Computing Systems
An Introduction to C programming

Andrea Masciadri, PhD
andrea.masciadri@polimi.it

Politecnico di Milano – Dipartimento di Elettronica, Informazione e Bioingegneria

Agenda

- Introduction
- Syntax
- Data structures
- Exercise - Hello World
- Functions
- Exercises – part 1

- APPENDIX
    - Advanced topics
    - Exercises – part 2

INTRODUCTION

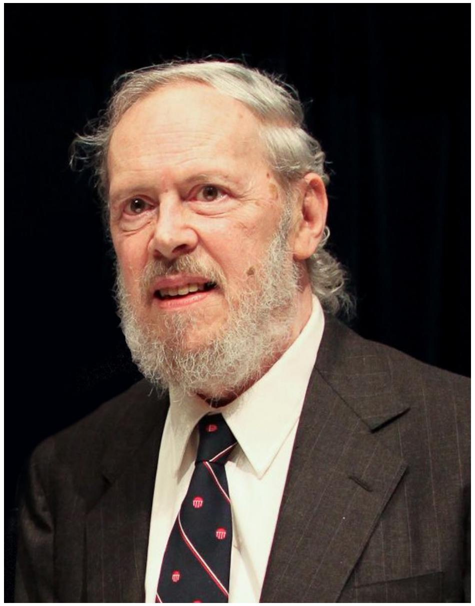
History

Developed in the 1970s , by Dennis Ritchie at Bell Telephone Laboratories, Inc. in conjunction with development of UNIX operating system.

C is a

- **high level**,
- **general–purpose,**
- **structured**

programming language



History

UNIX originally written in low-level assembly language but there were problems:

- code hard to maintain: no structured programming (e.g. encapsulating routines as “functions”, “methods”, etc.);
- not portable: code worked only for specific hardware.

```
8048374:    55                      push   %ebp
8048375:    89 e5                   mov    %esp,%ebp
8048377:    83 ec 08                sub    $0x8,%esp
804837a:    83 e4 f0                and    $0xfffffff0,%esp
804837d:    b8 00 00 00 00          mov    $0x0,%eax
8048382:    29 c4                   sub    %eax,%esp
8048384:    c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
804838b:    83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
804838f:    7e 02                   jle    8048393 <main+0x1f>
```

History

In 1978, Brian Kernighan and Dennis Ritchie published the first edition of The C Programming Language: this version of C is commonly referred to as "K&R C" .

In 1989, the C standard was ratified as ANSI X3.159-1989 "Programming Language C". This version of the language is often referred to as ANSI C, Standard C, or sometimes “C89". In 1990, the ANSI C was adopted by the **International Organization for Standardization (ISO)** as ISO/IEC 9899:1990, which is sometimes called C90. Therefore, the terms "C89" and "C90" refer to the same programming language.

The C standard was further revised in the late 1990s, leading to the publication of ISO/IEC 9899:1999 in 1999, which is commonly referred to as "C99".

History

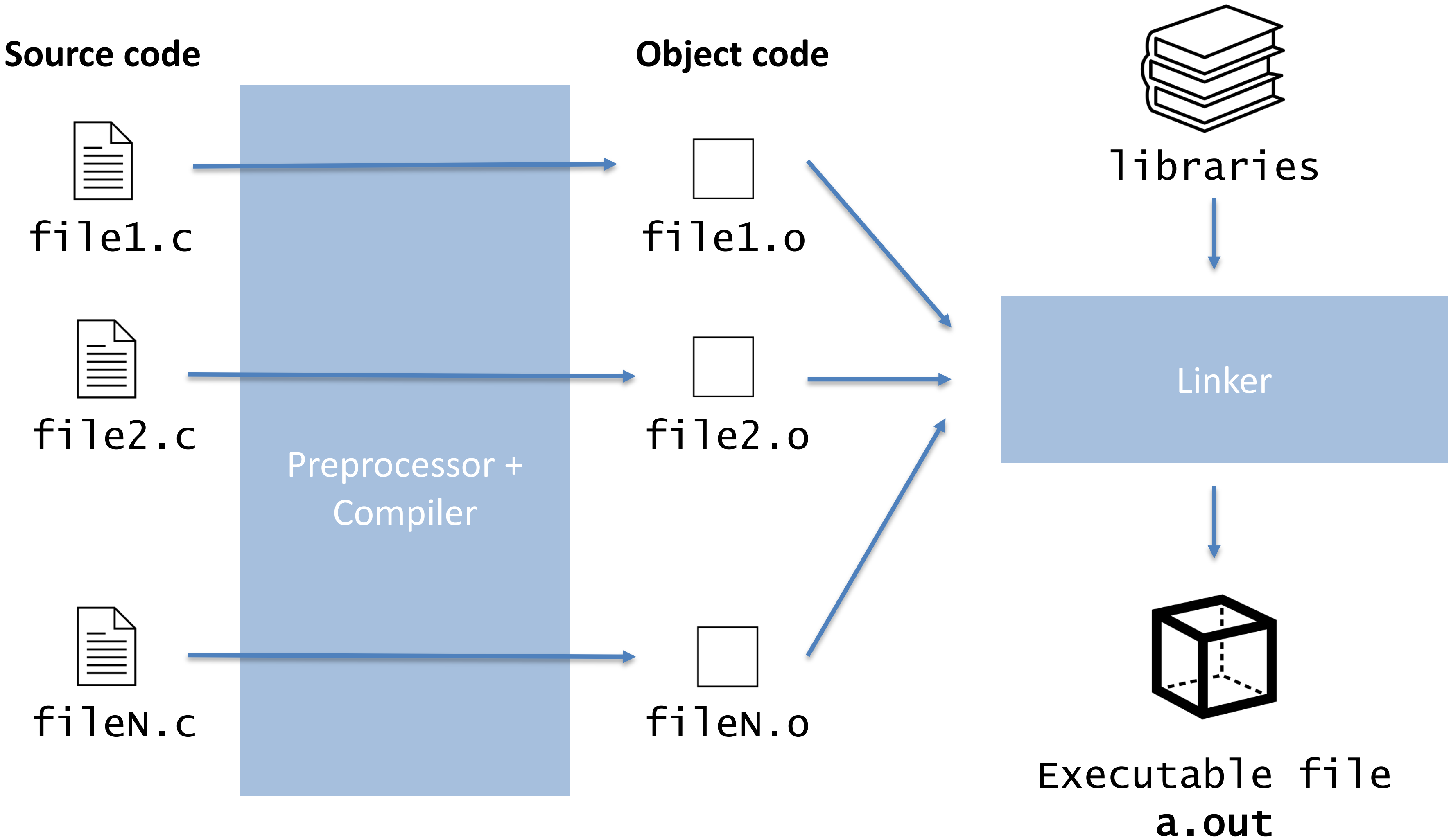
In 2011 was officialy publicated the C11 revision of the C standard.This version adds numerous new features to C.

Published in June 2018, C18 is the **current standard for the C programming** language. It introduces no new language features, only technical corrections, and clarifications to defects in C11.

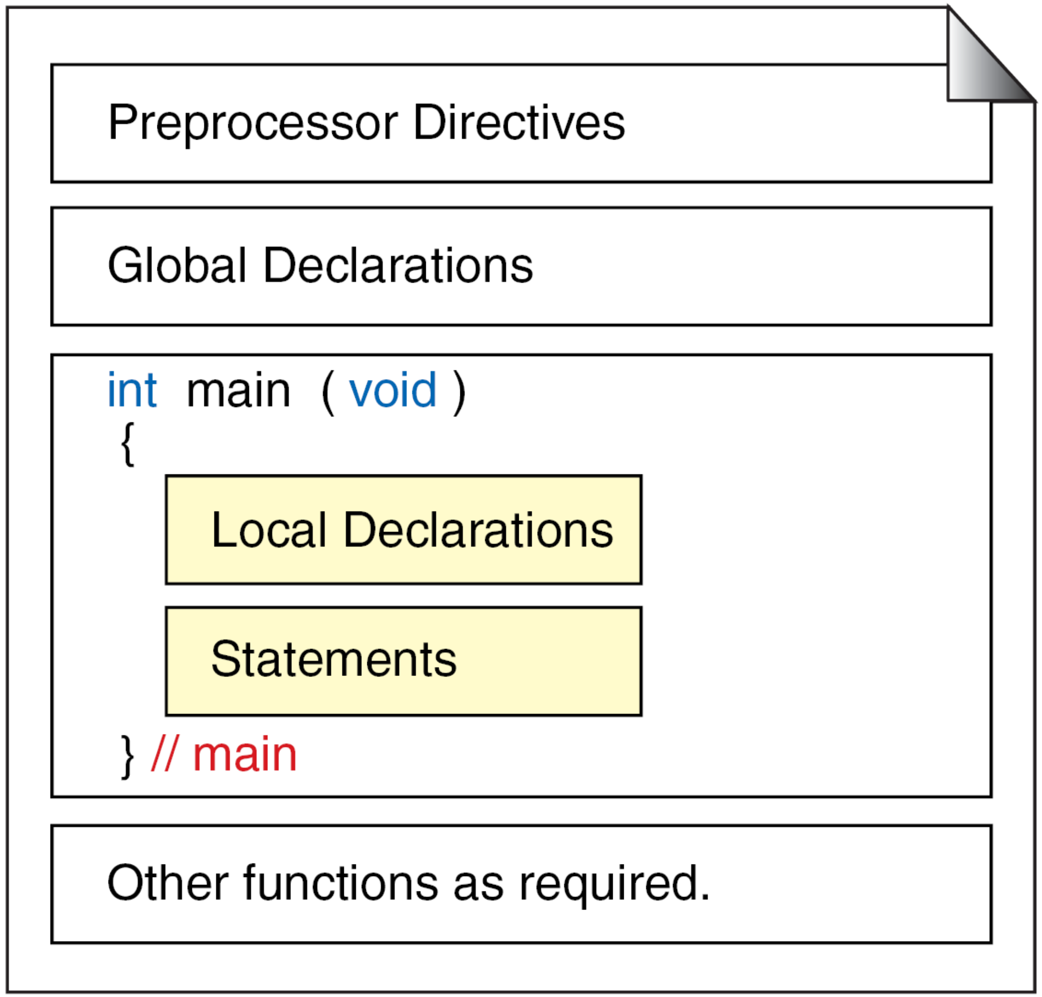
Embedded C

In 2008, the C Standards Committee published a technical report extending the C language to address these issues by providing a common standard for all implementations to adhere to. It includes a number of features not available in normal C, such as fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

Programming in C



Source code: <file.c>



C SYNTAX

C Syntax

The following tokens are the building blocks to write C programs:

1) Identifiers
2) Keywords
3) Operators
4) Punctuation Symbols

Identifiers

An identifier is used to name a variable, function, or label.

- First character must be alphabetic character or underscore
- C is case-sensitive
- Must be different from a keyword

name ≠ NAME

myFunction	OK
Variable2	OK
2Variable	KO
_name	OK

int numWheels = 4

Associated value

Integer Data type

Identifier

Keywords

Keywords are system defined identifiers. Examples:

break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Example: const keyword

Using the «const» keyword makes the value associated with an identifier constant: it cannot be altered in a program. For example:

The following identifier is a constant

```
const double PI = 3.14
```

Associated value

“Double” Data type

Identifier

Arithmetic operators

+	addition	a + b
-	subtraction	a - b
*	multiplication	a * b
/	division	a / b
%	modulo	a % b

Relational operators

==	Equal to	5 == 3	returns 0
>	Greater than (GT)	5 > 3	returns 1
<	Less than (LT)	5 < 3	returns 0
!=	Not equal to	5 != 3	returns 1
>=	GT or equal to	5 >= 3	returns 1
<=	LT or equal to	5 <= 3	returns 0

Assignment operators

```
x = y    assign y to x
x++      post-increment x
++x      pre-increment x
x--      post-decrement x
--x      pre-decrement x
```

```
x += y  assign (x+y) to x
x -= y  assign (x-y) to x
x *= y  assign (x*y) to x
x /= y  assign (x/y) to x
x %= y  assign (x%y) to x
```

Note the difference between ++x and x++:

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don’t confuse = and ==! The compiler will warn “suggest parens”.

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

Logical and Bitwise operators

Logical

!	**NOT**	**!(espr.)**
||	**OR**	**(espr.) || (espr.)**
&&	**AND**	**(espr.) && (espr.)**

Bitwise

~	**bitwise NOT**
|	**bitwise OR**
&	**bitwise AND**
^	**bitwise XOR**
<<	**shift left**
>>	**shift right**

Punctuation symbols

“;” terminates every **instruction**

“{“ and “}” to create **blocks of code**

A program is

a sequence of instructions and…

Structured program theorem (Böhm–Jacopini theorem)

It is possible to compute any computable function combining subprograms using only 3 control structures:

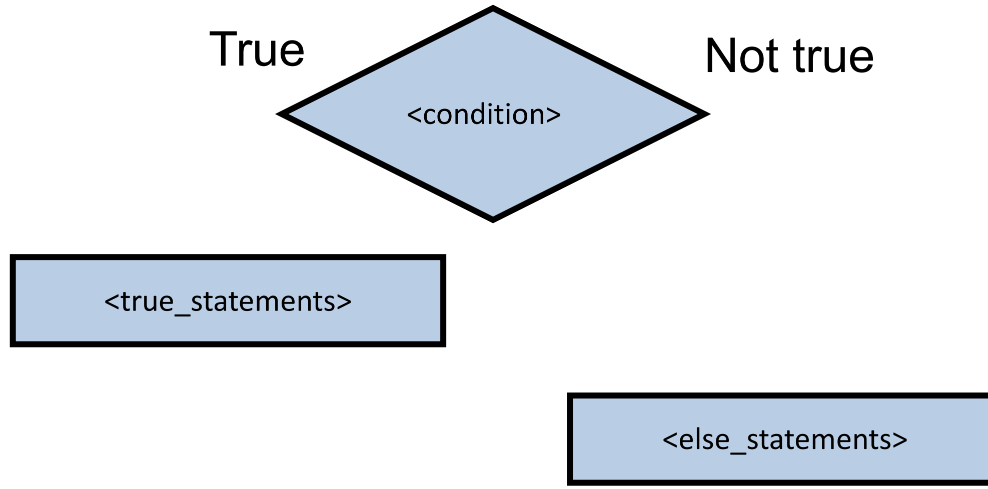
Sequence

Selection -> If, Ternary operator, Switch

Iteration -> While, Do-while, For Loops

Selection: If

```
If (condition) {
        /* SEQUENCE 1 */
} else {
        /* SEQUENCE 2 */
}
```



Example:

```
if (a < b) {
        c = a;
} else {
        c = b;
}
```

Selection: Ternary operator

```
condition ? value_if_true : value_if_false
```

Example:

```
c = (a < b) ? a : b;
```

Selection: Switch

```
switch (variable) {
        case v1:
                /* SEQUENCE 1 */
                break;
        case v2:
                /* SEQUENCE 2 */
                break;
        case v3:
                /* SEQUENCE 3 */
                break;
        default:
                /* SEQUENCE 4 */
                break;
}
```

Example:

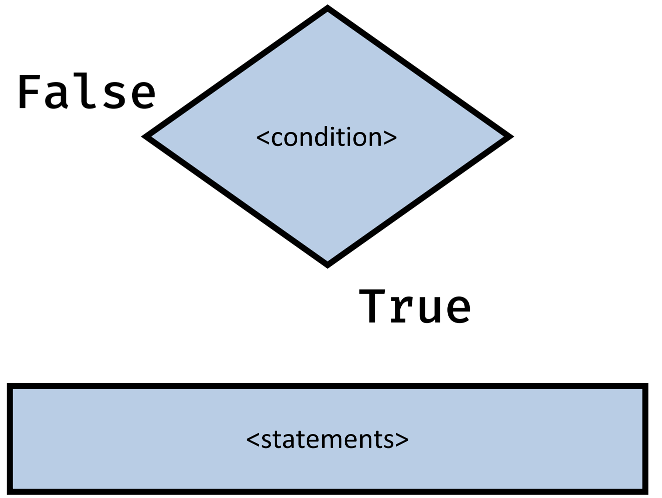
```
switch (a) {
        case 0:
                c == 1
                break;
        case 1:
                c == 2
                break;
        default:
                c == 3
                break;
}
```

Equal to:

```
if (a == 0) {
        c = 1
} else {
        if (a == 1) {
                c = 2;
        } else {
                c = 3;
        }
}
```

Iteration: while

```
while (condition) {

        /* SEQUENCE 1 */

}
```

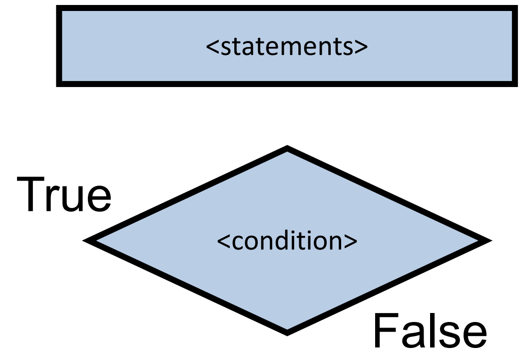


Example:

```
i = 0
while (i < 10) {
        /* do things */
        i = i + 1;
}
```

Iteration: do-while

```
do {

        /* SEQUENCE 1 */

} while (condition)
```

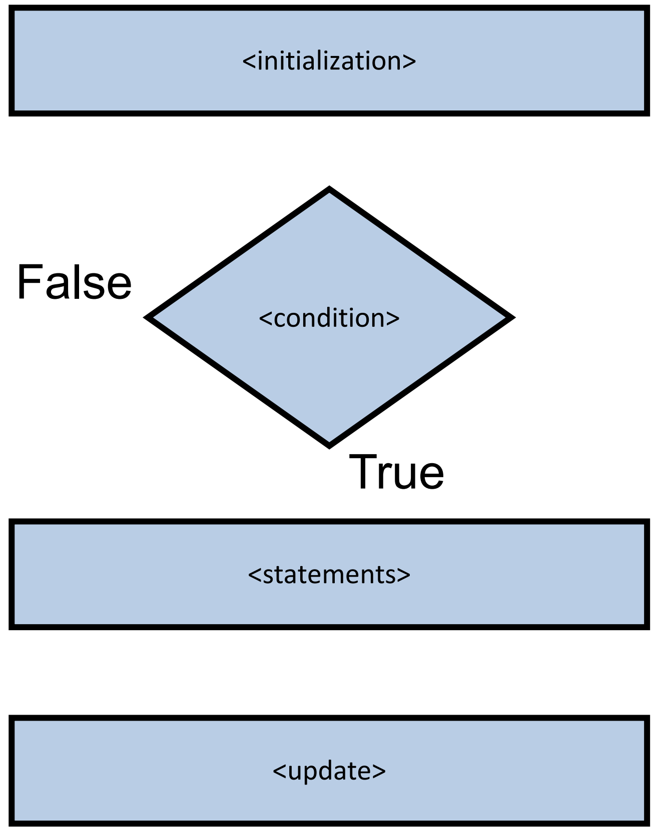


Example:

```
i = 0
do {
        /* do things */
        i = i+ 1;
} while (i < 10)
```

Iteration: for

```
for(initialization; condition; update) {

        /* SEQUENCE 1 */

}
```



Example:

```
for (i=0; i<10; i++) {
        /* do things */
}
```

BREAK: keyword to skip to the end of the for
CONTINUE: keyword to skip to next iteration

Comments

Comments in C are enclosed by slash/star pairs:

/* .. comments .. */   which may cross multiple lines.

NOTE:

C++ introduced a form of comment started by two slashes and extending to the end of the line:

// comment until the line end

The // comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

Preprocessing

- The preprocessor takes your source code and – following certain directives that you give it – tweaks it in various ways before compilation.
- A directive is given as a line of source code starting with the # symbol
- The preprocessor works in a very crude, “word-processor” way, simply cutting and pasting – it doesn’t really know anything about C!

Pre-processor directives

```
#define MAX_COLS  20
#define MAX_INPUT 1000
```

The `#define` directives perform "global replacements":

- every instance of `MAX_COLS` is replaced with `20`, and every instance of `MAX_INPUT` is replaced with `1000`.

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

- The `#include` directives "paste" the contents of the files `stdio.h`, `stdlib.h` and `string.h` into your source code, at the very place where the directives appear.
- These files contain information about some library functions used in the program:
  - `stdio` stands for "standard I/O", `stdlib` stands for "standard library", and `string.h` includes useful string manipulation functions.
- Want to see the files? Look in `/usr/include`

#include

stdio.h	file input and output
ctype.h	character tests
string.h	string operations
math.h	mathematical functions such as sin() and cos()
stdlib.h	utility functions such as malloc() and rand()
assert.h	the assert() debugging macro
stdarg.h	support for functions with variable numbers of arguments
setjmp.h	support for non-local flow control jumps
signal.h	support for exceptional condition signals
time.h	date and time

Pre-processor directives

```
#if <value1>
	/* code to execute if value1 is true */
#elsif <value2>
	/* code to execute if value2 is true */
#else
	/* code to execut otherwise */
#endif
```

#if 1 includes the code until the closing #endif.

#if 0 the code is removed from the copy of the file given to the compiler prior to compilation (but it has no effect on the original source code file).

Pre-processor directives

#pragma once

```
#pragma once // header file code
```

Include guard

```
#ifndef _FILE_NAME_H_
        #define _FILE_NAME_H_
        /* code */
#endif
```

They both prevent the header file from being processed multiple times.

DATA STRUCTURES

Memory

Addr	Value
0	
1	
2	
3	
4	‘H’ (72)
5	‘e’ (101)
6	‘l’ (108)
7	‘l’ (108)
8	‘o’ (111)
9	‘\n’ (10)
10	‘\0’ (0)
11	
12	

Variables

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value

```
char x;
char y='e';
```

Initial value of x is undefined

Initial value

Type is single character (char)

The compiler puts them somewhere in memory.

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

Data type

Category	Type	C Implementation
Void	Void	*void*
Integral	Boolean	*bool*
	Character	*char, wchar_t*
	Integer	*short int, int, long int, long long int*
Floating-Point	Real	*float, double, long double*
	Imaginary	*float imaginary, double imaginary, long double imaginary*
	Complex	*float complex, double complex, long double complex*

Signed integers

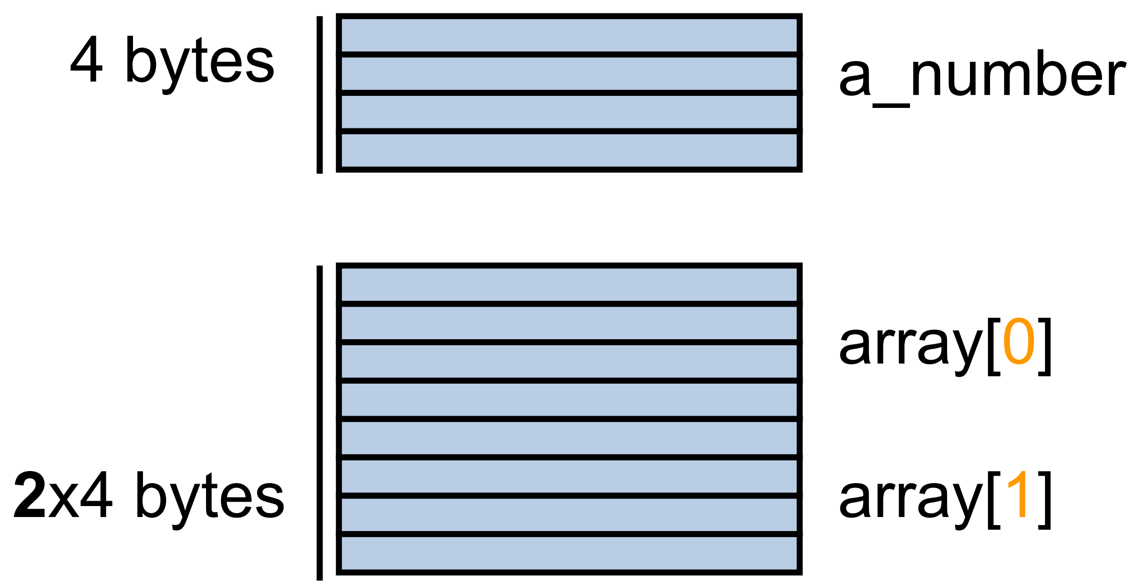
Type	Byte Size	Minimum Value	Maximum Value
short int	2	–32,768	32,767
int	4	–2,147,483,648	2,147,483,647
long int	4	–2,147,483,648	2,147,483,647
long long int	8	–9,223,372,036,854,775,807	9,223,372,036,854,775,806

Array

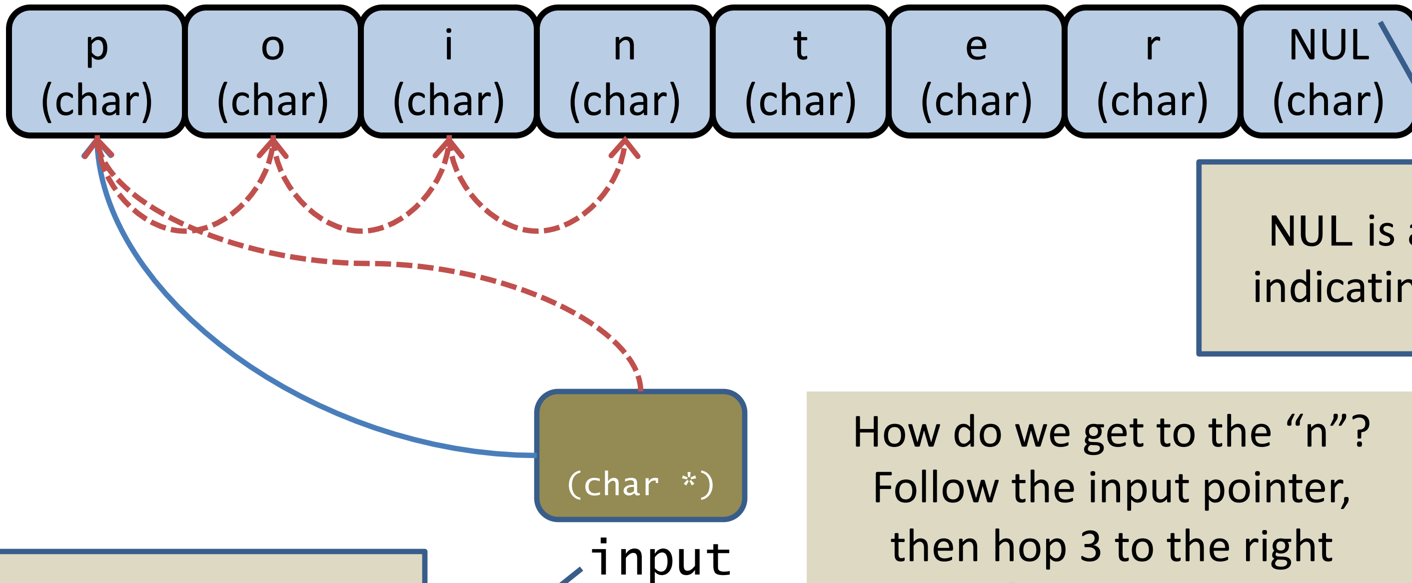
A **collection** of data items of the **same type**

```
int a_number;
int array[2];
```

The number of stored elements



Strings



NUL is a special value indicating end-of-string

How do we get to the “n”?
Follow the input pointer,
then hop 3 to the right
`*(input + 3)`
- or -
`input[3]`

What is `input`?
It’s a string!
It’s a pointer to `char`!
It’s an array of `char`!

`More on pointers later on!`

Constants

Constants are data values that cannot be changed during the execution of a program. Like variables, constants have a type.

A char constant is written with single quotes (') like 'A' or 'z'. The char constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for uppercase 'A'.

Control character

ASCII Character	Symbolic Name
null character	'\0'
alert (bell)	'\a'
backspace	'\b'
horizontal tab	'\t'
newline	'\n'
vertical tab	'\v'
form feed	'\f'
carriage return	'\r'
single quote	'\''
double quote	'\"'
backslash	'\\'

Integer and real constants

Representation	Value	Type
+123	123	int
–378	–378	int
–32271L	–32,271	long int
76542LU	76,542	unsigned long int
12789845LL	12,789,845	long long int

Representation	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
–2.0f	–2.0	float
3.1415926536L	3.1415926536	long double

Cast

```
(<type>) <expression>;
```

```
float side = 3.8;
int val_1 = (int) side;        // = 3
int val_2 = (int) 3.1;         // = 3
long area = val_1 * val_2;     // = 9
float areaf = side * side;     // = 14.44
```

Common mistakes - part 1

```
int a;
int b, c = 0;   /* A and B are not initialized */

const int d = 5;
b = –11;
c = d;      /* OK */
d = c;      /* KO */
```

Common mistakes - part 2

```
float a;
float b, c = 0;
const float d = 5.0;
b = –11;
a = d;                  /* OK */
d = a;                  /* KO */
a = 4 / 5;              /* Output? why? */
a = 4.0 / 5.0;          /* Output? why? */
b = 4 / 5.0;            /* Output? why? */
```

Common mistakes - part 3

```
char a;
char b, c = 'Q’;        /* OK’ */
const char d = 'q’;     /* OK’ */

a = "q";                /* KO: "q" is a string */
a = '\n’;               /* OK escape char*/
b = "ps";               /* KO: string assigned to a char */
c = 'ps’;               /* KO: ps is not a char */
a = 75;                 /* Output? */
```

Struct

A new type, that combines data items of different types.

```
struct <struct_name>
{
      <type1> <name1>;
      <type2> <name2>;
      …
      <typen> <namen>;
};
```

Example:

```
struct book
{
   char name[100];
   float price;
   int num_pages;
};
```

Struct

```
// declare a variable
struct book dark_tower;

// assign values to its fields using '.'
dark_tower.name = "The Dark Tower: The Gunslinger";
dark_tower.price = 5.49;
dark_tower.num_pages = 280;

// all together (initialization)
struct book  dark_tower2 =
{
	"The Dark Tower: The Gunslinger",   // name
	5.49,                               // price
	280                                 // num_pages
};

// assignment of the whole variable
struct book dark_tower3 = dark_tower2; // copy
```

EXERCISE: HELLO WORLD

Requirements

- A text editor (e.g. Notepad)
- gcc compiler

Or

- Any IDE with a C compiler (e.g. code blocks)

Hello World: code

#include inserts another file. “.h” files are called “header” files. They contain stuff needed to interface to libraries and code in other “.c” files.

This is a comment. The compiler ignores this.

The main() function is always where your program starts running.

Blocks of code (“lexical scopes”) are marked by { … }

```c
/* The greeting program. This program demonstrates
   some of the components of a simple C program.
      Written by:   your name here
      Date:         date program written
*/
#include <stdio.h>

int main (void)
{
// Local Declarations

// Statements

   printf("Hello World!\n");

   return 0;
} // main
```

Return ‘0’ from this function

Print out a message. ‘\n’ means “new line”.

Hello World

1. **CODE:** Write the source code using a text editor and save the file as: *hello_world.c*

2. **COMPILE:** Run the compiler to obtain the executable file:

```
1.  $ gcc –Wall –g hello_world.c –o my_program
```

3. **RUN IT!**

```
$ ./my_program
Hello World
```

printf

Character	Description
`%`	Prints a literal `%` character (this type doesn't accept any flags, width, precision, length fields).
`d`, `i`	`int` as a signed integer. `%d` and `%i` are synonymous for output, but are different when used with `scanf` for input (where using `%i` will interpret a number as hexadecimal if it's preceded by `0x`, and octal if it's preceded by `0`.)
`u`	Print decimal `unsigned int`.
`f`, `F`	`double` in normal (fixed-point) notation. `f` and `F` only differs in how the strings for an infinite number or NaN are printed (`inf`, `infinity` and `nan` for `f`; `INF`, `INFINITY` and `NAN` for `F`).
`e`, `E`	`double` value in standard form (*d.ddde±dd*). An `E` conversion uses the letter `E` (rather than `e`) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is `00`. In Windows, the exponent contains three digits by default, e.g. `1.5e002`, but this can be altered by Microsoft-specific `_set_output_format` function.
`g`, `G`	`double` in either normal or exponential notation, whichever is more appropriate for its magnitude. `g` uses lower-case letters, `G` uses upper-case letters. This type differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included. Also, the decimal point is not included on whole numbers.
`x`, `X`	`unsigned int` as a hexadecimal number. `x` uses lower-case letters and `X` uses upper-case.
`o`	`unsigned int` in octal.
`s`	null-terminated string.
`c`	`char` (character).
`p`	`void*` (pointer to void) in an implementation-defined format.
`a`, `A`	`double` in hexadecimal notation, starting with `0x` or `0X`. `a` uses lower-case letters, `A` uses upper-case letters.[5][6] (C++11 iostreams have a `hexfloat` that works the same).
`n`	Print nothing, but writes the number of characters successfully written so far into an integer pointer parameter. Java: indicates a platform neutral newline/carriage return.[7] Note: This can be utilized in Uncontrolled format string exploits.

Examples

- Variables and constants
- Struct
- Arrays
- Count number of digits

Example 1: variables and constants

```c
/* This program demonstrates three ways to use con-
stants.
      Written by:
      Date:
*/
#include <stdio.h>
#define PI 3.1415926536

int main (void)
{
// Local Declarations
   const double cPi = PI;

// Statements
   printf("Defined constant PI: %f\n", PI);
   printf("Memory constant cPi: %f\n", PI);
   printf("Literal constant:    %f\n", 3.1415926536);
   return 0;
}  // main
```

```
Results:
Defined constant PI:   3.141593
Memory constant cPi:   3.141593
Literal constant:      3.141593
```

FUNCTIONS

Functions

A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.

“main()” is a Function. It’s only special because it always gets called first when you run your program.

Return type, or void

Function Arguments

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf(“Hello World\n”);
  return 0;
}
```

Calling a Function: “printf()” is just another function, like main(). It’s defined for you in a “library”, a collection of functions you can call from your program.

Returning a value

Function prototypes

```
int   read_column_numbers( int columns[], int max );
void  rearrange( char *output, char const *input,
       int n_columns, int const columns[] );
```

These look like function definitions – they have the name and all the type information – but each ends abruptly with a semicolon. Where's the body of the function – what does it actually *do*?

(Note that each function *does* have a real definition, later in the program.)

Function prototypes

- Q: Why are these needed, if the functions are defined later in the program anyway?
- A: C programs are typically arranged in “top-down” order, so functions are used (called) before they’re defined.
  - (Note that the function main() includes a call to `read_column_numbers()`.)
  - When the compiler sees a call to `read_column_numbers()` , it must check whether the call is valid (the right number and types of parameters, and the right return type).
  - But it hasn’t seen the definition of `read_column_numbers()` yet!
- The prototype gives the compiler advance information about the function that’s being called.
  - Of course, the prototype and the later function definition must match in terms of type information.

Function declaration

Provides to the compiler the **information** on how to **use** the **function** (prototype).

Syntax:

```
<return_type> <name>( <parameters> );
```

Example:

```c
double compute_mean( int values[], int
num_values );
```

Function definition

Provides to the compiler the **statements** that compose the function.

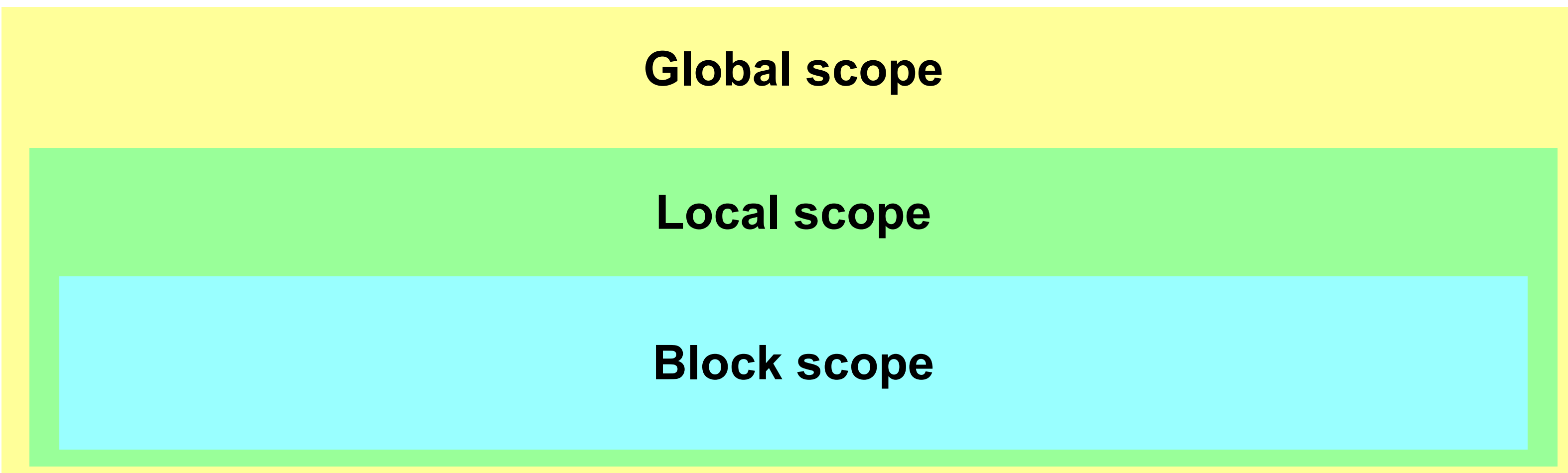
Syntax:

```
<return_type> <name>( <parameters> )
{
    <statements>
    return <expression>;
}
```

```
double compute_mean( int values[], int num_values )
{
    int index;
    double mean = 0;
    for( index = 0; index < num_values; ++index )
    {
        mean += ((double) values[index]) / num_values;
    }
    return mean;
}
```

Scope

- A **part** of the program **where** a function or a variable **name** is **valid**.
- The are **three** levels of scope:
  - **Global** scope (anything declared in the source file)
  - **Local** scope (anything declared in a function)
  - **Block** scope (anything declared in a {}-block)

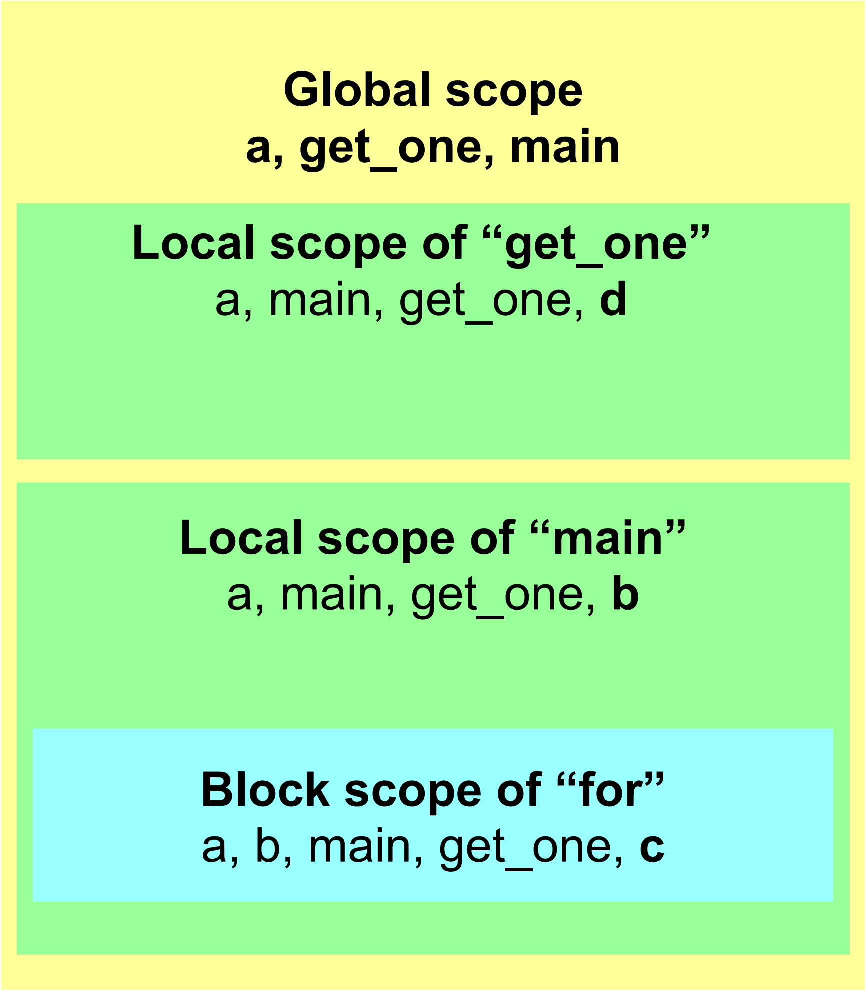


Scope

Source file:

```
int a;
int get_one();
int main();

int get_one()
{ int d =1;
  return d; }

int main()
{       int b;
        for(b=0; b<4; ++b)
        {
                int c;
        }
        return 0;}
```

Scope:



“Static” keyword

- A **local variable** might be declared as **static**
- A static variable it is:
  - **Initialized** just **once** (as for global variables)
  - It is **not created/destroyed** at each call
- **Example:**

```
void make_car() {
    static int vehicle_id = 0;
    …
    vehicle_id = vehicle_id + 1;
    printf(“VIN: &d\n”, vehicle_id);
}
int main() {
    make_car();
    make_car();
    return 0;}
```

Output:

```
VIN: 1
VIN: 2
```

Input parameters (by value) and return value

```
int add( int a, int b){
    int sum = 0;
    sum = a + b;
    a = 3;
    return sum;
}
```

All the values are copied between the local variables

Local variables of "add" function

a b sum

```
int main(){
    int a = 1, b = 2, sum = 0;
    // here a==1, b==2 and sum==0
    sum = add(a,b);
    // here a==1, b==2 and sum==3
}
```

a b sum

Local variables of "main" function

printf() is a function!

```
printf( "Original input : %s\n", input );
```

`printf()` is a library function declared in `<stdio.h>`

Syntax: `printf(` *`FormatString`*`,` *`Expr`*`,` *`Expr`*`...)`

- *`FormatString`*: String of text to print
- *`Expr`*s: Values to print
- *`FormatString`* has placeholders to show where to put the values (note: #placeholders should match #*`Expr`*s)
- Placeholders: `%s` (print as string), `%c` (print as char),
  %d (print as integer),
  %f (print as floating-point)
- \n indicates a newline character

Example

- Write a function tfor binary to decimal conversion

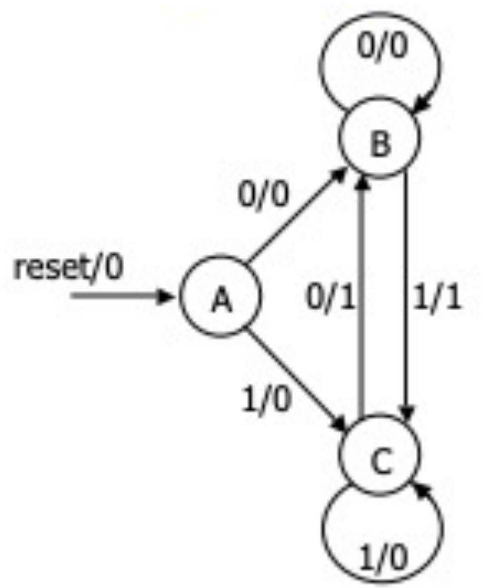
EXERCISES – PART 1

Example: FSM

Let's implement a very simple **Finite State Machine**:

Sequence detection for 01 and 10

Input	I:	{0, 1}
Output	O:	{0, 1}
States	S:	{A, B, C}



Mealy machine

Example FSM: DEFINITIONS

We need to use stdin e stdout

```
#include <stdio.h>

#define  STATE_A  'A'
#define  STATE_B  'B'
#define  STATE_C  'C'

#define  INPUT_0  '0'
#define  INPUT_1  '1'

#define  OUTPUT_0 '0'
#define  OUTPUT_1 '1'
```

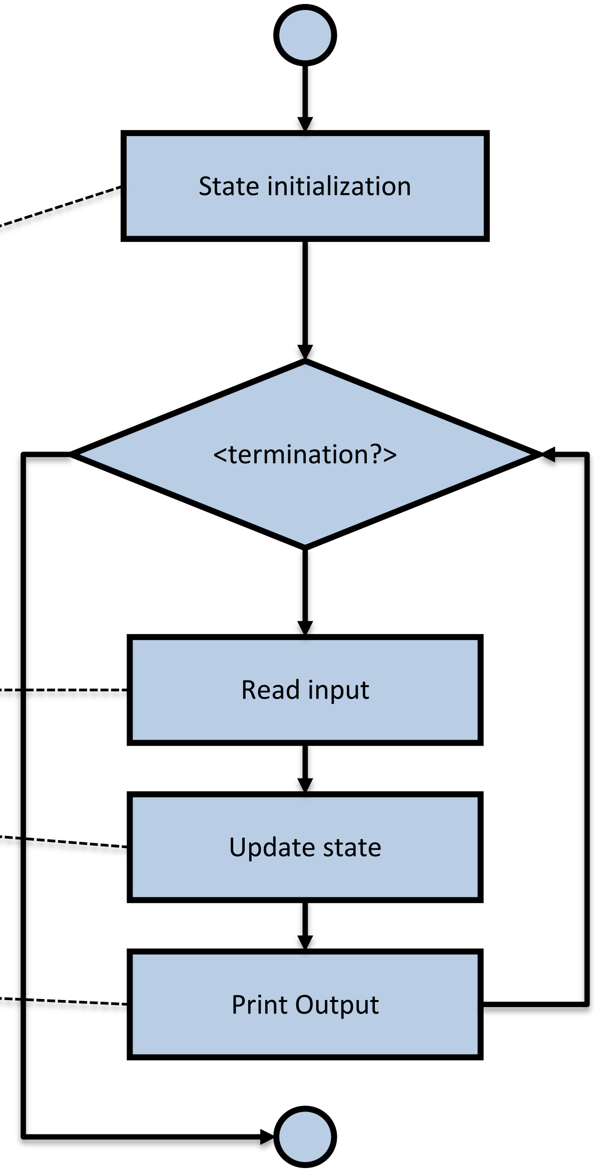
STATES

INPUT ALPHABET

OUTPUT ALPHABET

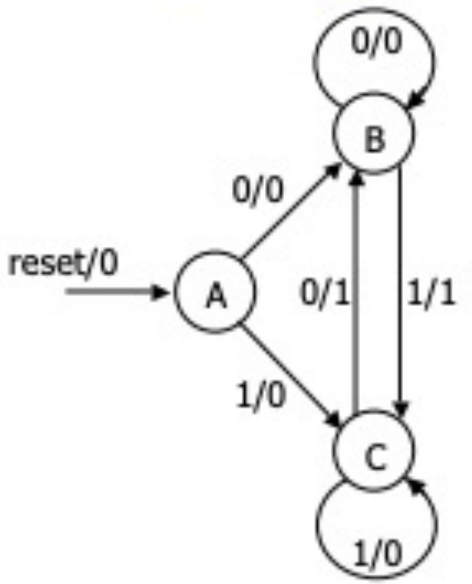
Example FSM: Implementation

```
/* Definitions here */

int main(void)
{
	char state = STATE_A;
	char input, output;

	While (1){  /* continuous loop */
		printf(“INPUT: ”)
		input = scanf("\n%c", &input);

		switch (state) {
			/* Logic here */
		}

		printf(“\n OUTPUT: %c \n”, output);
	}
}
```



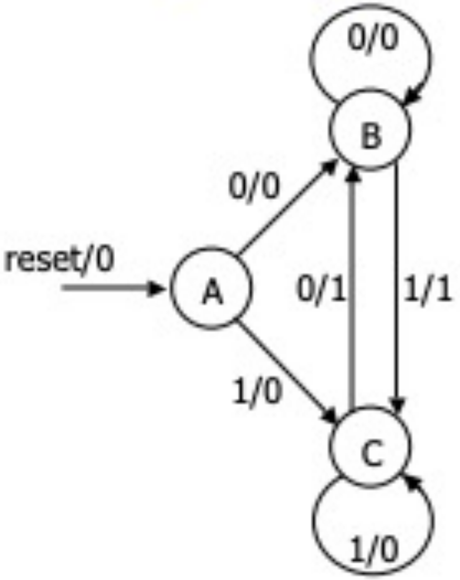
Example FSM: Logic state A

```
switch (state) {
            case STATE_A :
            if (input  == INPUT_0 ){
               output = OUTPUT_0;
               state = STATE_B;
            }
            if (input  == INPUT_1 ){
               output = OUTPUT_0;
               state = STATE_C;
            }
            break;
      [.. Continue..]
}
```



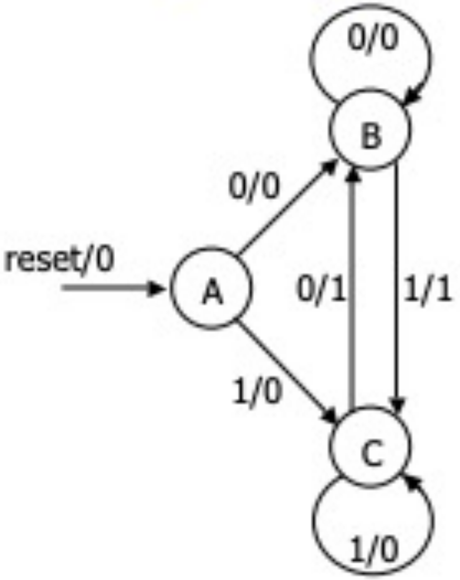
Example FSM: Logic state B

```
switch (state) {
        [.. Continue..]
                case STATE_B :
                 if (input  == INPUT_0 ){
                   output = OUTPUT_0;
                   state = STATE_B;
                 }
                 if (input  == INPUT_1 ){
                   output = OUTPUT_1;
                   state = STATE_C;
                 }
                 break;
        [.. Continue..]
}
```



Example FSM: Logic state C

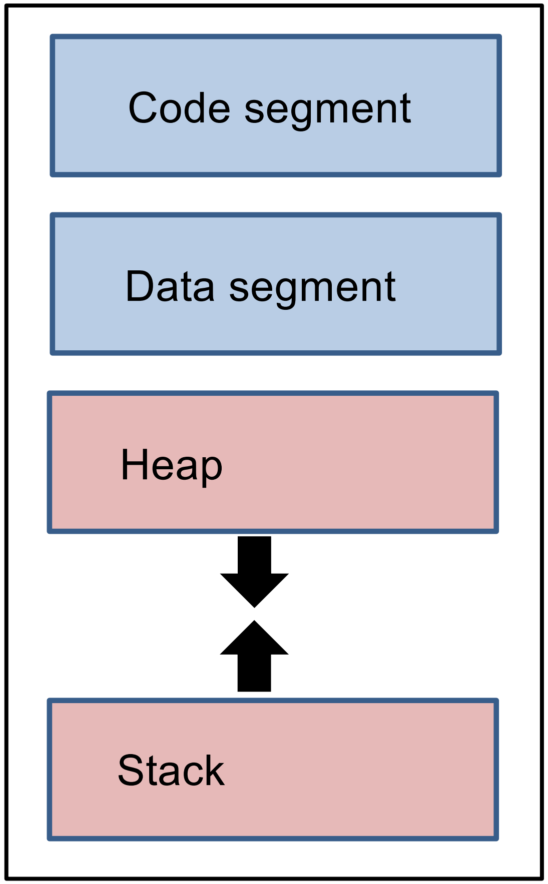
```
switch (state) {
        [.. Continue..]
                case STATE_C :
                 if (input  == INPUT_0 ){
                    output = OUTPUT_1;
                    state = STATE_B;
                 }
                 if (input  == INPUT_1 ){
                    output = OUTPUT_0;
                    state = STATE_C;
                 }
                 break;
}
```



APPENDIX

ADVANCED TOPICS

Address space



It contains the **executable code**

It contains the **global variables**

It contains the **dynamic variables**

It contains the **function stack frames**

Address space

Code segment:

Its dimension is FIXED at compiling time.

Data segment:

Its dimension is FIXED at compiling time.

It contains global variables and static variables.

Heap:

“Large” pool of memory that can be allocated in blocks at **run-time**.

Since the heap has a limited maximum size, it is important to deallocate unused space.

Heap

C provides access to the heap features through the <stdlib.h> library functions:

void* malloc (size_t size)

Request a contiguous block of memory of the given size in the heap (in byte). malloc() returns a pointer to the heap block or NULL if the request could not be satisfied

void free (void* block)

free takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for re-use. After the free(), the client should not access any part of the block or assume that the block is valid memory. The block should not be freed a second time.

“Referencing” and “Dereferencing”

The C language defines also:

- The **referencing** operator: *
- The **dereferencing** operator: **&**

The **referencing operator** retrieves the address of the target variable

0x000ffff9	-
0x000ffffa	-
0x000ffffb	k
0x000ffffc	-

System memory

```c
char my_c = ‘k’;
```

In this example:

- **my_c** refers to the **content** (‘k’)
- **&my_c** refers to its **address** (0x000ffffb)

Pointer to a type

With the deferencing operator, we get the address of that variable.

The C language defines a type suitable to hold such addresses:

Syntax:

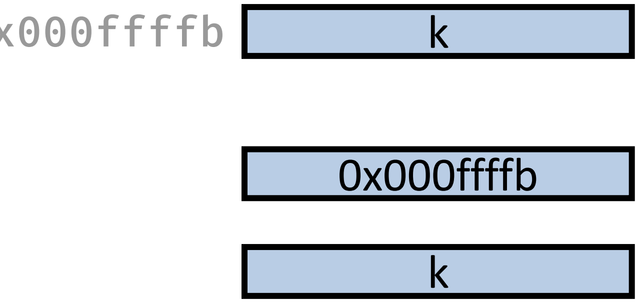
```
<type>* <name>;
```

Example:

```c
char my_c = 'k';
char* my_c_ptr = &my_c;
```

How to use a pointer

The reference operator “*” is used to retrieve the content from a pointer:



System memory

```
char my_c = ‘k’;
char* my_c_ptr = &my_c;

char my_c2 = *my_c_ptr;
printf(“%u\n”, my_c_ptr);
printf(“%c\n”, my_c2);
```

Output:

```
0x000ffffb
k
```

How to use a pointer

```c
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);
printf("%d", c);
```

Input parameters (by reference) and return value

```
int add( int* a, int b){

    int sum = 0;
    sum = *a + b;
    *a = 3;

    return sum;
}

int main(){
    int a = 1, b = 2, sum = 0;
    // here a==1, b==2 and sum==0
    sum = add(&a,b);
    // here a==3, b==2 and sum==3
}
```

Note that ‘a’ is now a pointer

We use the reference operator to use the content
of the pointer (write and read)

We need to retrieve the address of ‘a’

Call by Value and Call by Reference

Call by Value

```
int add( int a, int b){
	int sum = 0;
	sum = a + b;
	a = 3;
	return sum;
}

int main(){
	int a = 1, b = 2, sum = 0;
	// here a==1, b==2 and sum==0
	sum = add(a,b);
	// here a==1, b==2 and sum==3
}
```

Call by Reference

```
int add( int* a, int b){
	int sum = 0;
	sum = *a + b;
	*a = 3;
	return sum;
}

int main(){
	int a = 1, b = 2, sum = 0;
	// here a==1, b==2 and sum==0
	sum = add(&a,b);
	// here a==3, b==2 and sum==3
}
```

Examples

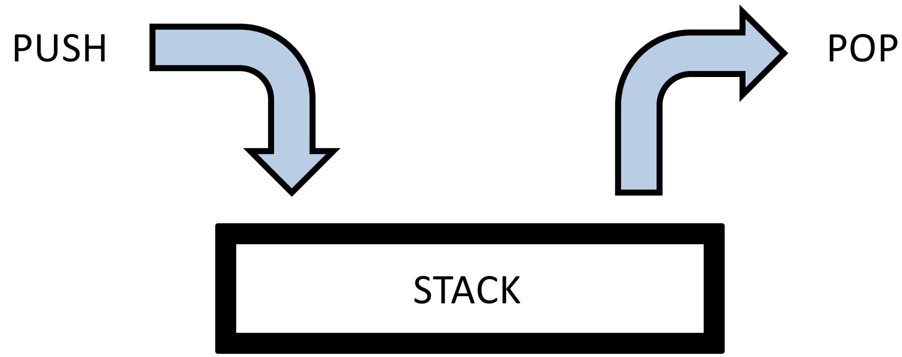
- Basic pointer
- Swap a sequence
- Memory allocation

Stack

Data structure LIFO (Last In, First out)

PUSH: it adds an element on the top of the stack

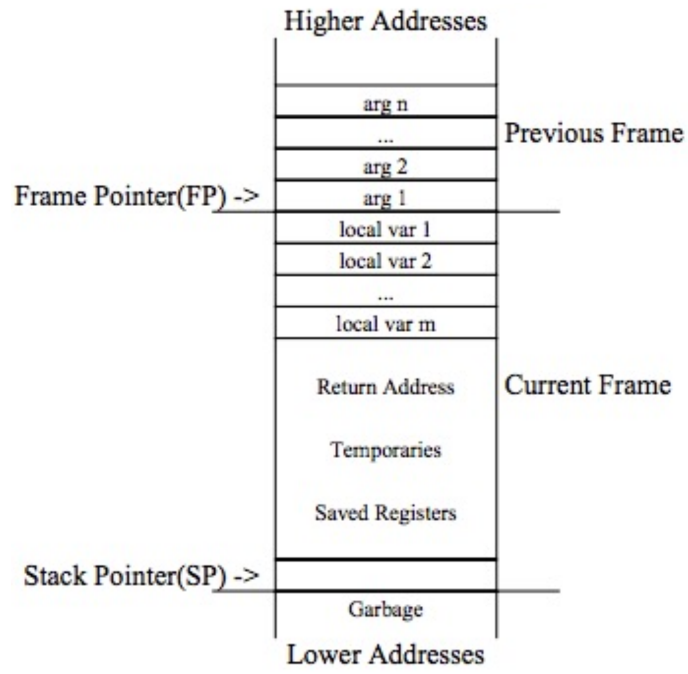
POP: it retrieves the last element from the top of the stack



Activation record

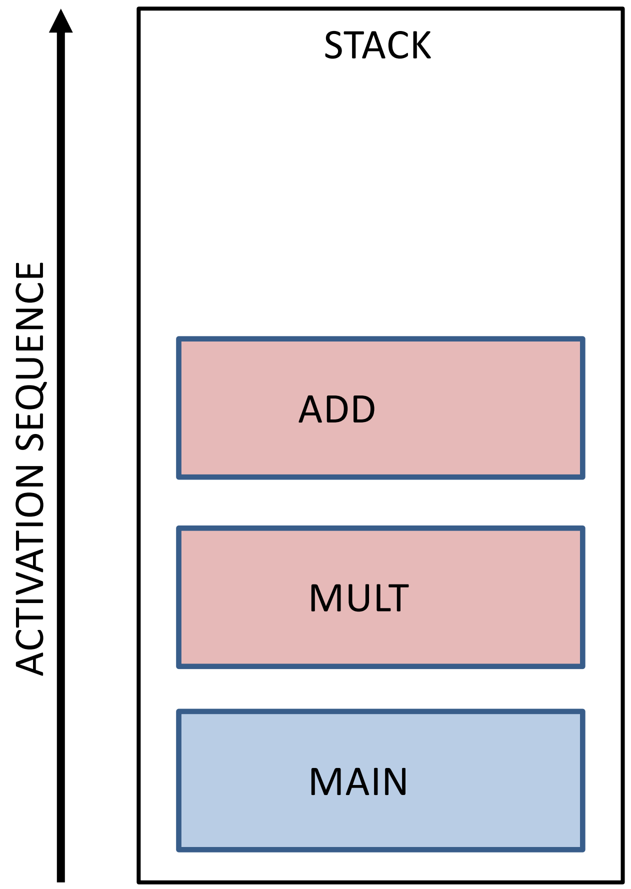
Every **function call** involves creating an **activation record** (also called **function stack frame**) that is saved on the stack.

- Parameters
- Local variables
- Return Address



Nested function calls

```
int add( int a, int b){
    int sum = 0;
    sum = a + b;
    return sum;
}


int mult (int a, int b){
    int mult = 0;
    for (int i= 0, I < a,
i++){
        mult = add(mult,
b);
    }
    return mult;
}

int main(){
    int a = 2, b = 2,
result = 0;
    result = mult(a,b);
}
```



Recursion

The function calls itself.

What happens to the stack?

```
float pow(float x, uint exp)
{
  float result;

  /* base case */
  if (exp == 0)
    return 1.0;

  /* x^(2*a) == x^a * x^a */
  result = pow(x, exp >> 1);
  result = result * result;

  /* x^(2*a+1) == x^(2*a) * x */
  if (exp & 1)
    result = result * x;

  return result;
}
```

Recursion vs iteration

Recursive

```
float pow(float x, uint exp)
{
  float result;

  /* base case */
  if (exp == 0)
    return 1.0;

  /* x^(2*a) == x^a * x^a */
  result = pow(x, exp >> 1);
  result = result * result;

  /* x^(2*a+1) == x^(2*a) * x */
  if (exp & 1)
    result = result * x;

  return result;
}
```

Iterative

```
float pow(float x, uint exp)
{
  float result = 1.0;

  int bit;
  for (bit = sizeof(exp)*8-1;
       bit >= 0; bit--) {
    result *= result;
    if (exp & (1 << bit))
      result *= x;
  }

  return result;
}
```

Which is better? Why?

Macros

Macros can be a useful way to customize your interface to C and make your code easier to read and less redundant. However, when possible, use a static inline function instead.

Macros and static inline functions must be included in any file that uses them, usually via a header file. Common uses for macros:

```
/* Macros are used to define constants */
#define FUDGE_FACTOR   45.6
#define MSEC_PER_SEC   1000
#define INPUT_FILENAME “my_input_file”

/* Macros are used to do constant arithmetic */
#define TIMER_VAL      (2*MSEC_PER_SEC)

/* Macros are used to capture information from the compiler */
#define DBG(args...) \
  do { \
    fprintf(stderr, “%s:%s:%d: “, \
      __FUNCTION__, __FILE__, __LINENO__); \
    fprintf(stderr, args...); \
  } while (0)

/* ex. DBG(“error: %d”, errno); */
```

Float constants must have a decimal point, else they are type int

Put expressions in parens.

Multi-line macros need \

args... grabs rest of args

Enclose multi-statement macros in do{}while(0)

Macros

Sometimes macros can be used to improve code readability… but make sure what's going on is obvious.

```
/* often best to define these types of macro right where they are used */
#define CASE(str) if (strncasecmp(arg, str, strlen(str)) == 0)

void parse_command(char *arg)
{
  CASE("help") {
    /* print help */
  }
  CASE("quit") {
    exit(0);
  }
}

/* and un-define them after use */
#undef CASE
```

```
void parse_command(char *arg)
{
  if (strncasecmp(arg, "help", strlen("help")) {
    /* print help */
  }
  if (strncasecmp(arg, "quit", strlen("quit")) {
    exit(0);
  }
}
```

Macros can be used to generate static inline functions. This is like a C version of a C++ template. See emstar/libmisc/include/queue.h for an example of this technique.

Using headers files

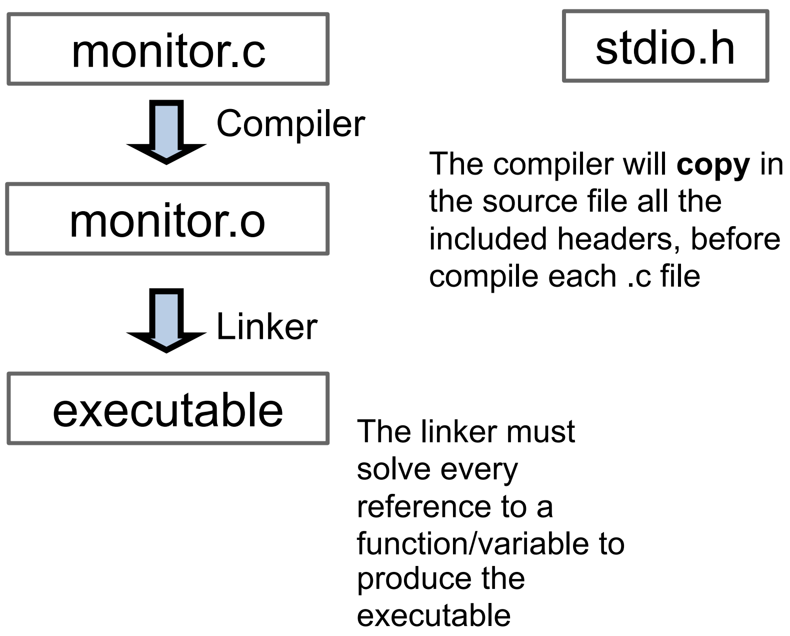
For any non-trivial application is recommended to split the code in multiple source files.The idea is to divide the problem in sub-problems and code the solution of each sub-problem in two files:

A **source** file (with the .c extension)

- Definition of global variables
- Function definition

A **header** file (with the .h extension)

- Declaration of data structures
- Function prototype declaration



Common mistakes: Multiple declaration

monitor.h

```c
struct monitor { ... };
```

network_monitor.h

```c
#include "monitor.h"
```

memory_monitor.h

```c
#include "monitor.h"
```

main.c

```c
#include "network_monitor.h"
#include "memory_monitor.h"
```

main.c

```c
struct monitor { ... };
struct monitor { ... };
```

Error: we have defined the struct monitor more than one time!

Solution: include guard

monitor.h

```
#ifndef MY_MONITOR_HDR
#define MY_MONITOR_HDR
struct monitor { ... };
#endif
```

network_monitor.h

```
#include “monitor.h”
```

memory_monitor.h

```
#include “monitor.h”
```

main.c

```
#include “network_monitor.h”
#include “memory_monitor.h”
```

main.c

```
struct monitor { ... };
```

It’s **ok**, only the first include of the file is really used, the other one is discarded

Common mistakes: Multiple definition

main.c

```
...
int var = 5;
...
```

common.c

```
...
int var = 5;
...
```

`main.o`

`common.o`

`executable`

Error at **linking** time: both files **allocate** memory for the variable var

Solution: extern keyword

main.c

```c
...
extern int var;
...
```

common.c

```c
...
int var = 5;
...
```

`main.o`

`common.o`

`executable`

It's ok, only common.o allocate memory for the variable var

EXERCISES – PART 2

Exercises; its your turn now!

1) Calculate the factorial of a number entered by the user;
2) calculate the average of n number of elements entered by the user using arrays (n<=100);
3) store information of a student and display it using a struct;
4) store a sentence entered by the user in a file;
5) read a line from a file and display it;
6) Array sort: bubble sort;
7) Binary search in array.

Exercise 1 - factorial

```c
#include <stdio.h>
int main() {
    int n, i;
    unsigned long long fact = 1;
    printf("Enter an integer: ");
    scanf("%d", &n);

    // shows error if the user enters a negative integer
    if (n < 0)
        printf("Error! Factorial of a negative number doesn't exist.");
    else {
        for (i = 1; i <= n; ++i) {
            fact *= i;
        }
        printf("Factorial of %d = %llu", n, fact);
    }

    return 0;
}
```

Is it possible to use recursion?

Exercise 2 – array average

```c
#include <stdio.h>
int main() {
    int n, i;
    float num[100], sum = 0.0, avg;

    printf("Enter the numbers of elements: ");
    scanf("%d", &n);

    while (n > 100 || n < 1) {
        printf("Error! number should in range of (1 to 100).\n");
        printf("Enter the number again: ");
        scanf("%d", &n);
    }

    for (i = 0; i < n; ++i) {
        printf("%d. Enter number: ", i + 1);
        scanf("%f", &num[i]);
        sum += num[i];
    }

    avg = sum / n;
    printf("Average = %.2f", avg);
    return 0;
}
```

Exercise 3 - struct

```c
#include <stdio.h>
struct student {
    char name[50];
    int roll;
    float marks;
} s;

int main() {
    printf("Enter information:\n");
    printf("Enter name: ");
    fgets(s.name, sizeof(s.name), stdin);

    printf("Enter roll number: ");
    scanf("%d", &s.roll);
    printf("Enter marks: ");
    scanf("%f", &s.marks);

    printf("Displaying Information:\n");
    printf("Name: ");
    printf("%s", s.name);
    printf("Roll number: %d\n", s.roll);
    printf("Marks: %.1f\n", s.marks);

    return 0;
}
```

Exercise 4 – file out

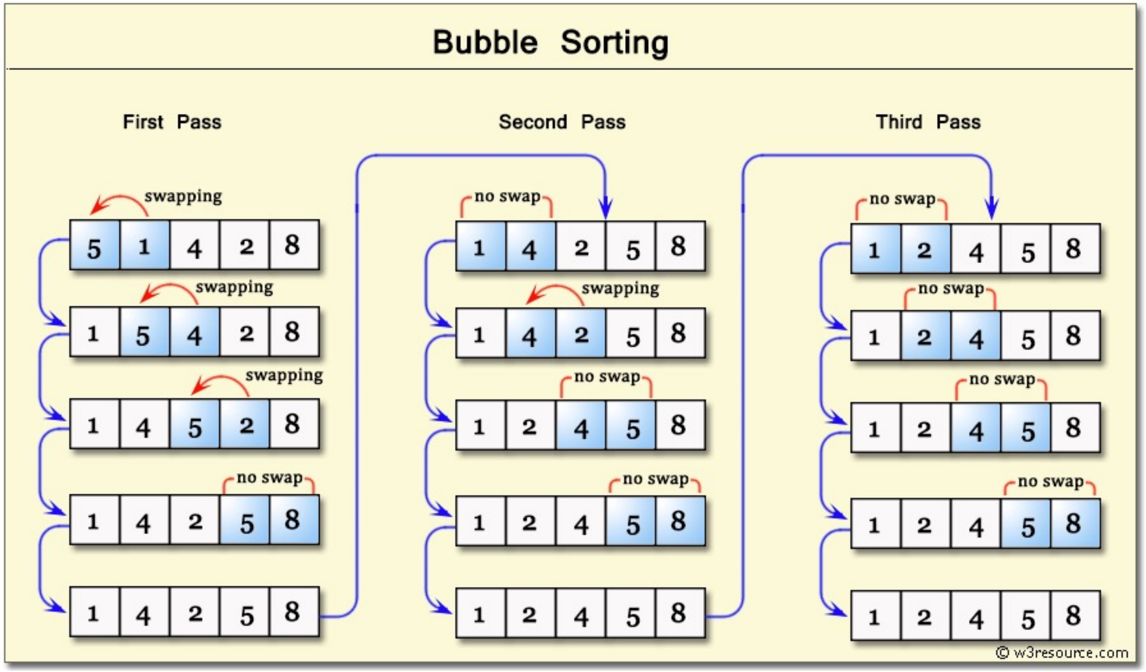
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char sentence[1000];

    // creating file pointer to work with files
    FILE *fptr;

    // opening file in writing mode
    fptr = fopen("program.txt", "w");

    // exiting program
    if (fptr == NULL) {
        printf("Error!");
        exit(1);
    }
    printf("Enter a sentence:\n");
    fgets(sentence, sizeof(sentence), stdin);
    fprintf(fptr, "%s", sentence);
    fclose(fptr);
    return 0;
}
```

Exercise 5 – file in

```c
#include <stdio.h>
#include <stdlib.h> // For exit() function
int main() {
    char c[1000];
    FILE *fptr;
    if ((fptr = fopen("program.txt", "r")) == NULL) {
        printf("Error! opening file");
        // Program exits if file pointer returns NULL.
        exit(1);
    }

    // reads text until newline is encountered
    fscanf(fptr, "%[^\n]", c);
    printf("Data from the file:\n%s", c);
    fclose(fptr);

    return 0;
}
```

Exercise 6 - bubble sort

Note: Bubble Sort works by repeatedly swapping the adjacent elements if they are in wrong order.



https://www.w3resource.com/

Exercise 6

```c
#include <stdio.h>

void bubble_sort (int *x, int n) {
    int i, t, j = n, s = 1;
    while (s) {
        s = 0;
        for (i = 1; i < j; i++) {
            if (x[i] < x[i - 1]) {
                t = x[i];
                x[i] = x[i - 1];
                x[i - 1] = t;
                s = 1;
            }
        }
        j--;
    }
}

int main () {
    int x[] = {15, 56, 12, -21, 1, 659, 3, 83, 51, 3, 135, 0};
    int n = sizeof x / sizeof x[0];
    int i;
    for (i = 0; i < n; i++)
        printf("%d%s", x[i], i == n - 1 ? "\n" : " ");
    bubble_sort(x, n);
    for (i = 0; i < n; i++)
        printf("%d%s", x[i], i == n - 1 ? "\n" : " ");
    return 0;
}
```

https://www.w3resource.com/

Exercise 7 – binary search

Binary Search : In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomies divide-and-conquer search algorithm and executes in logarithmic time.

https://www.w3resource.com/

Exercise 7

```c
#include<stdio.h>
  void main()
  {
  int  arra[100],i,n,x,f,l,m,flag=0;
  printf("Input no. of elements in  an array\n");
  scanf("%d",&n);
  printf("Input  %d value in ascending order\n",n);
  for(i=0;i<n;i++)
  scanf("%d",&arra[i]);
  printf("Input  the value to be search : ");
  scanf("%d",&x);
  /* Binary Search  logic */
  f=0;l=n-1;
  while(f<=l)
  {
  m=(f+l)/2;
  if(x==arra[m])
  {
  flag=1;
  break;
  }
  else  if(x<arra[m])
  l=m-1;
  else
  f=m+1;
  }
  if(flag==0)
  printf("%d  value not found\n",x);
  else
  printf("%d value  found at %d position\n",x,m);
  }
```

https://www.w3resource.com/

References

- Davide Gadioli, Politecnico di Milano, *C overview*
- Lewis Girod, CENS Systems Lab, http://lecs.cs.ucla.edu/~girod/talks/c-tutorial.ppt
- Reek Chs, Safety Critical Programming in C, *Introduction to C*
- Gaikwad Varsha P., Govt. College of Engg. Aurangabad, *Basics of C*
- Nick Parlante , *Essential C*, Copyright 1996-2003
- Wikipedia: https://en.wikipedia.org/wiki/C_(programming_language)
- Examples: https://www.programiz.com/c-programming/examples
- Examples: https://www.w3resource.com/

POLITECNICO
MILANO 1863

Questions?

andrea.masciadri@polimi.it