = 0
a = 1, b = 1
a = 1, b = 1
```

```
/* Condition the stream to
display booleans as words*/
std::cout << std::boolalpha;
```

```
a = true, b = false
a = true, b = true
a = true, b = true
```

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;
}
```

Testing conditions

C has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false

- **Use switch to specify many alternative blocks of code to be executed**

- **Use the ternary operator as a shorthand way to write an if-else statement**

Switch statement

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. It provides a way to streamline multiple if-else conditions when the same variable or expression is being compared to different values.

```
switch (expression) {
    case constant1:
        // Code to be executed if expression matches constant1
        break;

    case constant2:
        // Code to be executed if expression matches constant2
        break;

    // Additional cases as needed

    default:
        // Code to be executed if expression does not match any constant
}
```

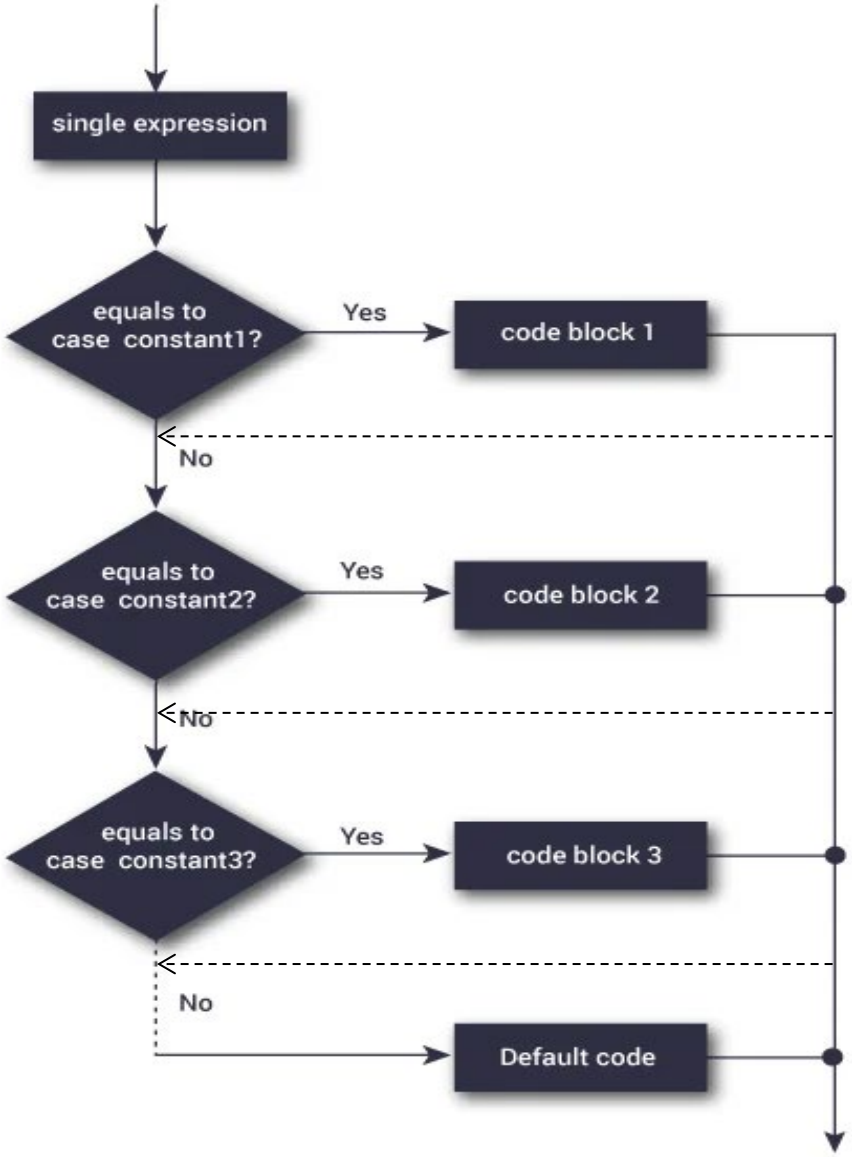
Switch statement

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- All case expressions must be different.
- Each case is labeled by one or more integer-valued constants or constant expressions or enumerated types.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- If a case matches the expression value, execution starts at that case until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement. This will stop the execution of more code and case testing inside the block.

Switch statement

- The break statement is crucial to avoid "fall-through." Without break, after a matching case executes, the program will continue executing all subsequent cases, regardless of whether they match or not.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached. Falling through from one case to another is not robust, better avoiding it.
- The case labeled default is executed if none of the other cases are satisfied.
  - A default is optional; if it isn't there and if none of the cases match, no action at all takes place.
- As a matter of good form, put a break after the last case (usually the default one) even though it’s logically unnecessary. If another case gets added at the end, this break could save your program correctness.
- Cases and the default clause can occur in any order.

Switch statement flowchart



Switch statement

Let's consider an example where we want to determine the day of the week based on a numeric code. In this case, the numeric code represents the day of the week, and we'll use a switch statement to handle different cases.

```
#include <stdio.h>
int main() {
   int dayCode = 3;  // Example code for Wednesday
   switch (dayCode) {
      case 1:
         printf("Monday\n");
         break;
      case 2:
         printf("Tuesday\n");
         break;
      case 3:
         printf("Wednesday\n");
         break;
      case 4:
         printf("Thursday\n");
         break;
      case 5:
         printf("Friday\n");
         break;
      case 6:
         printf("Saturday\n");
         break;
      case 7:
         printf("Sunday\n");
         break;
      default:
         printf("Invalid day code\n");
   }
   return 0;
}
```

Switch statement

Università degli Studi di Padova

```
#include <stdio.h>

int main () {

   /* local variable definition */
   char grade = 'B';

   switch(grade) {
      case 'A' :
         printf("Excellent!\n" );
         break;
      case 'B' :
      case 'C' :
         printf("Well done\n" );
         break;
      case 'D' :
         printf("You passed\n" );
         break;
      case 'F' :
         printf("Better try again\n" );
         break;
      default :
         printf("Invalid grade\n" );
   }

   printf("Your grade is  %c\n", grade );

   return 0;
}
```

Conditional or ternary operator ‘?:’

The statements

```
if (a > b)
z = a;
else
z = b;
```

compute in z the maximum of a and b. The conditional expression, written with the ternary operator ``?:'', provides an alternate way to write this and similar constructions. The conditional operator can be in the form

variable = expression1 ? expression2 : expression3;

or in the form

variable = (condition) ? expression2 : expression3;

or, simply,

expression1 ? expression2 : expression3;

Conditional or ternary operator '?:'

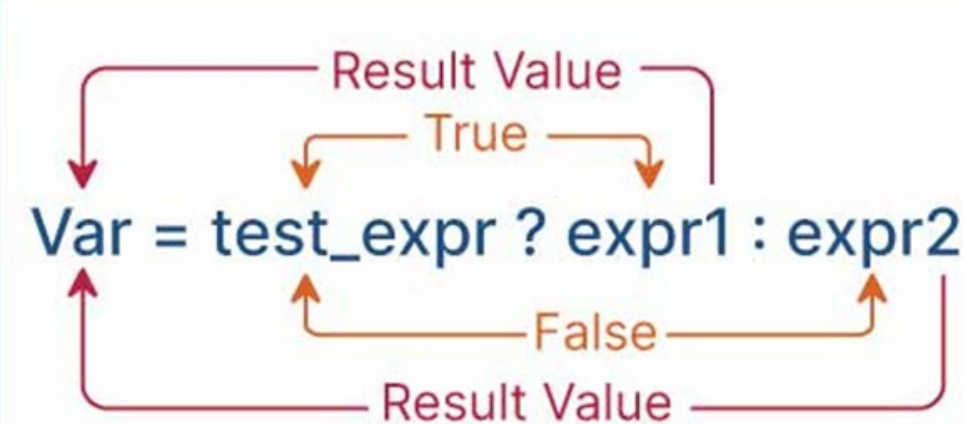
In the form

expression1 ? expression2 : expression3

the expression expression1 is evaluated first. If it is non-zero (true), then the expression expression2 is evaluated, and that is the value of the conditional expression. Otherwise expression3 is evaluated, and that is the value. Only one of expression2 and expression3 is evaluated.

Thus to set z to the maximum of a and b, we can use the following statement

```
z = (a > b) ? a : b; /* z = max(a, b) */
```



Conditional or ternary operator ‘?:’


```
#include <stdio.h>

int main() {
    int n1, n2;

    printf("Enter two numbers: ");
    scanf("%d %d", &n1, &n2);

    // Using the ternary operator to check which number is greater
    (n1 > n2) ? printf("%d > %d", n1, n2) : printf("%d > %d", n2, n1);

    return 0;
}
```

Conditional or ternary operator ‘?:’


```
#include <stdio.h>

int main() {
    int age;

    printf("Enter your age: ");
    scanf("%d", &age);

    // Using the ternary operator to check if the person is a minor or not
    const char* status = (age < 18) ? "minor" : "adult";
    printf("You are: %s", status);

    return 0;
}
```

Conditional or ternary operator ‘?:’

Advantages of Ternary Operators Over If-Else Statements.

- They allow for more compact and efficient code.
- They can improve code readability by reducing the number of lines needed.

However, whether or not to use ternary operators is ultimately a matter of personal preference and coding style.

You can nest ternary operators, but this can reduce readability. For instance:

int result = (x > 0) ? ((x == 1) ? 100 : 200) : -1;

This should be used with care to avoid making the code hard to understand.

The ternary operator should **only** be used for simple conditions and expressions. Overusing it for complex conditions can make the code harder to follow.

Goto and labels

The goto statement in C is a jump statement that allows the program's control to jump to a labeled statement within the same function or block.

While the use of goto is generally discouraged due to its potential to create complex and less readable code, and indeed the it is never necessary, and in practice it is almost always easy to write code without it, the goto statement can be employed in specific scenarios where other control flow structures are not suitable.
The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop.

Goto and labels

While this example might be better solved using other control structures, it demonstrates the basic syntax of goto.

```
#include <stdio.h>
int main() {
    int i, j;
    for (i = 1; i <= 5; ++i) {
        for (j = 1; j <= 5; ++j) {
            if (i == 3 && j == 3) {
                // Use goto to break out of nested loops
                goto endLoop;
            }
            printf("%d %d\n", i, j);
        }
    }
    // Label to jump to
    endLoop:
    printf("Loop ended\n");

    return 0;
}
```

Goto syntax and flow diagram

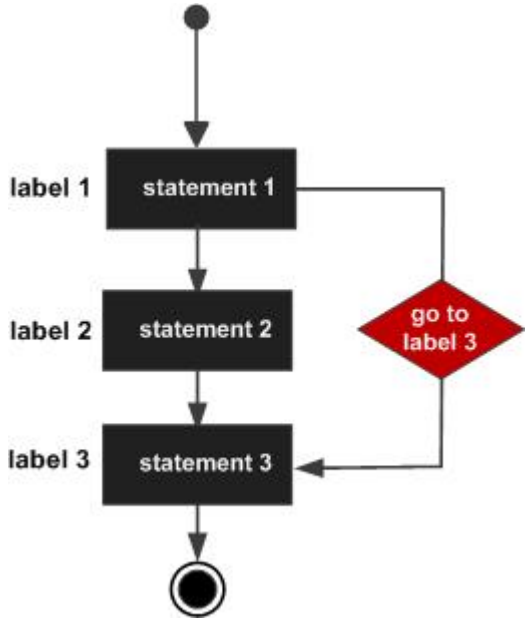
The syntax for a goto statement in C is as follows –

goto label;

...

label: statement;

Here label can be any plain text except a C keyword and it can be set anywhere in the C program above or below to goto statement.



Reasons to avoid goto

The use of goto statement may lead to code that is buggy and hard to follow. For example,

```
one:
for (i = 0; i < number; ++i)
{
    test += i;
    goto two;
}
two:
if (test > 5) {
  goto three;
}
... ... ...
```

Should you use goto?
If you think the use of goto statement simplifies your program, you can use it. That being said, goto is rarely useful and you can create any C program without using goto altogether.

Here's a quote from Bjarne Stroustrup, creator of C++, "**The fact that 'goto' can do anything is exactly why we don't use it.**"

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


```
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.