jgromero@inf.uc3m.es

Universidad
Carlos III de Madrid
www.uc3m.es

Lesson 3
Introduction to Programming in C

Programming

Grade in Industrial Technology Engineering

1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

1. **Introduction to the C programming language**
2. Basic program structure
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

C is closely related to the development of the UNIX operating system at AT&T Bell Labs

1968-1971

First versions of UNIX
Towards a better programming language: B, NB

1971-1972

C is created (**K. Thompson)**

UNIX is rewritten in C; versions of C are developed for other platforms (Honeywell 635, IBM 360/370)

1978

Kernighan and Ritchie

Publication of "The C programming language"

Johnson

Development of pcc (C compiler)

1989

C becomes standard (ISO/IEC 9899-1990)

New languages have been developed from C: Objective C, C++, C#, etc.

Different compilers, development platforms and language derivations may lead to C code targeted to a specific machine

- E.g.: Win32 graphic libraries

"Unambiguous and machine-independent definition of the language C"

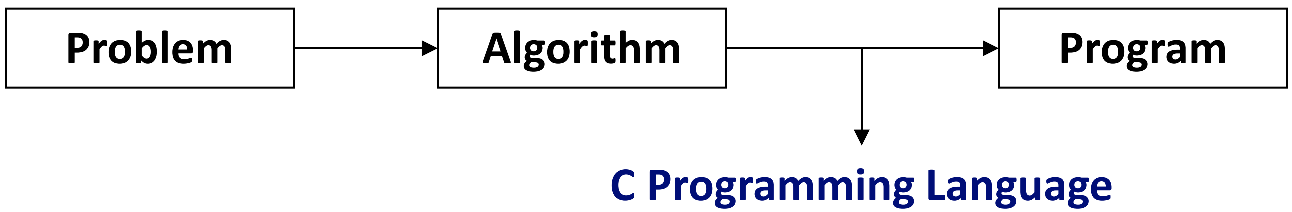
- A program in ANSI C must be compiled by any C compiler and must work in any platform

ANSI C is a **standard subset of the language**:

- Well-defined syntax
- Restricted set of functions

- Several specifications
  - C89/C90
  - **C99**
  - C11

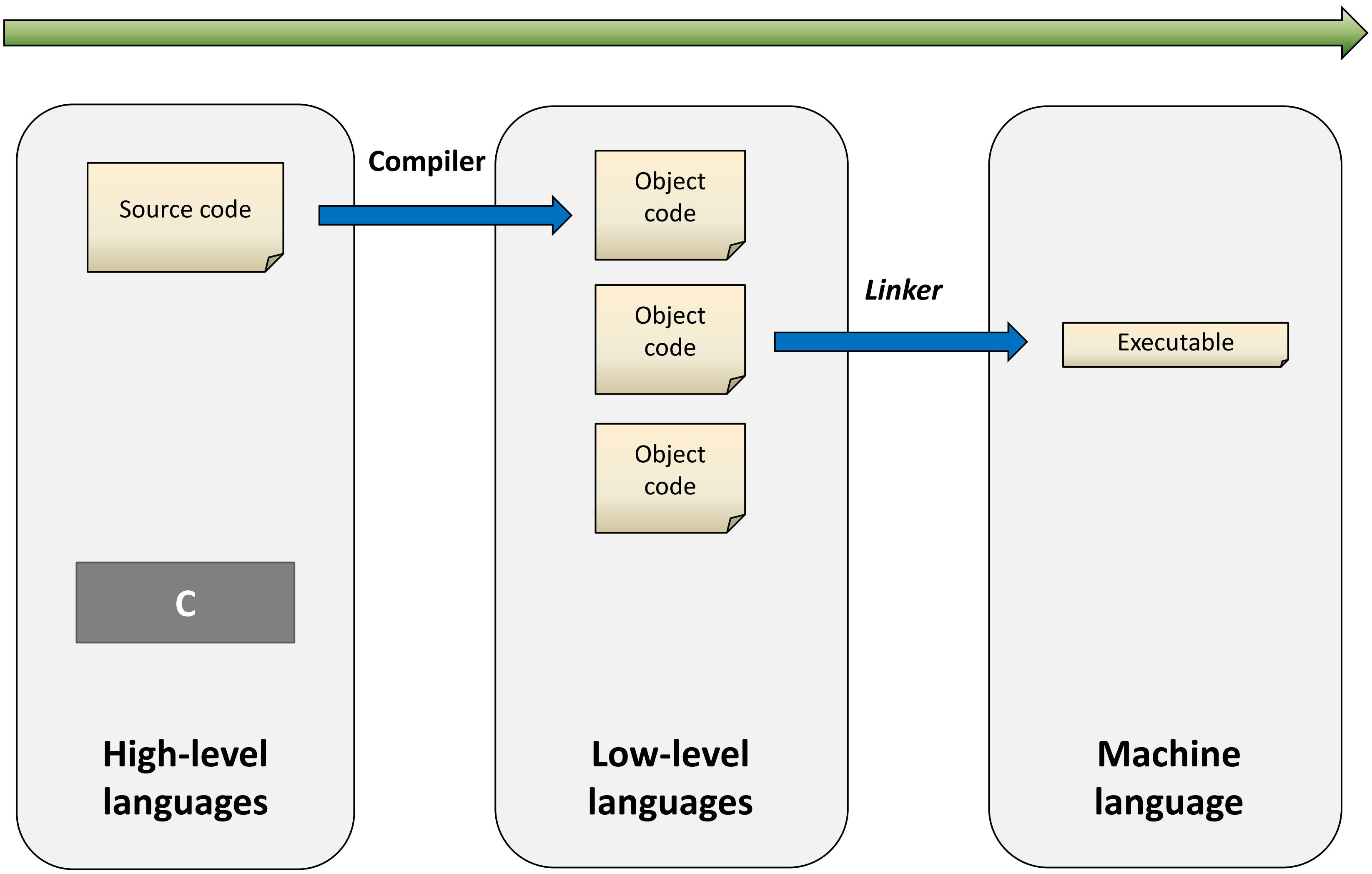
- **Program**: Set of orders (instructions or sentences) written in a programming language that are provided to the computer to develop a task.



- High-level programming languages:
  - Source code must be converted into machine code
    - Compilation
  - In C, there are two steps:
    - Compilation
    - Linking

1. Introduction to the C programming language

Compilation + Linking process



Development environments

Dev C/C++ (integrated MinGW 3.4.2 compiler)
http://www.bloodshed.net/dev/devcpp.html
(Download)

Orwell Dev C++ (integrated MinGW 4.7.0 compiler, portable version)
http://orwelldevcpp.blogspot.com.es/
(Download)

code::blocks (integrated MinGW compiler)
http://www.codeblocks.org/downloads/26
(Download)

Eclipse IDE for C/C++ developers (no integrated compiler)
http://www.eclipse.org/cdt/
(Download)

XCode (integrated LLVM compiler)
https://developer.apple.com/xcode/
(download from Mac App Store)

1. Introduction to the C programming language

First C program

HelloWorld.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");

    system("pause");
}
```

D:\Trabajo\docencia\11-12\2c\Programming - 39\Labs\0\HelloWorld.exe

```
Hello world!
Presione una tecla para continuar . . .
```

A programming language is characterized by:

Alphabet

Allowed characters

Lexicon

Words

Syntax

Rules for word combination to make meaningful programs

C *alphabet*

- Symbols that can appear in a C program
  - Letters
    - All but 'ñ' and accents (only in comments!)
  - Numbers
  - Special characters
- C is case sensitive: uppercase and lowercase letters are different
- Keywords are written in lowercase

The lexicon includes the primitive elements to build sentences

Keywords

Terms with a specific meaning

Lowercase (`include`, `define`, `main`, `if`, etc.)

Delimiters

Blank spaces, tabs, line breaks

Operators

Represent operations: arithmetic, logic, assignment, etc. (+, -, `*`, etc.)

Identifiers

Keywords cannot be used as identifiers

Variable names (`user_age`) – cannot start with a number

Function names (`printf`, `scanf`)

Literals

Values that do not change:

Numbers: `2`, `3.14159`

Strings: `"Hello world"`

Characters: `'a'`

Data

Values processed by the program

Expressions

Combination of operands and operators with a single value as a result

May include function calls, even though they do not return a value

```
user_age >= 18
3.14159*radius*radius
```

Statements/Instructions/Statements

Complete *action*

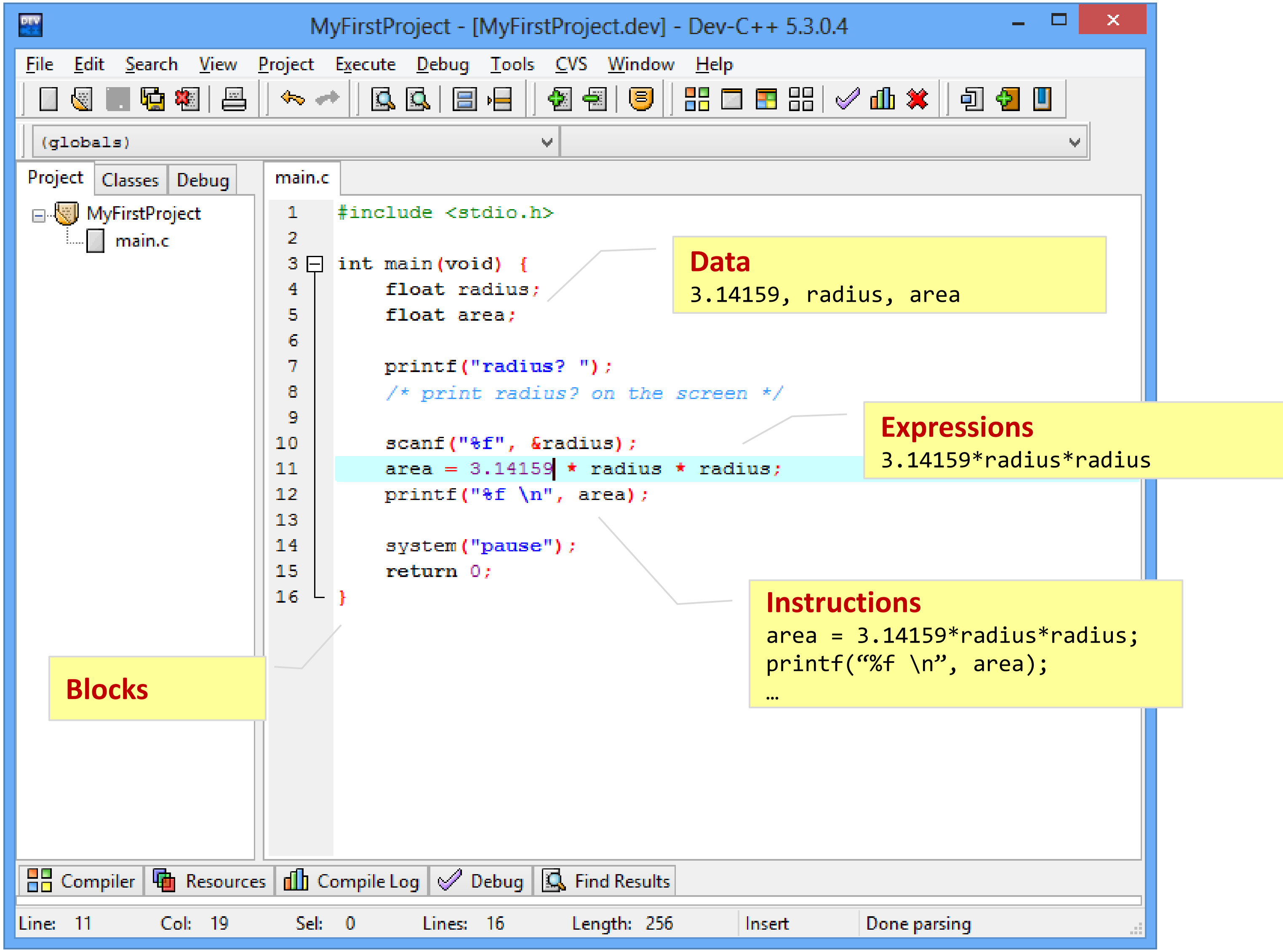
```
area=3.14159*radius*radius;
printf("Hello world");
int a;
```

Blocks or compound statements

Group of statements

Braces `{ }`

The statements of the `main` function are enclosed in a block


MyFirstProject - [MyFirstProject.dev] - Dev-C++ 5.3.0.4
File Edit Search View Project Execute Debug Tools CVS Window Help
(globals)
Project Classes Debug
MyFirstProject
main.c
main.c
#include <stdio.h>
int main(void) {
float radius;
float area;
printf("radius? ");
/* print radius? on the screen */
scanf("%f", &radius);
area = 3.14159 * radius * radius;
printf("%f \n", area);
system("pause");
return 0;
}
Data
3.14159, radius, area
Expressions
3.14159*radius*radius
Instructions
area = 3.14159*radius*radius;
printf("%f \n", area);
…
Blocks
Compiler Resources Compile Log Debug Find Results
Line: 11 Col: 19 Sel: 0 Lines: 16 Length: 256 Insert Done parsing

1. Introduction to the C programming language

Example

Greetings.c

```c
#include <stdio.h>

int main(void) {
     int user_age;

     printf("Hello, friend! \n");
     printf("How old are you? \n");
     scanf("%d", &user_age);
     printf("You said you are %d years old \n", user_age);

     system("pause");
     return 0;
}
```

1. Introduction to the C programming language

Example

Party.c

```
#include <stdio.h>

int main(void) {
    int user_age;

    printf("Hello, friend! \n");
    printf("How old are you? \n");
    scanf("%d", &user_age);

    if(user_age >= 18) {
        printf("Let's grab some beer and cigarretes! \n");
    } else {
        printf("Milk and cookies is enough for you! \n");
    }

    system("pause");
    return 0;
}
```

Lesson 4. Control flow and loops

1. Introduction to the C programming language
2. **Basic program structure**
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

2. Basic program structure

Program elements

HelloWorld.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");

    system("pause");
}
```

File inclusion

***Main* function**

Output instruction

Notice the parentheses and the braces!

The basic building block in C is the **function**

A C program is a collection of functions

A function is a piece of code that performs a task when it is called/invoked

Input values >> Output values

Functions include:

Lesson 6. Functions

Variable declaration (for storing data)

Statements (for performing operations)

All C programs have a *main* function

Starting point of the program

- Automatically started when the program is run

The simplest C program:

```
int main(void) {}
```

- Valid, but useless

`return` is optional, but recommended

***main* function structure**

```
int main(void) {
   …

   return 0;
}
```

`system("pause")`
In old versions of Dev C++ (Windows

C encourages the **use of previous code**

- New functions can be created and reused
- C provides **functions in libraries that can be used in our programs**
  - Input and output functions in *stdio.h*
    - `printf()` and `scanf()`

To include a file, use the directive #include with the name of the file:

`#include "file.h"`	Searches in the current folder
`#include <file.h>`	Searches in the default compiler folder

Comments are **notes** to the code that are not executed

- The compiler ignores comments (they are not *real* code)
- They can be used at any point of the program

Its very important to comment the code well:

- Make the code readable and understandable
- Although we now know perfectly what a program does, maybe we will have to reuse it in the future
- Perhaps other programmers reuse our code and need to understand it
- It is a good practice to introduce a comment at the beginning of each file describing what it does

Syntax for multi-line comments

- `/*` : Open comment block
- `*/` : Close comment block

```
/* print radius? on the screen */
/* This program solves a
   second grade equation. */
```

Comments can span several lines

Comments cannot be nested

In-line comments

- `//` : The remainder of the line is considered a comment

```
printf("%f \n", area);    // print area value
```

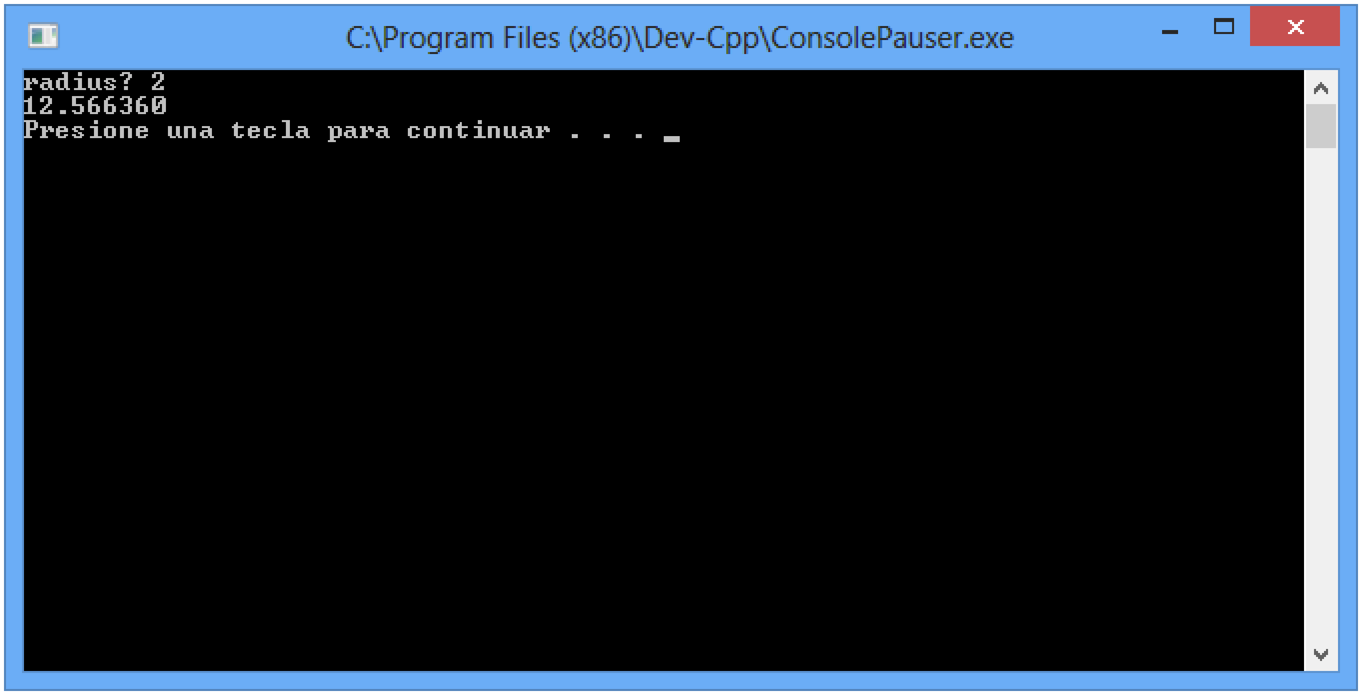
1. Introduction to the C programming language
2. Basic program structure
3. **Variables and constants**
4. Simple data types
5. Expressions and instructions
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

3. Variables and constants

Storing and using values

main.c

```c
#include <stdio.h>

#define PI 3.14159

int main(void) {
    float radius;
    float area;

    printf("radius? ");
    /* print radius? on the screen */

    scanf("%f", &radius);
    area = PI * radius * radius;
    printf("%f \n", area);

    system("pause");
    return 0;
}
```


C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
radius? 2
12.566360
Presione una tecla para continuar . . .

Data

- Information processed by the program
  - Read, used in calculations, written

- Types of data
  - **Variables**
    - Symbols whose value change during the program execution
      - `radius, area`
  - **Constants**
    - Symbols whose value do not change during the program execution
      - `PI`

Variables and constants have:

Name

Label or identifier of the symbol

`radius, area, PI`

Type

Determines which values that can be assigned to the symbol

Integer number, real number, single letter,…

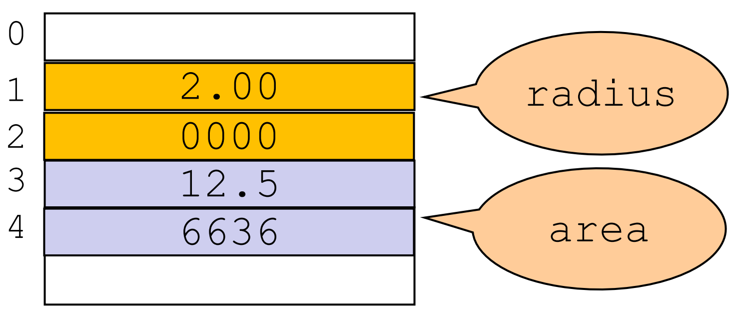
Value

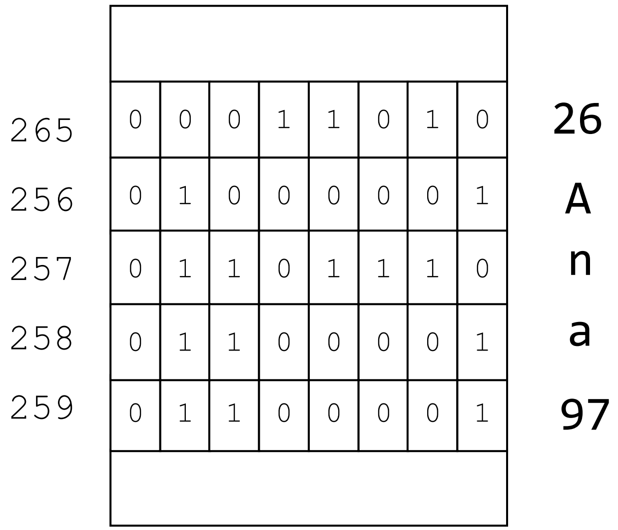
Value of the symbol at a given moment

2, 12.566360

Variables can be seen as a piece of the memory to store a piece of data

- User-defined name for a group of cells of the memory
- When the name (or identifier) of the variable is used in the program, the information at the address of the variable is accessed
- The memory size allocated for the variable depends on its type, which must be set when the variable is declared





Before using a variable, it is necessary to **declare it**

The declaration instruction **allocates a piece of the memory to store the value** of the variable

In the declaration, we specify:

name of the variable

data type

A variable can be declared only once

Syntax

<data type> <variable name>;

Examples

```
float average_mark;
int num1, sum;
char letter;
```

Self-explanatory names in **lowercase** are recommended

…but not too long

```
counter = counter + 1;
num_registered_students = 56;
```

Variables should be declared at the beginning of the block in which they are used. They are valid only in this block (scope)!

```
int main(void) {
    int a;
    int b;

    a = 10;
    printf("%i", a);
}
```

3. Variables and constants

Data types (see later)

Type	Description	Size (bytes)	Range
`int`	Integer number	2 bytes	-32768 to 32767
`float`	Real number with simple precision (7 decimal values)	4 bytes	$3.4x10^{-38}$ to $3.4x10^{38}$
`double`	Real number with double precision (up to 16 decimal values)	8 bytes	$1.7x10^{-308}$ to $1.7x10^{308}$
`char`	Alphanumeric characters	1 byte	Unsigned: 0 to 255

Assigning a value to a variable means that the value on the right is stored on the variable on the left

A single value or the result of an expression can be assigned

`variable <---- value or expression`

A variable **can be assigned several times**

The previous value is overwritten

The assignment operator is **=**

`x=3;`

Value 3 is stored at the memory position assigned to *x*

`x=(a+b)/2;`

Result of the expression (a+b)/2 is stored at the memory position assigned to *x*

`x=x+3;`

Result of the expression x+3 is stored at the memory assigned to *x*

Assignments can must done between a variable and an expressions with **compatible types**

same type
int <--- **int**

compatible types
float <--- **int** adds .0 to the *int*
int <--- **char** assigns the ASCII code of the *char* to the *int*
char <--- **int** if the value of the *int* is out of range, it is truncated
int <--- **float** the decimal part of the *float* is truncated

```
int a=5, b;
char c='Z';
float x, y=3.1;

b=a;
x=a;
b=c;
c=a;
b=y;
```

Variable initialization: first value assignment

In the declaration:

```
int a=8;
```

After the declaration:

```
int a;
a = 8;
```

Multiple declaration/initialization is allowed

```
int a, b, c;
int a=5, b=4, c=8;
int a=1, b, c=a;
```

Uninitialized variables have junk values

We cannot assume that they are 0

A **C constant** is a symbol whose value **is set at the beginning of the program and does not change later**

Two alternatives:

`#define` directive

`#define` <name> <value>

```
#define PI 3.14159
#define KEY 'a'
#define MESSAGE "Press INTRO to continue…"
```

`const` qualifier to a variable

`const` <type> <name> = <value>;

```
const float PI = 3.14159;
const char KEY = 'a';
const char MESSAGE [] = "Press INTRO to continue…";
```

Constant identifiers are usually written in **uppercase letters**

3. Variables and constants

Constants

main.c

```c
#include <stdio.h>

#define PI 3.14159

int main(void) {
    float radius;
    float area;

    printf("radius? ");
    /* print radius? on the screen */

    scanf("%f", &radius);
    area = PI * radius * radius;
    printf("%f \n", area);

    system("pause");
    return 0;
}
```

From this point on, the symbol `PI` represents the value *3.14159*

3. Variables and constants

Constants

main.c

```c
#include <stdio.h>

int main(void) {
    const float PI = 3.14159;
    float radius;
    float area;

    printf("radius? ");
    /* print radius? on the screen */

    scanf("%f", &radius);
    area = PI * radius * radius;
    printf("%f \n", area);

    system("pause");
    return 0;
}
```

From this point on, the symbol `PI` represents the value *3.14159*

Differences between `const` and `#define`

`const` declarations are for typed variables, finish with `;`, and are assigned just like variables

`#define` is a directive, does not specify a data type, does not use an assignment instruction, and does not finish with `;`

Advantages of `const` versus `#define`

The compiler generates more efficient code

The compiler can check if the type and the assigned value are compatible

Advantages of `#define` versus `const`

`const` values cannot be used in places where the compiler expects a literal value (e.g., array definition)

3. Variables and constants

Operate with data

main.c

```c
#include <stdio.h>
#define PI 3.14159

int main (void) {
    float radius;
    float area;

    printf ("radius? ");
    /* prints radius? on the screen */

    scanf("%f", &radius);
    area = PI*radius*radius;
    printf("%f \n", area);


    system("pause");
    return 0;
}
```

Constant definition

Variable declaration

Read value

Assign result of the calculation

Print value

1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. **Simple data types**
5. Expressions and instructions
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

Data can be structured or unstructured

Simple data types

Symbols with a single element and a single value

Numbers: integer numbers, real numbers, …

Characters: single letters

Structured data types

Symbols with an internal structure, not a single element

Character strings

Arrays and matrices

Structures

Lesson 5. Structured data types

Type	Description	Size (bytes)	Range
`int`	Integer number	2 bytes	`-32768 to 32767`
`float`	Real number with simple precision (7 decimal values)	4 bytes	$3.4x10^{-38}$ to $3.4x10^{38}$
`double`	Real number with double precision (up to 16 decimal values)	8 bytes	$1.7x10^{-308}$ to $1.7x10^{308}$
`char`	Alphanumeric characters	1 byte	`Unsigned: 0 to 255`

Size in bytes may be different in different operating systems and platforms

Other simple data types

`void`

Pointers

Modifiers

`int`, `char`: *signed, unsigned*

`int`: *long, short*

int datatype is used to represent integer values

- int literals
- int variables
- int expressions

%i specifier in printf and scanf

int literals can be expressed with different notations

(conversely, integers can be formatted to different notations – see later)

Decimal (base 10): 2013
Octal (base 8): 011 (leading 0)
Hexadecimal (base 16): 0x2B (leading 0x)

```
printf("number: %i \n", 2013);    // 2013
printf("number: %i \n", -2013);   // -2013
printf("number: %i \n", 011);     // 1*8+1*1 --> 9
printf("number: %i \n", 0x2B);    // 2*16+11 --> 43
```

float and double data types are used to represent real values

- `double` more precision, but also larger memory size

%f specifier in `printf` and `scanf`

The decimal separator for literals is .

Scientific notation can be used

- Regular: 82.3473
- Without leading 0: .34
- Scientific notation: 2.4E-4

```
printf("number: %f \n", 82.3473); // 82.34730
printf("number: %f \n", 2.4E-4);  // 0.000240
```

char data type is used to represent ASCII characters

Literals are enclosed in single quotation marks `'  '`

%c specifier in `printf` and `scanf`

```
char letter = 'b';
printf("%c", letter);
```

Special and escape characters can be used

```
char lineBreak = '\n';
```

`void` data type is used to indicate that no value is expected in specific parts of the program

1. A function has no parameters

```
int main(void)
```

is equivalent to

```
int main()
```

2. A function does not return any value

```
void main(void)
```

3. Generic pointers

```
void *p;
```

`void` variables are not allowed

Character strings are used to represent a sequence of characters

Stored in the memory as a strip of characters ended with the null character `'\0'`

`%s` specifier in `printf` and `scanf`

String literals are enclosed in double quotation marks `"  "`

String variables and constants are declared as arrays:

```
char message [] = "Hello world";    // string constant

char name[100];      // string variable of 100 characters at most
scanf("%s", name);   // beware: & is not used with strings
                     //         error if name has more than 100 chars
                     //         do not consider text after blank space
```

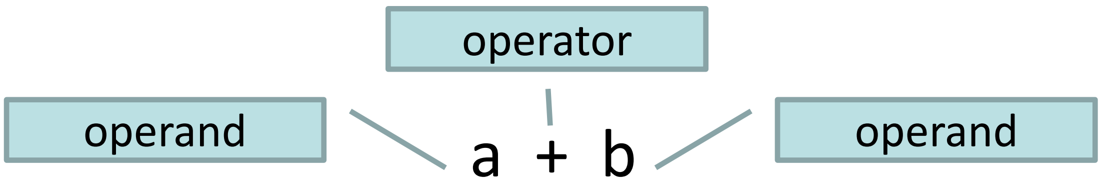
1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. Simple data types
5. **Expressions and instructions**
6. Operators
7. Pointers
8. Basic input/output: `printf` and `scanf`

An **expression** is a combination of **data** by means of one or several **operators**

Data can be literal values, variables, constants, and other expressions

- Even calls to functions can be included

Data symbols in an expression are called operands



Expression composition is guided by rules

- Operands must have a concrete type to be used in an operation

Examples

```
a + b
x == y
x <= y
```

main.c

```
#include <stdio.h>
#define PI 3.14159

int main (void) {
    float radius;
    float area;

    printf ("radius? ");
    /* prints radius? on the screen */

    scanf("%f", &radius);
    area = PI*radius*radius;
    printf("%f \n", area);


    system("pause");
    return 0;
}
```

Number of operands

Unary

-: negative number
++: variable increment
--: variable decrement
!: logic negation

Binary

Operation type

Arithmetic

+ : Addition or positive sign
- : Subtraction or negative sign
*: Product
/: Division
%: Module

Assignment

= : Assign
<op>= : Operation and assignment

Relational

== : Equal
< : Less than
<= : Less or equal than
> : Larger than
>= : Larger or equal than
!= : Different from

Logical

! : NOT (negation)
&, &&: AND (conjunction)
|, ||: OR (disjunction)

Instructions or sentences

- Orders of the program to accomplish a task
- Keywords: short terms interpreted as a command by the computer
  - Are applied on operators and expressions

Types

- According to the function
  - Declaration
  - Assignment
  - Input and output
  - Control
- According to the overall structure of the program
  - Data process
  - Input
  - Output

5. Expressions and instructions

Example

main.c

```
#include <stdio.h>
#define PI 3.14159

int main (void) {
    float radius;
    float area;

    printf ("radius? ");
    /* prints radius? on the screen */

    scanf("%f", &radius);
    area = PI*radius*radius;
    printf("%f \n", area);


    system("pause");
    return 0;
}
```

Variable declaration

Read value

Assign result of the expression

Print value

1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. **Operators**
7. Pointers
8. Basic input/output: `printf` and `scanf`

Operator	Operation
+	Addition
-	Substraction
*	Multiplication
/	Division
%	Remainder or Module

The result of arithmetic operators is a numerical value. The type of the result depends on the type of the operands

The % operator requires two integer operands, being the second one different to 0

The / requires the second operand to be different to 0. **When both operands are integers, the result is also an integer value** (no decimals!)

There is no operator for exponentiation, but the *pow* function of the mathematical library `math.h` can be used (*sqrt* for square roots)

6. Operators

Arithmetic operators - Example

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe

```
integer division (assignment to integer var):    3
float division (assignment to float var):    3.500000
float-integer division (assignment to integer var):    3
float-integer division (assignment to float var):    3.500000


Presione una tecla para continuar . . . _
```

ArithmeticOperators.c

```c
#include <stdio.h>

int main(void) {
    int u=3;
    int a=7, b=2, c;
    float x=7, y=2, z;

    printf("integer division (assignment to integer var):    ");
    c = a/b;
    printf("%i \n", c);

    printf("float division (assignment to float var):    ");
    z = x/y;
    printf("%f \n", z);

    printf("float-integer division (assignment to integer var):    ");
    c = x/b;
    printf("%i \n", c);

    printf("float-integer division (assignment to float var):    ");
    z = x/b;
    printf("%f \n", z);

    printf("\n\n");
    system("pause");
    return 0;
}
```

6. Operators

Arithmetic operators – *pow* and *sqrt*

SqrtPow.c

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

int main(void) {
    int u=3;
    float v=16;

    u = pow(2, 3);
    printf("%i \n", u);

    u = sqrt(v);
    printf("%i \n\n", u);

    system("pause");
    return 0;
}
```

C:\Program Files (x86)\Dev-C

```
8
4

Presione una tecla para continuar . . .
```

Unary operators

++ --

Increase / decrease a variable

They can be used in prefix or suffix mode:

`++x` : increment `x` in 1 and then proceed with the expression evaluation

`x++` : evaluate the expression and then increment `x` in 1

```
int a=100, b=10;
```

1) Pre-increment

```
c = a + ++b;    // --> c=100+11=111, a=100, b=11
```

2) Post-increment

```
c = a + b++;    // --> c=100+10=110, a=100, b=11
```

The result of **relational operators** is a *boolean* value

true: 1, false: 0

Operator	Operation
<	Less than
<=	Less or equal than
>	Larger than
>=	Larger or equal than
==	Equals
!=	Different from

AND, OR, NOT

They are applied on *boolean* expressions –which may be the result of relational operations or other logic operations

Examples:

To pass the lecture, exam **and** exercises must be passed

Pass = Pass_Exer AND Pass_Exam

To pass the lecture, **at least one** of the parts needs to be passed

Pass = Pass_Exer OR Pass_Exam

AND

Operand values

	T	F
T	T	F
F	F	F

Result of the expression

OR

	T	F
T	T	T
F	T	F

NOT

T	F
F	T

6. Operators
Logic operators

Operation	Operator
and	&&
or	||
not	!

Let us suppose that i=7, f=5.5, c='w'

Expression	Result	Value
`c == 'w'`	True	1
`c == "w"`	False	0
`(i >= 6 ) && (c == 'w' )`	True	1
`(i >= 6) || (c == 119)`	True	1
`(c != 'p') || ( (i+f) <= 10 )`	True	1
`!(i > f)`	False	0

Basic assignment

= operation for setting the value of a variable

The previous value, if any, is replaced

Operation and assignment

Change the value of the variable on the left by the result of the operator applied on the same variable and the expression on the right

+=   -=   *=   /=   %=

`<var> <op>= <exp>` is equivalent to `<var> = <var> <op> (<exp>)`

```
int x = 10, y = 2;
y += x;        // y = y + x;        (y : 12, x : 10)
y -= ++x;      // y = y - (++x);    (y : -9, x : 11)
```

Special abbreviation involving *boolean* expressions:

```
<variable> =
    <logical expression> ?
    <value if true> : <value if false>;
```

If more than one operator appears in an expression, **precedence rules** are applied to determine which operators are firstly evaluated

```
a + b > c || c < 0
```

Precedence rules are very similar in all programming languages

Parenthesis should be used

- Expressions enclosed with parenthesis are evaluated first, from the inner-most to the outer-most

```
( (a + b) > c ) || (c < 0)
```

Operators are classified according to their precedence

From higher to lower precedence (**a, b are expressions** with proper type)

Expressions with operators of the same category are evaluated from left to right

Category			
Unary	!	NOT (negación lógica)	!a
	++	Increment	++a
	--	Decrement	--a
	-	Sign change	-b
	*	Indirection	*p
	&	Address	&a
Multiplication	*	Multiplication	a*b
	/	Division	a/b
	%	Module	a%b
Addition	+	Addition	a+b
	-	Substraction	a-b
Relational	<	Less than	a<b
	<=	Less or equal than	a<=b
	>	Larger than	a>b
	>=	Larger or equal than	a>=b
Equality	==	Equals to	a == b
	!=	Different to	a != b
Logic	&&	AND	a && b
	||	OR	a || b
Assignment	=	assignment	a = b

Outline

1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. Operators
7. **Pointers**
8. Basic input/output: `printf` and `scanf`

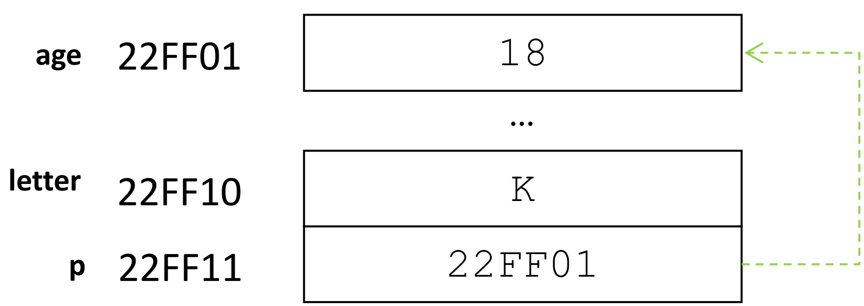
A **pointer** is a variable that **stores a memory address** (it does not contain a *normal* value, but a number corresponding to the position of a memory cell)

Let us suppose that T is a data type

Then, T* is a pointer to a variable of type T

`int age;`	Integer variable
`int *p;`	Integer pointer variable

Usually, the address value is the memory address of another variable




```
int age = 18;
char letter = 'K';

int *p;
p = &age;
```

Pointer declaration

<data type to point to> <* symbol> <name of the pointer variable>;

`int *p;`	pointer *p* to an integer variable
`char *ppt;`	pointer *ppt* to a char variable

address-of operator (`&`)

&<variable> : obtains the memory address of the *variable*

```
&age
```

indirection operator (`*`)

*<pointer> : obtains the variable pointed by the *pointer*

```
*p
```

Pointers must be always initialized

How can we **assign** a value to a pointer?

1) directly

```
int *p;
p = 0x22FF01;
```

Not recommended: we do not know the memory address of a variable

2) indirectly (address operator)

```
<pointer> = &<variable>;
```

```
int *p;
int age = 18;
p = &age;
```

Recommended: we say that the pointer (p) points to the variable (age)

We can **indirectly change the value of the variable** through the pointer (indirection operator)

```
*<pointer> = <expression>;
```

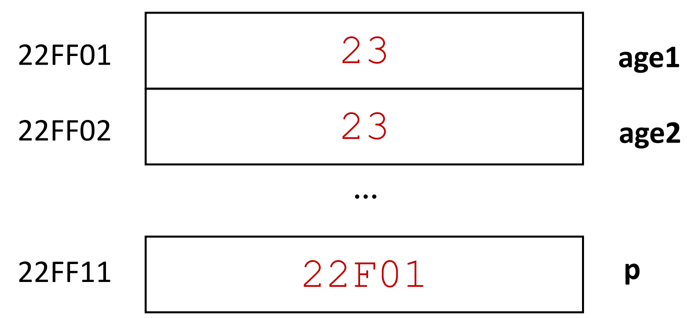
```
*p = 21;
```

After the pointer `p` has been assigned the address of `age`, `*p` is the value of the variable `age`

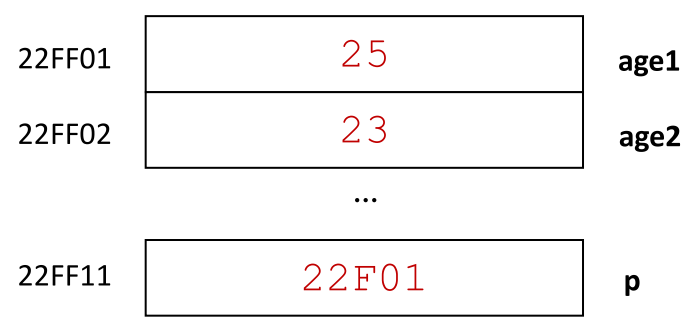
Pointers **can be assigned only address values of variables of the pointer type**

```c
int main(void) {
    int age1 = 18;
    int *p;
    int age2;

    printf("%i \n", age1);    // 18
    age2 = 0;
```


```c
    p = &age1;
    age1 = age1 + 5;
    age2 = *p;
```




```c
    *p = 25;
    printf("%i \n", age1);   // 25
}
```



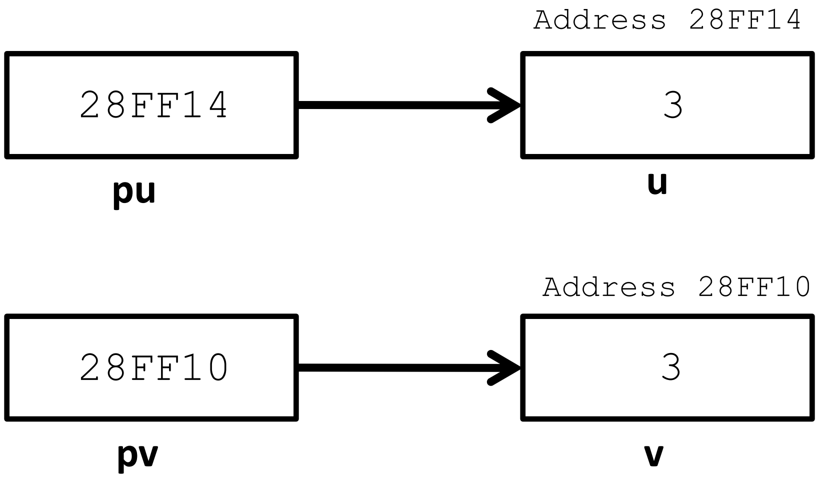
main.c

```c
#include <stdio.h>
int main() {

    int u = 3;
    int v;
    int *pu;     /* pointer to an integer variable pu */
    int *pv;     /* pointer to an integer variable pu */

    pu = &u;         /* u address are assigned as the value of pu */
    v  = *pu;        /* content of address in pu is assigned as the value of v */
    pv = &v;         /* v address are assigned as the value of pv */

    /* Print instructions*/
    printf("\n u=%d &u=%x pu=%x *pu=%d",u,&u,pu,*pu);
    printf("\n v=%d &v=%x pv=%x *pv=%d",v,&v,pv,*pv);
    return 0;
}
```

The output of the program is

```
u=3        &u=28FF14    pu=28FF14    *pu=3
v=3        &v=28FF10    pv=28FF10    *pv=3
```

The relation between the pointers and the variables is shown in this diagram:



The **void** can be used to declare a generic pointer:

```
void *pointer;
```

NULL is a special value to explicitly indicate that the pointer is not pointing to any valid memory address

```
#include <stdio.h>      // NULL is defined in stdio.h

int main(void) {
   int *p = NULL;
   …
```

1. Introduction to the C programming language
2. Basic program structure
3. Variables and constants
4. Simple data types
5. Expressions and instructions
6. Operators
7. Pointers
8. **Basic input/output: `printf` and `scanf`**

Programs receive input data (e.g., keyboard) and provide output data (e.g., screen)

Input and output (I/O) functions allow reading and printing data

C does not provide input/output instructions

- I/O is achieved with **functions included in the standard library** –this library is part of the core of the language

It is necessary to include at the beginning of the program the file *stdio.h*, where these functions are declared

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

printf()

Prints information on the standard output device

Usually, on the screen

Syntax

```
printf("argument format", arguments)
```

```
#include <stdio.h>

int main  ( ) {
     int n=10;
     printf ( "%i", n);

     return 0;
}
```

Placeholders

`%[flags][width][.precision][length]<type>`

Flags

+ : prints number sign

space: prefixes non-negative values with a blank space

- : left-aligns the output

: trailing numbers and decimal values are always printed

0: uses 0 instead of spaces for padding

Width

Minimum number of characters to output (pads if necessary)

Precision

Maximum limit of characters to output (rounds if necessary)

8. Basic input and output

`printf`

Type	Argument format
%c	Character
%d, %i	Integer
%O	Integer, octal format
%u	Integer, unsigned
%x	Hexadecimal
%f	Float
%e	Float, scientific notation
%lf	Double
%s	Character string
%p	Pointer

Special characters

\n : Line break

\t : Tabulation

\b : backspace

Escape characters

\' : to print the ' character

\" : to print the " character

\\ : to print the \ character

scanf()

Reads information from the standard input device

- Usually, the keyboard

Syntax

```
scanf("argument format", &variable)
```

- The & operator means that the variable in the arguments is passed by reference
  - Pass by reference: the address of the variable is passed; the value is changed in the function
- More than one variable can be read in the same `scanf` instruction

```
#include <stdio.h>

int main  ( void ) {
     int n;
     float mark;

     printf ( "Enter student number and mark:\n");
     scanf ("%i %f", &n, &mark);
     printf ("\n The mark of the student %i is %f\n", n, mark);
}
```

Reading strings with `scanf`

Do not use &

```
char name[100];
scanf("%s", name);
```

`scanf %s` stops reading when it finds a blank space in the input

```
scanf("%s", name);
Miguel de Cervantes

printf("Hello %s", name);
Hello Miguel
```

To read a string including blank spaces we use:

```
scanf ("%[^\n]", name);
```

`%[^\n]` means that `scanf` reads until a line break character is found

```
#include                Pre-processor directives
#define
```

```
/* Global declarations */
Function prototypes
```

```
/* Main function */
int main (void)
{
      Local variable and constant declaration
      Instructions
}
```

```
/* Definition of other functions */
type function_name (...)
{
      ...
}
```

Basic

- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapters 1, 2
- Stephen G. Kochan. *Programming in C.* Sams, 2004 (3rd Edition), Programming in C – Chapters 3, 4

Additional information

- Stephen Prata. *C Primer Plus.* Sams, 2004 (5th Edition) – Chapters 1-4

1. **Introduction to the C programming language**
2. **Basic program structure**
3. **Variables and constants**
4. **Simple data types**
5. **Expressions and instructions**
6. **Operators**
7. **Pointers**
8. **Basic input/output: `printf` and `scanf`**