CRAFT #2

Introduction to Programming

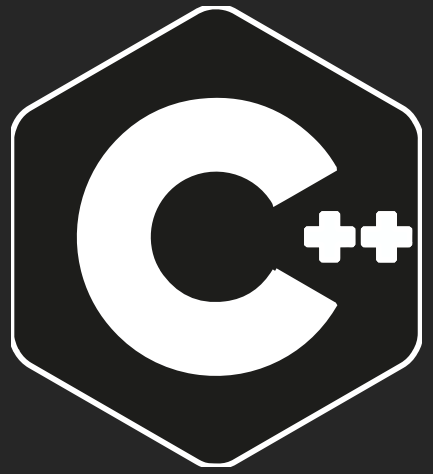
Warm-up

Micael S. Couceiro micael@ingeniarius.pt

01/07/2025

CONTENT

- Why C?
- Software: CodeBlocks (or any other)
- C Programming



C LANGUAGE

WHY C?

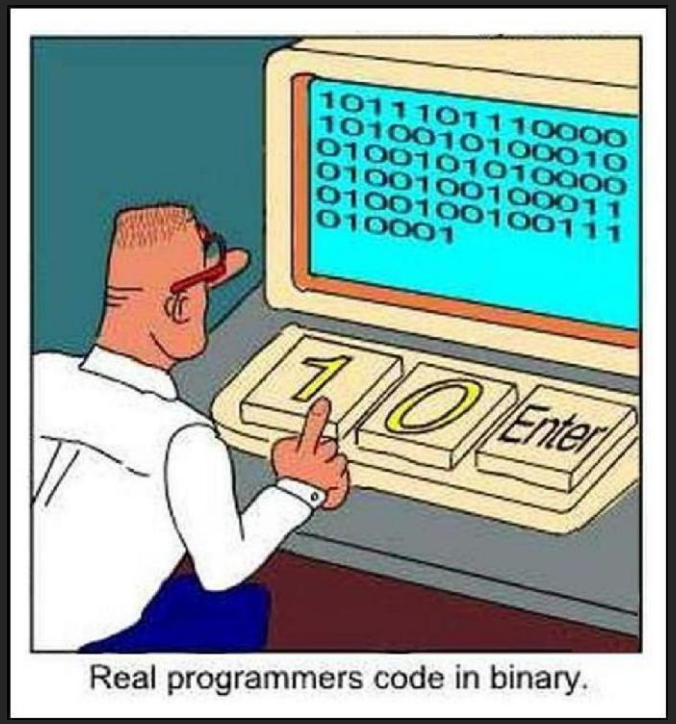
The C language forms the basis for many programming languages, such as C++, Objective-C, C# and others, which add an object oriented 'layer' over C itself. In order to program those languages, ***you first need to understand C****.*

WHY C?

- C Language is old school (from the 60's) and still one of the most popular programming languages in the world
- It is very flexible and powerful, with applicability ranging from software apps to embedded systems
- The good: it allows to develop high-level applications while interacting directly with hardware (e.g., memory, …)
- The bad: it is complex when compared to high(er)-level languages, such as Python, Ruby, JS, …; therefore, it is easier to make mistakes!

WHY C?

- Still better than programming in the language of machines…



Real programmers code in binary.

SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

- Open Source
- Cross-platform (Linux, Mac and Windows)
- No interpreted languages or proprietary libs needed
- Extensible through plugins

SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

- Download at Ingeniarius e-learning platform or

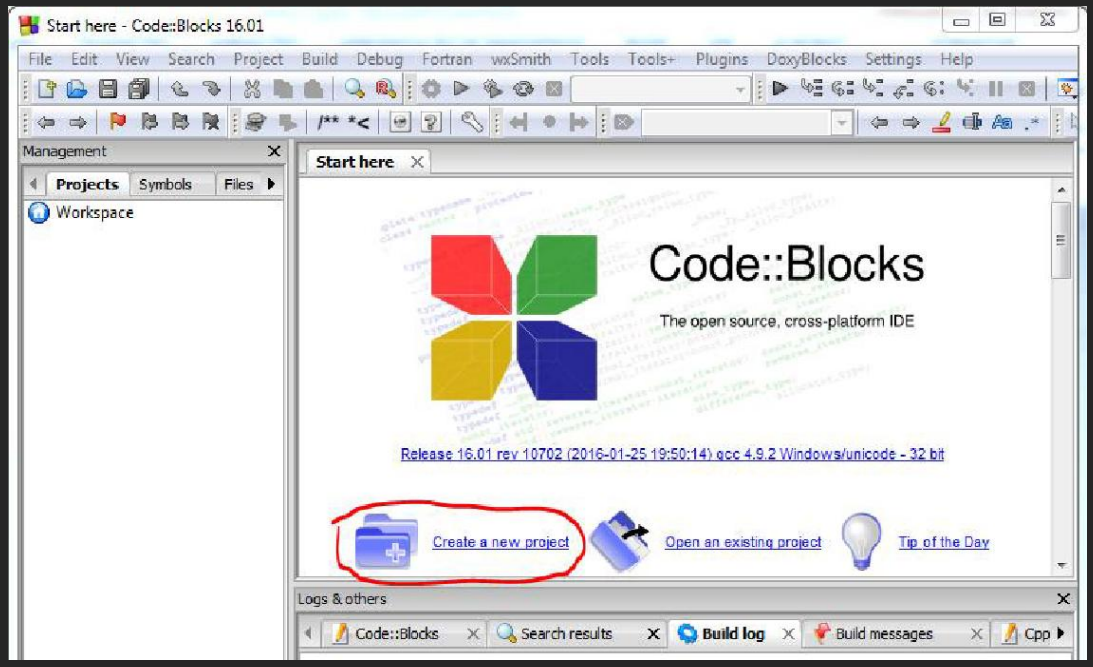
  http://www.codeblocks.org/downloads/26

- **Do not forget to install a compiler** (e.g., MinGW)

SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

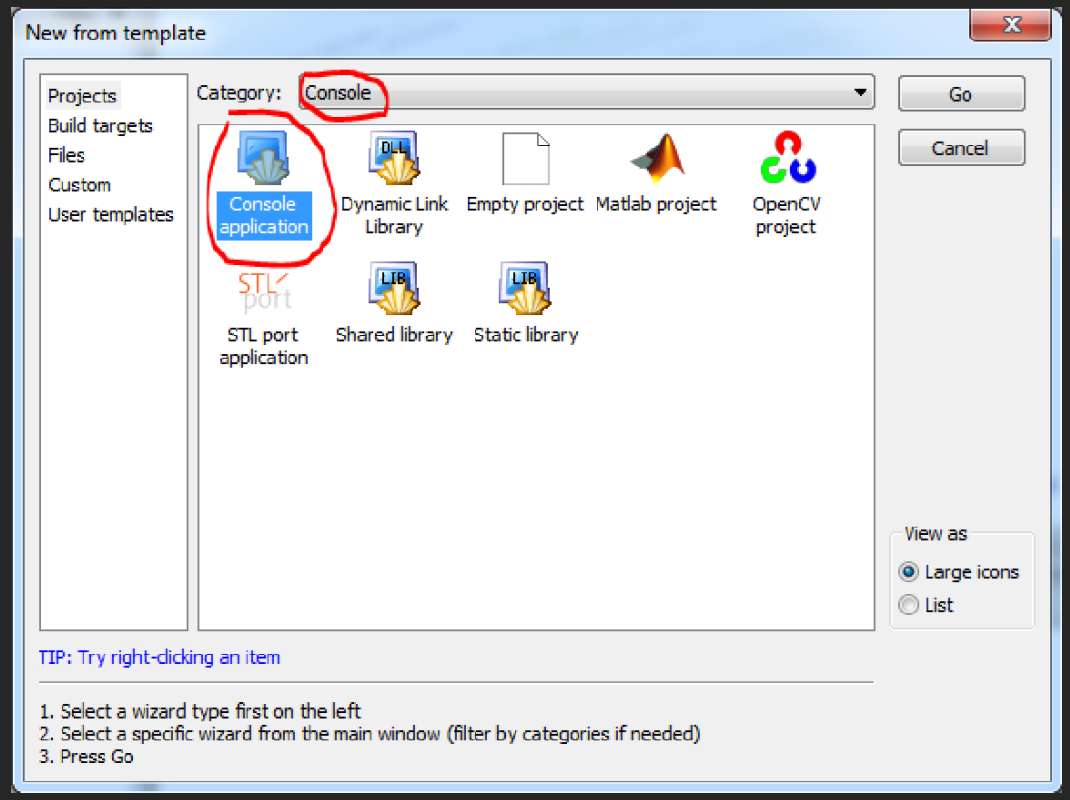
Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

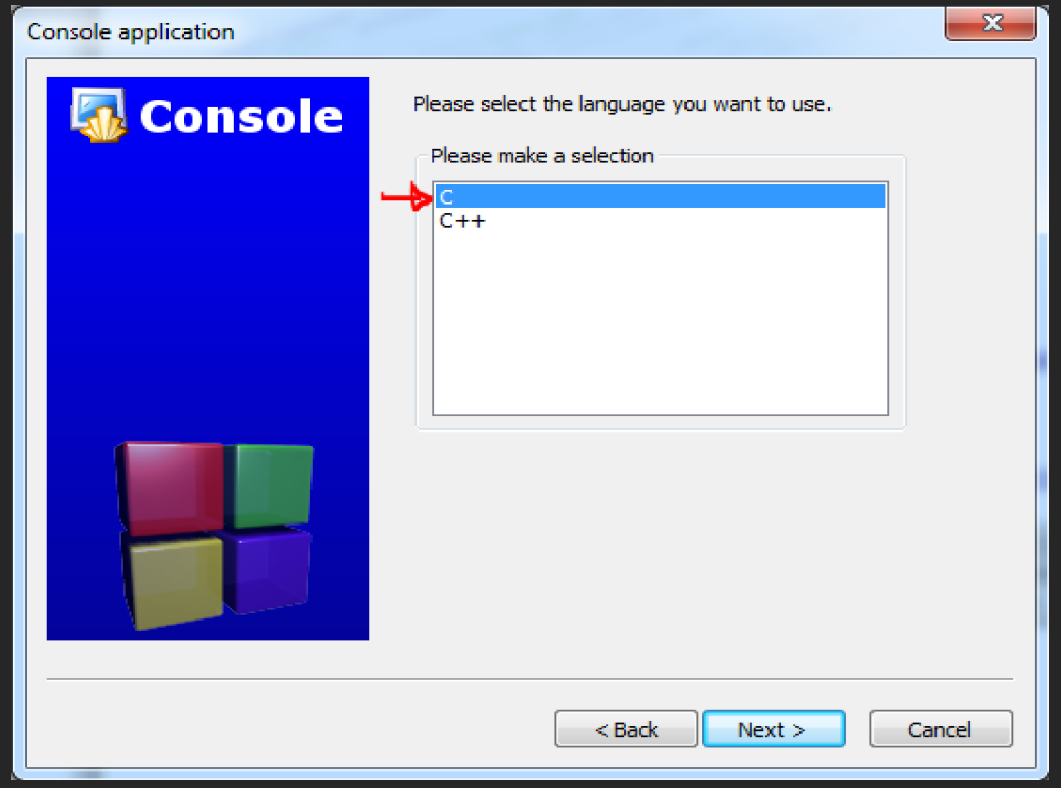
Code::Blocks

Create your first C-program

SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

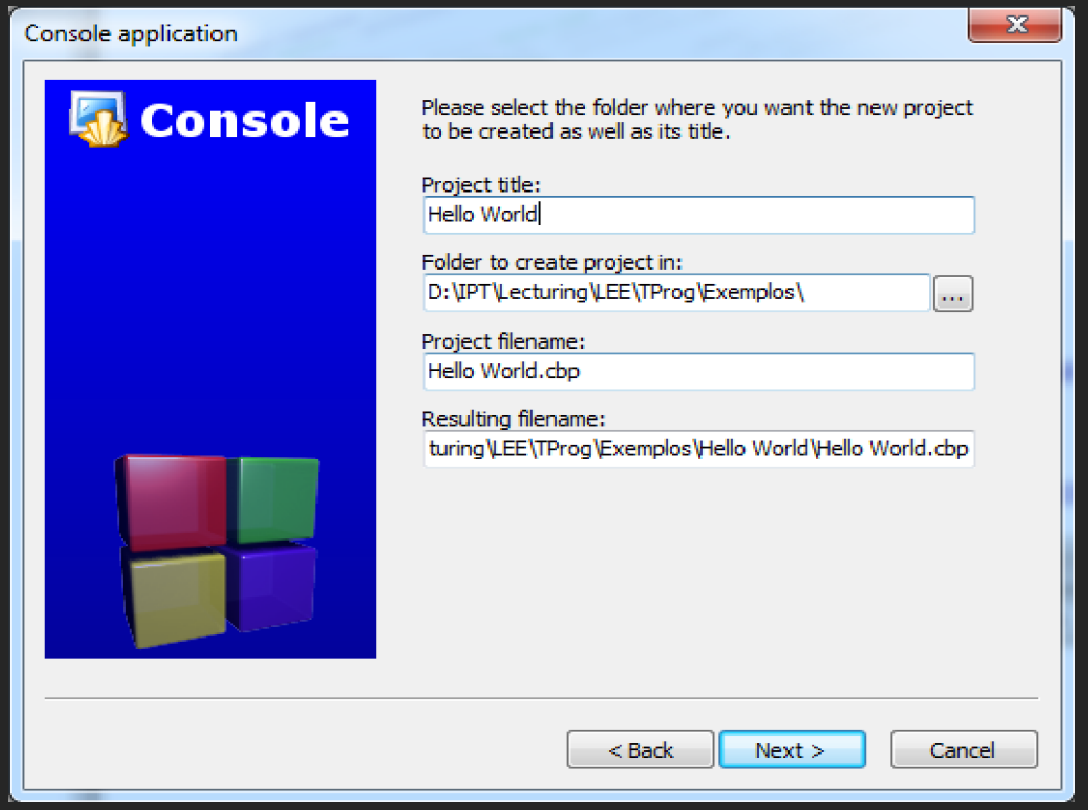
Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

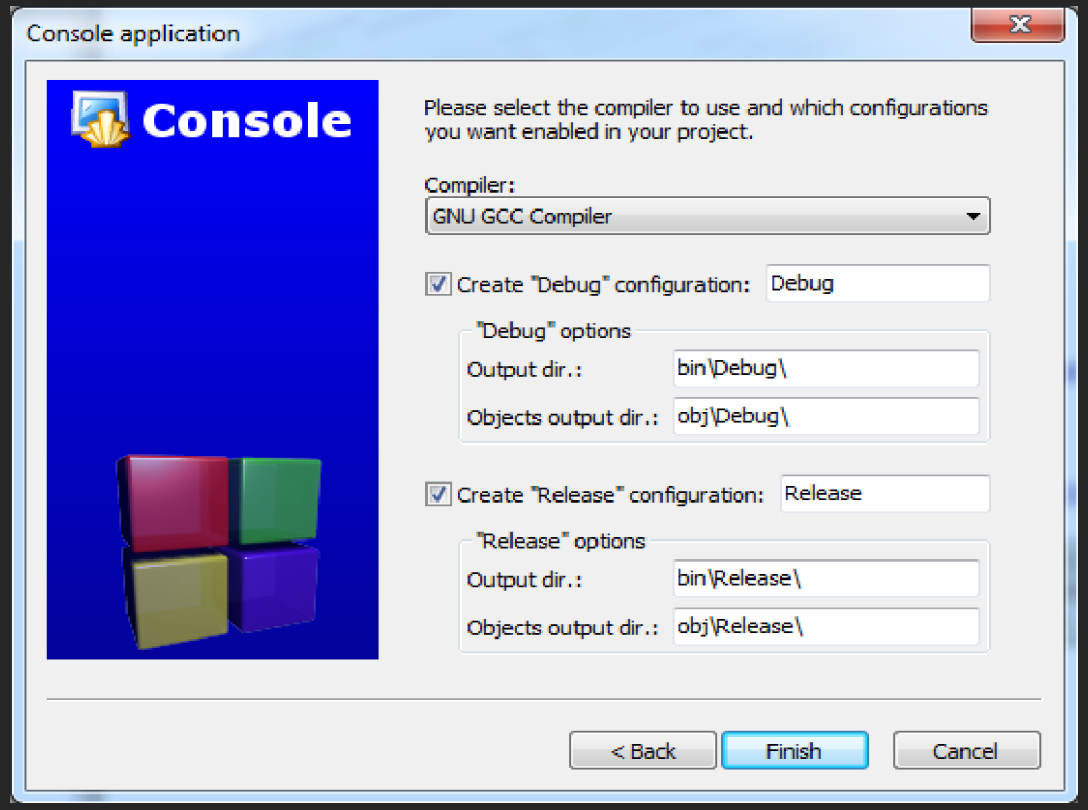
Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

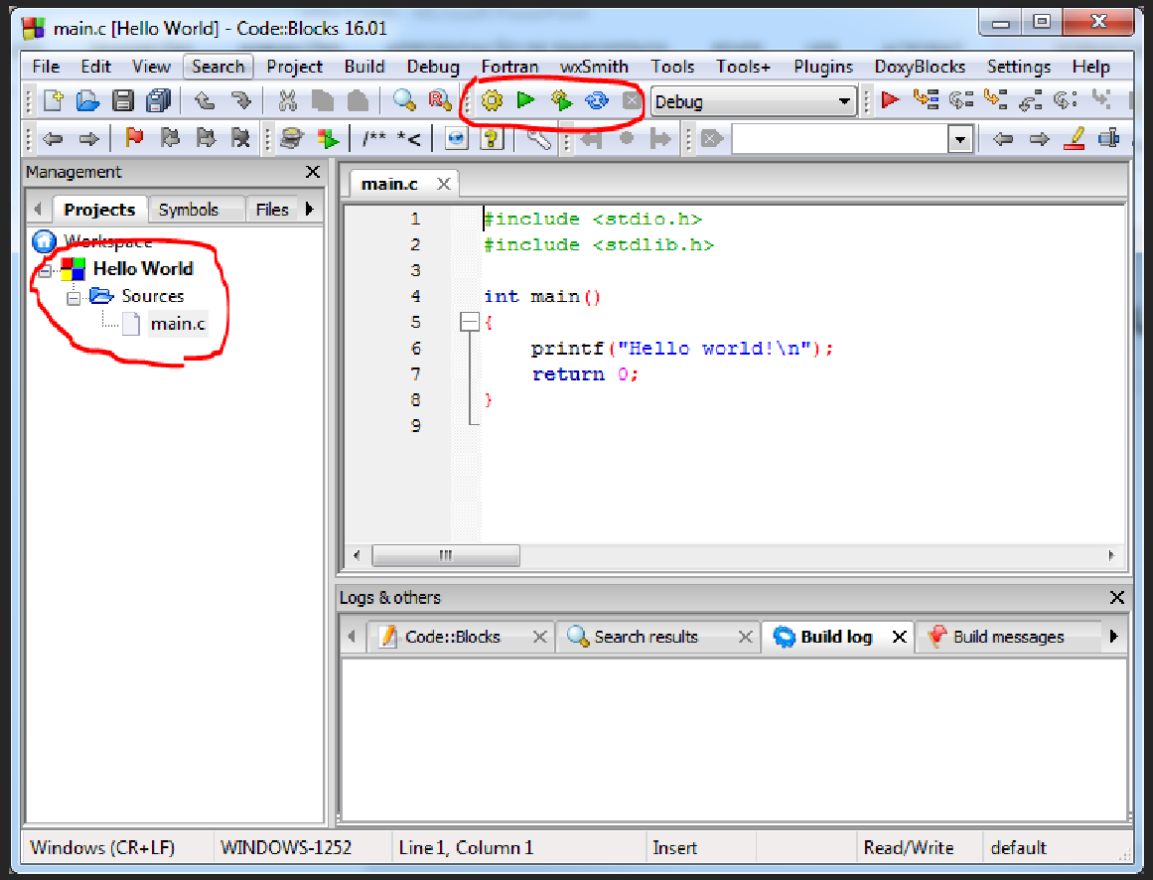
Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

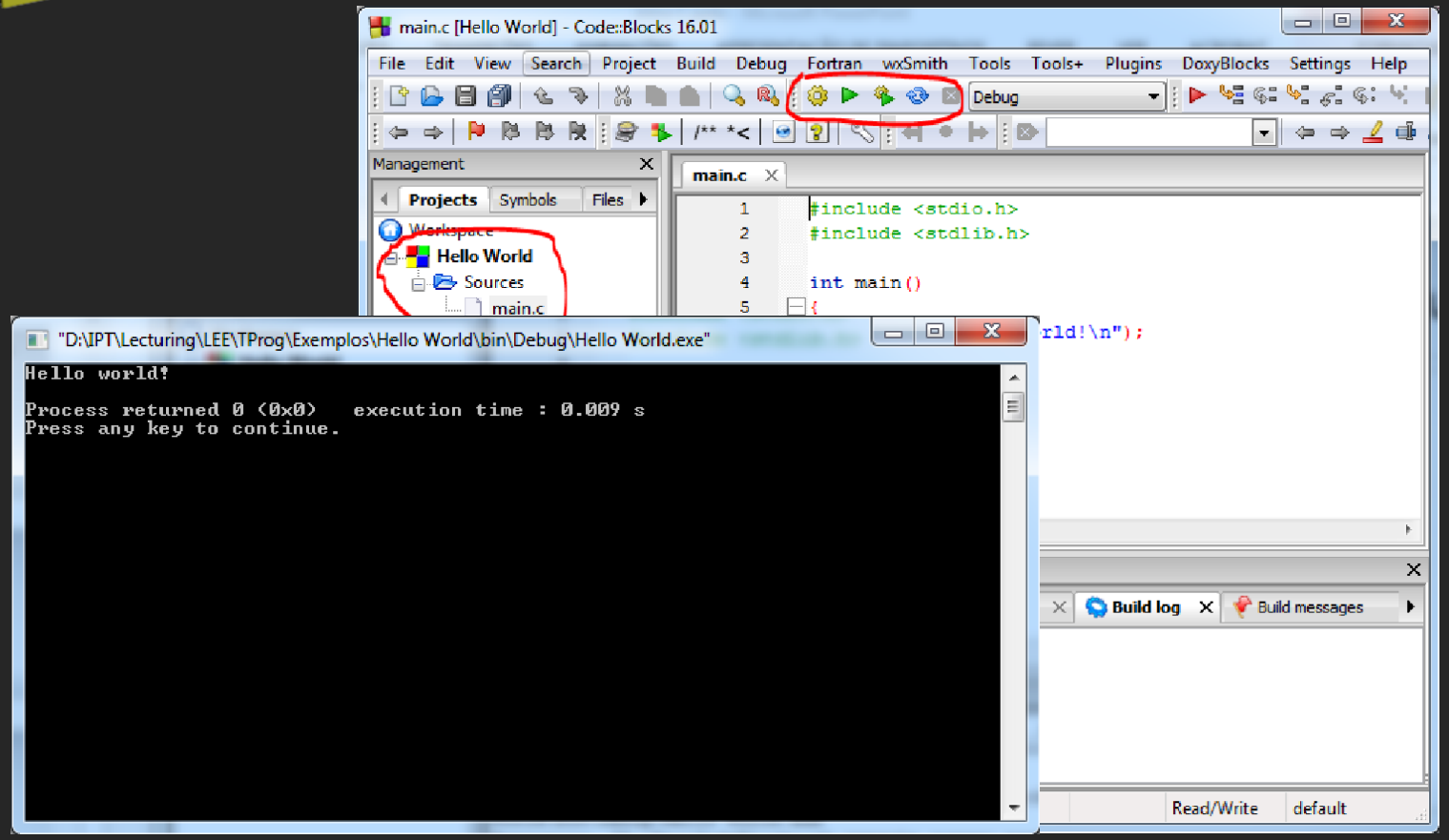
Create your first C-program



SOFTWARE: CODEBLOCKS (OR ANY OTHER)

Code::Blocks

Create your first C-program



C PROGRAMMING

- Comments (line 4)
- Pre-processing directives (line 1)
- Functions (line 3)
- Variables (not in this code)
- Statements and expressions (lines 5 and 7)

main.c

```
1  #include <stdio.h>
2
3  int main() {
4      /* um comentario */
5      printf("Hello, World!\n");
6
7      return 0;
8  }
```

C PROGRAMMING

Comments

```c
//Single line comment
/*multi
 line
 comment*/
int main()
{
    printf("code goes here");
}
```

Comments are not processed by the compiler

There are two ways to write a comment in C

- // - single line comment
- /* and */ - multi-line comment

C PROGRAMMING

Pre-processing directives

```
1 #include <stdio.h>
```

Provides access to *printf* and other input/output functions

Tells the compiler to include the content from the *stdio.h* library before compiling the code

- Why is it needed?
- Why is it between <>?

C PROGRAMMING

Pre-processing directives

Provides access to *printf* and other input/output functions

```
1 #include <stdio.h>
```

- H-files are called “header files” containing functions, variables and constants
  - They contain only the declaration of the functions (prototypes)
  - The implementation is usually in a C-file with the same name
- By including the H-file in our code, its functions become available
- The library should be included between <> as long as it belongs to the standard C libraries (e.g., <*string.h*>); otherwise, if it is a user-created library, then it should be included between “” (e.g., “*mystring.h*”)

C PROGRAMMING

Functions

```
3 int main(int argc, char **argv)
```

Main function of your code – this is where the program will start

- Group of statements that together perform a task - every C program has at least one function, which is *main()*
  - Return Type – A function may return a value (e.g., int) or not (void) (output of the function)
  - Function Name – The function name and the parameter list together constitute the function signature
  - Parameters – When a function is invoked, you pass a value to the parameter or argument (inputs of the function)
  - Function Body – The function body contains a collection of statements that define what the function does

C PROGRAMMING

Functions

```
6       return 0;
```

In this case, the function returns 0 – the return value needs to be of the same type as the return type of the function

Functions cannot return arrays

- Non-void functions need a return, which ends the execution of the function.
- For the *main()* function it ends the whole program

C PROGRAMMING

Variables

```
17 |   int a = 10;
```

This is a variable called **a**, of type *int*, initialized with the value 10

- A variable is nothing but a name given to a storage area that our programs can manipulate
- A variable can be **local**, if defined within a function, and only accessed in such function, or **global**, if defined outside functions, being accessed in any function of the same file
- Each variable in C has a **specific type**, which determines:
  - the size and layout of the variable's memory
  - the range of values that can be stored within that memory
  - the set of operations that can be applied to the variable.

C PROGRAMMING

Variables

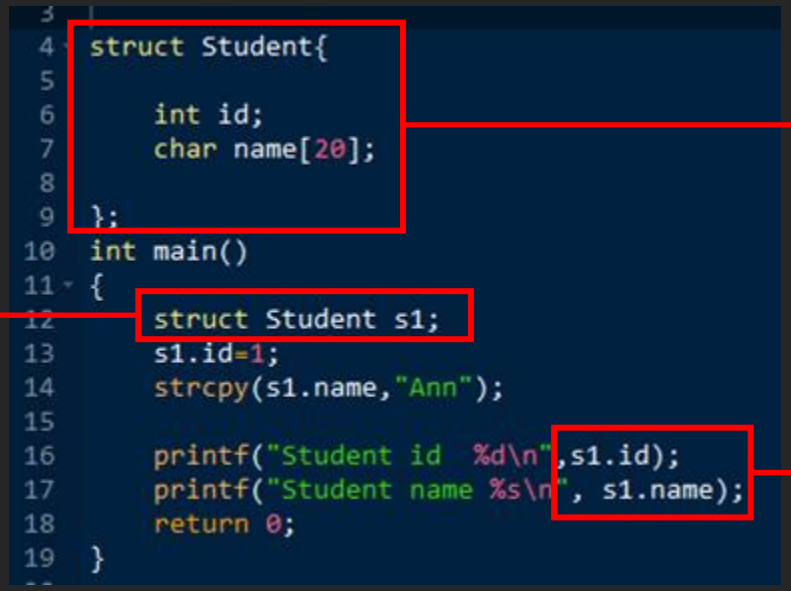
- Types of variables can be:

Type	Storage size	Value range	Precision
char	1 byte	-128 to 127 or 0 to 255	0 decimal places
unsigned char	1 byte	0 to 255	0 decimal places
signed char	1 byte	-128 to 127	0 decimal places
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	0 decimal places
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295	0 decimal places
short	2 bytes	-32,768 to 32,767	0 decimal places
unsigned short	2 bytes	0 to 65,535	0 decimal places
long	8 bytes	-9223372036854775808 to 9223372036854775807	0 decimal places
unsigned long	8 bytes	0 to 18446744073709551615	0 decimal places
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

C PROGRAMMING

Variables - structures

- There are special “variables” defined as **structure…**



The name of the user-defined type of variable is *Student*

Here we create variable *s1* of the type *Student*

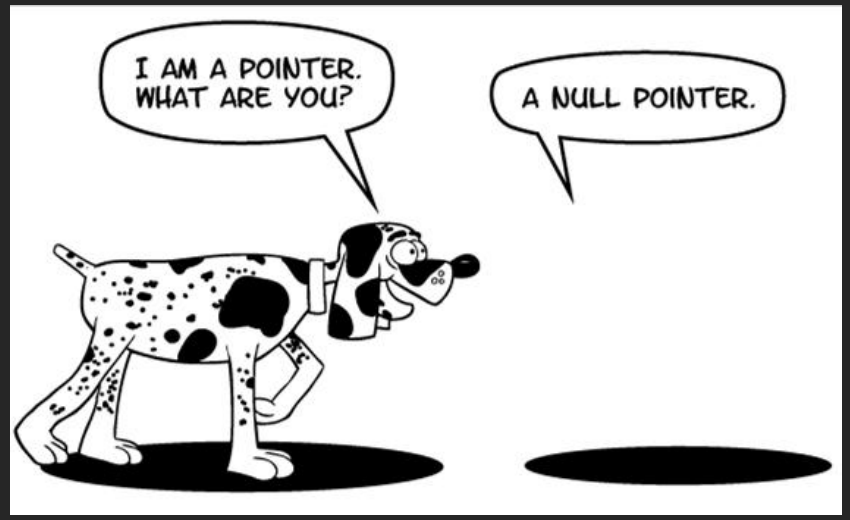
- **Structure** is a user-defined data type that allows to **combine data items of different kinds**.

C PROGRAMMING

Variables - pointers

- And then we have **pointers**…

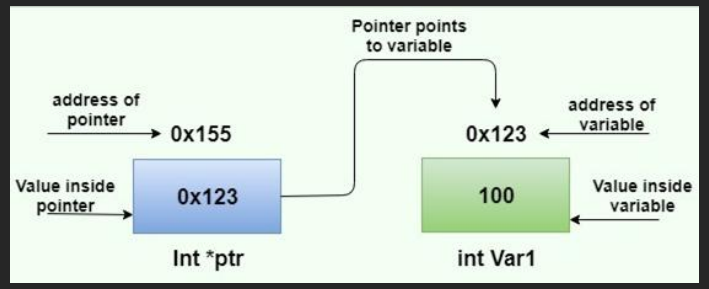
…the nightmare of many (for some reason)



C PROGRAMMING

Variables - pointers

- As previously stated, a variable is nothing but a name given to a storage area that you can manipulate



- What if you could manipulate memory directly? You could:
  - ✓ Directly manipulate a given variable, even among different functions
  - ✓ Change parameters within functions
  - ✓ Manipulate dynamic arrays
  - ✓ Optimize memory
  - ✓ Send and "return" arrays to and from functions

This is what pointers do.

C PROGRAMMING

Variables - pointers

```c
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < MAX; i++) {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }

   return 0;
}
```

C PROGRAMMING

Variables - pointers

```c
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main () {

   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */
   printf("Average value is: %f\n", avg );
   return 0;
}
```

```c
double getAverage(int *arr, int size) {

   int  i, sum = 0;
   double avg;

   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }

   avg = (double)sum / size;
   return avg;
}
```

C PROGRAMMING

Variables - pointers

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

/* function to generate and return random numbers. */
int * getRandom( ) {

   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );

   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf("%d\n", r[i] );
   }

   return r;
}
```

```c
/* main function to call above defined function */
int main () {

   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();

   for ( i = 0; i < 10; i++ ) {
      printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }

   return 0;
}
```

C PROGRAMMING

Statements and expressions

- Operators
  - Arithmetic
  - Relational
  - Logical

- Decision-Making
  - IF-ELSE
  - SWITCH

- Loops
  - WHILE
  - DO-WHILE
  - FOR

C PROGRAMMING

Operators: Arithmetic

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

C PROGRAMMING

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) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	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) is not true.
<=	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) is true.

C PROGRAMMING

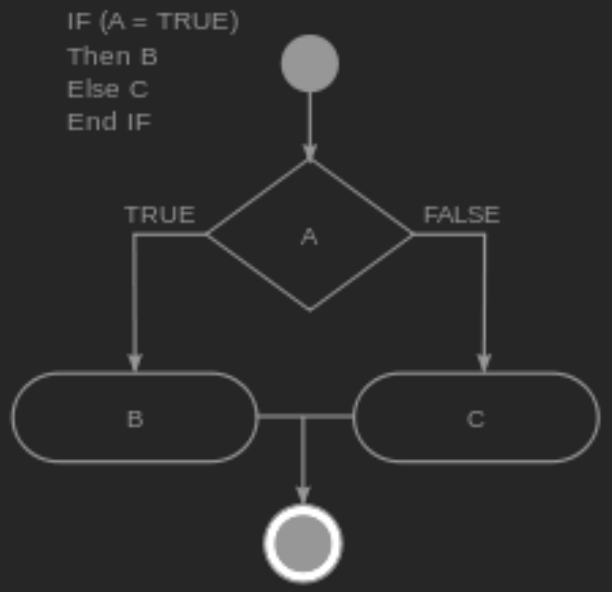
Operators: Logical

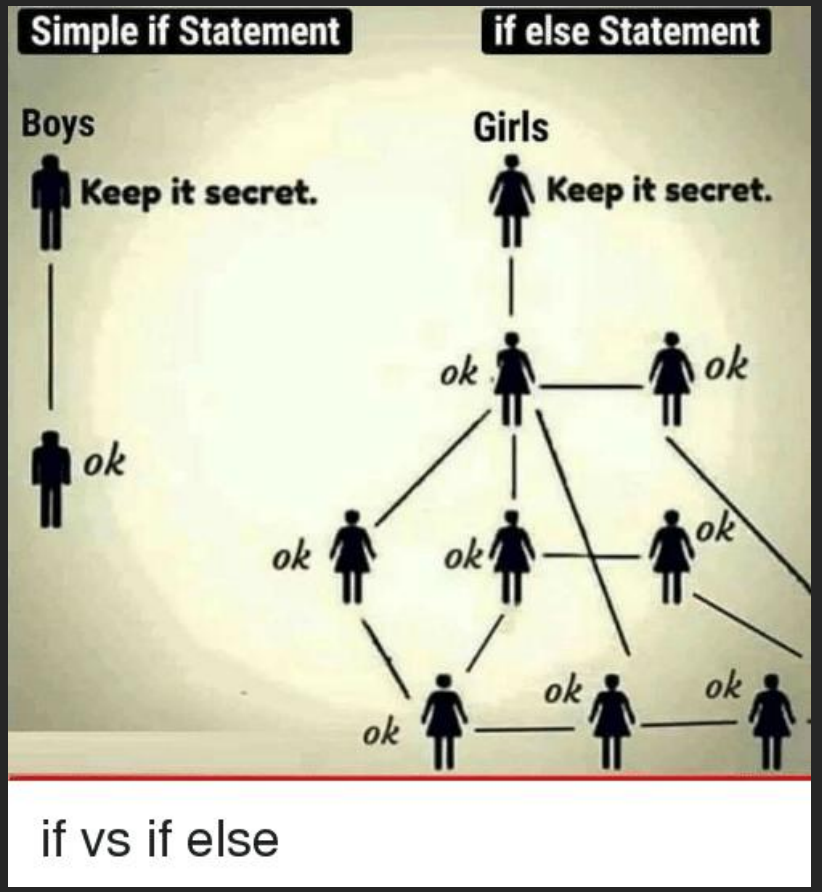
Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. 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 && B) is true.

C PROGRAMMING

Decision-Making: IF-ELSE

- An IF statement consists of a boolean expression followed by one or more statements. An IF statement can be followed by an optional ELSE statement, which executes when the boolean expression is false.





if vs if else

C PROGRAMMING

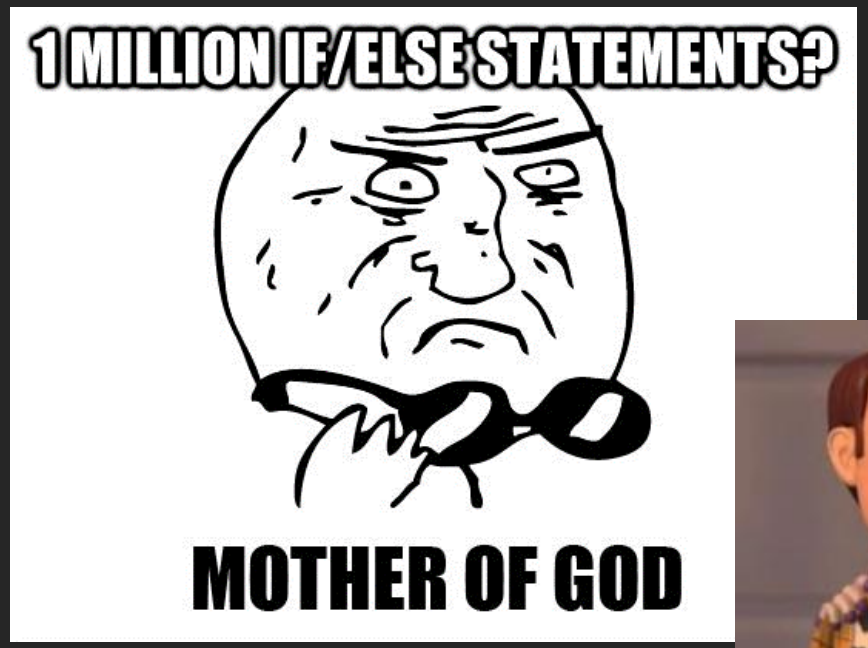
Decision-Making: IF-ELSE

```c
#include <stdio.h>

int main (void) {
    int powerLevel = 9000;

    if (powerLevel > 9000) {
        printf ("It's over 9000!!\n");
    } else {
        printf ("You are weak!\n");
    }

    return 0;
}
```

BODY BLOCK

C PROGRAMMING

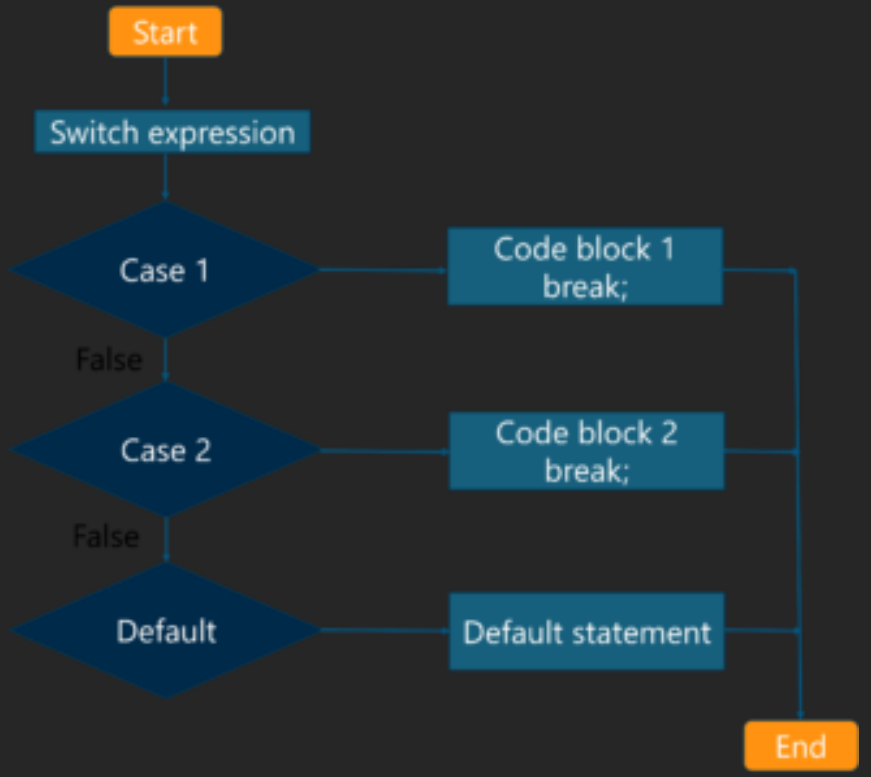
Decision-Making: SWITCH



C PROGRAMMING

Decision-Making: SWITCH

- A switch statement allows a variable to be tested for equality against a list of values. It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.



C PROGRAMMING

Decision-Making: SWITCH

```
What operation you want to perform?
 1. Addition
 2. Subtraction
 3. Multiplication
 4. Division
4

Enter two numbers to be divide
0
0

 zero / zero = Undefined form!

Enter two numbers to be divide
5
0

a number / zero = Undeterminate form!

Enter two numbers to be divide
5
5

The qoutient of 5.000000 / 5.000000 = 1.000000

Do you want more? Y or N
```

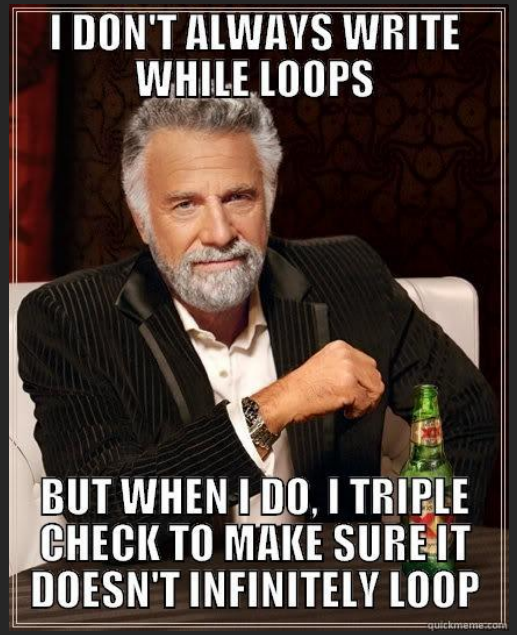
wednesday.c

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

//C Calculator with menus
//Using switch case and some if else
//AnonLonewulf ftw

main()
{
        float a, b, c;
        int n;
        char choice;
        char var[10];

        printf("\n What operation you want to perform?\n 1. Addition\n 2. Subtraction\n 3. Multiplication\n 4. Division\n");
        scanf("%d", &n);

        switch(n)
        {
                case 1:
                        printf("\nEnter two numbers to be added\n");
                        scanf("%f %f", &a, &b);

                        c=a+b;
                        printf("\nThe total of %f + %f = %f\n", a, b, c);
                        printf("\nDo you want more? Y or N\n");
                        scanf("%s", &choice);
                        if(choice=='y'||choice=='Y')
                                main();
                        break;
                case 2:
                        printf("\nEnter two numbers to be subtract\n");
                        scanf("%f %f", &a, &b);
```

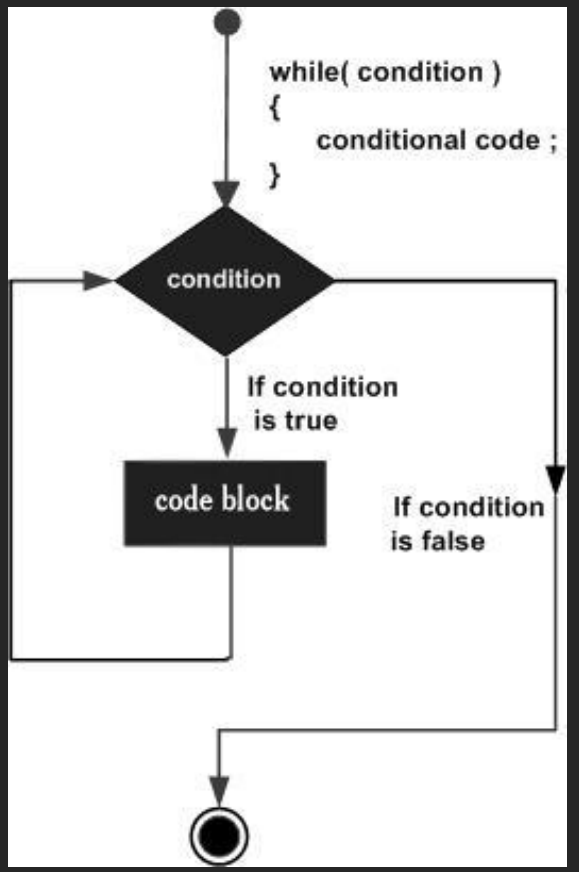
C PROGRAMMING

Loops: WHILE

- Repeats a statement or group of statements while a given condition is true
- **It tests the condition before executing the loop body**




```
while(alive)
{
 eat();
 coffee();//sleep();
 code();
}
```



C PROGRAMMING

Loops: WHILE

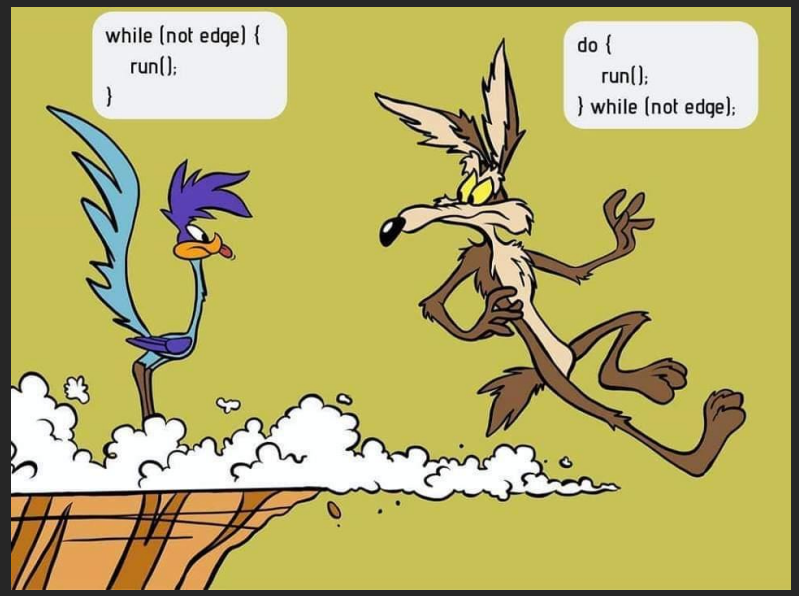
```
int main () {

   /* local variable definition */
   int a = 10;

   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

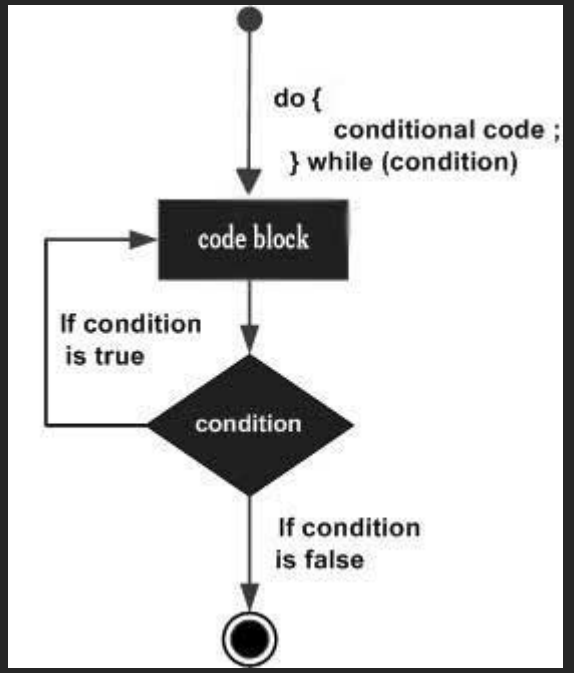
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C PROGRAMMING

Loops: DO-WHILE

- Unlike WHILE, which test the loop condition at the top of the loop, the DO-WHILE checks its condition at the bottom of the loop
- **A DO-WHILE loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time**





C PROGRAMMING

Loops: DO-WHILE

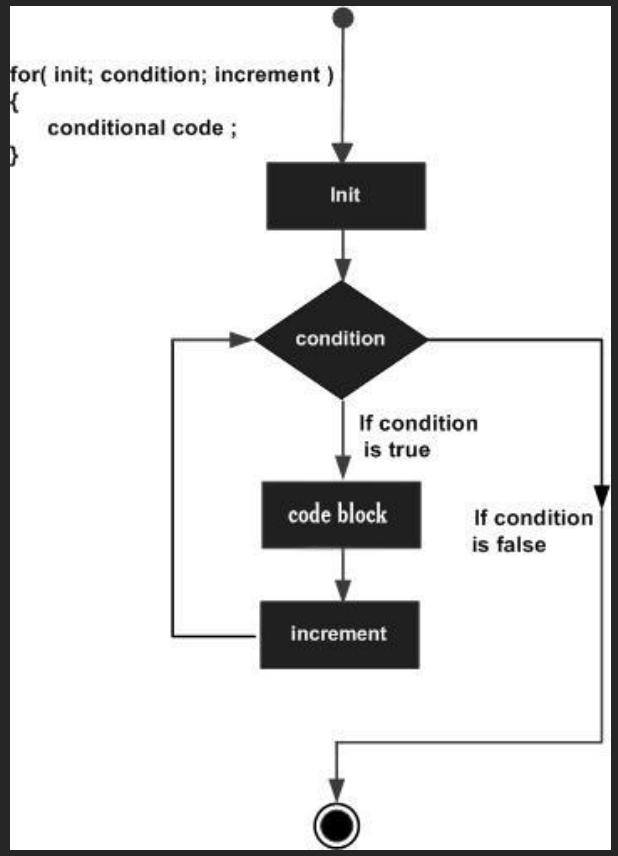
```
2
3  int main () {
4
5     /* local variable definition */
6     int a = 10;
7
8     /* do loop execution */
9     do {
10       printf("value of a: %d\n", a);
11       a = a + 1;
12    }while( a < 20 );
13
14    return 0;
15 }
16
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C PROGRAMMING

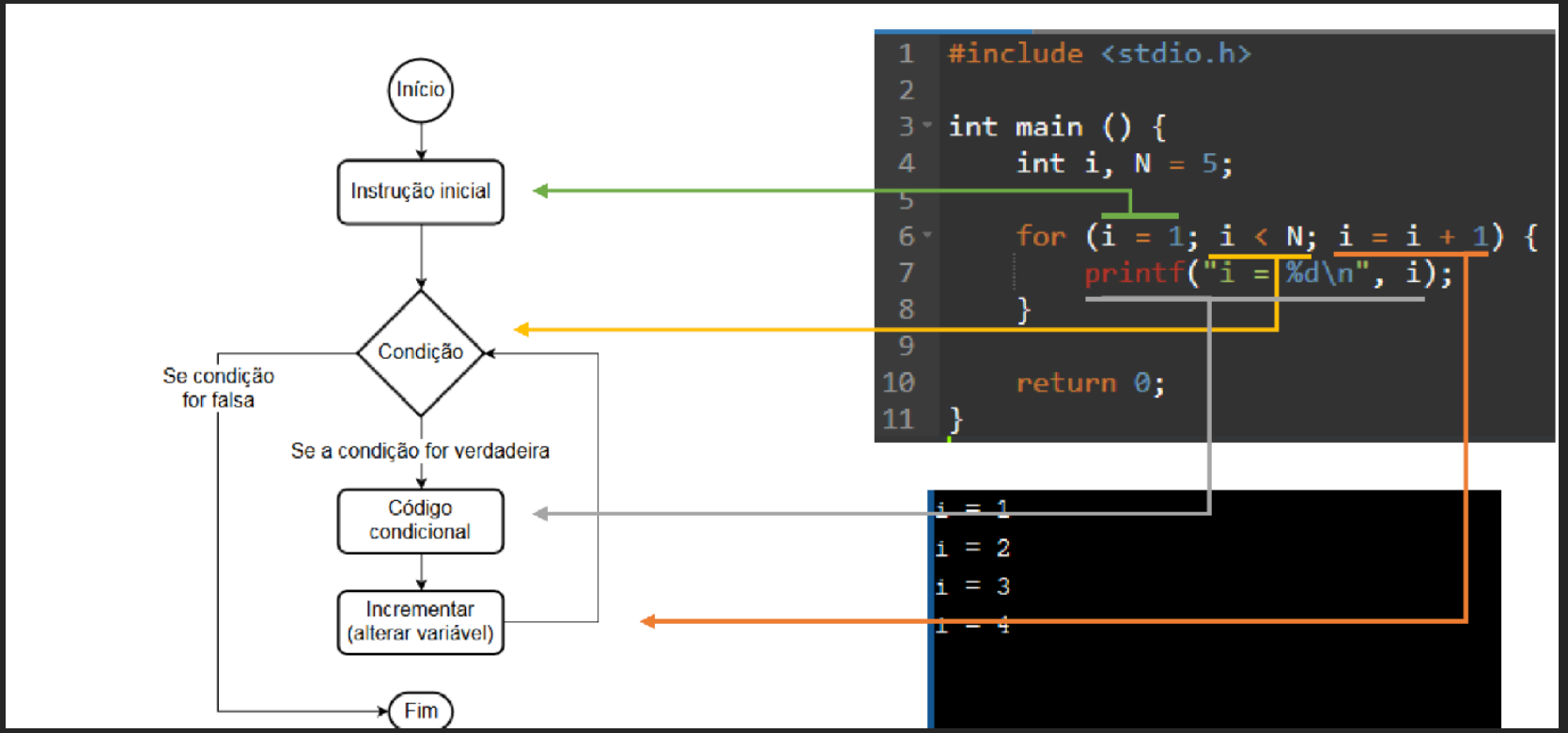
Loops: FOR

- FOR is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times
- As opposed to WHILE and DO-WHILE, FOR has a slightly more complex flow of control [for (init; condition; increment) { }]
  - The init is executed first and only once, allowing to declare and initialize any loop control variables
  - The condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps out of the FOR.
  - After the body of the loop executes, the flow of control jumps back up to the increment statement, updating any loop control variables.
  - The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the FOR terminates.



C PROGRAMMING

Loops: FOR



C PROGRAMMING

Loops: FOR

```
2
3  int main () {
4
5     int a;
6
7     /* for loop execution */
8     for( a = 10; a < 20; a = a + 1 ){
9        printf("value of a: %d\n", a);
10    }
11
12    return 0;
13 }
```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

C++ PROGRAMMING

C++ is an object-oriented programming language.
Everything in C++ is associated with **classes** and **objects**, along with its attributes and methods.

For example: in real life, a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake.

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

C++ PROGRAMMING

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create **objects**.
To create an object of MyClass, specify the class name, followed by the object name.

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

C++ PROGRAMMING

Methods are functions that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

Note: You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

C++ PROGRAMMING

Outside class definition example

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Inside class definition example

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

C++ PROGRAMMING

As in functions, you can also add parameters to the method:

```
#include <iostream>
using namespace std;

class Car {
  public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
  return maxSpeed;
}

int main() {
  Car myObj; // Create an object of Car
  cout << myObj.speed(200); // Call the method with an argument
  return 0;
}
```

C++ PROGRAMMING

A constructor in C++ is a special method that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses ():

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass
  return 0;
}
```

```
class Car {         // The class
  public:           // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

C++ PROGRAMMING

```cpp
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

C++ PROGRAMMING

The public keyword is an access specifier. Access specifiers define how the members (attributes and methods) of a class can be accessed. If the members are public they can be accessed and modified from outside the code.

In C++, there are three access specifiers:

public - members are accessible from outside the class
private - members cannot be accessed (or viewed) from outside the class
protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};
```

C++ PROGRAMMING

To hide the data from users you must declare class variables/attributes as **private** (cannot be accessed from outside the class).

If you want others to read or modify the value of a private member, you can provide **public** get and set methods

To access a private attribute,
use public "get" and "set" methods:

```
int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

```
class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};
```

MOST COMMON C-FUNCTIONS

Function	Library	Function Prototype	Description
abs	stdlib.h	int abs(int n);	Calculates the absolute value of an integer argument n.
acos	math.h	double acos(double x);	Calculates the arc cosine of x.
asctime_r	time.h	char *asctime_r (const struct tm *tm, char *buf);	Converts tm that is stored as a structure to a character string. (Restartable version of asctime.)
asin	math.h	double asin(double x);	Calculates the arc sine of x.
atan	math.h	double atan(double x);	Calculates the arc tangent of x.
atan2	math.h	double atan2(double y, double x);	Calculates the arc tangent of y/x.
atof	stdlib.h	double atof(const char *string);	Converts string to a double-precision floating-point value.
atoi	stdlib.h	int atoi(const char *string);	Converts string to an integer.
atol	stdlib.h	long int atol(const char *string);	Converts string to a long integer.
ceil	math.h	double ceil(double x);	Calculates the double value representing the smallest integer that is greater than or equal to x.
cos	math.h	double cos(double x);	Calculates the cosine of x.
exp	math.h	double exp(double x);	Calculates the exponential function of a floating-point argument x.
fabs	math.h	double fabs(double x);	Calculates the absolute value of a floating-point argument x.
floor	math.h	double floor(double x);	Calculates the floating-point value representing the largest integer less than or equal to x.
fmod	math.h	double fmod(double x, double y);	Calculates the floating-point remainder of x/y.
getc	stdio.h	int getc(FILE *stream);	Reads a single character from the input stream.
getchar	stdio.h	int getchar(void);	Reads a single character from stdin.
gets	stdio.h	char *gets(char *buffer);	Reads a string from stdin, and stores it in buffer.
labs	stdlib.h	long int labs(long int n);	Calculates the absolute value of n.
log	math.h	double log(double x);	Calculates the natural logarithm of x.
log10	math.h	double log10(double x);	Calculates the base 10 logarithm of x.
malloc	stdlib.h	void *malloc(size_t size);	Reserves a block of storage.

MOST COMMON C-FUNCTIONS

Function	Library	Function Prototype	Description
pow	math.h	double pow(double x, double y);	Calculates the value x to the power y.
printf	stdio.h	int printf(const char *format-string, arg-list);	Formats and prints characters and values to stdout.
putc	stdio.h	int putc(int c, FILE *stream);	Prints c to the output stream.
putchar	stdio.h	int putchar(int c);	Prints c to stdout.
puts	stdio.h	int puts(const char *string);	Prints a string to stdout.
qsort	stdlib.h	void qsort(void *base, size_t num, size_t width, int(*compare)(const void *element1, const void *element2));	Performs a quick sort of an array of num elements, each of width bytes in size.
rand	stdlib.h	int rand(void);	Returns a pseudo-random integer.
scanf	stdio.h	int scanf(const char *format-string, arg-list);	Reads data from stdin into locations given by arg-list.
sin	math.h	double sin(double x);	Calculates the sine of x.
sqrt	math.h	double sqrt(double x);	Calculates the square root of x.
srand	stdlib.h	void srand(unsigned int seed);	Sets the seed for the pseudo-random number generator.
strcasecmp	strings.h	int srtcasecmp(const char *string1, const char *string2);	Compares strings without case sensitivity.
strcat	string.h	char *strcat(char *string1, const char *string2);	Concatenates string2 to string1.
strchr	string.h	char *strchr(const char *string, int c);	Locates the first occurrence of c in string.
strcmp	string.h	int strcmp(const char *string1, const char *string2);	Compares the value of string1 to string2.
strcpy	string.h	char *strcpy(char *string1, const char *string2);	Copies string2 into string1.
strlen	string.h	size_t strlen(const char *string);	Calculates the length of string.
strtok	string.h	char *strtok(char *string1, const char *string2);	Locates the next token in string1 delimited by the next character in string2.
tan	math.h	double tan(double x);	Calculates the tangent of x.

CRAFT #2

Thank you

Micael S. Couceiro micael@ingeniarius.pt

01/07/2025