

# Introduction to programming

Taha ARBAOUI  
— G101 —

Université de Technologie de Troyes

– Spring 2021 –



# Lecture chapters

- Introduction
- 1 C Language: The Basics
- 2 Types, Instructions and Operators
- 3 Control Structures
- 4 Functions and procedures
- 5 Arrays
- 6 Structures
- 7 Pointers and Dynamic Memory Allocation
- 8 Files

# 0. Introduction

- Introduction
  - Programming Languages Classification
  - C Language: Properties
  - A C program
- 1 C Language: The Basics
- 2 Types, Instructions and Operators
- 3 Control Structures
- 4 Functions and procedures
- 5 Arrays
- 6 Structures
- 7 Pointers and Dynamic Memory Allocation
- 8 Files

# Programming languages classification

## First Classification

- Interpreted languages (Perl, Python, Tcl, ...) or compiled languages (Pascal, C, ...)

## Second Classification

- 1st generation : machine code
- 2nd generation : Assembly language
- 3rd generation : procedural language (Fortran, Basic, Pascal, C, ...) or objected-oriented languages (Java, C++, Python, ...).
- 4th generation : languages using a user interface and having a simplified human language (Microsoft Access avec SQL, Labview, Matlab, Mathematica)

## Third Classification

- Structured programming (Fortran, Pascal, C, Perl, Tcl )
- Functional (Lisp) or logical (Prolog) programming
- Object-oriented programming (C++, Java, VB.net, C#, ...)

## Fourth Classification

- **Low** and **high** level languages

# History of C language

- Created in 1970 by D. Ritchie (BELL labs)
- Chronological naming : BCPL (Basic Combined Programming Language), B (Simplified version of BCPL), CPL (Combined Programming Language)
- C was initially introduced for the Unix system
- Unix is mostly written in C language (for performance reasons)
- C is today used in many applications (especially for its fast performance)
- Several languages were derived from C (C++, C#, Objective-C, ...)

# Properties of C

- C is also considered as a low-level language, in the sense that it handles elementary operations : *Bit (binary digit), byte (8 bits), & addresses*
- At the same time, it is a general-purpose language, it allows to develop sophisticated applications (scientific, data analysis, Word & Excel are written using C and C++)
- C is among the first languages that introduced modular programming :  
a program can be made up of several modules (.c files)

# Properties of C

## Low-level Language

- A programming language is used to communicate with the machine. The *machine code* uses two symbols: 0 and 1.  
**Example** : Number 5 is coded on the machine using the following: 101 - The binary representation of number 5.
- Machine operations are also done in the binary representation, using 0 and 1.  
**Example** : the machine instruction 00011010 0001 0010 requires the computer to do 1+2.
- The binary representation is not the only language understood by the machine. Assembly language (add \$1 \$2) can be used but it is not portable



# C characteristics

1/2

- Type: procedural programming  
The program is divided into multiple blocks that contain their variables (and sub-blocks).
- Control structures:
  - Conditional (`if`, `else`, `switch`)
  - Loops (`for`, `do`, `while`)
- Blocks and sub-blocks:
  - Procedures
  - Functions
  - Library calls
- Several data types:
  - Integer (`int`, `short`, ...)
  - Continuous (`float`, `double`, ...)
  - Characters (`char`)
  - Pointers
  - Structures and type definitions
  - Files, ...

# C characteristics

2/2

- High-level language
  - Structured
  - Modular
  - Portable
  - ...
- At the same time, low-level language: address manipulation, pointers, registers, ...
- Limited in terms of flexibility and mathematical computations. However, libraries can be used to cope with this limitation.
- Very small machine code after compilation
- Allows a great freedom to the developer but requires a great proficiency of the language

# How to write a C program

## Steps to write and compile a C program

- 1 **Write a ".c" file:** (or .cpp) using a text editor (Sublime Text, Notepad, ...)
- 2 **Compile:** using any C compiler (Code Warrior, CodeBlocks, Borland C/C++, Visual C++, GCC, KDevelop, Xcode, ...)
- 3 Hidden step: transform the C code into assembly
- 4 Linking: linking allows to link the different object files
- 5 **Result:** a ready-to-be-executed file containing the machine code

# C Programming

## A first code

```
1  #include <stdio.h>  /* header */
2
3  /* This program prints a text using the input/output standard library stdio.h */
4
5  int main(void)
6  {
7
8      printf("Hello UTT\n");  // One line is enough
9
10     return 0;
11 }
```

# C programming

In ANSI C, C++, C#, Objective-C

## C (ANSI)

```
1 #include <stdio.h>
3 int main(void)
{
5     printf(" Hello UTT\n");
    return 0;
7 }
```

## C++ (ISO)

```
1 #include <iostream>
3 using namespace std;
5 int main()
{
7     cout << " Hello UTT" << endl;
    return 0;
9 }
```

## C#

```
1 public class HelloWorld
{
3     static void Main()
    {
5         System.Console.WriteLine(" Hello UTT\n");
    }
7 }
```

## Objective-C (object oriented)

```
1 #import <stdio.h>
   #import <objc/Object.h>
3
   @interface Hello : Object
5 {
   }
7 - hello;
   @end
9
   @implementation Hello
11 - hello
    {
13         printf(" Hello UTT\n");
    }
15 @end
17 int main(void)
    {
19         id obj;
        obj = [Hello new];
21         [obj hello];
        [obj free];
23         return 0;
    }
```

# 1. C Language: The Basics

## ● Introduction

### 1 C Language: The Basics

- The main blocks of C program
- `printf` and `scanf`
- Reserved Identifiers and Keywords

### 2 Types, Instructions and Operators

### 3 Control Structures

### 4 Functions and procedures

### 5 Arrays

### 6 Structures

### 7 Pointers and Dynamic Memory Allocation

### 8 Files

# The main blocks of a C program

Every C program has mainly four blocks:

# The main blocks of a C program

Library calls, new types and structures declaration

Call of the 1<sup>st</sup> function or procedure

Call of the 2<sup>nd</sup> function or procedure

⋮

Call of the  $n^{\text{th}}$  function or procedure

Call of the **main** function



## The main blocks of a C program: header

- Libraries are .h files. They can be used/called using instruction `#include`. Existing libraries (such as the standard libraries or others) can be used. The user can also define their own libraries
- Constants are also defined in the header using instruction `#define` (example : `#define m 7`)
- New types and structures are defined in the header using commands `struct` and `typedef`

```
#include <stdio.h>
2 #define m 7           // Constant m is defined

4 //Define a new structure couple
typedef struct {int l; int j; } couple;
6
void main()
8 {
    couple cup;           // define variable cup
10  cup.j=m; cup.l=2;      // initialize variable cup
    printf("Sum = %d\n",cup.j+cup.l);
12 }
```

# The main blocks of a C program: functions and procedures calls

```
<type> FunctionName(<type1> var1, <type2> var2, , <typen> varn)
2 {
  // Local variables declaration
4 Instruction1;
  InstructionN;
6 }
```

- FunctionName: is the function name/identifier
  - <type>: is the type of data returned by the function (example : `int`)
  - <type1>: is the type of variable var1
  - <typen>: is the type of variable varn
- If a function does not return any value, it is called a **procedure**. In this case, the return type to be specified is `void`
- Functions/procedures are declared one after another. **Warning: a function/procedure must be declared before called in any part of the program!**
- The returned data is dependent on input variables (arguments)
- Arguments/Variables of the function/procedure are put between parentheses after its name. Two variables are separated using “,”
- A function/procedure can have on or multiple instructions
- A variable can be declared inside the function/procedure. In this case, it is called a **local** variable. They are not **visible** outside the function
- A function/procedure can be called for an unlimited number of times

# The main blocks of a C program: instructions

A function is made up of one or several instructions

- Every instruction must be ended by a semicolon “;”
- The number of instructions in a function is not limited
- The set of instructions of a function is put inside braces. Braces are used to delimit blocks
- An instruction can be written on multiple lines.
- Multiple instructions can be written on the same line. The instruction delimiter is the semicolon.

# The main blocks of a C program: the `main` function

- The last (as mostly written) function of a program is the `main` function
- This function must have the name `main`. This function represents the entry point of the program (its first instruction is the first instruction executed). Its variables are global variables.
- In terms of declaration, it is similar to other functions (declarations, instructions, syntax, ...)

Do not forget:

- Predefined functions can be called using existing libraries (e.g. function `printf` in the standard library `stdio.h`)
- Comments can be put in the program using `//` for a one-line comment or `/* */` for multi-line comments

## Example: declaring a function

```
/* Max function */
2 int imax(int n, int m)    // Function header
{
4     int max;              // Local variable

6     if (n>m)
        max = n;
8     else
        max = m;
10    return max;           // Returned value
}
```

## Example: surface of a rectangle

Write a C program that computes the surface of a rectangle.

Execute the program for width  $w = 5$  and length  $L = 24$ .

```
1  #include<stdio.h>
3  void main()
4  {
5      float w;
6      float L;
7      float Surface;
8
9      w=5;
10     L=24;
11     Surface=w*L;
12     printf("Surface   = %.2f\n",Surface);
13 }
```

# A first C program

## Example.c

```
1  #include <stdio.h>
   #include <math.h>
3  #define NTimes 5

5  void main()
   {
7     int i;
     float x;
9     float rootx;

11    printf("Hi!\n");
    printf("I will compute  %d square roots\n", NTimes);
13
    for(i=0;i<NTimes;i++)
15    {
        printf("Enter a number: ");
17        scanf("%f",&x);
        if (x<0)
19            printf("The number %f does not have a square root\n", x);
        else
21            {
                rootx=sqrt(x);
23                printf("The number %f has the following square root: %f\n", x, rootx);
                ;
            }
25    }
    printf("Bye!");
27 }
```

## Another C program

### Example2.c

```
1 #include<studio.h>          // Library call
3 void procedure()             // procedure declaration
{
5     printf("The procedure has been executed\n");
}
7 void main()                  // The main function
{
9     int i;                   // Variable declaration
    i=0;                       // Variable initialization
11
13     printf("Hi, the procedure will be executed\n");
15     procedure();             // First call
    i=i+1;                     // incrementation of variable i
17     procedure();
    i=i+1;
19     printf("We have executed the procedure  %d times\n",i);
21 }
```



printf

## printf

### syntax

#### syntax of printf

```
printf(<format>, <id1>, ... , <idn>);
```

```
printf("%d square = %d", i, i*i);
```

**format** specifies how variables will be displayed. The format is a string that includes specific characters to include the variables in the string.

Format	Data type	Example
d ( <sup>ou</sup> i)	Signed integer in decimals	392
u	Unsigned integer in decimals	7235
o	Unsigned integer in octals	610
x ( <sup>ou</sup> X)	Unsigned integer in hexadecimals in small or capital characters	7fa ou 7FA
f	Decimal floating point	392.65
e ( <sup>ou</sup> E)	Scientific notation (mantissa/exponent) in small or capital characters	3.9265e+2
g ( <sup>ou</sup> G)	Shortest representation between e or f	392.65
c	Character	a
s	String	NF05A
%	%% prints %	%
p	Pointer	B8000000

## scanf

## syntax

## syntax of scanf

```
scanf(<format>, <id1>, ... , <idn>);
```

```
scanf(" %d %d", &i, &j);
```

**format** specifies how variables will be read (scanned).

Format	Data type	Example
d ( <sup>ou</sup> i)	Signed integer in decimals	392
u	Unsigned integer in decimals	7235
o	Unsigned integer in octals	610
x ( <sup>ou</sup> X)	Unsigned integer in hexadecimal in small or capital characters	7fa ou 7FA
f	Decimal floating point	392.65
c	Character	a
s	String	NF05A

**printf & scanf**

## Example

```

#include<stdio.h>

2  int i;                                // Global variable
4  void main()
{
6      i=23;                             // Initialization of i
    printf(" i(dec) = %d\n",i);          // Printing i in decimal
8    printf(" i(octal) =%o\n",i);        // Printing i in octal
    printf(" i(hex)= %x\n",i);           // Printing i in Hexadecimal
10   scanf("%d", &i);                     // Reading i
    printf("New value of i = %d\n", i);  // Print the new value of i
12 }

```

Special characters	What will be displayed
%%	'%'
\\	\
\'	'
\"	"
\Onn	Octal value (ASCII) of nn
\Xnn	Hexadecimal value of nn
\n	New line (line break)
\r	carriage return
\b	backspace
\0	Null character
\f	Page break
\t	Horizontal tab
\v	Vertical tab

**printf**

## Example

```

#include<stdio.h>
2
#define M_PI 3.14159265358979323846 /* pi */
4
#define YEAR 2007
6
int main()
8 {
    printf ("Characters: %c %c %c %c\n", 'a', 65, 0x30, '0');
10    printf ("Integers: %d %ld\n", YEAR, 650000);
    printf ("With spaces: |%10d|\n", YEAR);
12    printf ("With zeros: |%010d|\n", YEAR);
    printf ("Using different systems: %d %x %o %#x %o \n", 100, 100, 100, 100,
100);
14    printf ("Floating point: |%4.2f| |%.4e| |%E|\n", M_PI, M_PI*YEAR, M_PI);
    printf ("Length of an argument: |%*d|\n", 5, 10);
16    printf ("%s\n", "Debian GNU/Linux");
    return 0;
18 }

```

```

Characters: a A 0 0
2 Integers: 2007 650000
With spaces: |          2007|
4 With zeros: |00000002007|
Using different systems: 100 64 144 0x64 0144
6 Floating point: |3.14| |+6.3052e+03| |3.141593E+00|
Length of an argument: |    10|
8 Debian GNU/Linux

```

## scanf and the line break problem

Here is an example:

```
#include<stdio.h>
2
int main()
4 {
    char c1, c2;
    printf ("Enter two characters: ");
    scanf ("%c", &c1);
    scanf ("%c", &c2); // reading a line break '\n'
    // The second character is not read
    printf ("c1 = %c \n c2 = %c end of characters\n", c1, c2);
    return 0;
12 }
```

**Result:**

```
Enter two characters: t
2 c1 = t
  c2 =
4 end of characters
```

### Problem

The line break is not read by the scanf function

## scanf and the line break problem

readChar et readString

```
#include<stdio.h>
2 char readChar()
{
4     char c ;
    c = getchar(); // read a character
6     while(getchar() == '\n') ; // delete all line breaks after the character
    return c;
8 }
void readString(char str[], int len)
10 {
    int idx=0;
12     char c;
    while ((c = getchar()) != '\n' && c != EOF) {
14         if(idx < len-1)
            {
16             str[idx]=c;
                idx++;
18         }
    }
20     str[idx] = '\0';
    printf("I read: %s \n" , str);
22 }
int main(int argc, char* argv[])
24 {
    char c, t[10];
26     printf("Enter a string and then a character:\n");
    readString(t, 10);
28     c = readChar();
    printf("String = %s --- c = %c end of reading\n" , t, c);
30     return 0;
}
```

```
1 Enter a string and then a character:
  NF05estTMdeTC
3 I read: NF05estTM
  n
5 String = NF05estTM --- c = n end of reading
```

# Reserved Identifiers and Keywords in C language

## Identifier:

- The expressed used to name a variable or a function.
  - An identifier must start with a character (upper and lower cases and `_`).
  - An identifier can be made up of characters and numbers.
  - Be careful: there exist some expressions that are prohibited to be used as identifiers (examples: `int`, `float`, ...).

## Reserved keywords:

- C language uses specific keywords for the names of types, functions, control sequences, etc. These expressions cannot be used to name variables or functions as they are reserved. Here is the list of specific keywords in C:

<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>	<code>const</code>	<code>continue</code>	<code>default</code>	<code>do</code>
<code>double</code>	<code>else</code>	<code>enum</code>	<code>extern</code>	<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>
<code>int</code>	<code>long</code>	<code>register</code>	<code>return</code>	<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>
<code>struct</code>	<code>switch</code>	<code>typedef</code>	<code>union</code>	<code>unsigned</code>	<code>void</code>	<code>volatile</code>	<code>while</code>





## 2. Types, Instructions and Operators

- Introduction
- C Language: The Basics
- **Types, Instructions and Operators**
  - Information Encoding in Computer Science
  - Data Types
  - Instructions
  - Operators
- Control Structures
- Functions and procedures
- Arrays
- Structures
- Pointers and Dynamic Memory Allocation
- Files

# Information Encoding in Computer Science

The objective of this recall is to explain the link between information encoding on a machine and source codes (in any programming language).

- In computer science, machines are electric components and can only interpret two states:
  - High voltage state (5v)
  - Low voltage state (0v)
- These two states correspond to two logical values: 1 for the high voltage and 0 for the low voltage
- Based on this, we use the **binary system** {0, 1} to represent all data in a computer
- **Bit**: a bit is a binary variable that takes either 0 or 1
- **Byte**: a byte is a sequence of 8 bits. Using a byte, we can represent  $2^8$  different states (i.e. 256 different states)

**Question:** using the bits (or bytes), how can we represent numbers and characters?

**Answer:** depending on the variable to be represented, we can determine how many bits are needed to represent all its possible values.

# Information Encoding in Computer Science

## Conversion

### Theorem

*Given  $x$  and  $y$  (integers) such that  $y > 1$ , there exists an integer  $n$  and a sequence  $(a_0, a_1, \dots, a_n)$  that gives:*

$$x = a_0 \cdot y_0 + a_1 \cdot y_1 + \dots + a_n \cdot y_n$$

*and  $a_i < y_i$  for  $i \leq n$ .*

To determine the sequence  $(a_0, a_1, \dots, a_n)$ , suffice it to apply the Euclidean division of  $x$  by  $y$  and to substitute  $x$  by the result of its division by  $y$  if the result is  $> y$ .

# Information Encoding in Computer Science

## Conversion Example

### Conversion example:

if  $x_{\text{binary}} = 1101$ , then  $x_{\text{decimal}} = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 13$

For the reverse operation (i.e. convert a decimal to binary), one can iteratively apply the Euclidean division by 2. Here is an example:

$$135_{\text{decimal}} = 1000\ 0111_{\text{binary}}$$

$$\begin{array}{r|l} 135 & 2 \\ \hline 1 & 67 \end{array} \quad \begin{array}{r|l} 67 & 2 \\ \hline 1 & 33 \end{array} \quad \begin{array}{r|l} 33 & 2 \\ \hline 1 & 16 \end{array} \quad \begin{array}{r|l} 16 & 2 \\ \hline 0 & 8 \end{array} \quad \begin{array}{r|l} 8 & 2 \\ \hline 0 & 4 \end{array} \quad \begin{array}{r|l} 4 & 2 \\ \hline 0 & 2 \end{array} \quad \begin{array}{r|l} 2 & 2 \\ \hline 0 & 1 \end{array} \quad \begin{array}{r|l} 1 & 2 \\ \hline 1 & 0 \end{array}$$

$$\begin{aligned} 135_{\text{decimal}} &= 2 \cdot 67 + 1 \\ &= 2 \cdot (2 \cdot 33 + 1) + 1 \\ &= 2^2 \cdot 33 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &\vdots \\ &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1000\ 0111_{\text{binary}} \end{aligned}$$

# Information Encoding in Computer Science

## Octal System

### Octal system:

A system that uses digits between 0 and 7 to encode numbers.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	...
Octal	0	1	2	3	4	5	6	7	10	11	12	13	...

Conversion between octal and binary is immediate. Every octal digit is represented by 3 bits.

### Example:

$$145_{\text{decimal}} = 221_{\text{octal}}$$

$$\begin{array}{r|l} 145 & 8 \\ \hline 1 & 18 \end{array} \quad \begin{array}{r|l} 18 & 8 \\ \hline 2 & 2 \end{array} \quad \begin{array}{r|l} 2 & 8 \\ \hline 2 & 0 \end{array}$$

Then,  $145_{\text{decimal}} = 010\ 010\ 001_{\text{binary}}$ .

# Information Encoding in Computer Science

## Hex System

### Hexadecimal system:

A system that uses digits between 0 and 9 and alphabets between A and F to encode numbers.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

Conversion between hex and binary is immediate. Every hex digit/alphabet is replaced by its equivalent 4 bits.

$45_{\text{hex}} = 4 \cdot 16^1 + 5 \cdot 16^0 = 69_{\text{decimal}} = 0100\ 0101_{\text{binary}}$  (note that  $4_{\text{decimal}} = 0100_{\text{binary}}$  and  $5_{\text{decimal}} = 0101_{\text{binary}}$ ).

To convert a decimal to hex, iterative Euclidean division is applied.

$$\begin{array}{r|l} 69 & 16 \\ 5 & 4 \end{array} \quad \begin{array}{r|l} 4 & 16 \\ 4 & 0 \end{array}$$

How can we convert  $177_{\text{decimal}}$ ?

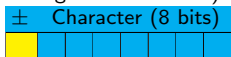
$$\begin{array}{r|l} 177 & 16 \\ 1 & B \end{array} \quad \begin{array}{r|l} B & 16 \\ B & 0 \end{array}$$

$177_{\text{decimal}} = B1_{\text{Hex}}$ .

# Information Encoding in Computer Science

## ASCII Code

- Integers (in the different systems) can be coded using 0 and 1
- Floating point numbers can also be represented using the binary system
- Characters are represented using 7 bits (one byte). For each character, we associate a number between 0 and 127, called **ASCII Code** (example: character **B** has the following ASCII code **66**).



- Floating-point numbers (`float` and `double`) are encoded using the following:

$$\text{value} = (\text{sign}) \text{ mantissa} \cdot 2^{\text{exponent}}$$



- The sign is represented by only one bit (highest ranking bit)
- The exponent is represented by 8 consecutive bits after the sign
- Mantissa (numbers after the point) is represented using the remaining 23 bits.



# Information Encoding in Computer Science

## ASCII Code

binnaire	oct	déc	hex		binnaire	oct	déc	hex		binnaire	oct	déc	hex	
0100000	040	32	20	espace	1000000	100	64	40	@	1100000	140	96	60	'
0100001	041	33	21	!	1000001	101	65	41	A	1100001	141	97	61	a
0100010	042	34	22	"	1000010	102	66	42	B	1100010	142	98	62	b
0100011	043	35	23	#	1000011	103	67	43	C	1100011	143	99	63	c
0100100	044	36	24	\$	1000100	104	68	44	D	1100100	144	100	64	d
0100101	045	37	25	%	1000101	105	69	45	E	1100101	145	101	65	e
0100110	046	38	26	&	1000110	106	70	46	F	1100110	146	102	66	f
0100111	047	39	27	'	1000111	107	71	47	G	1100111	147	103	67	g
0101000	050	40	28	(	1001000	110	72	48	H	1101000	150	104	68	h
0101001	051	41	29	)	1001001	111	73	49	I	1101001	151	105	69	i
0101010	052	42	2A	*	1001010	112	74	4A	J	1101010	152	106	6A	j
0101011	053	43	2B	+	1001011	113	75	4B	K	1101011	153	107	6B	k
0101100	054	44	2C	,	1001100	114	76	4C	L	1101100	154	108	6C	l
0101101	055	45	2D	-	1001101	115	77	4D	M	1101101	155	109	6D	m
0101110	056	46	2E	.	1001110	116	78	4E	N	1101110	156	110	6E	n
0101111	057	47	2F	/	1001111	117	79	4F	O	1101111	157	111	6F	o
0110000	060	48	30	0	1010000	120	80	50	P	1110000	160	112	70	p
0110001	061	49	31	1	1010001	121	81	51	Q	1110001	161	113	71	q
0110010	062	50	32	2	1010010	122	82	52	R	1110010	162	114	72	r
0110011	063	51	33	3	1010011	123	83	53	S	1110011	163	115	73	s
0110100	064	52	34	4	1010100	124	84	54	T	1110100	164	116	74	t
0110101	065	53	35	5	1010101	125	85	55	U	1110101	165	117	75	u
0110110	066	54	36	6	1010110	126	86	56	V	1110110	166	118	76	v
0110111	067	55	37	7	1010111	127	87	57	W	1110111	167	119	77	w
0111000	070	56	38	8	1011000	130	88	58	X	1111000	170	120	78	x
0111001	071	57	39	9	1011001	131	89	59	Y	1111001	171	121	79	y
0111010	072	58	3A	:	1011010	132	90	5A	Z	1111010	172	122	7A	z
0111011	073	59	3B	;	1011011	133	91	5B	[	1111011	173	123	7B	{
0111100	074	60	3C	<	1011100	134	92	5C	\	1111100	174	124	7C	
0111101	075	61	3D	=	1011101	135	93	5D	]	1111101	175	125	7D	}
0111110	076	62	3E	>	1011110	136	94	5E	^	1111110	176	126	7E	~
0111111	077	63	3F	?	1011111	137	95	5F	-	1111111	177	127	7F	suppr

... plus 32 control characters

# Standard Data Types

- `int`
  - to be used with integer variables
  - occupies 2 or 4 bytes (depending on the machine)
- `char`
  - to be used with characters or strings
  - occupies 1 byte
- `float`
  - to be used with floating point numbers
  - occupies 4 bytes
- `double`
  - to be used with floating point numbers with double precision
  - occupies 8 bytes
- `short`
  - to be used with integer variables
  - occupies 2 bytes
  - When `int` is encoded on more than 4 bytes, `short` is used with short integers
- `long`
  - to be used with integer variables
  - occupies 4 bytes
- `unsigned`
  - to be used with positive-only integer variables
- `signed`
  - to be used with integer variables
  - occupies 4 bytes

# How to declare a variable?

- Variable declaration is done using the following: `<type> variableName;`
- The number of bytes occupied by a variable depends on its type. To know the size of a variable, we use the function `sizeof`.

```
#include <stdio.h>
2 int main()
{
4     int A=40000;
    short B=32768;
6     printf("A = %d B= %d\n", A,B);
    printf("Size int = %d bytes\n", sizeof(int));
8     printf("Size char = %d bytes\n", sizeof(char));
    printf("Size short = %d bytes\n", sizeof(short));
10    printf("Size long = %d bytes\n", sizeof(long));
    printf("Size float = %d bytes\n", sizeof(float));
12    printf("Size double = %d bytes\n", sizeof(double));
    printf("Size long double = %d bytes\n", sizeof(long double));
14    return 0;
}
```

```
1 Size int = 4 bytes
  Size char = 1 bytes
3 Size short = 2 bytes
  Size long = 4 bytes
5 Size float = 4 bytes
  Size double = 8 bytes
7 Size long double = 12 bytes
```

# How to declare a variable?

## Syntax

syntax of variable declaration

```
1      <type> <id1>, ... , <idn>;
```

```
char i;
```

type            identifier

- **Type** determines the nature of the variable and its operations. If `unsigned` precedes the type of a variable (except `float` and `double`), the variable is unsigned and the bounds of its variables are extended.
- The **identifier** is the variable's name. The number of characters of a name is limited in most compilers.
  - Be careful of case-sensitivity:  
MyId  $\neq$  myid  $\neq$  MYID
  - The first character of a variable's name cannot be a digit
  - Characters with accents are not accepted in identifiers

# How to declare a variable?

## Example

```
/* Floating point declaration */
2 float r;
/* Integer declaration */
4 int i,j;
/* Character declaration */
6 char t;
/* Floating point (double) declaration */
8 double pi;
/* Byte declaration */
10 unsigned char BY;
/* Register variable declaration */
12 register unsigned char port;

14 void main()
{
16     r=10.14;
    i=2;
18     j=3;
    t='A'; /* t=65 ASCII code of A */
20     pi=3.14159;
    BY=129; /* We can exceed 127 */
22     port=34;
}
```

# How to declare a variable?

## Octal and Hexadecimal Notations

### Remark:

When assigning a variable, if 0 (zero) is put before a number, this value is considered in the octal system. If we put 0x (zero x), then it is considered in the hex system.

```
1  /* Integer declaration */
   int i;
3
   void main()
5  {
       i=21;      /* Decimal */
7
       i=025;     /* Octal */
9
       i=0x15;    /* Hex */
11 }
```

# How to declare a variable?

## Initialization

### Two ways to initialize a variable

```
1  /* Declaration */
   int i;
3
   void main()
5  {
       /* Initialization */
7      i=15;
       .
9      .
   }
```

```
   /* Declaration */
2  int i=15;

4  void main()
   {
6      .
       .
8      .
       .
10 }
```

# Pointers

- Every variable is stored in the memory which is made up of bytes, each having an address.
- To find (or use) a variable, suffice it to have its address
- A pointer is a variable that contains a memory address. It can be used to reference a variable and used as a type.

syntax of a pointer declaration

```
<type> *Variable;
```

- Variable is a pointer that contains the memory address of \*Variable

```
#include <stdio.h>
2 int main()
{
4     float *ptr;
    float a;
6     a=3.4;
    ptr=&a;
8     printf("The value referenced by ptr = %1.2f \n",*ptr);
    printf("The address = %d \n",ptr);
10    printf("The address of a is %d \n",&a);
    *ptr=a+1;
12    printf("The value referenced by ptr = %1.2f \n",*ptr);
    printf("The address = %d \n",ptr);
14    printf("The address of a is %d \n",&a);
    printf("The value of a = %1.2f \n",a);
16    return 0;
}
```



# Constants

- A constant is a memory space that does not change value during the program's execution.
- Different types of constants exist:
  - Integer: 4, 77, -98, ...
  - Floating point: 332.88, 15, 0.094, - 12.06, ....
  - Character: delimited using apostrophes 'a', 'E', ' ', '{', ....
  - Strings: a sequence of charactes delimited using quotation marks " 1a7", " 123o0"....

```

1 #include <stdio.h>
  int main()
3 {
    char car;
5    int i;
    float r;
7    char *str;
    car='a';
9    i=4;
    i=-5;
11   r=3.98;
    str="reau";
13   printf("String: %s\n",str);
    printf("Character: %c\n",car);
15   printf("Character's ASCII code: %d\n",car);
    printf("Integer: %d\n",i);
17   printf("Floating point: %f\n",r);
    return 0;
19 }
```

```

1 String: reau
  Character: a
3 Character's ASCII code: 97
  Integer: -5
5 Floating point: 3.980000
```

# Arrays

- Arrays are very useful when dealing with multiple variables of the same type
- We can define an array for all types (int, float, char, ...)
- In the C language (as in most programming languages), indexation starts from 0

syntax of array declaration

```
1      <type> array[nb_elements];
```

- Example: `int tab[5]`. The array `tab` has 5 elements
- `tab[0]` represents the first element, `tab[1]` the second, and so on ... the last element is `tab[4]` (`tab[5]` does not exist)
- In C, arrays can also be created using pointers (we'll see this later in the lectures).  
In this case, the size is not precised and should be handled differently?

syntax of an array declaration using a pointer

```
<type> *array;
```

- Example : `int *tab;`
- The same logic is applied whichever type is chosen for the elements.

# Arrays

## Example

```
#include <stdio.h>
2 int main()
{
4     char car[4];
    int tab[4];
6     int *table;
    char *str;
8     str="oiseau";
    car[0]=str[0];
10    car[1]=str[1];
    car[2]=str[2];
12    car[3]=str[3];
    printf("%c %c %c %c \n",car[0],car[1],car[2], car[3]);
14    printf("Size of car =%d bytes\n",sizeof(car));
    printf("Size of str =%d bytes\n",sizeof(str));
16    printf("Size of str =%d bytes\n",sizeof(*str));
    tab[0]=9;
18    tab[1]=2;
    tab[2]=3;
20    tab[3]=4;
    table=tab;
22    printf("%d %d %d %d\n",table[0],table[1],table[2], table[3]);
    printf("Size of table =%d bytes\n",sizeof(table));
24    printf("Size of table =%d bytes\n",sizeof(*table));
    printf("Size of tab =%d bytes\n",sizeof(tab));
26    return 0;
}
```

```
1 o i s e
  Size of car =4 bytes
3 Size of str =4 bytes
  Size of str =1 bytes
5 9 2 3 4
  Size of table =4 bytes
7 Size of table =4 bytes
  Size of tab =16 bytes
```

# Strings

- In C, a string is an array of characters that ends with `\0`.

syntax of a string declaration

```
char variableName[nbCharacters + 1];
```

```
1 char messageA[4] = "utt";
3 char messageB[] = "utt";
5 char messageC[4];
7 void main()
8 {
9     messageC[0]='u';
10    messageC[1]='t';
11    messageC[2]='t';
12    messageC[3]='\0'; /* End character */
13 }
```

- `messageC` is considered as a pointer by the compiler.  
Writing `messageC="utt"` is an error.  
Either we initialize each character in the string or we use the function `strcpy` from the library `string.h` to copy the string.

# Instructions

# Assignment

- Assignment is done using the operator =

syntax of assignment

```
variable=value;
```

After assignment, the variable takes value

```
2   int var;  
    var=7;
```

- We can also assign a variable to another one

syntax of assignment

```
variable1=variable2;
```

In this case, variable1 takes the value of variable2

```
2   int var1, var2;  
    var2=5;  
    var1=var2;
```

# Assignment

```
1 #include <stdio.h>
   int main()
3 {
   char car1, car2;
   float a=5, b=7;

7   car1='6';
   car2='z';
9   printf("%c and %c \n",car1, car2);
   car1=car2;
11  printf("%c and %c \n",car1, car2);

13  printf("%.2f and %.2f \n",a, b);
   a=a+b;
15  b=a-b;
   printf("%.2f and %.2f \n",a, b);

17
   return 0;
19 }
```

```
1 6 and z
   z and z
3 5.00 and 7.00
   12.00 and 5.00
```

# (Assignment Using) Casting

Assignment in C can be done between variables having different types while satisfying certain conditions:

- `int i; double x=4.3; i=x;` Here, `i` takes the value 4  
We have information loss after the casting.
- `int j; double y=6.1; j=(int)y;` The value 6 is assigned to `j`  
In this case, the casting is explicit.
- `int j=5; char y='d'; j=j+y;`  
In this case, the casting is implicit.

```

1 #include <stdio.h>
2 int main()
3 {
4     int i;
5     double x=4.3;
6     i=x;
7     printf("x=%1.1f and i=%d \n",x,i);
8     return 0;
9 }
```

```

1 #include <stdio.h>
2 void main()
3 {
4     char ca;
5     int x=114;
6     ca=(char)x;
7     x=x+ca;
8     printf("ca=%c and x=%d \n",ca, x);
9     ;
10    x=0;
11    ca='d'+14;
12    x=x+ca;
13    printf("ca=%c and x=%d \n",ca, x);
14    ;
15 }
```

```

1 ca=r and x=228
2 ca=r and x=114
```



# Cast

There exist two types of casting:

- Implicit casting is done between similar types (int, char, long). Casting is done from the type of the smallest size (eg. short) to the biggest (eg. long) to avoid information loss.

Example: from `char` to `int`, from `float` to `double`

- Explicit casting is done when changing the type is necessary. It is applied by using (type) in front of the identifier.

```
#include <stdio.h>
2  /* Variables declaration */
   char car;
4  int a,b,c;
   float g;
6
   void main()
8  {
       a=4;
10      b=7;
       c=0x41; /* ASCII Code of 'A' */
12
       /* Implicit Casting between a and b in float */
14      g= (a+b)/100. ;
       printf("g= %f\n",g);
16
       /* Explicit Cast c in char */
18      car = (char) c +3;
       /* c is integer and will be casted to char */
20      printf("car = %c\n",car);
   }
```

```
1  g=0.110000
   car=D
```

# Operators

## Arithmetic operators `+, -, *, /, \%`

### Operator `+`

- It can be used for increment: `x++`; and `++x`; are equivalent to `x=x+1`;
- It can be used in an expression:
  - `x++`; increments `x` after it has been executed.  
Example: `y=x++`;  $\iff$  `y=x`; `x=x+1`; (assign then increment)
  - `++x`; increments `x` before it has been treated.  
Example: `y=++x`;  $\iff$  `x=x+1`; `y=x`; (increment then assign)
  - `y += x`; allows to add `x` to `y`  $\iff$  `y = y + x`
- It can be used in multi-operation expressions
  - Example: addition of multiple variables: `y+=z+d`;  $\iff$  `y = y + z + d`;

### Operator `-`

- It can be used to represent a negative value of an expression. Example: `x = -x`;
- It can be used for decrement: `x--`; and `--x`;  $\iff$  `x = x - 1`;
- It can be used in an expression:
  - `x--`; decrements `x` after it has been executed.  
Example: `y = x--`;  $\iff$  `y=x`; `x=x-1`; (assign then decrement)
  - `--x`; decrements `x` before it has been treated.  
Example: `y = --x`;  $\iff$  `x=x-1`; `y=x`; (decrement then assign)
  - `y -= x`; allows to subtract `x` to `y`  $\iff$  `y = y - x`
- It can be used in multi-operation expressions
  - Example: `y-=z`;  $\iff$  `y=y-z`;

# Operators

## Operator `*`

- The classical multiplication operator.
  - Example: `x*=y;`  $\iff$  `x=x*y;`
- In a declaration, it allows to create a pointer.
- Example : pointer `int *ptr;`

## Operator `/`

- The classical division operator.
  - Example: `x/=y;`  $\iff$  `x=x/y;`
- If both variables are integer, the result is the result of the division (not the remainder).
  - Example: `x= 2/3;` `x=0` (and not 0.66 because it's the result of the Euclidean division)

## Operator `%`

- This operator allows to compute the remainder after division (modulo)
  - Example: `int x=303, y=4, z;` `z=x%y;` provides the remainder of the division `x/y` and puts it in `z`
- This operator can only be used between `int` variables (it doesn't work if one of the variables is `float`)

# Boolean Operators (expression)

As we have seen, any non-null value (conventionally 1) is considered as "True". 0 is considered as "False".

## Relational Operators



Operator	Meaning	Example
<code>==</code>	equal to	<code>7==7</code>
<code>!=</code>	is different than	<code>5!=7</code>
<code>&lt;</code>	less than	<code>5&lt;7</code>
<code>&gt;</code>	greater than	<code>5&gt;7</code>
<code>&lt;=</code>	less than or equal to	<code>5&lt;=7</code>
<code>&gt;=</code>	greater than or equal to	<code>5&gt;=7</code>

## Logical Operators

Operator	Meaning	Example
<code>!</code>	Logical negation	<code>int x=2, y=4, z; z=!((y&gt;x)&amp;&amp;(y*x&gt;7));</code>
<code>&amp;&amp;</code>	Logical AND	<code>int x=2, y=4, z; z=((y&gt;x)&amp;&amp;(y*x&gt;7));</code>
<code>  </code>	Logical OR	<code>int x=2, y=4, z; z=!((y&gt;x)  !(y*x&gt;7));</code>

# Bitwise Operators

## Bitwise Operators

Operator	Meaning	Example
	Bitwise Complement (to 1)	<code>unsigned short x=7, y; y=~x;</code> 7=0000000000000111 then y takes the value 65528 = 111111111111000
<code>&amp;</code>	Bitwise AND	<code>short i=5,j=11; i=i&amp;j;</code> i=1
<code> </code>	Bitwise OR	<code>short i=5,j=10; i=i j;</code> i = 15
	Bitwise Exclusive OR	<code>short i=5,j=11; i=i^j;</code> i=14
<code>&lt;&lt;</code>	Left shift	<code>int i, j; j=4; i=j&lt;&lt;2;</code> i = 16
<code>&gt;&gt;</code>	Right shift	<code>int i, j; j=4; i=j&gt;&gt;2;</code> i=1

Left shift: `0x01<<4;` becomes `0x10`

Right shift: `0x010>>4;` becomes `0x0`

# Ternary Operator

## Ternary Operator

```
x = condition ? <expression1> : <expression2>;
```



```
1 if (condition)
   x=<expression1>;
3 else
   x=<expression2>;
```

## Examples:

```
sign=x>0?-1;
2 // is equivalent to
if (x>0)
4   sign=1;
else
6   sign=-1;
```

```
is_odd=x%2==1?1:0;
2 // is equivalent to
if (x%2==1)
4   is_odd=1;
else
6   is_odd=0;
```

# Ternary Operator

## Examples

```

1  int i, j;
2  j=4;
   i=j>>2;
4  i=(i<j)?i+1:i-1;

```

```

#include <stdio.h>
2
int main()
4 { int x=4, y=6, z;
   int i, j;
6   z=x!=y;
   j=4;
8   i=j<<2;
   printf("i=%d and z=%d\n",i,z);
10  i=(i<=j)?i+1:i-3;
   printf("i=%d and j=%d \n",i, j);
12  i+=j%i;
   z+=!((i>j)||!(j*i>7));
14  printf("i=%d and z=%d \n",i, z);
   --i=i+(j>z);
16  printf("%i=d and j=%d \n",i, j);

18  return 0;
}

```

```

1  i=16 and z=1
   i=13 and j=4
3  i=17 and z=1
   17=d and j=4

```

# Priority and Associativity of Operators

Priority	Operators	Associativity
14	<code>++ -- ()</code>	Left
13	<code>++ -- ! ~ sizeof &amp;</code>	Right
12	<code>* / %</code>	Left
11	<code>+ -</code>	Left
10	<code>&lt;&lt; &gt;&gt;</code>	Left
9	<code>&lt; &lt;= &gt; &gt;=</code>	Left
9	<code>== !=</code>	Left
7	<code>&amp;</code>	Left
6	<code>^</code>	Left
5	<code> </code>	Left
4	<code>&amp;&amp;</code>	Left
3	<code>  </code>	Left
2	<code>?:</code>	Right
1	<code>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;=  =</code>	Right





## 3. Control Structures

- Introduction
- C Language: The Basics
- Types, Instructions and Operators
- **Control Structures**
  - Control Structures
  - while, do --- while, for
  - if, if --- else, switch
  - break, continue, goto
- Functions and procedures
- Arrays
- Structures
- Pointers and Dynamic Memory Allocation
- Files

# Control Structures

# Control Structures

- Allow the developer to have flexibility in coding
- Improve the code readability  
Example: execute an instruction 100 times. Loop structures simplify this instead of writing the instruction 100 times.
- Example: we want to compute the following sum:  $1*1+2*2+3*3+...+100*100$ 
  - Solution 1: write the 100 terms one after another (is it really a good solution?)
  - Solution 2: use a loop

# Basic Control Structures

**Loops** are used to repeat a set of instructions

- `while`
- `do --- while`
- `for`

**Conditional branching structures** select which sections of the code to execute and in which order.

- `if`
- `if --- else`
- `switch`

**Non-conditional branching structures** are used to move from one instruction to another one that does not come immediately after.

- `break`
- `continue`
- `goto`

# Control Structure: while

## Control Structure: while

The **while** loop executes iteratively a set of instructions. Its syntax is the following:

```
2      while (condition)
      {instruction1; instruction2; instructionN;};
```

- While the condition is verified, the instructions within the braces (called the **while block**) are executed
- When the condition is not verified initially, the while block is not executed
- Example: compute the sum  $1*1+2*2+3*3+\dots+100*100$

```
#include <stdio.h>
2 int main()
  {
4     int S=0, i=1;
     while(i<=100)
6     {
           S+=i*i;
8           i++;
        }
10    printf("Sum S = %d\n", S);
     return 0;
12 }
```

## Control Structure: do -- while

The structure `do -- while` is another form of the `while` but acts a little bit differently. Its syntax is the following:

```
2      do {instruction1; instruction2; instructionN;}
      while (condition);
```

The instructions within the braces will be executed at least one time, even if the condition is not satisfied initially.

- The following program computes the following sum:  $1*1+2*2+3*3+...+100*100$

```
#include <stdio.h>
2 int main()
{
4     int S=0, i=1;
    do
6     {
        S+=i*i;
8        i++;
    }
10    while (i<=100);
    printf("Sum S = %d\n", S);
12    return 0;
}
```



# Control Structure: for

The structure `for` executes iteratively a set of instructions. Its syntax is the following:

```
for (expression1_init; expression2_cond; expression3_update;)
2   {instruction1; instruction2; instructionN;}
```

- `expression1_init`: expression that initializes the variable
- `expression2_cond`: condition to be verified by the variable
- `expression3_update`: variable update instructions

The `for` loop can be transformed into a while loop by doing the following:

```
expression1_init;
2 while (expression2_cond)
   { instruction1; instruction2; instructionN;
4   expression 3_update;
   }
```

# Control Structure: for

## Example

Example: compute the sum  $1*1+2*2+3*3+...+100*100$

```
1 #include <stdio.h>
   int main()
3 {
   int S=0,i;
5   for(i=1;i<=100;i++)
       S+=i*i; // notice that we don't have braces since we have one instruction
7   printf("Sum S = %d\n", S);
   return 0;
9 }
```

The three expressions in the `for` loop can use multiple instructions separated by commas.

```
#include <stdio.h>
2 int main()
{
4   int S,i;
   for(i=1,S=0; i<=100; S+=i*i,i++);
6   printf("Sum S = %d\n", S);
   return 0;
8 }
```

## Control Structures: Exercise

**Exercise:** Write a program that takes an integer between 1 and 10 from the user using structures `while`, `do --- while` and `for`

structure `for`

```
#include <stdio.h>
2 int main()
{
4     int S,i;
    for(printf("Enter an integer between 1 and 10:"),
6         scanf("%d",&i);
        (i<=0||i>10);
8         printf("\nEnter an integer between 1 and 10:"),
        scanf("%d",&i));
10    printf("i = %d\n", i);
    return 0;
12 }
```

# Control Structures: Exercise

## structure do --- while

```
#include <stdio.h>
2 int main()
{
4     int i;
    do {
6         printf("\nEnter an integer between 1 and 10:");
        scanf("%d",&i);
8     } while ((i<=0)|| (i>10));
    printf("Entered integer = %d\n", i);
10    return 0;
}
```

## structure while

```
#include <stdio.h>
2 int main()
{
4     int i;
    printf("\nEnter an integer between 1 and 10:");
6     scanf("%d",&i);
    while ((i<=0)|| (i>10)){
8         printf("\nEnter an integer between 1 and 10:");
        scanf("%d",&i);
10    }
    printf("Entered integer = %d\n", i);
12    return 0;
}
```

# Control Structure: if

## Control Structure: if

The structure `if` is a conditional structure that executes a set of instructions only if a condition is verified. Its syntax is the following:

```
if (condition) {instruction1; instruction2; instructionN;};
```

Example: absolute value of an integer

```
1 #include <stdio.h>
   int main()
3 {
    int i;
5    printf("Enter an integer:");
    scanf("%d",&i);
7    if(i<0)
        i=-i; // notice that we don't have braces since we have one
              instruction
9    printf("Absolute value = %d\n", i);
    return 0;
11 }
```

## Control Structure: if --- else

The structure `if --- else` is a conditional structure that executes a set of instructions only if a condition is verified. If it is not, another set of instructions is executed. Its syntax is the following:

```
if (condition)
2   {instruction1; instruction2; instructionN;}
else
4   {instruction1; instruction2; instructionK};
```

Example: check whether an integer is positive or negative

```
#include <stdio.h>
2 int main()
{
4     int i;
    printf("Enter an integer:");
6     scanf("%d",&i);
    if(i<0)
8         printf("Your integer is negative\n");
    else
10        printf("Your integer is positive or null\n");
    return 0;
12 }
```

## Control Structure: if --- else if --- else

```
if (condition1)
2   {instruction1; instruction2; instruction N1;}
else if (condition2)
4   {instruction1; instruction2; instructionN2;}
else if (condition3)
6   {instruction1; instruction2; instructionN3;}
else
8   {instruction1; instruction2; instructionK};
```

Example: check if an integer is positive, negative or null

```
#include <stdio.h>
2 int main()
{
4   int i;
   printf("Enter an integer:");
6   scanf("%d",&i);
   if(i<0)
8       printf("Your integer is negative\n");
   else if (i==0)
10      printf("Your integer is null\n");
   else
12      printf("Your integer is positive\n");
   return 0;
14 }
```



## Control Structure: switch

The structure `switch` is a conditional structure that executes a set of instructions according to the value of a variable. Its syntax is the following.

```
switch (variable){  
2     case val1 : instruction1; break;  
     case val2 : instruction2; break;  
4     default  : instructionN;}
```

If the variable does not have any of the listed values, the set of instructions that correspond to the `default` is executed.

Example: input an integer and verify if it is a multiple of 5.

```
#include <stdio.h>  
2 int main(){  
    int i,j;  
4    printf("Enter an integer:");  
    scanf("%d",&i);  
6    j=i%5;  
    switch(j) {  
8        case 1 : printf("i=5n+1\n"); break;  
        case 2 : printf("i=5n+2\n"); break;  
10       case 3 : printf("i=5n+3\n"); break;  
        case 4 : printf("i=5n+4\n"); break;  
12       default : printf("It's a multiple of 5\n"); break;  
    }  
14    return 0;  
}
```

# Control Structure: break

## Control Structure: break

The structure `break` exits a loop before it finishes. The program then executes the first instruction after the loop.

`break` can be used in all loops seen previously.

Example: compute the sum  $1*1+2*2+3*3+\dots+100*100$  using structure `for` and `if`  
--- else

```
#include <stdio.h>
2 int main()
{
4     int S=0,i;
    for(i=1; ;i++)
6     {
        if (i<=100)
8            S+=i*i;
        else
10           break;
    }
12    printf("Sum S = %d\n", S);
    return 0;
14 }
```

## Control Structure: continue

The structure `continue` skips the current loop iteration to the next iteration without finishing the remaining instructions of the current iteration.

Continue can be used in all loops seen previously.

Example: compute the sum  $2*2+4*4+\dots+100*100$  using `for` and `if`

```
#include <stdio.h>
2 int main()
{
4     int S=0,i;
    for(i=1;i<=100;i++)
6     {
        if (i%2!=0)
8            continue;
        S+=i*i;
10    }
    printf("Sum S = %d\n", S);
12    return 0;
}
```

## Control Structure: goto

Structure `goto` jumps from one instruction to another using labels.

It can be used everywhere, **but ...**

Example: determine the least common multiple of two integers.

```
#include <stdio.h>
2 int main()
{
4     int i,j,max,lcm;
    printf("\nEnter the first integer:");
6     scanf("%d",&i);
    printf("\nEnter the second integer:");
8     scanf("%d",&j);
    max =i;
10    if (j>max)
        max =j;
12    for(lcm=max; ;lcm+=max)
    {
14        if ((lcm%i==0)&&(lcm%j==0))
            goto display;
16    }
    i+=j;
18    display: printf("The lcm of %d and %d = %d\n", i,j,lcm);
    return 0;
20 }
```

## 4. Functions and procedures

- Introduction
- C Language: The Basics
- Types, Instructions and Operators
- Control Structures
- **Functions and procedures**
  - Functions/procedures
  - Global/Local Variables
  - Argument passing by value / by address
  - Recursivity
  - Mathematical Functions
  - `main` Function
  - Examples
  - Generic Functions
- Arrays
- Structures
- Pointers and Dynamic Memory Allocation
- Files

# Functions

# Functions

- Functions and procedures are used to improve the code modularity.
- They also allow to improve the code's compactness
- A function/procedure can be called several times in the same program

```

1 <type> function_name(<type1> arg1, <type2> arg2, <typeN> argN)
  // function's parameters arg1, arg2, argN
3 {
    instruction1;
5    instruction2;
    instructionN;
7
    return val;
9 }

```

A function returns a value using **return**.

**return** has the same impact a **break** has on a loop: it terminates the function.

```

/* function area returns an int, its parameters are int length, int
   width */
2 int area(int length, int width)
  {
4    int res;          /* local variables */
    res=length*width;
6    return res;
  }

```



# Functions

```
1 #include <stdio.h>

3 int area(int length, int width)
4 {
5     int res;
6     res=length*width;
7     return res;
8 }

9 void main()
10 {
11     int l,w,AR;
12     printf("\n Enter the length:");
13     scanf("%d", &l);
14     printf("\n Enter the width:");
15     scanf("%d", &w);
16     AR=area(l,w); // call for the function area
17     printf("\n Area = %d\n",AR);
18 }
19 }
```

# Procedures

```

1 void Procedure_name(<type1> arg1, <type2> arg2, <typeN> argN)
  // procedure's parameters arg1, arg2, argN
3 {
    instruction1;
5    instructionN;
  }

```

A procedure is a function that does not return any value. The procedure can modify the values of certain parameters (passed by address).

```

void area(int length, int width, int *res){
2    *res=length*width;
  }

```

```

1 #include <stdio.h>
void area(int length, int width, int *res){
3    *res=length*width;
  }
5 void main() {
    int l,w,AR;
7    printf("\n Enter the length:");
    scanf("%d", &l);
9    printf("\n Enter the width:");
    scanf("%d", &w);
11   area(l,w, &AR);    // procedure call of area
    printf("\n Area = %d\n",AR);
13 }

```

## Function/Procedure call

- A function can be declared using its prototype without detailing its instruction (without definition). Once the function is declared using the prototype, it can be called before its definition.
- Function call: Variable = function\_name(arg1, arg2, argN);
- Procedure call: procedure\_name(arg1, arg2, argN);
- A function/procedure can have 0 arguments/arguments

```
void print () {printf("\n Enter a value: ");}
```

```
1 double pi() { return(3.14159); }
```

```
1 #include <stdio.h>

3 int area(int length, int width);
  void message();
5
  void main() {
6     int l,w,AR;
      message(); scanf("%d", &l);
9     message(); scanf("%d", &w);
      AR=area(l,w);      /* call of area function before its call*/
11    printf("\n Area = %d\n",AR);
  }
13
  int area(int length, int width) { return length*width; }
15 void message() { printf("\nEnter a value:");}
```

# Global Variables

# Global Variables

- A global variable is a variable declared outside any function
- Global variables are *static* or *permanent* (a permanent variable is kept in memory as long as the program is running)
- Static variables are initialized to 0 by the compiler

```
#include <stdio.h>
2
int n; // Global variable
4
void fct()
6 {
    n++;
8     printf("call number %d\n",n);
}
10
int main()
12 {
    int i; // Local variable
14     printf("Initial value of n = %d\n",n);
    for (i = 0; i < 5; i++)
16         fct();
    return 0;
18 }
```

# Local Variables

- A local variable is declared inside a block (a function/procedure, a loop, a condition, etc.)
- Local variables are temporary, i.e. when their block ends, they are destroyed (at the }).

```
#include <stdio.h>
2
int n; // Global variable
4
void fct()
6 {
    n++;
8     printf("call number %d\n",n);
}
10
int main()
12 {
    int i; // Local variable
14     printf("Initial value of n = %d\n",n);
    for (i = 0; i < 5; i++)
16         fct();
    return 0;
18 }
```

# Static Local Variables

```
static <type> variable_locale;
```

- The value of a static local variable is kept between the calls. The variable is not destroyed at the end of the block.
- Static local variables are initialized to 0 by the compiler

```
1  #include <stdio.h>

3  int n;  // Global variable

5  void fct()
{
7      static int n;
      n++;
9      printf("call number %d\n",n);
}

11

13 int main()
{
14     int i;  // Local variable
15     printf("Initial value of n = %d\n",n);
16     for (i = 0; i < 5; i++)
17         fct();
18     return 0;
19 }
```

# Argument passing

To call a function with parameters, they have to be provided with the call and can be changed inside the function.

Two argument passing modes exist:

- Argument passing by value: parameters are copied and their copies are used inside the function.  
The initial argument are **never modified**.
- Argument passing by address: parameters are passed to the function using their addresses. Using their addresses, they **can** be modified. They are not necessarily modified inside the function



# Argument passing by value / by address

## Argument passing by value

```
1 #include <stdio.h>
2 void effective_area
3   (float length, float width)
4 {
5   float res;
6   length*=0.9;
7   width*=0.8;
8   res=length*width;
9   printf("area = %f\n",res);
10 }
11
12 void main()
13 {
14   float l,w;
15   printf("\nEnter the length: ");
16   scanf("%f", &l);
17   printf("\nEnter the width:");
18   scanf("%f", &w);
19   effective_area(l,w);
20   printf("length = %f\n",l);
21   printf("width = %f\n",L);
22 }
```

## Argument passing by address

```
1 #include <stdio.h>
2 void effective_area
3   (float *length, float *width)
4 {
5   float res;
6   *length*=0.9;
7   *width*=0.8;
8   res=*length*(*width);
9   printf("area = %f\n",res);
10 }
11
12 void main()
13 {
14   float l,w;
15   printf("\nEnter the length: ");
16   scanf("%f", &l);
17   printf("\nEnter the width:");
18   scanf("%f", &w);
19   effective_area(&l,&w);
20   printf("length = %f\n",l);
21   printf("width = %f\n",L);
22 }
```

# Recursivity

# Recursivity

A function is recursive if it calls itself.

Example: the factorial function can be defined as follows:

$\text{fact}(0) = 1$  et  $\text{fact}(n) = n \times \text{fact}(n-1)$  pour  $n > 0$

```
#include <stdio.h>
2 int fact(int n)
{
4     if(n==0) return 1;
    else return (n*fact(n-1));
6 }
void main()
8 {
    printf("fact(7)=%d\n", fact(7));
10 }
```

Example: Compute  $x^n$  using recursivity

```
#include <stdio.h>
2 float power(float x, int n)
{
4     if(n==0) return 1;
    else return (x*power(x,n-1));
6 }
void main()
8 {
    printf("Power(2,2)=%f\n", power(1.5,2));
10 }
```

# Recursivity

Disadvantage of recursivity: considerable computing time (several useless function calls)

```

#include <stdio.h>
2 int prime(int n)
{
4     if(n==2) return 1;
    else
6         {
            int k=2;int z=1;
8             while(k<n)
                {
10                 while ((prime(k)==0)&&(k<n-1)) {k++;};
                    if (n%k==0) {z=0;k++;} else k++;
12                };
            return z;
14        };
}
16 void main()
{
18     int i;
    for(i=2;i<100;i++)
20         printf("prime(%d)=%d\n",i, prime(i));
}

```

# Mathematical Functions

# Mathematical Functions

- Most C compiler have mathematical libraries with several predefined functions
- To be able to use these functions, it is necessary to include `<math.h>`. Other functions can be used by including `<stdlib.h>`.

Function	library	description
<code>fabs</code>	<code>&lt;math.h&gt;</code>	Absolute value of a float
<code>exp</code>	<code>&lt;math.h&gt;</code>	Exponential of a value (int or float)
<code>cos</code>	<code>&lt;math.h&gt;</code>	cosine of input value (float or int)
<code>sin</code>	<code>&lt;math.h&gt;</code>	sin of input value (float or int)
<code>acos</code>	<code>&lt;math.h&gt;</code>	arccosine of input value (float or int)
<code>asin</code>	<code>&lt;math.h&gt;</code>	arcsine of input value (float or int)
<code>ceil</code>	<code>&lt;math.h&gt;</code>	rounds a float up to the next largest integer
<code>log</code>	<code>&lt;math.h&gt;</code>	logarithm of a float
<code>min</code>	<code>&lt;stdlib.h&gt;</code>	Minimum of two input values (float or int)
<code>max</code>	<code>&lt;stdlib.h&gt;</code>	Maximum of two input values (float or int)
<code>pow</code>	<code>&lt;math.h&gt;</code>	power of a float ( $x^n$ )
<code>rand</code>	<code>&lt;stdlib.h&gt;</code>	generates a random int between 0 and the input value (int)

# main

# main

- Function/procedure `main` without a parameter:
  - `void main (void)`
  - `int main () return 0;`  
The returned value (0) corresponds to the successful execution of the program (EXIT\_SUCCESS)
  - `int main () return 1;`  
The returned value (1) corresponds to an error during the execution of the program (EXIT\_FAILURE)
- The main function can have parameters which correspond to the arguments that the program receives when launched
- The command line (or in the IDE) is made of the program's name followed by the parameters that are passed to the program

`main`

```
int main (int argc, char *argv[]) { return 0; /* or 1 */ }
```

- `argc` (argument count): variable of type `int` that provides the number of arguments passed to the program (including the program's name).
- `argv` (argument vector): array of strings that correspond to the different parameters. The size of the array is `argc`.
- `argv[0]` is the program's name
- `argv[1]` is the first parameter
- `argv[2]` is the second parameter



# Example 1

# Example 1

Example 1: Compute the sum:  $1*1+2*2+3*3+...+N*N$ .  $N$  is given to the program using command line.

```

1  #include <stdio.h>
   #include <stdlib.h>
3
   int main(int argc, char *argv[])
5  {
       int a;
7      printf("\nNumber of parameters =%d",argc);
       if (argc != 2)
9      {
           printf("\nError: invalid number of arguments");
11          printf("\nUsage: %s int\n",argv[0]);
           return(1);
13      }
       printf("\n argc= %d; argv[0]= %s; argv[1]=%s\n", argc,argv[0],argv[1]);
15       a = atoi(argv[1]); // convert to an int
       int S=0, i=1;
17       while(i<=a)
       {
19           S+=i*i; i++;
       }
21       printf("Sum (n=%d): S = %d\n",a, S);
       return(0);
23 }

```

## Example 2

Example 2: write a program that allows to input a number of integers and that finds their maximum. Then, it calls the previous program (called Sum) with the computed maximum

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

4  int main()
    {
6      int i,n,j,k=0;
        printf("Enter the number of elements:");
8      scanf("%d",&n);
        for (i=0;i<n;i++)
10     {
            printf("Enter %d ith integer: ",i+1);
12         scanf("%d",&j);
            if (j>k)
14             k=j;
        }
16     char str[6];
        sprintf(str,"%d",k); // converts an int to a string
18     execl("C:\\Code", "Sum", str, NULL);
        // execl executes "Sum" located in "C:\\Code", with the argument
        // "str"
20     // (char*)0 or NULL to indicate the end of list of arguments
        return 0;
22 }
```

# Generic Functions

# Generic Functions

Tip: write a generic function without precising the types

```
#include <stdio.h>
2 #define maxi(a,b) (a>b)?a:b

4 int main()
{
6     float l=2,L=4,m;
      int i=2,j=9,mij;
8     char ca='r',cb='Z',mc;

10     m=maxi(l,L);
      mij=maxi(i,j);
12     mc=maxi(ca,cb);

14     printf("maxi_float = %f\n",m);
      printf("maxi_int = %d\n",mij);
16     printf("maxi_char = %c\n",mc);
      return 0;
18 }
```

## 5. Arrays

- Introduction
- ① C Language: The Basics
- ② Types, Instructions and Operators
- ③ Control Structures
- ④ Functions and procedures
- ⑤ **Arrays**
  - Generalities on Arrays
  - One Dimensional Array
  - Multi-dimensional Arrays
  - Arrays and Functions/Procedures
  - Strings
- ⑥ Structures
- ⑦ Pointers and Dynamic Memory Allocation
- ⑧ Files

# Generalities on Arrays

# Generalities on Arrays

- Arrays are mandatory and practical structures to manage several variables of the same type
- In C, we can define arrays of any type:  
`int, float, double, char, ...`
- Types of arrays
  - One dimensional array
  - Multi-dimensional array
  - Strings



# One Dimensional Arrays

# One Dimensional Arrays

## Array

```
1 <type> tab[Nb_elements];
```

- The size of an element of an array is the number of bytes of the type of the element
- In C, the indexation starts from 0

Example: `int tab[5];`

- `tab` is a constant pointer whose value is the address of the first element of the array. In other words:
  - the value of `tab` is `&tab[0]`: it's an address
  - the value of `tab+i` is `&tab[i]`: it's also an address
  - the value of `*(tab+i)` is `tab[i]`: it's a value!
- The array `tab` is made up of 5 elements, each of 4 bytes (representing an `int`).

```
#include <stdio.h>
2 void main()
{
4   char tab[]={'2', '5', '7', 'r'};
   int i;
6   printf("\nSize = %d",sizeof(tab));
   printf("\nMemory address of tab[0]: %d, tab[1]:%d",&tab[0], &tab[1]);
8   printf("\nMemory address of pointer tab: %d, tab+2: %d\n",&tab[0], &tab[2]);
   for (i=0;i<4;i++)
10    printf("\nValue of tab[%d]= %c, of *(tab+%d)=%c ",i,tab[i],i, *(tab+i));
}
```

# One Dimensional Arrays

## Initialization

### Array Initialization:

- During the declaration:
  - `char tab[4]={ '2', '5', '7', 'r' }`; . The size is defined.
  - `char tab[]={ '2', '5', '7', 'r' }`; The size is not determined
  - `char tab[10]={ '2', '5', '7', 'r' }`; the first fourth elements are initialized.
  - `char tab[10]={ [4] = '1', [7] = '8' }`; the 5<sup>th</sup> and the 8<sup>th</sup> elements are initialized.
- After the declaration using a loop (`for`, `while`, ...) :

```
#include <stdio.h>
2 void main()
{
4     int tab[10]={0,1,4,9,16};
    int i;
6     for(i=5;i<10;i++) tab[i]=i*i;
    for(i=0;i<10;i++)
8         printf("tab[%d]=%d\n",i,tab[i]);
}
```

# One Dimensional Arrays

## Pointers and Arrays

### Pointers and Arrays

- The C language allows to declare an array without providing its size by using a pointer. In this case, it's called a **dynamic array**:

Array

```
1      <type> *tab;
```

Example: `int *point;`

- `point` is a pointer. In the following example, `tab` and `point` refer to the same memory address:

```
#include <stdio.h>
2 void main()
{
4     char tab[]={'y', '5', '7', 'r'};
    char *point;
6     point=tab;
    printf("point[3] = %c\n",point[3]);
8 }
```

# One Dimensional Arrays

## Access

### Access :

In C, it is possible to access to the  $(k+1)^{\text{th}}$  element of the array `tab` using `tab[k]`. This expression represents an independent variable and is used as any other variable of the same type (assignment, access, printing, ...)

```
#include <stdio.h>
2 void main()
{
4     char tab[7];
    int i;
6     for(i=0;i<=6;i++)
    {
8         tab[i]=(char)i+100;
        printf("tab[%d]=%c\n",i,tab[i]);
10    }
}
```

# Multi-dimensional Arrays

# Multi-dimensional Arrays

- To declare a multi-dimensional array, the same syntax of one-dimensional array is used:

```
1      <type> tab[N1][N2][Nn];
```

- The following declaration is also allowed:

```
1      <type> tab[][N2][Nn];
```

```
1  #include <stdio.h>
   void main()
3  {
   int t[][2]={1, 2, 3, 4};
5   int i,j;
   t[2][0]=5;
7   t[2][1]=5;
   for(i=0;i<2;i++)
9   {
       for(j=0;j<2;j++)
11      printf("t[%d][%d]=%d\n",i,j,t[i][j]);
   }
13 }
```

# Multi-dimensional Arrays

## Exercise

### Exercise:

Class management: there is N students (N is given by the user) and it is required to code a program that takes all their marks (tests, midterm and final exams). The proposed program must compute for each student their grade by combining the grades of the three assessments using the following weighting: 30%, 25% and 45% respectively. The program has to print the list of students and their final grades in a descending order.



# Arrays and Functions/Procedures

# Arrays and Functions/Procedures

## Arrays and functions:

Recall that the elements of an array can be considered as independent variables (input/output) and they can be used as parameters of functions and procedures

```
#include <stdio.h>
2 int sum(int k, int tab[]) // or int sum(int k, int *tab)
{
4     int i, s=0;
    for(i=0;i<k;i++)
6         s+=tab[i];tab[i]+=1;
    return s;
8 }
int main()
10 {
    int i, tab[5]={1, 2, 3, 4, 5};
12    printf("Input array:");
    for(i=0;i<5;i++)
14        printf(" %d ",tab[i]);

16    printf(". Somme =%d \n",su(5,tab));
    printf(" Obtained array:");

18    for(i=0;i<5;i++)
20        printf(" %d ",tab[i]);

22    printf(". Sum =%d \n",sum(5,tab));
    return 0;
24 }
```

# Arrays and Functions/Procedures

## Exercise

**Exercise:** write a function/procedure to input an integer array of size N (input by the user). Write a function that sorts the array in an ascending order.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int *input(int *k)
4 {
5     int *res, i;
6     printf("Provide the number of
7         elements of the array:");
8     scanf("%d",k);
9     res = (int*)malloc(*k*sizeof(int));
10    printf("\n Input the elements of the
11        array:");
12    for(i=0;i<*k;i++)
13        scanf("%d",&res[i]);
14    return res;
15 }
16 void sort_func(int k, int *tab)
17 {
18     int i, j, aux;
19     for(i=0;i<k-1;i++)
20     {
21         for(j=i+1;j<k;j++)
22         {
23             if(tab[i]>tab[j])
24             {
25                 aux=tab[i];
26                 tab[i]=tab[j];
27                 tab[j]=aux;
28             }
29         }
30     }
31 }

```

```

1 int main()
2 {
3     int *ta;
4     int k=1,i;
5     ta=input(&k);
6     printf("\nArray provided:");
7     for(i=0;i<k;i++)
8         printf(" %d ",ta[i]);
9     sort_func(k,ta);
10    printf("\nSorted Array:");
11    for(i=0;i<k;i++)
12        printf(" %d ", ta[i]);
13    return 0;
14 }

```

# Strings

# Strings

- Strings are arrays of characters that end with the character `NULL` or `\0`,

a	n		e	x	a	m	p	l	e	\0
---	---	--	---	---	---	---	---	---	---	----

- Always allocate an additional element for the `NULL` character and make sure that it is assigned to the last character
- String functions (of the library `string.h`) are dependent on the `NULL` character
- it is not necessary to know the size of a string as the `NULL` character allows to detect the end of the string (which is an array)
- Warning: if you assign this character at the middle of a string, it is broken and the characters that come after the `NULL` character are not considered in the different functions (in the `printf` function for example).
- In some cases, the compilers adds the `NULL` character automatically.

# Strings

```
#include <stdio.h>
2 void main()
{
4     char str1[20];
    char *str2="and this is a second string";
6     int i;

8     for(i=0;i<19;i++)
        str1[i]=i+100;
10    str1[i]='\0';

12    printf("str1 = %s\n", str1);
    printf("str2 = %s\n", str2);
14
    str1[10]='\0';
16    printf("str1 = %s\n", str1);

18 }
```

# Predefined Functions for Strings

Function	Library	Description
strlen	<string.h>	Length of the a string
strcpy	<string.h>	Copies a string in another
strcat	<string.h>	Appends a string using another
strcmp	<string.h>	Compares two strings (0 if equal, 1/-1 otherwise)
atof	<stdlib.h>	Converts a string into a float
atol	<stdlib.h>	Converts a string into a long
atoi	<stdlib.h>	Converts a string into an int
isalpha	<ctype.h>	Checks whether a character is a number
isprint	<ctype.h>	Checks whether a character is printable
ispunct	<ctype.h>	Checks whether a character is a punctuation character
isascii	<ctype.h>	Checks whether a character is an ASCII (code is between 0 and 127)

# Arrays and Strings

How to determine the size?

- For a static array, use the function `sizeof()` with static arrays. **Never use `sizeof` with a pointer array or an array allocated dynamically.**
- For a string, always use `strlen()` (while including `<string.h>`)



## 6. Structures

- Introduction
- C Language: The Basics
- Types, Instructions and Operators
- Control Structures
- Functions and procedures
- Arrays
- **Structures**
  - struct
  - typedef
  - enum
  - union
- Pointers and Dynamic Memory Allocation
- Files

# Structures

# Structures

- In C, we can define new types (on top of the standard types). This can be done using **structures**.
- A structure is made up of multiple other variables that can be of standard or non-standard types.
- Each element of structure, called **member** is identified by its name
- A structure is defined using the following syntax:

## Structure

```
1 struct structure_name{<type1> var1; <type2> var2; <typeN> varN};;
```

```
1 struct student
{
3     char *name;
   float midterm_grade;
5     float final_grade;
   float average;
7 };;
```

Remark: the different members of the structure do not occupy a back-to-back memory spaces. They can be allocated in different addresses that are far from each other.

# Structures

## Variable of a structure type

```
1 struct structure_name Variable;
```

To access a member of a structure, we use the operator `.`

```
1 #include <stdio.h>
   struct student
3 {
   char *name;
5   float midterm_grade;
   float final_grade;
7   float average;
   };
9
   int main()
11 {
   struct student Dupont, Ami;
13   Dupont.name="DUPONT";
   Dupont.midterm_grade=17;
15   Dupont.final_grade=8.5;
   Dupont.average=0.4*Dupont.midterm_grade+0.6*Dupont.final_grade;
17   Ami=Dupont;
   Ami.name="RAYAN";
19   printf("Name : %s Average : %1.2f\n",Dupont.name, Dupont.average);
   printf("Name : %s Average : %1.2f\n",Ami.name, Ami.average);
21   return 0;
   }
```

# Structures

```
struct structure_name Variable;
```

To access a member of a structure using its pointer (or its address), we use the operator  
->

```
#include <stdio.h>
2 struct student {
    char *name;
4    float midterm_grade;
    float final_grade;
6    float average;
} ;
8
void modify_grades(float a, float b, struct student *s) {
10    s->midterm_grade=a; s->final_grade=b; s->average=0.4*a+0.6*b;
}
12 int main() {
    struct student Dupont={"DUPONT", 17, 8.5, 0};
14    struct student Ami;
    Dupont.average=0.4*Dupont.midterm_grade+0.6*Dupont.final_grade;
16    Ami=Dupont; Ami.name="RAYAN";
    modify_grades(9,14,&Ami);
18    printf("Name: %s Average: %1.2f\n",Dupont.name, Dupont.average);
    printf("Name: %s Average: %1.2f\n",Ami.name, Ami.average);
20    return 0;
}
```

# Structures

- Structures can be nested (one contained in another)
- Structures can contain arrays
- An array whereby elements are of a structure type is considered as standard-type arrays

```

1  struct university
   {
3      struct room {int number; int capacity;} sl[4];
      char majors[2][10];
5  };
   void main()
7  {
      struct university utt;
9      utt.sl[0].number = 0;
      utt.sl[0].capacity = 25;
11     utt.sl[1].number = 1;
      utt.sl[1].capacity = 25;
13     utt.sl[2].number = 2;
      utt.sl[2].capacity = 12;
15     utt.sl[3].number = 3;
      utt.sl[3].capacity = 100;
17     strcpy(utt.majors[0], "Tronc C");
      strcpy(utt.majors[1], "Branche");
19     printf("UTT has two majors: %s and %s\n", utt.majors[0], utt.majors[1]);
      printf("UTT has four rooms: \n");
21     for(int i = 0; i < 4; ++i)
        {
23         printf("\t Room number %d with capacity %d\n",
                utt.sl[i].number, utt.sl[i].capacity);
25     }
   }

```

# Structures

## Initialization and Assignment

- A structure can be initialized
- One or multiple members can be initialized separately
- Two structure variables can be assigned to each other using ordinary assignment

```

struct university
2 {
    struct room {int number; int capacity;} sl[4];
4     char majors[2][10];
    };
6 void main()
    {
8         struct university utt = {{0,25},{1,25},{2,12},{3,100}},
                                   {"Tronc C", "Branche"}};
10        printf("UTT has two majors: %s and %s\n", utt.majors[0], utt.majors[1]);
12        printf("UTT has four rooms: \n");
        for(int i = 0; i < 4; ++i)
14        {
            printf("\t Room number %d with capacity  %d\n", utt.sl[i].number, utt.sl
                [i].capacity);
        }
16 }

```

# Structures

## Initialization and Assignment

- A structure can be initialized
- One or multiple members can be initialized separately
- Two structure variables can be assigned to each other using ordinary assignment

```

1 struct university
2 {
3     struct room {int number; int capacity;} sl[4];
4     char majors[2][10];
5 };
6 void main()
7 {
8     struct university u = {.sl = {{0,25},{1,25},{2,12},{3,100}},
9                             .majors = {"Tronc C", "Branche"}};
10
11    struct university utt;
12    utt = u; // assignment
13    printf("UTT has two majors: %s and %s\n", utt.majors[0], utt.majors[1]);
14    printf("UTT has four rooms: \n");
15    for(int i = 0; i < 4; ++i)
16    {
17        printf("\t Room number %d with capacity  %d\n", utt.sl[i].number, utt.sl
18            [i].capacity);
19    }
20 }

```



# Personalized Types

# Personalized Types

- C also allows to create personalized types

```
typedef <type> newType;
```

- Example: `typedef int newint;` In this case, `newint` is used **exactly** as `int` is used

```
#include <stdio.h>
2 typedef int newint;
  int main()
4 {
    newint test;
    printf("\n Input an integer:");
    scanf("%d",&test);
    printf("Your input=%d\n",test);
    return 0;
10 }
```

# Structures and Personalized Types

- **typedef** can be used when defining a new structure to define a new type

```
typedef struct { <type1> var1; <type2> var2; <typeN> varN;
2 } structure_name; // structure_name represents the new type
```

- Therefore, a variable can be defined in the following manner: structure\_name Variable;

```
#include <stdio.h>
2 struct student {
    char *name;
4     float midterm_grade;
    float final_grade;
6     float average;
};
8 int main()
{
10     struct student Dupont, Ami;
    Dupont.name="DUPONT";
12     Dupont.midterm_grade=17;
    Dupont.final_grade=8.5;
14     Dupont.average=0.4*Dupont.midterm_grade+0.6*Dupont.final_grade;
    Ami=Dupont;
16     Ami.name="RAYAN";
    printf("Name : %s Average : %1.2f\n",Dupont.name, Dupont.average);
18     printf("Name : %s Aveage : %1.2f\n",Ami.name, Ami.average);
    return 0;
20 }
```

# Structures and Personalized Types

## Summary

Two definitions are possible:

```
// Definition 1
2 struct structure_name
{
4     <type1> var1;
      <type2> var2;
6     <typeN> varN;
      };
8
// Variable definition:
10 struct structure_name Variable;
```

```
// Definition 2
2 typedef struct
{
4     <type1> var1;
      <type2> var2;
6     <typeN> varN;
      } structure_name;
8
// Variable definition:
10 structure_name Variable;
```

If we add to Definition 1:

```
typedef struct structure_name structure_name ;
```

Then, we can again use the following variable definition: `structure_name Variable;`

# Structures and Personalized Types

Exercise: supermarkets

## Exercise: supermarket management

a supermarket is organized in shelves, each containing several products.

- ① Define a type *produit* that corresponds to a product. The type must contain the name of the product and the number of its items
- ② Define a type *shelf*  
Define type *supermarket* that contain the different shelves
- ③ Define functions/procedures to perform operations on products:
  - Add a product
  - Checks the number of items of a product
  - Delete a product
  - Supply a product
- ④ Define functions/procedures to perform operations on shelves:
  - Display the content of a shelf
  - Add a shelf
  - Delete a shelf

# Enumeration Types

# Enumeration Types

- Enumerations allow to define a type using a list of names assigned to integer values
- To define a new enumeration type, the following syntax is used:

```
1 enum enum_type_name {expression1, expression2, expressionN};
```

```
1 #include <stdio.h>

3 enum colors{blue, white, red, green, yellow};

5 int main()
{
7     enum colors C;
    C=green;
9     printf("\nThe integer representing the color = %d\n", C);
    return 0;
11 }
```

# Enumeration Types

Every symbolic name corresponds to an integer value .

- By default, this value is determined by the indexation that starts from 0.

```
1 enum BOOLEAN
  {
3     false,      // false = 0
      true        // true = 1
5 };

7 enum BOOLEAN myEnum;
```

- It is also possible to define the integer value of a symbolic name

```
1 #include <stdio.h>

3 enum colors{blue=100, white=20, red, green=1000, yellow};

5 int main()
  {
7     colors C;
      C=green;
9     printf("\nThe integer representing the color = %d\n", C);
      return 0;
11 }
```



# Enumeration Types

## Examples

```
1 enum { yes, no } response; // only the variable is defined
```

```
1 enum Day
  {
3     Saturday,
    Sunday = 0,
5     Monday,
    Tuesday,
7     Wednesday,      // here, Wednesday is given the integer value 3
    Thursday,
9     Friday
  } workdays;
11
enum Day tomorrow = Thursday;
```

```
typedef enum thecolors {red, green, white, blue} colors;
2
struct carte {
4     colors c;
    short int value;
6 } main[13];
```

# Unions

# Unions

C also allows to create specific types that group multiple members **that occupy the same memory space**: unions.

A variable of a `union` type stores a single value corresponding the last member that was assigned. The values of the other members are overwritten.

```
union union_name {expression1, expression2, expressionN};
```

```
1 #include <stdio.h>
   typedef union group
3 {
       short a;
5       char car;
       char cac;
7 } group;

9 int main()
   {
11     group gr;
       gr.car='z';gr.a=128;
13     printf("a=%d, car=%d and cac=%d\n",gr.a,gr.car,gr.cac);
       printf("address of a %d\n", &(gr.a));
15     printf("address of car %d\n", &(gr.car));
       return 0;
17 }
```

## 7. Pointers and Dynamic Memory Allocation

- Introduction
- C Language: The Basics
- Types, Instructions and Operators
- Control Structures
- Functions and procedures
- Arrays
- Structures
- **Pointers and Dynamic Memory Allocation**
  - Pointers
  - Dynamic Memory Allocation
  - Pointers and Arrays
  - Examples of pointers
- Files

# Pointers

# Pointers

- A pointer can be considered as a link
- It contains the address of another variable
- Generally, a pointer is declared in the following manner:

```
<type> *Variable;
```

- Variable is a pointer that contains the address of \*Variable
- Example: `int *x`; Here, x is a pointer on an integer variable
- Where pointers are used:
  - Passing by address (cf. functions lecture)
  - Dynamic data structures (arrays, lists, ...)

# Pointers

- **Example:** `int *x; int y=7; x=&y; *x=9; //x contains the address of y`

```

#include <stdio.h>
2 int main()
{
4     int *x;
    int y=7;
6     x=&y;
    printf("\nValue of y = %d and its address = %d",y,&y);
8     printf("\nValue of x = %d and its address = %d",x,&x);
    printf("\nValue of *x = %d",*x);
10    *x=9;
    printf("\nValue of x = %d and its address = %d. Value of y = %d\n",x,&x,y);
12    x++;
    printf("\nNew value of x = %d and its address = %d",x,&x);
14    printf("\nNew value of *x = %d",*x);
    return 0;
16 }

```

- A pointer can be incremented to a new address. In this case, we do not know on which variable it points neither do we know what this variable represents.

```
int *x; int y=7; x=&y; x++;
```

# Pointers

- NULL: the NULL pointer represents address=0. In this case, the pointer does not point to any variable. Usually, it is used to initialize a pointer.
- A pointer generally occupies the same memory space.
- On a 64-bit architecture, a pointer occupies 8 bytes.

```
#include <stdio.h>
2 int main()
{
4     double *a, b=7.5;a=&b;
    int *x, i=5;x=&i;
6     char *y, c='c';y=&c;
    printf("\nSize of variable b = %d;
8     size of pointer a= %d",sizeof(b),sizeof(a));
    printf("\nSize of variable i = %d;
10    size of pointer x= %d",sizeof(i),sizeof(x));
    printf("\nSize of variable c = %d;
12    size of pointer y= %d",sizeof(c),sizeof(y));
    return 0;
14 }
```



# Dynamic Memory Allocation

# Dynamic Memory Allocation

- What is the dynamic memory allocation?  
Dynamically allocate a memory space to accomplish a certain processing.
- Why:
  - Allocate the necessary memory space to the program's needs without excess
  - Define a variable-size array
  - Create dynamic data structures
- When to be used?  
When the necessary memory space is not known before execution.

# Dynamic Memory Allocation

In previous examples, the memory is automatically allocated after a declaration (example: `int i`; 4 bytes are allocated to variable `i`).

The dynamic memory allocation allows the programmer to allocate memory spaces depending on the need.

```
#include <stdio.h>
2 #include <stdlib.h>
int main()
4 {
    int *tab;
    int i;
    tab=(int*)malloc(5*sizeof(int));
    8 for(i=0;i<5;i++)
        {
            10 printf("Input tab[%d]:\n",i);
                scanf("%d",&tab[i]);
        }
    12 return 0;
    14 }
```

# Dynamic Memory Allocation

- Allows to perfectly control the memory allocation and thus optimize the memory used by the program
- Very useful when the program manipulates large-scale data
- The dynamic memory allocation is used by functions of library `<stdlib.h>`

## Four steps are necessary:

- Declaration of the pointer
- Dynamic allocation of memory using functions `malloc`, `calloc`, `realloc`
- Use of the allocated space
- Free the allocated memory using function `free`

# Dynamic Memory Allocation

## Syntax

```
(pointer type *) malloc (number of bytes);
```

- number of bytes is usually expressed using function `sizeof(variable_type)`. This function returns the number of bytes required for a variable of type `type_variable`.
- The memory is freed using function `free`. This function is useful to optimize memory allocation during the program's execution.

```
#include <stdio.h>
2 #include <stdlib.h>
void main()
4 {
    int *tab;
6     int i;
    tab=(int*)malloc(5*sizeof(int));
8     for(i=0;i<5;i++)
    {
10         printf("\n Input tab[%d]= ",i);
        scanf("%d",&tab[i]);
12     }
    free(tab);
14 }
```

# Dynamic Memory Allocation

## Function `malloc`

### The function `malloc`

- `malloc` allows to allocate a memory space whose address is given by the returned pointer. The size of the allocated memory space is given by the argument of the function.
- `malloc` returns the address of the allocated memory if the allocation succeeds, 0 (NULL) otherwise.

# Dynamic Memory Allocation

## Function calloc

- calloc plays the same role as malloc but additionally initializes the allocated space to 0.

```
(pointer type *) calloc (number of elements, size of an element);
```

```
1  #include <stdio.h>
   #include <stdlib.h>
3  void main()
   {
5     int *tab;
     int i;
7     tab=(int*)calloc(5, sizeof(int));
     for(i=0;i<5;i++)
9     {
         printf("\n Input tab[%d]= ",i);
11        scanf("%d",&tab[i]);
     }
13    free(tab);
   }
```

# Dynamic Memory Allocation

## Function realloc

### Syntax

```
(pointeur type*) realloc (pointer, new size in bytes);
```

- new size in bytes is expressed using function `sizeof`.
- `realloc` cannot be used to reallocate a memory space that is not allocated before the function call using `malloc` and `calloc`
- If the new size is smaller, the size of the memory space is reduced.

```
#include <stdio.h>
2 #include <stdlib.h>
void main()
4 {
    int *tab;
    int i;
    tab=(int*)malloc(5*sizeof(int));
    8 for(i=0;i<5;i++)
    {
        10 printf("\n Input tab[%d]= ",i);
        scanf("%d",&tab[i]);
    }
    12 tab = realloc (tab, 10*sizeof(int));
    14 for(i=0;i<10;i++)
    {
        16 printf("\n Input tab[%d]= ",i);
        scanf("%d",&tab[i]);
    }
    18 free(tab);
    20 }
```



# Dynamic Memory Allocation

## Function `free`

### Function `free`

- Function `free` allows to free a dynamically allocated memory space pointed by the argument pointer.
- **Warning:** Never free a memory that was not dynamically allocated.

# Dynamic Memory Allocation

## NULL pointer

### NULL pointer

- NULL represents address 0. It doesn't point to any variable. If you assign NULL to a pointer, it does not free any dynamically allocated space!
- **Warning:** If you do so, you're just giving your pointer another address!

# Dynamic Memory Allocation

```
#include <stdio.h>
2 #include <stdlib.h>
   int main()
4 {
    int *tab;
    int i;
    int *tab2;
    8 tab=(int*)malloc(3*sizeof(int));
    for(i=0;i<3;i++)
10 {
        printf("\n Input tab[%d] =",i);
12        scanf("%d",&tab[i]);
    }
14    for(i=0;i<3;i++) printf("tab[%d]=%d\n",i,tab[i]);
    tab2=tab;
16    for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
    printf("ADDRESS of tab before assignment=%d\n",tab);
18    printf("ADDRESS of tab2 before assignment=%d\n",tab2);
    tab=NULL;
20    tab2[2]=777;
    for(i=0;i<3;i++) printf("tab2[%d]=%d\n",i,tab2[i]);
22    printf("ADDRESS of tab after assignment=%d\n",tab);
    printf("ADDRESS of tab2 after assignment=%d\n",tab2);
24    return 0;
}
```

# Pointers and Arrays

# Pointers and Arrays

## Multi-dimensional arrays

- In a static array, the size of each dimension is defined
- Example: `int tab[M][N];` //an array of M rows and N columns
- Dynamic array  $\Rightarrow$  pointer !!

```
type **pointeur_name; // a 2D array pour un tableau à deux
dimensions
2 type ***pointeur_name; // a 3D array
```

```
#include <stdio.h>
2 #include <stdlib.h>
int main()
4 {
    int k=2, n=2,m=3,i;
    int **tab;
    tab = (int**)malloc(k * sizeof(int*));
    8     tab[0] = (int*)malloc(n*sizeof(int));
        tab[1] = (int*)malloc(m*sizeof(int));
    10     tab[0][0]=1;tab[0][1]=2;
        tab[1][0]=3;tab[1][1]=4;tab[1][2]=5;
    12     printf("\n%d %d",tab[0][0],tab[0][1]);
        printf("\n%d %d %d\n",tab[1][0],tab[1][1],tab[1][2]);
    14     for (i = 0; i < k; i++)
        free(tab[i]);
    16     free(tab);
        return 0;
    18 }
```

# Examples of pointers

# Examples of pointers

## Pointers and strings

```
#include <stdio.h>
2 #include <string.h>
int main()
4 {
    char *str;
6     str="This is a string";
    printf("%s\n",str);
8     printf("\nNumber of characters = %d\n",strlen(str));
    return 0;
10 }
```

# Examples of pointers

```
#include <stdio.h>
2 #include <stdlib.h>
int main()
4 {
    int *tab;
    int i;
    int *tab2;
    8 tab=(int*)malloc(3*sizeof(int));
    for(i=0;i<3;i++) {
10         printf("\n Input tab[%d] = ",i);
            scanf("%d",&tab[i]);
12     }
    for(i=0;i<3;i++)
14         printf("tab[%d]=%d\n",i,tab[i]);
    tab2=tab;
16     for(i=0;i<3;i++)
        printf("tab2 [%d]=%d\n",i,tab2[i]);
18     printf("ADDRESS of tab before free=%d\n",tab);
    printf("ADDRESS of tab2 before free=%d\n",tab2);
20     free(tab);
    for(i=0;i<3;i++)
22         printf("tab[%d]=%d\n",i,tab[i]);
    for(i=0;i<3;i++)
24         printf("tab2 [%d]=%d\n",i,tab2[i]);
    printf("ADDRESS of tab after free=%d\n",tab);
26     printf("ADDRESS of tab after free=%d\n",tab2);
    return 0;
28 }
```



# Examples of pointers

## Pointers and structures

```
#include <stdio.h>
2 typedef struct
  { char *name;
4     float midterm;
      float final;
6     float average;
  } student;
8
void modify_grades(float a,float b, student *s)
10 {
    s->midterm=a;
12    s->final=b;
    s->average=0.4*a+0.6*b;
14 }

16 int main()
  {
18     student Dupont={"DUPONT", 17, 8.5, 0};
    student Ami;
20    Dupont.average=0.4*Dupont.midterm+0.6*Dupont.final;
    Ami=Dupont;
22    Ami.name="RAYAN";
    modify_grades(9,14,&Ami);
24    printf("Name : %s Average : %1.2f\n",Dupont.name, Dupont.average);
    printf("Name : %s Average : %1.2f\n",Ami.name, Ami.average);
26    return 0;
  }
```



## 8. Files

- Introduction
- C Language: The Basics
- Types, Instructions and Operators
- Control Structures
- Functions and procedures
- Arrays
- Structures
- Pointers and Dynamic Memory Allocation
- **Files**
  - Generalities on files
  - Opening a File: `fopen`
  - Some Useful Functions

# Generalities on files

# Generalities on files

## Introduction

- Used to store data in different extensions
- Files can be stored on different storage devices (hard drive, usb key, DVD, ...)
- Every file is identified by its name and its path
- Input/Output in C programs are managed using files

# Generalities on files

## Rules

- The file name is made up of two parts: the file name and the extension. They are separated by a period (.).  
example of a text file: *titi.txt*  
example of a binary file: *toto.bin*
  - Files must be opened before any use
  - The content of the files can only be read sequentially
  - Files must be closed after operations are done
- 
- The keyboard of the computer are the default input device (`stdin`)
  - The computer screen is the default output device (`stdout`)  
The screen is also the default error output (`stderr`)

# fopen

## Creating a File

- Declaration of a pointer on a structure FILE:

```
FILE *ptr;
```

- FILE is a structure defined in the library `stdio.h` and it contains the characteristics of a file
- File name: `char name[20] = {'t', 'o', 't', 'o', '.', 't', 'x', 't'};`
- Syntax of file opening:

```
ptr=fopen(name, "w");
```

- The function `fopen` allocates a memory space for the file. The address of this space is stored in the pointer (`ptr`). If the memory space is not allocated, then the NULL pointer is returned.

# fopen

## Creating a File

- The result of the function `fopen` is assigned to pointer `ptr`. This result represents the address of the memory associated with the file
- The second argument of the function allows to specify the opening mode: read, write, append
- From these three categories of modes, other modes are derived and can be used.
- There exists the *TEXT* and the *BINARY* modes



# fopen

## Opening Modes

### Opening Modes (text) of a file:

- **r**: read mode. The file must exist before opening.
- **w**: write mode. If the file exists, it will be erased and created again.
- **a**: append mode. If the file does not exist, it will be created.
- **r+**: read/write mode. The file must exist before opening.
- **w+**: read/write mode. If the file exists, it will be erased and created again.
- **a+** : append/read mode. If the file does not exist, it will be created.

In the append mode, appending will be done at the end of the file.

### Binary Opening Modes:

- **rb**, **wb**, **ab**, **r+b**, **w+b**, **a+b**: they represent the same as above but in binary.

# fclose

## Closing a File

- After the different operations on a file, it must be closed using function `fclose`
- This function will free the memory space that was allocated to the file.

```
#include <stdio.h>
2 void main()
{
4     FILE *ptr;
    char name[20]={ 't', 'o', 't', 'o', '.', 't', 'x', 't' };
6     char n[20];
    printf("Input a filename:");
8     scanf("%s",n);
    ptr=fopen(name,"w");
10    ptr=fopen(n,"w");
    fclose(ptr);
12 }
```

# Some Useful Functions

# Some Useful Functions

Writing in a file: `fprintf`

Function `fprintf` writes data (text) into the file. It works in the same manner as function `printf`.

```
#include <stdio.h>
2 int main()
{
4     FILE *ptr;
    char name[20];
6     printf("Input a filename:");
    scanf("%s",name);
8     ptr=fopen(name,"w");
    fprintf(ptr, "%s", "I write in the file ptr!\n");
10    fclose(ptr);
    return 0;
12 }
```

# Some Useful Functions

Reading from a file: `fscanf`

Function `fscanf` reads data from the file using their format. It works in the same manner as function `scanf`.

```
char str; fscanf(ptr, "%c", &str);
```

```
1 #include <stdio.h>
2 int main()
3 {
4     FILE *ptr;
5     char name[20]={ 't', 'o', 't', 'o', '.', 't', 'x', 't' };
6     char str[20];
7
8     ptr=fopen(name,"r");
9     int i=0,z=0;
10    do
11    {
12        fscanf(ptr, "%c", &str[i]);
13        printf("%c",str[i]);
14        if(str[i]=='\n')
15            z=1;
16        i++;
17    } while (z==0);
18    fclose(ptr);
19    return 0;
20 }
```

# Some Useful Functions

End of a file: `feof`

Function `feof` detects whether the end of the file has been reached. It returns 1 if the end of the file has been reached, 0 otherwise.

```
int test; test=feof(ptr);
```

```
1 #include <stdio.h>
   int main()
3 {
   FILE *ptr;
5   char name[20]={ 't', 'o', 't', 'o', '.', 't', 'x', 't' };
   char car;
7
   ptr=fopen(name, "r");
9   while (feof(ptr)==0)
   {
11       fscanf(ptr, "%c", &car);
       printf("%c", car);
13   };
   fclose(ptr);
15   return 0;
}
```

# Some Useful Functions

Reading a character from a file: `fgetc`

Function `fgetc` reads a character from a file whose pointer is the argument.

```
fgetc(ptr);
```

```
1 #include <stdio.h>
   #include <stdlib.h>
3 void main()
   {
5     int i, j;
     FILE *ptr;
7     char car, name[20]={ 't', 'o', 't', 'o', '.', 't', 'x', 't' };

9     ptr=fopen(name, "r");
     while(feof(ptr)==0)
11    {
        car=fgetc(ptr);
13        printf("%c", car);
        };
15    fclose(ptr);

17 }
```

# Some Useful Functions

Writing a character into a file: `fputc`

Function `fputc` writes a character (first argument) into a file (second argument).

```
char ca='\n'; fputc(ca,ptr);
```

```
1 #include <stdio.h>
   #include <stdlib.h>
3 void main()
   {
5     int j=32;
       FILE *ptr;
7     char car, name[20]={'t','o','t','o','.','t','x','t'};

9     ptr=fopen(name,"w");
       while(j<128)
11    {
           fputc((char)j,ptr);
13        fputc('\n',ptr);
           printf("%c\n",(char)j);
15        j++;
       }
17    fclose(ptr);

19 }
```



# Some Useful Functions

## Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define INPUT "input.txt"
5  #define OUTPUT "output.txt"
6
7  int main(void)
8  {
9      FILE *f_in, *f_out;
10     int c;
11
12     if ((f_in = fopen(INPUT,"r")) == NULL)
13     {
14         fprintf(stderr, "\n Error: unable to open file %s\n",INPUT);
15         return(EXIT_FAILURE);
16     }
17     if ((f_out = fopen(OUTPUT,"w")) == NULL)
18     {
19         fprintf(stderr, "\n Error: unable to open file %s\n", \OUTPUT);
20         return(EXIT_FAILURE);
21     }
22     while ((c = fgetc(f_in)) != EOF)
23         fputc(c, f_out);
24     fclose(f_in);
25     fclose(f_out);
26     return(EXIT_SUCCESS);
27 }
```

# Some Useful Functions

## fwrite

Function `fwrite` writes data into a file (first argument) from a memory space (fourth argument). The second and the third argument provide the size of the memory space.

```
fwrite(tab, sizeof(int), n_enreg, ptr);
```

This function is mainly used for the binary mode but can be used in the text mode.

```
1 #include <stdio.h>
   #include <stdlib.h>
3 int main()
   {
5     FILE *ptr;
     char name[20]={'t','o','t','o','.','t','x','t'};
7     char *tab;
     int i;
9     tab=(char*)malloc(10*sizeof(char));
     for(i=0;i<10;i++)
11         tab[i]=(char)48+i;
     ptr=fopen(name,"wb");
13     fwrite(tab, sizeof(char), 10, ptr);
     fclose(ptr);
15     return 0;
   }
```

# Some Useful Functions

## fread

Function `fread` reads data from a file (first argument) and saves them into a memory space (fourth argument). The second and the third argument provide the size of the memory space.

```
fread(tab, sizeof(int), n_enreg, ptr);
```

This function is mainly used for the binary mode but can be used in the text mode.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     FILE *ptr;
6     char name[20]={'t','o','t','o','.','t','x','t'};
7     char *tab;
8     int i;
9     tab=(char*)malloc(10*sizeof(char));
10    ptr=fopen(name,"r");
11    fread(tab,1,10,ptr);
12    for(i=0;i<10;i++)
13        printf("%c\n",tab[i]);
14    fclose(ptr);
15    return 0;
16 }
```

# Some Useful Functions

Moving the position indicator: `fseek`

Function `fseek` shifts the position indicator of a file from a certain position.

```
fseek(ptr, n_bytes, init_pos);
```

`init_pos` can be `SEEK_SET` (beginning of the file), `SEEK_CUR` (current position) or `SEEK_END` (end of the file).

This function is mainly used for the binary mode but can be used in the text mode.

```
#include <stdio.h>
2 #include <stdlib.h>
void main()
4 {
    int i,j;
    FILE *ptr;
    char name[20]={ 't','o','t','o','.','t','x','t' };
    ptr=fopen(name,"r+");
    for(j=0;j<10;j++)
10 {
        fseek(ptr, sizeof(int), SEEK_CUR);
12        fscanf(ptr, "%d", &i);
        printf("\n%d", i);
14    }
    fclose(ptr);
16 }
```

# Some Useful Functions

Current Position: ftell

Function `ftell` gives the current position of the position indicator. The given value is in bytes with regards to the file.

```
int position; position=ftell(ptr);
```

This function is mainly used for the binary mode but can be used in the text mode.

```
1 #include <stdio.h>
   unsigned int fsize(FILE *pointer) {
3     int size;
       fseek(pointer,0,SEEK_END);
5     size=ftell(pointer);
       fseek(pointer,0,SEEK_SET);
7     size-=ftell(pointer);
       return size;
9 }
   void main()
11 {
       int i;
13     FILE *ptr;
       char name[20]={ 't', 'o', 't', 'o', '.', 't', 'x', 't' };
15     ptr=fopen(name,"r");
       i=fsize(ptr);
17     printf("The size of the file = %d bytes\n",i);
       fclose(ptr);
19 }
```

