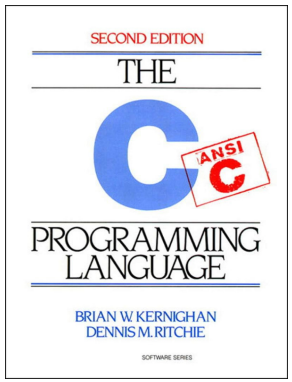
C Fundamentals

Pedro Ribeiro

DCC/FCUP

2024/2025



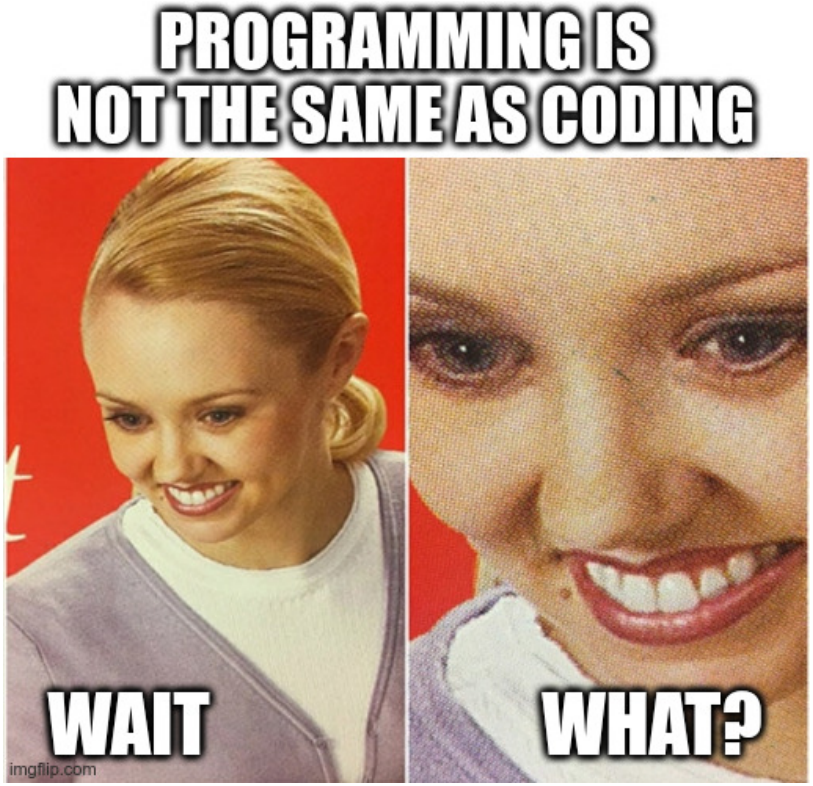
(based and/or partially inspired by Pedro Vasconcelos's slides for Imperative Programming)

What are we doing at this course?

Our course

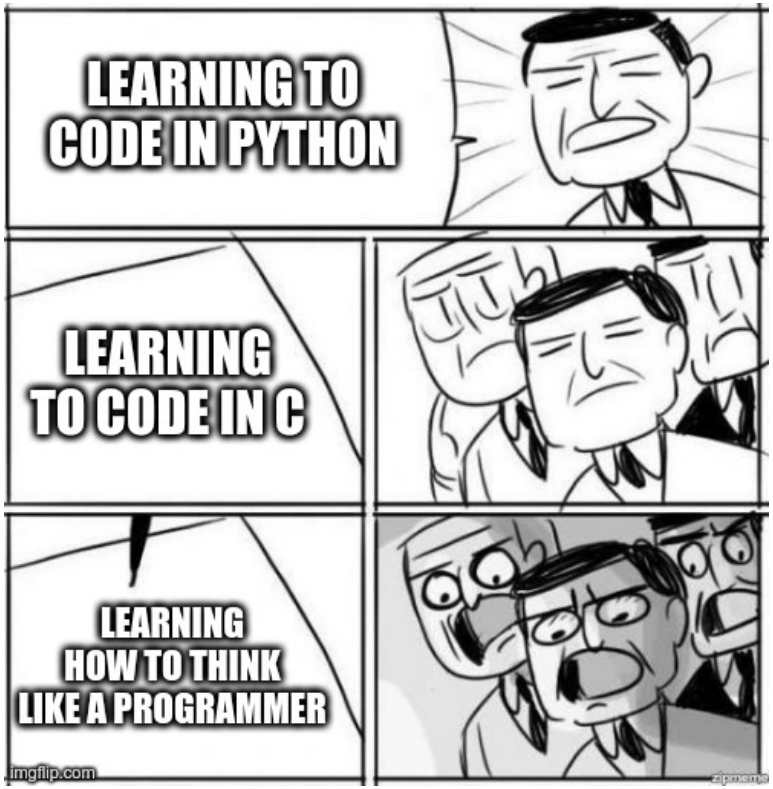
- What is the name of this course?

Programming vs Coding



- **Programming** is the mental process of **thinking** up instructions to give to a "machine" (like a computer).
- **Coding** is the process of transforming those ideas into a (written) **language** that a computer can understand.

Computational Thinking



- You will eventually learn other programming languages (including some that do not even exist nowadays...)
- The most fundamental aspect is **how to think** and to express our ideas as **algorithms**

Why C?

- C is a very **influential language** initially developed in the early 1970s by **Dennis Ritchie** *(more than 50 years ago!)*:

Influenced by
B (BCPL, CPL), ALGOL 68,[4] PL/I, FORTRAN
Influenced
Numerous: AMPL, AWK, csh, C++, C--, C#, Objective-C, D, Go, Java, JavaScript, JS++, Julia, Limbo, LPC, Perl, PHP, Pike, Processing, Python, Rust, Seed7, V (Vlang), Vala, Verilog (HDL),[5] Nim, Zig

C in Wikipedia

Dennis Ritchie in Wikipedia | Turing Award

Why C?

Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2025	2020	2015	2010	2005	2000	1995	1990	1985
Python	1	3	7	7	7	24	23	-	-
C++	2	4	4	4	3	2	1	2	13
C	3	2	1	2	1	1	2	1	1
Java	4	1	2	1	2	3	-	-	-
C#	5	5	5	6	9	9	-	-	-
JavaScript	6	7	8	9	10	7	-	-	-
Go	7	16	36	184	-	-	-	-	-
Visual Basic	8	19	234	-	-	-	-	-	-
SQL	9	9	-	-	100	-	-	-	-
Fortran	10	30	31	25	15	18	5	8	11
PHP	13	8	6	3	5	29	-	-	-
Ada	25	35	30	26	16	17	7	4	3
Lisp	28	31	18	17	14	16	6	3	2
Objective-C	35	10	3	23	40	-	-	-	-
(Visual) Basic	-	-	77	5	4	4	3	5	4

TIOBE Index

C has been on the top-3 for the last 40 years

Why C?

- Initially developed as a **system programming language** to write an operating system (Unix).
- C has **low-level access to memory**

C helps to understand the underlying architecture of how a computer works

Analogy:

- Imagine you learn how to drive on a car with **automatics gears**:
  - Automatic gears (*"Python"*) can make your life easier but,
  - You will not understand how gears work and its intricacies
  - You will not be able to drive a manual gear car if you need to
- C is like learning to drive on **manual gears**:
  - If you know how to "drive" it, you can also drive automatic gears
  - You will also understand better the mechanism and how the car works

Some characteristics of C

- C is a **"Middle-Level"** Language

  Somewhere between low-level machine understandable assembly languages and high-level super user friendly languages
  (bridging the gap between both levels)

- Helps to **understand the fundamentals** of computing

  Aspects such as networks, compiler, computer architecture, operating systems are based on C programming language and requires a good knowledge of C programming if you are working on them.

  In modern high level languages (such as Python), machine level details are hidden from the user, so in order to work with CPU cache, memory, network adapters, learning C programming is a must.

- C is very **portable**

  There are C compilers for practically all processors and operating syst.

Some characteristics of C

- **Fewer Libraries, simple set of keywords.**

  C programming language has fewer libraries in comparison with other high-level languages and will help you focus on the fundamentals.

  You will not be dependent on the programming language entirely for implementing some basic operations and implementing them on your own will also help you to build your analytical skills.

- **C is very fast in terms of execution time.**

  Programs written and compiled in C execute much faster than compared to (almost) any other programming language.

  C programming language is very fast in terms of execution as it does not have any additional processing overheads such as garbage collection or preventing memory leaks etc. The programmer must take care of these things on his own.

C vs Python

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

#define LIMIT 10000000

bool is_prime(int n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    if (n % 3 == 0) return n == 3;
    int p = 5;
    while (p * p <= n) {
      if (n % p == 0) return false;
      p += 2;
    }
    return true;
}

int main() {

  int count = 0;
  for (int n = 1; n < LIMIT; n++)
    if (is_prime(n))
      count++;
  printf("There are %d primes less than %d\n", count, LIMIT);

  return 0;
}
```

U:--- primes.c All L28 (C/*l Abbrev)

```python
LIMIT = 10000000

def is_prime(n):
    if n < 2: return False
    if n % 2 == 0: return n == 2
    if n % 3 == 0: return n == 3
    p = 5
    while p * p <= n:
        if n % p == 0: return False
        p += 2
    return True

def main():

    count = 0
    for n in range(1, LIMIT):
        if is_prime(n):
            count += 1
    print(f"There are {count} primes less than {LIMIT}")

if __name__ == "__main__":
    main()
```

U:--- primes.py All L24 (Python ElDoc)

Comparison between equivalent (naive) code to compute number of primes less than 10 million (see source code in C and Python):

- **C**: 2.6s
- **Python**: 1m33.2s (93.2s, 35× **slower**)

C vs Python

C also has (many) **disadvantages**:

- Forces the programmer to specify many implementation details
  - e.g. managing memory allocation/release explicitly
- C code can be difficult to understand and modify
- It is easy to introduce errors that are difficult to detect
  - e.g. *buffer overflows*, *memory leaks*, *use-after-free*, ...
  - currently one of the biggest sources of **reliability and security problems** in software

C vs Python

Some of the (practical) differences between C and Python:

C	**Python**
.c file extension	.py file extension
Compiled language	Interpreted language
Faster execution times	Slower execution times
Limited number of built-ins and libraries	Large collection of built-ins and libraries
Variables must be declared	No need to declare variables
Statically typed variables	Dynamically typed variables
Blocks of code are separated by {}	Uses indentation to separate blocks
Mandatory ; at the end of each instruction	Instructions can be terminated by end-of-line

Usages of C

Some examples of real life usage of **C** (and derivatives such as **C++**)

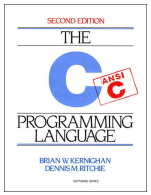
- Operating Systems
- Embedded Systems
- Hardware Drivers
- Scientific Computing
- Game (engine) development
- Libraries (even for other languages, such as Python)
- ...

Note that each language has it's own "niche" and some languages are much better than other for specific tasks:

- **Python** is much better for **AI** and **Data Science**
  (rich collection of libraries, easy to create prototype)
- **JavaScript** is much better for **web development**
  (e.g. for client-side scripting)

C Timeline

- Programming languages are dynamic and change
- The first C *"de facto"* standard was K&R book



Year	Informal name	Official standard
1972	first release	-
1978	K&R C	—
1989	ANSI C, C89	ANSI X3.159-1989
1990	ISO C, C90	ISO/IEC 9899:1990
1999	C99, C9X	ISO/IEC 9899:1999
2011	C11, C1X	ISO/IEC 9899:2011
2018	C17, C18	ISO/IEC 9899:2018
2024	C23, C2X	ISO/IEC 9899:2024

- At this course we will use **C17** as our standard

C Programs

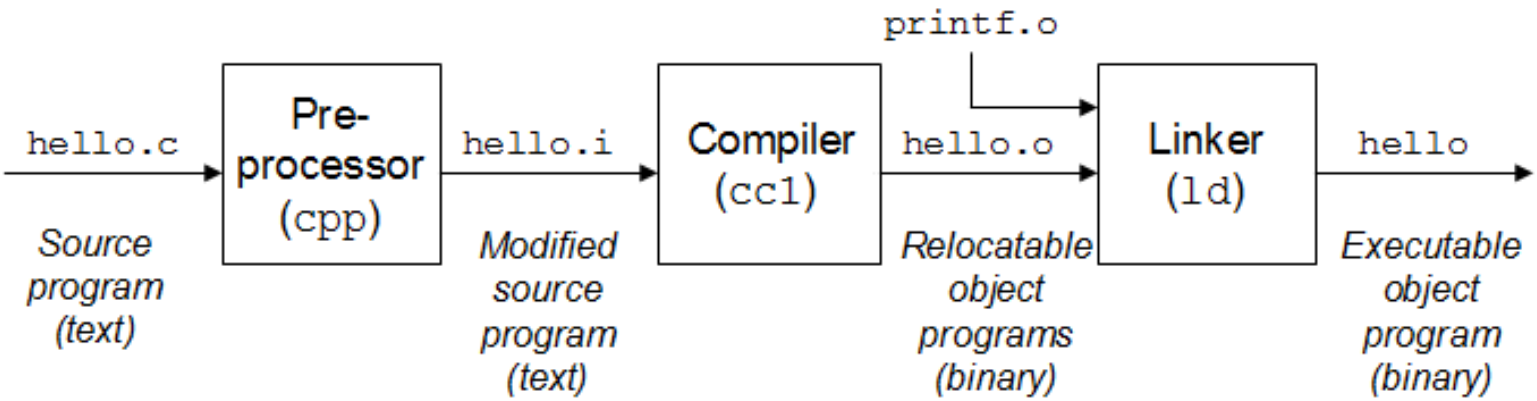
- C programs are text files
- They are composed using a text editor (or IDE):
  (Examples: Emacs, VIM, Atom, Sublime, Notepad++, VScode, CLion)
- Convention: file name ends with `.c` extension (lower case)

`hello.c` (source code)

```c
#include <stdio.h>

int main(void) {

    printf("To C or not to C, ");
    printf("that is the question.\n");

    return 0;
}
```

Execution

- To be executed, a C program must first be translated into **machine code**.
- The translation is done by a **compiler** program.
- In this course we will use **GCC** (*GNU Compiler Collection*).



Translation phases

- **Pre-processing**
  the preprocessor interprets directives (lines beginning with #)
- **Compilation**
  the compiler translates the C code into machine code
- **Linking**
  the linker combines the generated machine code with the necessary libraries

The preprocessor, compiler and linker are executed in sequence by the `gcc` command

Compile, link and run

- We invoke the compiler using the (Linux) command interpreter (shell):

```
$ gcc -o hello hello.c
```

- It produces an `hello` file that we can run:

```
$ ./hello
To C or not to C, that is the question.
```

- We can use other options, such as:

```
$ gcc -Wall -std=c17 -o hello hello.c
```

  - `-Wall` (turn on all warnings)
  - `-std=c17` (use C17 standard)
  - `-o hello` (name of the created executable should be hello)

  (if no `-o name` is passed, then the executable created is `a.out`)

Structure of simple programs

```
directives

int main(void) {
  instructions
}
```

Directives

- A directive is indicated by a line beginning with #; e.g.:

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

- The C language includes header files with library declarations
- `stdio.h` contains the definitions associated with input/output
- Example: `printf` is declared in this header

Functions

- A function groups together a sequence of instructions with a name
- An implementation of the C language provides several *libraries* with predefined functions
- The result of a function is specified with the `return` statement

Main Function

- A complete program must define a `main` function that is executed when the program starts.
- The value returned from `main` represents the error code for the operating system
- Returning zero means that the program ended correctly

```c
int main(void) {
  ...
  return 0;
}
```

Instructions

- The body of a function is a sequence of instructions

```
  printf("To C or not to C, ");
  printf("that is the question.\n");
  return 0;
```

- This example uses only two types of instructions: `printf` *function calls* and `return`
- The `printf("...")` call prints the text in quotes to the standard output (terminal).
- It prints the following message:

```
To C or not to C, that is the question.
```

Instructions

- Each instruction ends with a semicolon (`;`)
- An instruction can be divided into several lines:

```
printf(
    "To C or not to C, "
);
```

- You can also write several instructions on one line:

```
printf("To C or "); printf("not to C, ");
```

- *Directives* usually only take up one line: they **don't** need a semicolon.

Comments

- A *comment* starts with `/*` and ends with `*/`
- Comments can occur on separate lines or in the middle of lines of code
- They can extend over several lines

```
/* This is a comment */

/*
  Author: Pedro Ribeiro
  File: hello.c
  Program: Prints an example message
*/
```

- Warning: forgetting to close a comment may cause the compiler to ignore part of the program.

```
  printf("To be "); /* comment open
  printf("or not to be; "); /* closed */
  printf("that is the question.\n");
```

Comments

- We can also write single-line comments:

```
// This is a comment
```

- Starts with `//` and ends at the end of the line
- More succinct for short comments
- Avoids the risk of forgetting to close the comment

Variables and types

- C programs perform computation by *modifying values* in memory
- Places to store values are designated using **variables**
- Variables in C have a **type** associated with them
- Basic **numeric types**: `int` and `float`

Types - int and float

- A variable of type `int` can store positive and negative integer values:
  - e.g.: `0 1 -23 397`
- The *minimum* and *maximum* values of int depend on the implementation; e.g.:
  - GCC on Intel x86/x64 uses 32-bit (integers from 2^{31} to $2^{31} - 1$)
- A `float` variable stores single-precision *floating-point* values
- It can represent fractional values:
  - e.g.: `0.0253 -1.25 123.555`
- Also values of very large or small magnitudes (approximately between 10^{-38} to 10^{38})
- Disadvantages:
  - slower operations than with integers
  - rounding errors

Declarations

- Variables must be declared before use:

```
int height;
float radius;
```

- You can declare multiple variables of the same type at once:

```
int height, width, depth;
float radius, mass;
```

- Previously all declarations should occur before instructions.

```
int main(void) {
  /* variable declarations */
  int height, width;
  float radius;

  /* instructions follow */
  ...
}
```

- Since C99 declarations and instructions can be mixed.

  (as long as the declaration occurs before use)

Assignment

- We can define or modify the value of a variable using an assignment.

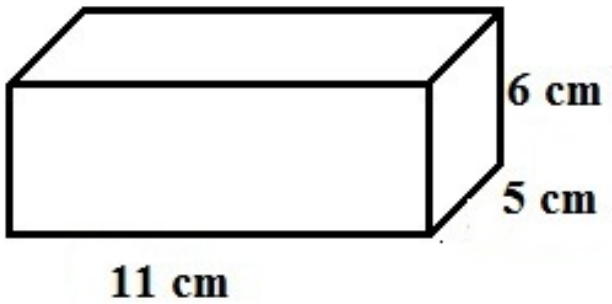
```
int height; // declaration
height = 8; // assignment
```

- The assignment must occur after the declaration
- In this case: we assign the constant 8 to the variable `height`
- On the right-hand side of an assignment we can use expressions e.g. constants, variables and operations.

```
int height, width, area;
height = 8;
width = 3;
area = height * width; // area is 24
```

Example - Volume of a Box

- A program to calculate the volume V of a rectangular box.
- Example:



$$V = 11cm \times 5cm \times 6cm = 330cm^3$$

Example - Volume of a Box

`volume.c` (source code)

```c
#include <stdio.h>

int main(void) {
  int l, w, h, v; // dimensions and volume

  l = 11;         // length
  w = 5;          // width
  h = 6;          // height
  v = l * w * h; // volume calculation

  printf("LxWxH: %d*%d*%d (cm)\n", l, w, h);
  printf("Volume: %d (cm^3)\n", v);

  return 0;
}
```

```
LxWxH: 11*5*6 (cm)
Volume: 330 (cm^3)
```

Printing values

- We can use the `printf` library function to print variable values. Example:

```
int alt;
alt = 6;
printf("Height: %d cm\n", alt);
```

- Prints the text:

```
Height: 6 cm
```

- `%d` is a field that is replaced by the value of an integer variable in decimal base.

Printing values

- For float values we use the `%f` specifier.

```
float cost;
cost = 123.45;
printf("Cost: EUR %f\n", cost);
```

```
Cost: EUR 123.449997
```

- It is not possible to represent 123.45 exactly as a float!
- `%f` displays the result rounded to 6 decimal places

- To force formatting to *n* decimal places we use `%.` *n* `f`:
- Example:

```
  printf("Cost: EUR %.2f\n", cost);
```

```
Cost: EUR 123.45
```

Printing values

- You can format several values in a single `printf`:

```
printf("Height: %d cm; Cost: EUR %.2f\n", alt, cost);
```

- Warning:
  - specify the same number of fields as arguments
  - use the correct fields for each type (`%d` for int, `%f` for float)

Reading values

- The `scanf` library function is used to read values from standard input (*keyboard*).
- Like `printf`, the 1st argument is the data format
- Example: read an integer value and store the result in the variable n

```c
int n;
scanf("%d", &n);
```

- The `&` sign must be placed before the name of the variable to be read (we'll see why later).
- To read a `float` we don't need to specify decimal places.

```c
float x;
scanf("%f", &x);
```

- Works with or without decimal places in the input; examples:

```
123
123.4
123.4567
```

Revised example

Let's modify the previous example program to read the dimensions of the box.

`volume_v2.c` (source code)

```c
#include <stdio.h>

int main(void) {
 int l, w, h, v; // dimensions and volume

 printf("L=? "); scanf("%d", &l);
 printf("W=? "); scanf("%d", &w);
 printf("H=? "); scanf("%d", &h);
 v = l * w * h;  // volume calculation

 printf("LxWxH: %d*%d*%d (cm)\n", l, w, h);
 printf("Volume: %d (cm^3)\n", v);

 return 0;
}
```

Initialisation

- Variables in C are not initialised automatically
- A variable that is not assigned a value is said to be uninitialised:

```c
int x, y;
y = x + 1; // uninitialised variable x
```

- The result of using uninitialised variables is unpredictable:
  - may have different values in each execution;
  - may terminate execution with an error (crash)
- The `gcc` compiler can detect uninitialised variables using the `-Wall` option (*all warnings*)

```
warning: 'x' is used uninitialized in this function
```

Initialisation

- We can initialise variables directly in the declaration:

```
int alt = 8;
```

- Also for multiple variables:

```
int alt = 8, width = 5, comp = 11;
```

- Each variable needs its own initialiser:

```
int alt, larg, comp = 11; // only initialises one variable (comp)
```

Identifiers

- The names of variables, functions and other entities are **identifiers**
- They can contain *letters*, *digits* and underscores but must start with letters or underscores
- Only unaccented letters (i.e. ASCII)
- Valid examples: `times10` `get_Next_Char` `_done`
- Invalid examples: `10times` `get-Next-Char` `máximo`

Identifiers

- Uppercase and lowercase are distinct; for example
  - `get_next_char`
  - `get_next_Char`
  - `get_Next_Char`

  are different identifiers (it would be confusing to use them in the same program...)

- There is no limit to the length of identifiers

Reserved words

- We can't use the following **reserved words** as identifiers (on C17):

```
auto        enum        restrict    unsigned    _Alignas
break       extern      return      void        _Alignof
case        float       short       volatile    _Atomic
char        for         signed      while       _Bool
const       goto        sizeof                  _Complex
continue    if          static                  _Generic
default     inline      struct                  _Imaginary
do          int         switch                  _Noreturn
double      long        typedef                 _Static_assert
else        register    union                   _Thread_local
```

See https://en.cppreference.com/w/c/keyword

Defining constants

- It is sometimes necessary to use **constants** or **parameters**
- Constants scattered throughout the code can obfuscate the meaning
- Instead: we can use `#define` directives to define macros

```
// conversion factor:  inches per metre
#define INCHES_PER_METER 39.3701
```

- Convention: names of constants in **upper case**
- The preprocessor replaces macros textually.
  Example:

```
inches = metres * INCHES_PER_METER;
```

  after preprocessing you get

```
inches = meters * 39.3701;
```

Example

- The area of a circle of radius r is $A = \pi r^2$ (where π is the constant 3.14159...)

`area.c` (source code)

```c
#include <stdio.h>

#define PI 3.14159

int main(void) {
  float radius, area;

  printf("Radius of the circle? ");
  scanf("%f", &radius);
  area = PI * radius * radius;
  printf("Area: %f\n", area);

  return 0;
}
```

Program structure

- A C program is a sequence of symbols ('tokens'):
  - identifiers
  - reserved words
  - operators
  - punctuation
  - constants
  - literal strings

Program structure

- Example: the statement

```
printf("Area: %f\n", area);
```

contains seven symbols:

symbol	description
`printf`	identifier
`(`	punctuation
`"Area: %f\n"`	character string
`,`	punctuation
`area`	identifier
`)`	punctuation
`;`	punctuation

Program layout

- Spaces between symbols are usually not important
- We can even omit spaces (except when two different symbols merge)
- However, this makes code more difficult to read

```
#include <stdio.h>
#define PI 3.14159
int main(void) {float radius,area;printf(
"Radius of the circle?");scanf("%f",&radius);
area=PI*radius*radius;printf("Area: %f\n",area);
return 0;}
```

Program layout

- We should use spaces, tabs and line changes to increase the readability of the code:
  - insert spaces after commas or between operators;
  - use tabs/spaces to align instructions;
  - use blank lines to visually separate blocks of code;
  - insert comments between lines (or even in the middle of a line).

Emphasising syntax

- Text editors have special modes for programming languages
- Among other things, they automatically highlight symbols with colors and/or styles
- They help with reading and writing code

```c
#include <stdio.h>

#define PI 3.14159

int main(void) {
  float radius, area;

  printf("Radius of the circle?");
  scanf("%f", &radius);

  // calculate the area
  area = PI * radius * radius;
  printf("Area: %f\n", area);

  return 0;
}
```

vs

```c
#include <stdio.h>

#define PI 3.14159

int main(void) {
  float radius, area;

  printf("Radius of the circle?");
  scanf("%f", &radius);

  // calculate the area
  area = PI * radius * radius;
  printf("Area: %f\n", area);

  return 0;
}
```