IEE 2463
Programmable Electronic Systems

C Programming Language

Electrical Engineering Department

Pontificia Universidad Católica de Chile

peclab.ing.uc.cl

Introduction
C Programming Language

This section provides a brief overview of the programming language and compilers.

Introduction

The C Language

- C is a **general-purpose** programming Language.
- It is a tool to easily **move data within a processor**. The most basic tool to do that is Assembly, which directly uses the set of instructions of a processor.
- *Basic Combined Programming Language (**BCPL**)*, was first published by Martin Richards in 1966 from Cambridge University. (improvement from **CPL** published in 1963). Both come from root language **ALGOL** 60 (published in 1960).
- **B** language is early attempt for high level programming language. Published in 1970 by Ken Thompson from **B**ell Labs.
- **B** and **BCLP** has **no types of variables**. The B language is the base for C language.
- **C** language **define types of variables**, as character, int and floats of different sizes. First version published in 1972.
- Standardization of C developed as: ANSI C (1989), ANSI/ISO C (1990), C99 (1999), C11 (2007) and C17 (2018). New version should be released in 2023. For details of improvements see here.
- C language is today the standard language to program microcontrollers.

Introduction

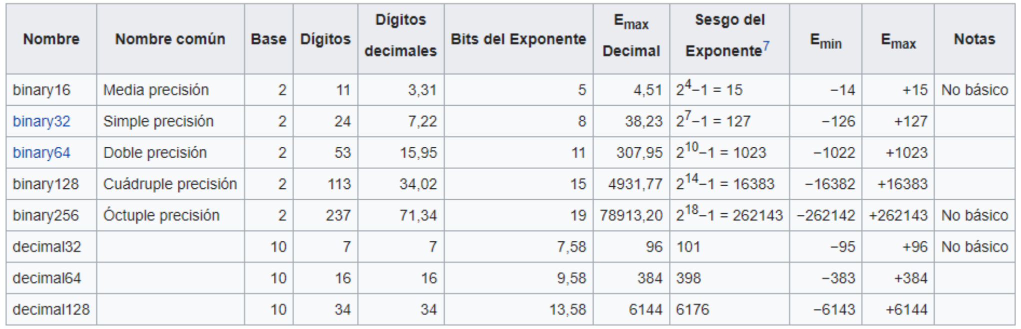
The C Language

- C language provides arrays , structures and data joint.
- C language provides arithmetic calculation based on **memory pointers.**
- C language provides a set of data **flow control instructions**: if, for, while switch, break.
- C is **a low-level programming language**. Uses characters and memory addresses. A C-Compiler translate C code into processor instructions and lately into a binary code to be loaded into processors program memory.
- C has no instructions to deal with objects, i.e. composed of "chain of characters", lists or arrays. **It not an object-oriented language.** For those tasks we create our own functions (e.g. to compare two texts).
- **C is a sequential language**. It can no execute parallel instruction or co-routines.
- The C compiler is called gcc (**G**NU **C**ompiler **C**ollection) (free software). GCC includes C, C++, Objective-C, Objective-C++, Fortran, Ada, D, and Go.

Introduction
The C Language

- **C is a language independent of the processor architecture**. Thereby, a C program can be easily exported to different microcontrollers architectures.

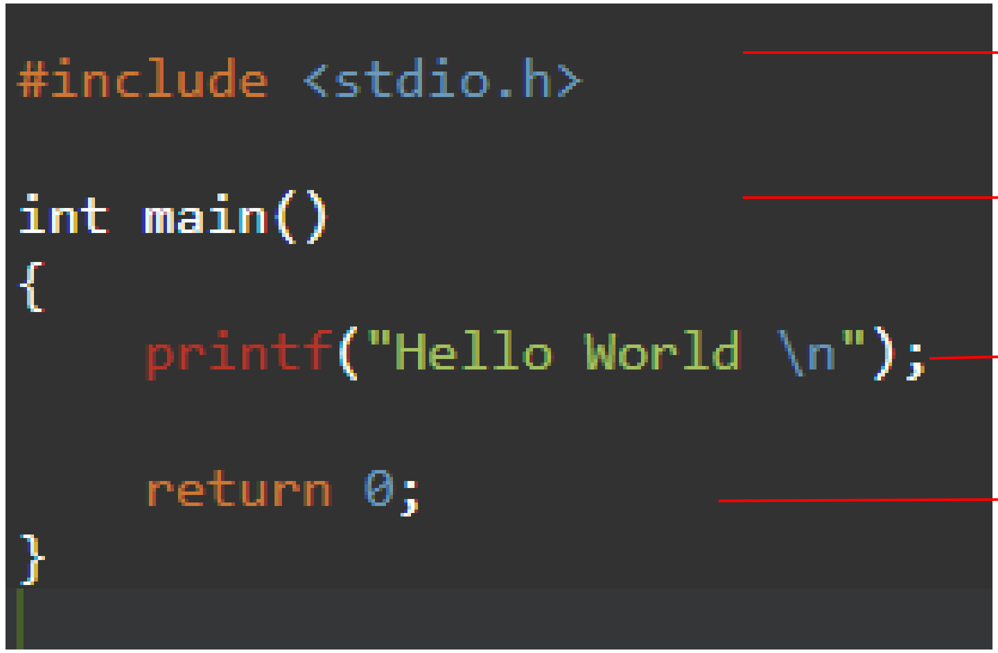
- To represent a decimal number, a certain numbers of bits are required for its integer and decimal part (floating point numbers). The standard IEEE 754 regulates the definition of floating-point numbers.



Nombre	Nombre común	Base	Dígitos	Dígitos decimales	Bits del Exponente	E_{max} Decimal	Sesgo del Exponente[7]	E_{min}	E_{max}	Notas
binary16	Media precisión	2	11	3,31	5	4,51	$2^4-1 = 15$	−14	+15	No básico
binary32	Simple precisión	2	24	7,22	8	38,23	$2^7-1 = 127$	−126	+127	
binary64	Doble precisión	2	53	15,95	11	307,95	$2^{10}-1 = 1023$	−1022	+1023	
binary128	Cuádruple precisión	2	113	34,02	15	4931,77	$2^{14}-1 = 16383$	−16382	+16383	
binary256	Óctuple precisión	2	237	71,34	19	78913,20	$2^{18}-1 = 262143$	−262142	+262143	No básico
decimal32		10	7	7	7,58	96	101	−95	+96	No básico
decimal64		10	16	16	9,58	384	398	−383	+384	
decimal128		10	34	34	13,58	6144	6176	−6143	+6144	

- **An important library in C code, is the stdio.h.** This library collect instructions to *talk* with the operative system. This means, to read/write files, inputs and outputs of the system, e.g. serial port, usb-port, screen,etc.

Introduction
The C Language – Hello World



```
#include <stdio.h>

int main()
{
    printf("Hello World \n");

    return 0;
}
```

Include base library.

Define a *main* function. It does not receive arguments and return an integer number. This function is the base of out program, and its name can not be used for other function.

Within the *main* function, the *printif("")* function is called. This is defined into the stdio.h library to print characters into the console. Command \n means new line.

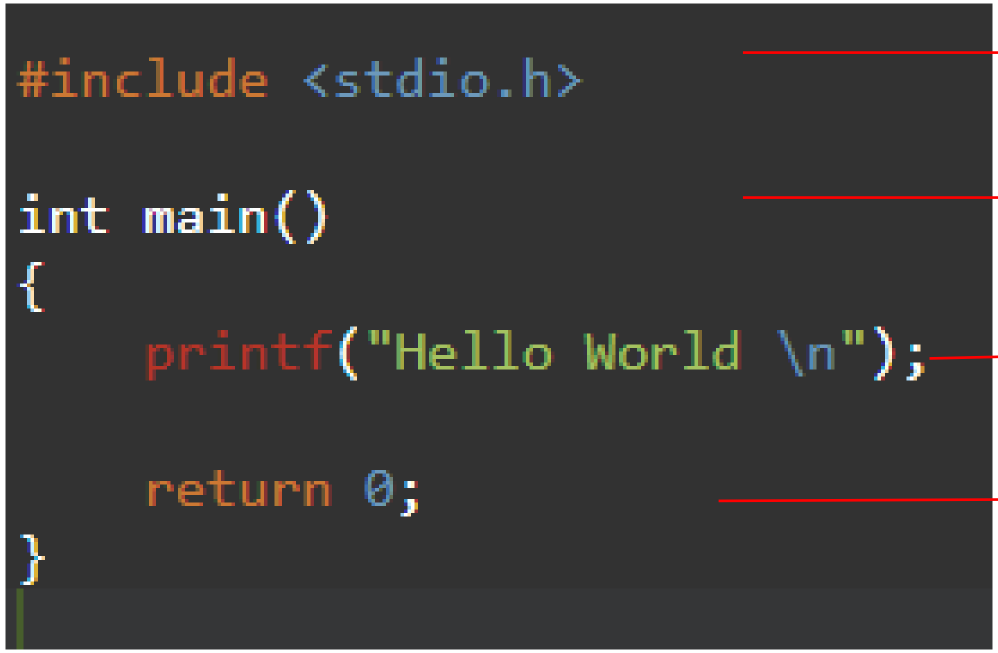
We arbitrary choose to return the number 0 when functions ends.

Important details for the syntaxis:

1. To end one instruction the symbol **;** is **mandatory.**
2. Functions content are delimited by brackets **{}**. These brakers are needed inly when we define/describe/create the function, but not when we invoke it.
3. Functions arguments are delimited by brackets **().**

Introduction

The C Language – Hello World



```c
#include <stdio.h>

int main()
{
    printf("Hello World \n");

    return 0;
}
```

Include base library.

Define a *main* function. It does not receive arguments and return an integer number. This function is the base of out program, and its name can not be used for other function.

Within the *main* function, the *printif("")* function is called. This is defined into the stdio.h library to print characters into the console. Command \n means new line.

We arbitrary choose to return the number 0 when functions ends.

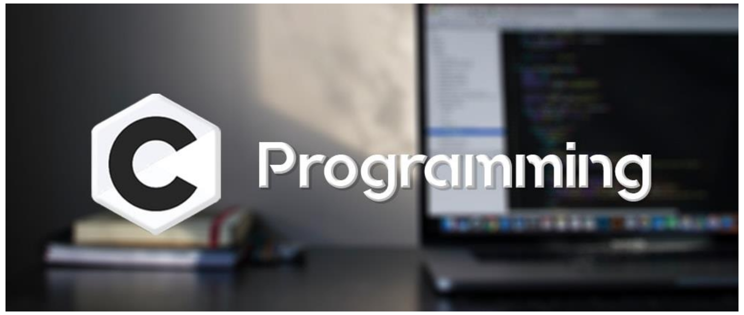
See the appendix A of this document to known how to install a C compiler in your PC.

Important details for the syntaxis:

1. To end one instruction the symbol **;** is **mandatory.**
2. Functions content are delimited by brackets **{}**. These brakers are needed inly when we define/describe/create the function, but not when we invoke it.
3. Functions arguments are delimited by brackets **().**

Basic Concepts
C Programming Language

Identifiers, types of variables, operators, expressions, precedence and more.



Keywords

These keywords are reserved by the compiler and con not be used by the programmer. The compiler understand it as an instruction.

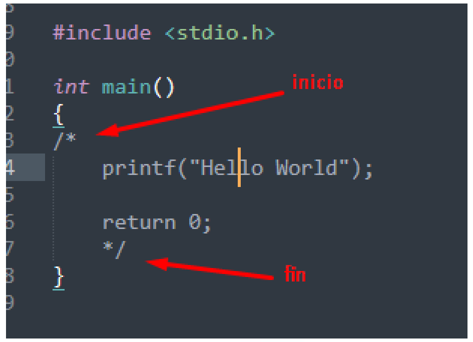
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

Basic Concepts
Identifiers

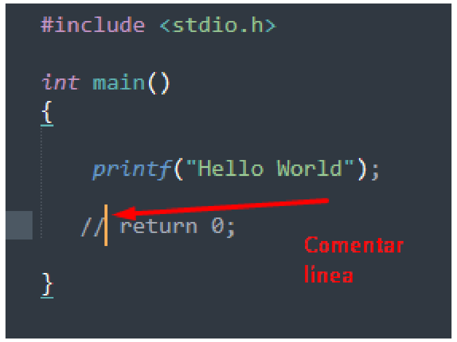
An identifier can be a name given to a variable, function or any assignment. Identifiers can be only alphanumeric symbols:

- a-z, A-Z, 0-9 and underline "_". E.g. name, name_number
- Note1: Avoid start with _ , as it is used by libraries routines.
- Note2: Generally, *lower case* is used for *variables* and *upper case* is symbolic *constant*.
- You can not start an identifier with a number.
- Identifier are case sensitive. *Number* is not the same as *number* . Be careful!

To comment one or several lines we use // or /* …text to be commented.. */, see this example.



```
#include <stdio.h>

int main()
{
/*
    printf("Hello World");

    return 0;
    */
}
```



```
#include <stdio.h>

int main()
{

    printf("Hello World");

    // return 0;

}
```

Basic Concepts

Type of Data

The **type** of a variable defines how much **space** it occupies and how the **bit pattern** stored is **interpreted**.

Data Type	Description
Basic Types	Arithmetic types, further classified into: **integers** and **floating-point.**
Enumerated Types	Arithmetic type but they take only certain discrete integer values.
Type void	Indicates the absence of value.
Derived types	pointers, array, structure, union and function types.

Basic Concepts
Type of Variables

- Syntaxis to define a variable is:
  - <datatype> <variable_1>, <variable_2>, <variable_3>......, <variable_N>;

- The type of variables are:

Variable	Size byte	Format identification	Description
char	1	%c	One byte character.
int	2 or 4	%i or %d	An integer number. Its size is same as the size of data bus of the system where it is executed.
float	4	%f	Standard/single precision floating-point number.
double	8	%lf	Double precision floating-point number.
void			Represent absence of type. Used only for defining **functions** and **pointer**. We

Basic Concepts

Type of Variables - Quiailifers

We can also add qualifiers to our variables:

Short and **long** applied to **int**. It maintains the type of variable but increase or decrease its size.

E.g.

- *short int variable1* or equivalently *short variable1*

Depending on the machine **int** can be 16 or 32 bit. Following rules must be fulfilled:

- **short** and **int** are at least 16bits.
- **longs** are at least 32 bits.
- **short** can not be longer than **int**.
- **int** can not be longer than **long**.

Example: in a machine of 32bits. short is 16bit, int is 32bits and long is 64 bits.

To know the bits assigned to the variables in my machine use the operator ***sizeof***

Basic Concepts

Type of Variables - Quialifiers

We can also add qualifiers to our variables:

signed and **unsigned** applied to **int** and **char**.

Unsigned number is a positive number between 0 and 2^n. Where n is its size in bits.

Signed numbers are from -2^(n-1) to 2^(n-1) – 1

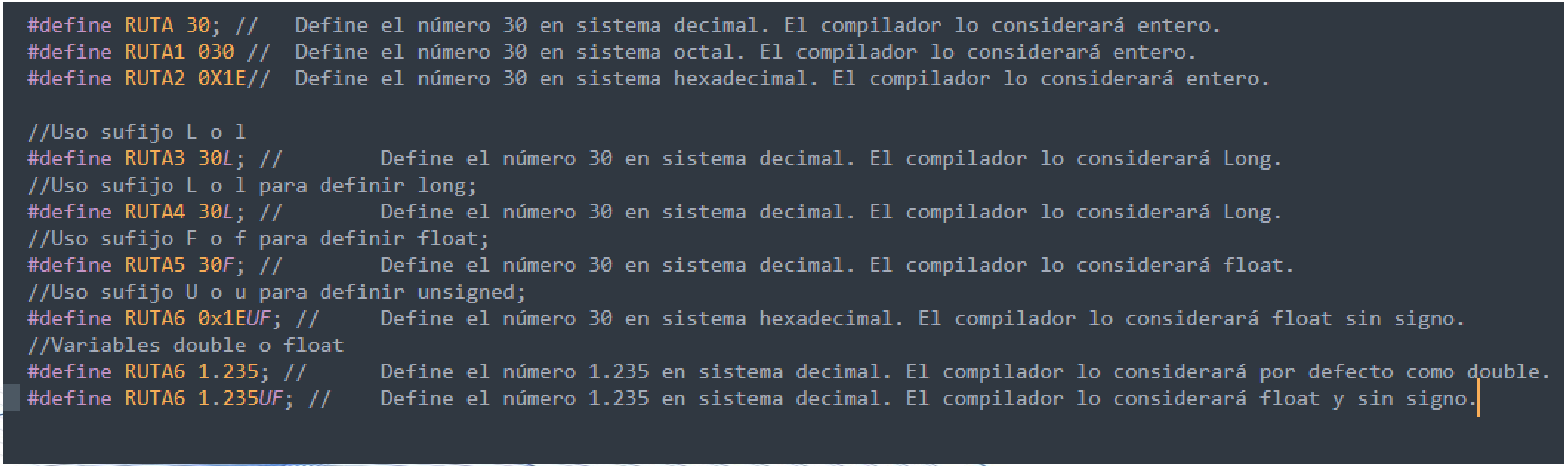
E.g. a 8-bits **signed char** is a number between -128 to 127.

Basic Concepts

Constants - int, double, float

- Constants does never change its value across the code.
- We define the *header* files with an extention .h. It contains all our constant definitions.
- We can define numbers and ASCII characters.

The following example shows several different definitions:



```
#define RUTA 30; //   Define el número 30 en sistema decimal. El compilador lo considerará entero.
#define RUTA1 030 //  Define el número 30 en sistema octal. El compilador lo considerará entero.
#define RUTA2 0X1E//  Define el número 30 en sistema hexadecimal. El compilador lo considerará entero.

//Uso sufijo L o l
#define RUTA3 30L; //        Define el número 30 en sistema decimal. El compilador lo considerará Long.
//Uso sufijo L o l para definir long;
#define RUTA4 30L; //        Define el número 30 en sistema decimal. El compilador lo considerará Long.
//Uso sufijo F o f para definir float;
#define RUTA5 30F; //        Define el número 30 en sistema decimal. El compilador lo considerará float.
//Uso sufijo U o u para definir unsigned;
#define RUTA6 0x1EUF; //     Define el número 30 en sistema hexadecimal. El compilador lo considerará float sin signo.
//Variables double o float
#define RUTA6 1.235; //      Define el número 1.235 en sistema decimal. El compilador lo considerará por defecto como double.
#define RUTA6 1.235UF; //    Define el número 1.235 en sistema decimal. El compilador lo considerará float y sin signo.
```

Basic Concepts

Constants - character

- To define a **character** is basically a **int** number, written as a character within apostrophes **'X'**.
- The numeric value of a character is given by the ASCII code.
- Constant as character are part of arithmetic operations as any other number. However, we can use them to be compared with other characters.
- See the following examples:

```
// Variable Simple
#define PT 't';    // La constante PT tiene por contenido el caracter t , de tipo char. EL cual en según ASCII representa el número 116 decimal.
// Secuencia de escape
#define N_LINEA '\n';    // La constante N_LINEA representa la secuencia de escape \n, que significa un nuevo salto de línea. (\n es el número 11 decimal en ASCII)
                         // Aunque se ve como dos caracteres, representa solo 1 (salto de línea).
                         // Este tipo de secuencias de escape pueden ser representadas también por un byte en número octal o hexadecimal
#define N_LINEA2 '\013'; // La constante N_LINEA2 representa la secuencia de escape \n
#define N_LINEA3 '\xb';  // La constante N_LINEA3 representa la secuencia de escape \n
#define NULO '\0';       // La constante NULO tiene el valor entero 0. El cuial se representa por el caracter \0
#define NULO_TEXT '0';   // La constante NULO_TEXT representa el caracter 0. Sin embargo, su número entero es 48 según ASCII.
```

- A list of scape sequences is also given here:

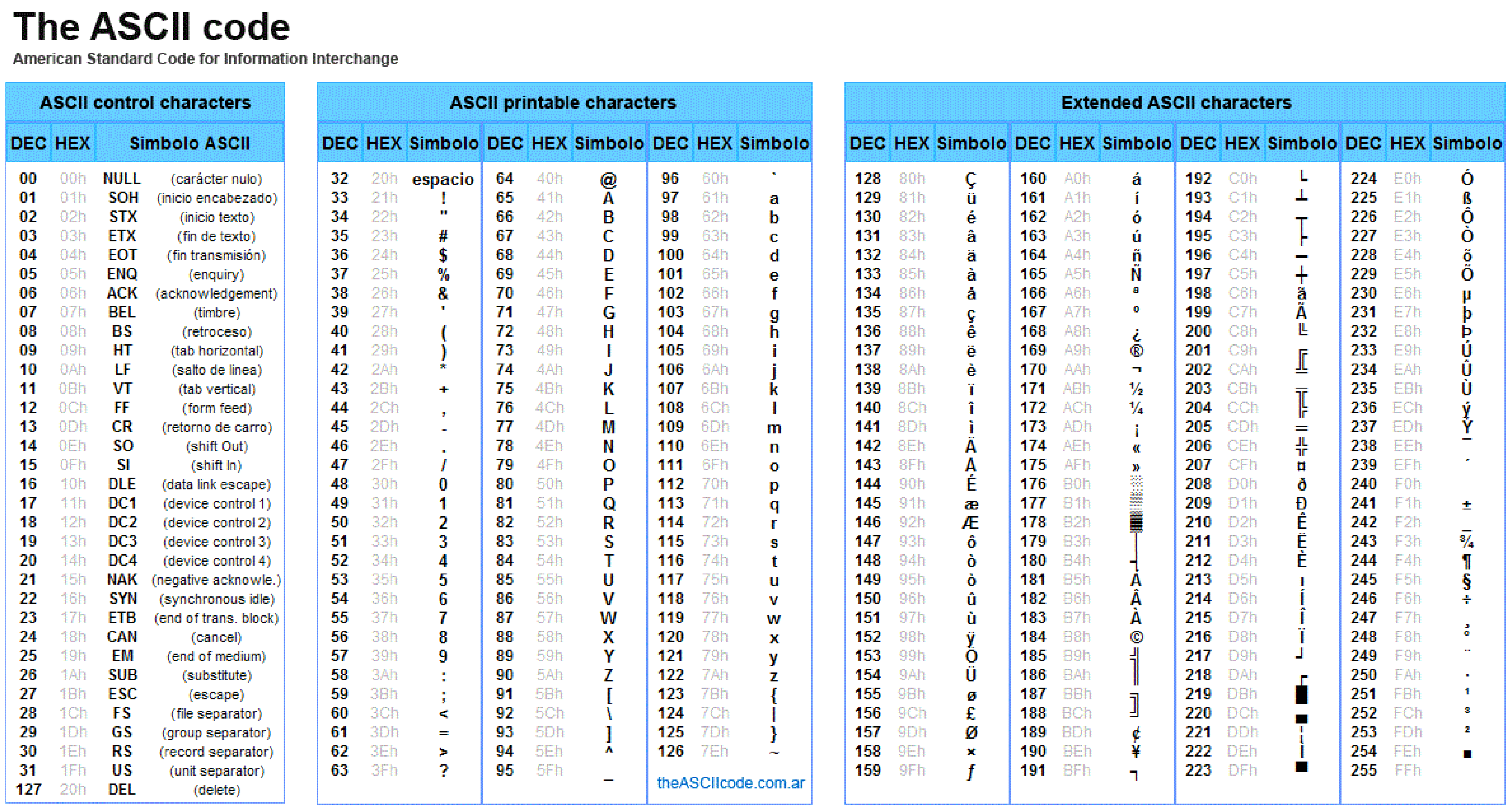
\a	*Alarm or Beep*	\t	*Tab (Horizontal)*	\?	*Question Mark*
\b	*Backspace*	\v	*Vertical Tab*	\ooo	*octal number*
\f	*Form Feed*	\\	*Backslash*	\xhh	*hexadecimal number*
\n	*New Line*	\'	*Single Quote*	\0	*Null*
\r	*Carriage Return*	\"	*Double Quote*		

Basic Concepts
Constants - A constant Expression

- A **string** constant is a sequence of characters. It must be written between quotation marks **" "**
    - E.g. "I am a string"
- A string has a null character at the end "\0". This limits is size and storage space in memory.
- Function strlen() from stdio.h library gives the length of a string. (not including null character)
- We must not be confused between ´**X**´ con **"X".** The **first** is used to reproduce the numeric representation of the character x, The **second** is recognized as a **string**, in this case of size 2, i.e. the character **x** and the null **\0**

Basic Concepts

ASCII Code



The ASCII code

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

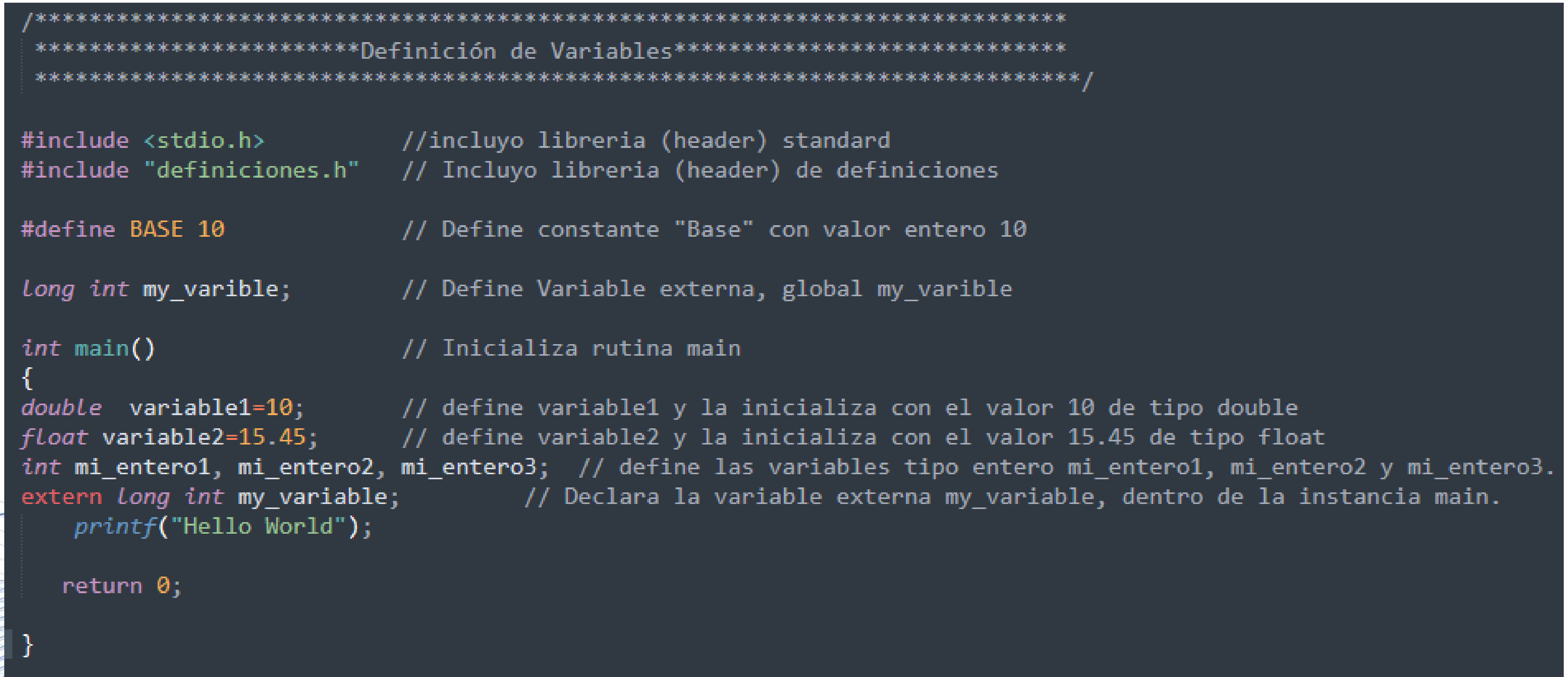
theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Basic Concepts
Variables Declaration

- All variables must be declared before being used. (at the beginning of the program)
- Variables should be initialized when declared. (this is a good practice)
- **Static** and **external** (or global) variables are initialized as **cero** by the compiler.
- **Local** (or automatic) variables (the variables within the main function or any function), have **undefined** values as default. Be careful with that!. These variables do not retain their value in a new function call.



```
/****************************************************************************
 ************************Definición de Variables*****************************
 *****************************************************************************/

#include <stdio.h>           //incluyo libreria (header) standard
#include "definiciones.h"    // Incluyo libreria (header) de definiciones

#define BASE 10              // Define constante "Base" con valor entero 10

long int my_varible;         // Define Variable externa, global my_varible

int main()                   // Inicializa rutina main
{
double  variable1=10;        // define variable1 y la inicializa con el valor 10 de tipo double
float variable2=15.45;       // define variable2 y la inicializa con el valor 15.45 de tipo float
int mi_entero1, mi_entero2, mi_entero3;  // define las variables tipo entero mi_entero1, mi_entero2 y mi_entero3.
extern long int my_variable;         // Declara la variable externa my_variable, dentro de la instancia main.
    printf("Hello World");

   return 0;

}
```

Basic Concepts

Variables Declaration – Storage Classes

- Four classes defines the life-time of variables and/or functions within a C program:
  - **auto:** default class for all local variables
  - **register:** local variable that should be stored in a register instead of RAM. The maximum size of the variable is given by the register size. No memory operators can be used.
  - **static:** It forces the compiler to maintain the value of a local variable between different functions execution. When used in global variables, restrict its use to the file were it is declared (not to other files).
  - **extern:** Makes the variable visible for all files. It can not be initialized.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
}
```

- We have **two** files. To compile them we use **$gcc main.c support.c**
- main.c declare and define variable **count** as **5**.
- Function **write_extrern** has no input or output, but uses the **extern variable count** to print in console. Now there is **only one count variable** for all files.
- Executing this program gives:
  - **count is 5**

Basic Concepts
Printf Function

- It belong to stdio.h library and can print messages in console. The formatting is controlled by “format identifiers” as follow:

Format Specifier	Type
%c	Character
%d	Signed integer
%e or %E	Scientific notation of floats
%f	Float values
%g or %G	Similar as %e or %E
%hi	Signed integer (short)
%hu	Unsigned Integer (short)
%i	Unsigned integer
%l or %ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or %lld	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

Flags: Format Identifier modifiers

```
-      Left justify.
0      Field is padded with 0's instead of blanks.
+      Sign of number always O/P.
blank  Positive values begin with a blank.
#      Various uses:
       %#o (Octal) 0 prefix inserted.
       %#x (Hex)   0x prefix added to non-zero values.
       %#X (Hex)   0X prefix added to non-zero values.
       %#e         Always show the decimal point.
       %#E         Always show the decimal point.
       %#f         Always show the decimal point.
       %#g         Always show the decimal point trailing
                   zeros not removed.
       %#G         Always show the decimal point trailing
                   zeros not removed.
```

Examples:

```
printf(" %-10d \n", number);
printf(" %010d \n", number);
printf(" %-#10x \n", number);
printf(" %#x \n", number);
```

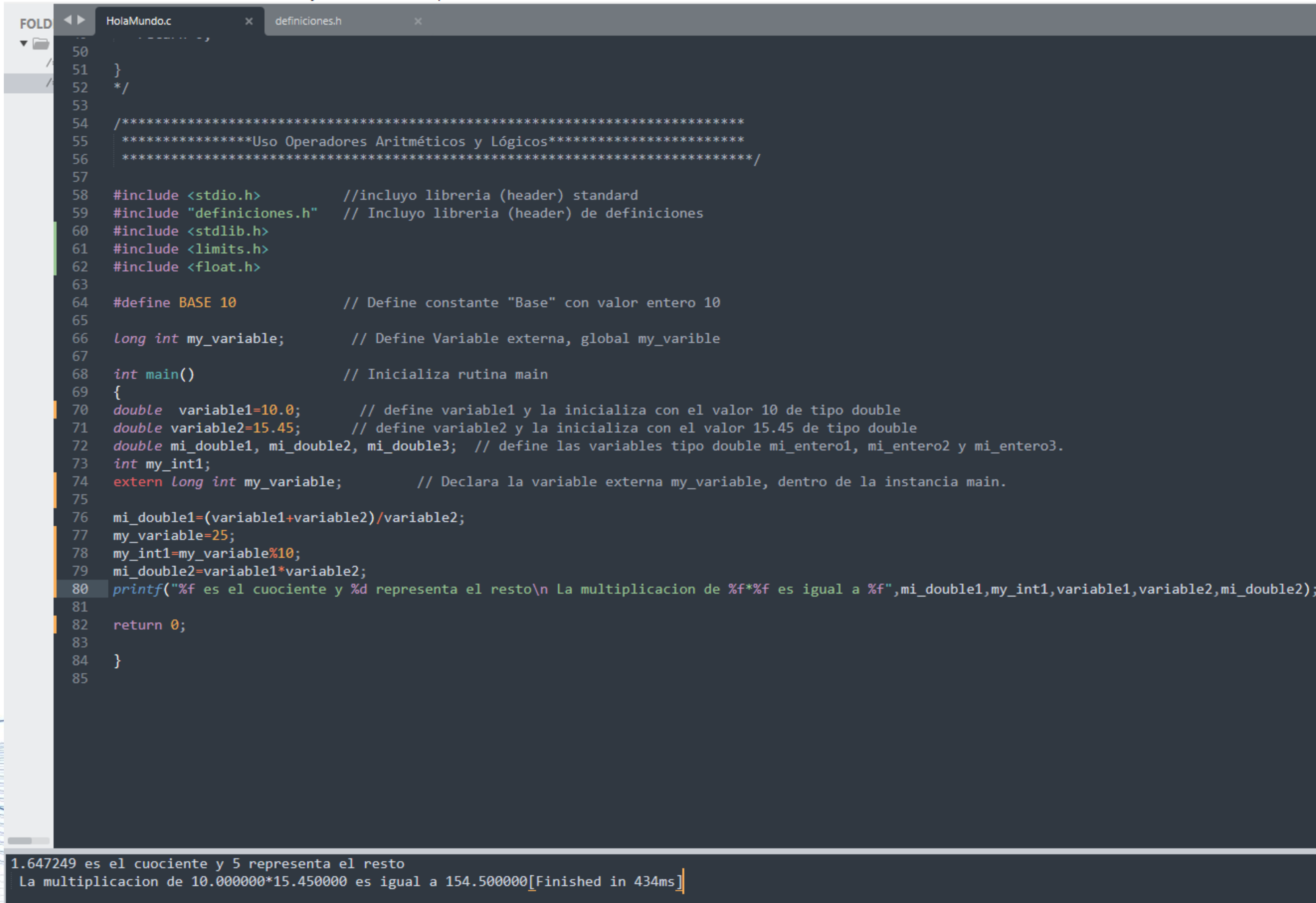
More details here

Basic Concepts

Operators - Arithmetic

The arithmetic operators are: -
addition **+** ,
subtraction **-** ,
multiplication *****
division **/**.
Increment **++**
decrement **--**

modulus **%,**
Provides the remainder of a division.
% is only applied to integer values.



```

}
*/

/***************************************************************************
 ***************Uso Operadores Aritméticos y Lógicos***********************
 ****************************************************************************/

#include <stdio.h>          //incluyo libreria (header) standard
#include "definiciones.h"   // Incluyo libreria (header) de definiciones
#include <stdlib.h>
#include <limits.h>
#include <float.h>

#define BASE 10             // Define constante "Base" con valor entero 10

long int my_variable;        // Define Variable externa, global my_varible

int main()                  // Inicializa rutina main
{
double  variable1=10.0;       // define variable1 y la inicializa con el valor 10 de tipo double
double variable2=15.45;      // define variable2 y la inicializa con el valor 15.45 de tipo double
double mi_double1, mi_double2, mi_double3;  // define las variables tipo double mi_entero1, mi_entero2 y mi_entero3.
int my_int1;
extern long int my_variable;        // Declara la variable externa my_variable, dentro de la instancia main.

mi_double1=(variable1+variable2)/variable2;
my_variable=25;
my_int1=my_variable%10;
mi_double2=variable1*variable2;
printf("%f es el cuociente y %d representa el resto\n La multiplicacion de %f*%f es igual a %f",mi_double1,my_int1,variable1,variable2,mi_double2);

return 0;

}

```

```
1.647249 es el cuociente y 5 representa el resto
 La multiplicacion de 10.000000*15.450000 es igual a 154.500000[Finished in 434ms]
```

Basic Concepts

Operators- Logical

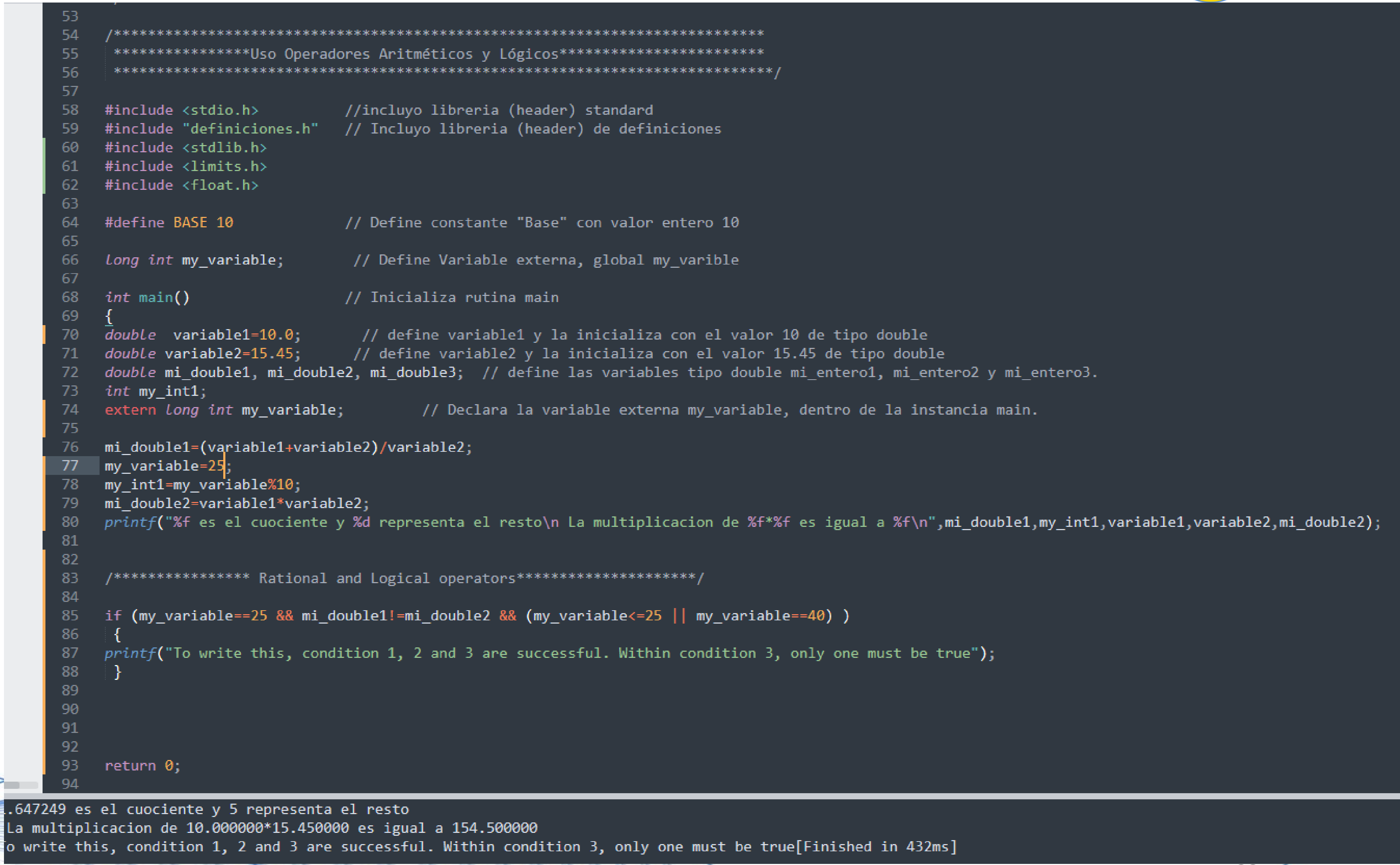
The logical operators are:

Relational:

>	Greater than
<	Smaller than
>=	Greater or equal than
<=	Smaller or equal than
==	Equal to
!=	Different to

Logical:

&& :	AND operator
II :	OR operator
!:	NOT operator

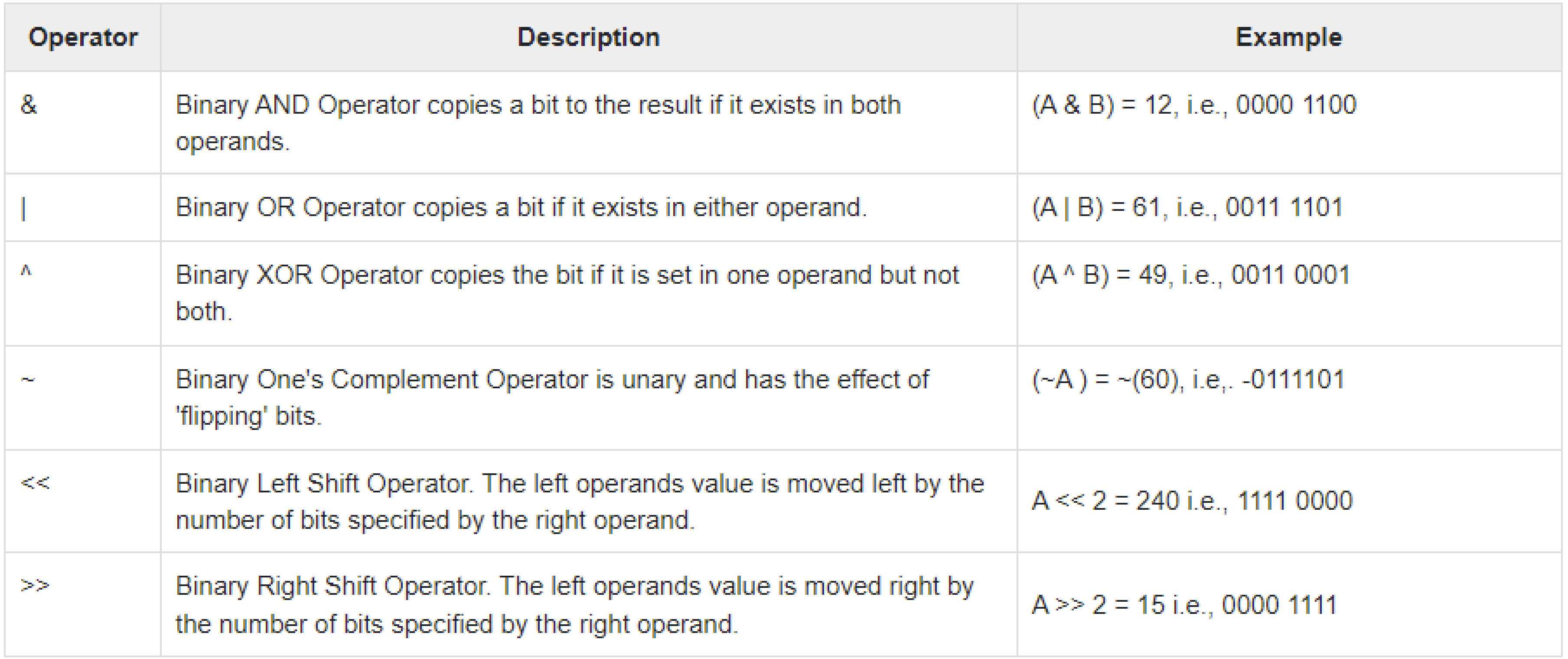

```
/***************************************************************************
 ****************Uso Operadores Aritméticos y Lógicos************************
 ***************************************************************************/

#include <stdio.h>          //incluyo libreria (header) standard
#include "definiciones.h"   // Incluyo libreria (header) de definiciones
#include <stdlib.h>
#include <limits.h>
#include <float.h>

#define BASE 10             // Define constante "Base" con valor entero 10

long int my_variable;        // Define Variable externa, global my_varible

int main()                  // Inicializa rutina main
{
double  variable1=10.0;       // define variable1 y la inicializa con el valor 10 de tipo double
double variable2=15.45;      // define variable2 y la inicializa con el valor 15.45 de tipo double
double mi_double1, mi_double2, mi_double3;  // define las variables tipo double mi_entero1, mi_entero2 y mi_entero3.
int my_int1;
extern long int my_variable;         // Declara la variable externa my_variable, dentro de la instancia main.

mi_double1=(variable1+variable2)/variable2;
my_variable=25;
my_int1=my_variable%10;
mi_double2=variable1*variable2;
printf("%f es el cuociente y %d representa el resto\n La multiplicacion de %f*%f es igual a %f\n",mi_double1,my_int1,variable1,variable2,mi_double2);


/**************** Rational and Logical operators*********************/

if (my_variable==25 && mi_double1!=mi_double2 && (my_variable<=25 || my_variable==40) )
 {
printf("To write this, condition 1, 2 and 3 are successful. Within condition 3, only one must be true");
 }




return 0;

```

```
.647249 es el cuociente y 5 representa el resto
La multiplicacion de 10.000000*15.450000 es igual a 154.500000
o write this, condition 1, 2 and 3 are successful. Within condition 3, only one must be true[Finished in 432ms]
```

Basic Concepts

Operator - Bitwise

Assume A = 60 and B = 13 in binary format, they are:
A = 0011 1100
B = 0000 1101

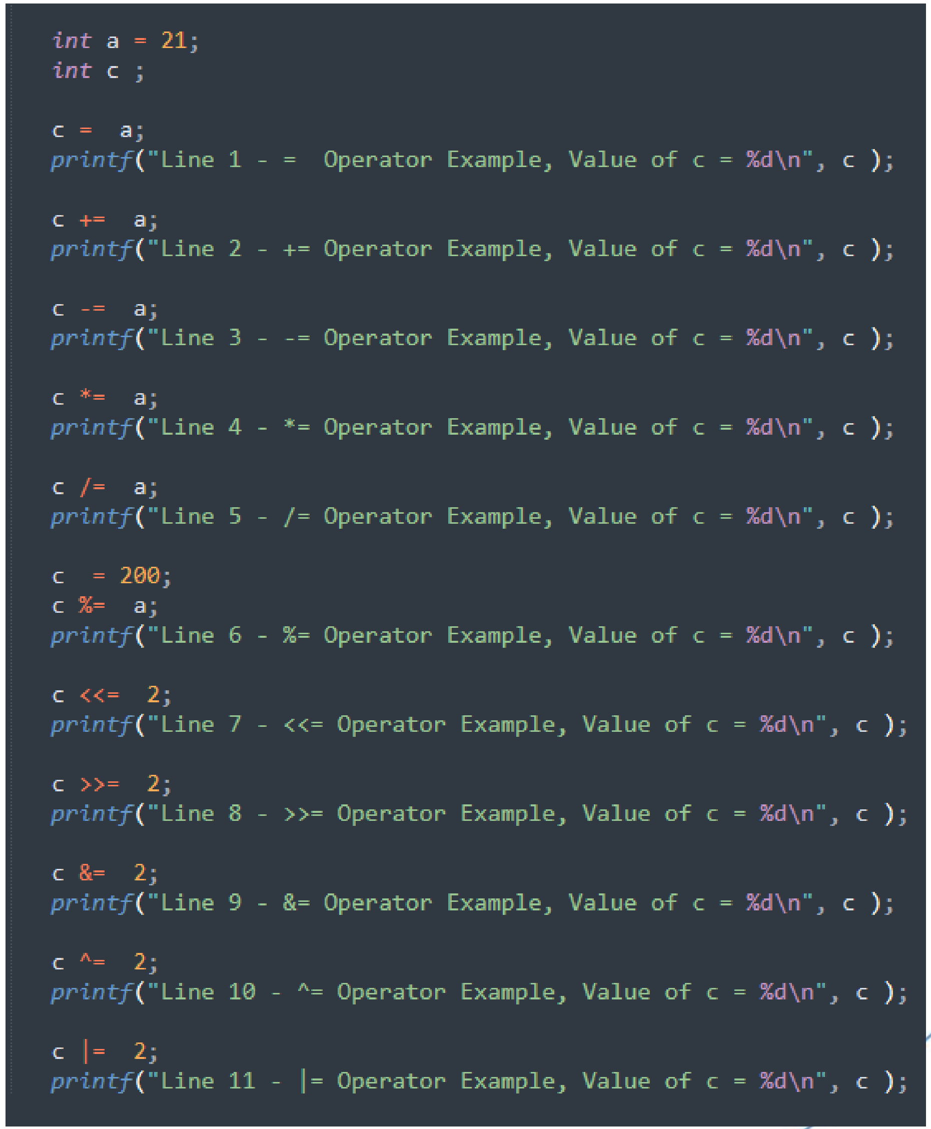


Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Basic Concepts

Operator-Assignment

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2



```
int a = 21;
int c ;

c =  a;
printf("Line 1 - =  Operator Example, Value of c = %d\n", c );

c +=  a;
printf("Line 2 - += Operator Example, Value of c = %d\n", c );

c -=  a;
printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

c *=  a;
printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

c /=  a;
printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

c  = 200;
c %=  a;
printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

c <<=  2;
printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

c >>=  2;
printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );

c &=  2;
printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

c ^=  2;
printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );

c |=  2;
printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
```

Basic Concepts

Operators - Miscelaneous

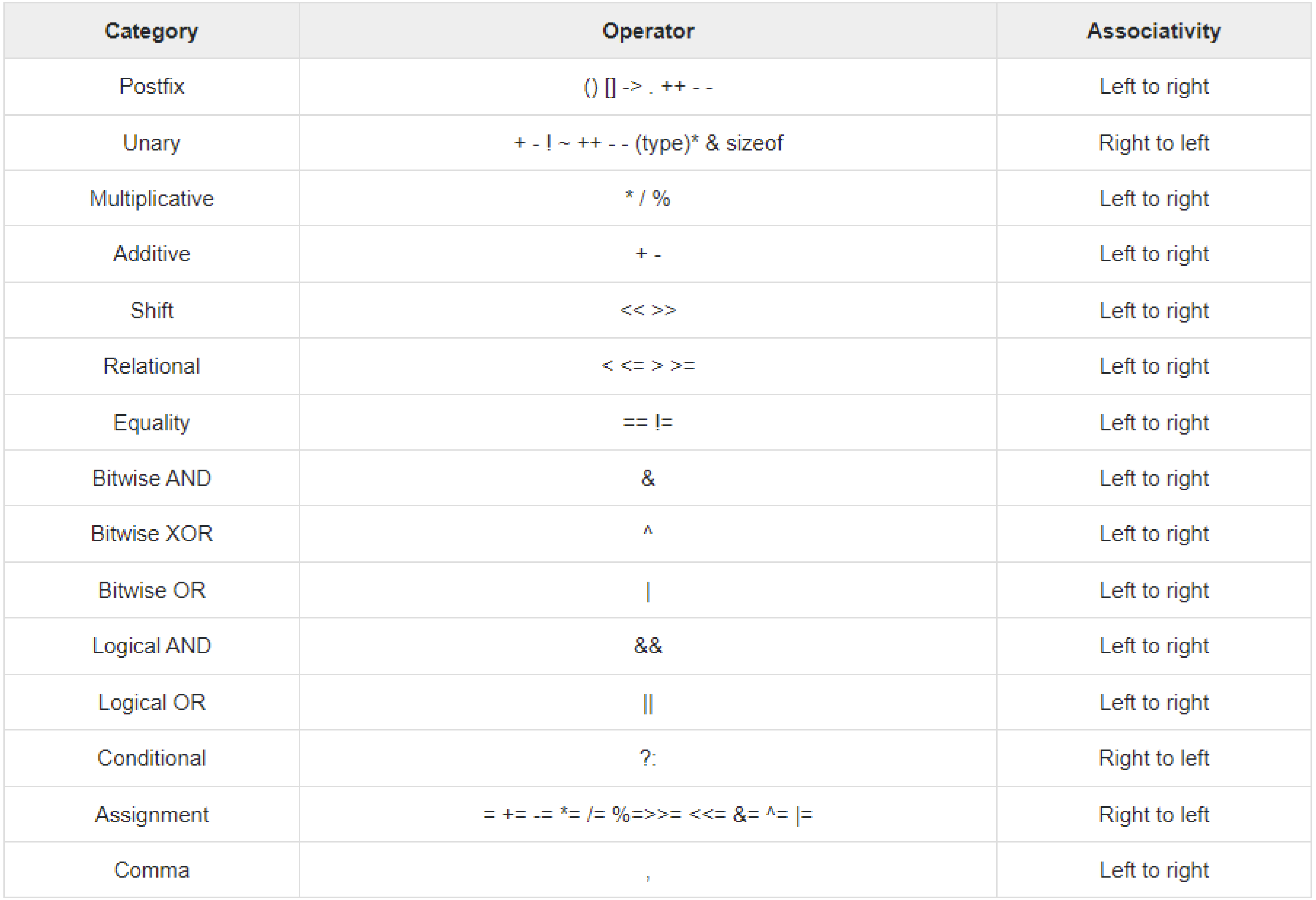
Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

```
main() {

   int a = 4;
   short b;
   double c;
   int* ptr;

   /* example of sizeof operator */
   printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
   printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
   printf("Line 3 - Size of variable c= %d\n", sizeof(c) );

   /* example of & and * operators */
   ptr = &a;    /* 'ptr' now contains the address of 'a'*/
   printf("value of a is  %d\n", a);
   printf("*ptr is %d.\n", *ptr);

   /* example of ternary operator */
   a = 10;
   b = (a == 1) ? 20: 30;
   printf( "Value of b is %d\n", b );

   b = (a == 10) ? 20: 30;
   printf( "Value of b is %d\n", b );
}
```

```
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c= 8
value of a is  4
*ptr is 4.
Value of b is 30
Value of b is 20
```

Basic Concepts

Operators – Precedence in C



Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

This defines the priority of an operator over other. For instance, in:
X=2+6*8;

Multiplication has priority over addition; therefore, first multiplication is achieved and then addition. This might be obvious for us, but it is not for the compiler.

Priority is ordered from top to bottom of the table in this slide.

Associativity: Direction in which the expression is evaluated. E.g.

A=b; The value of b is given to A. (R to L)

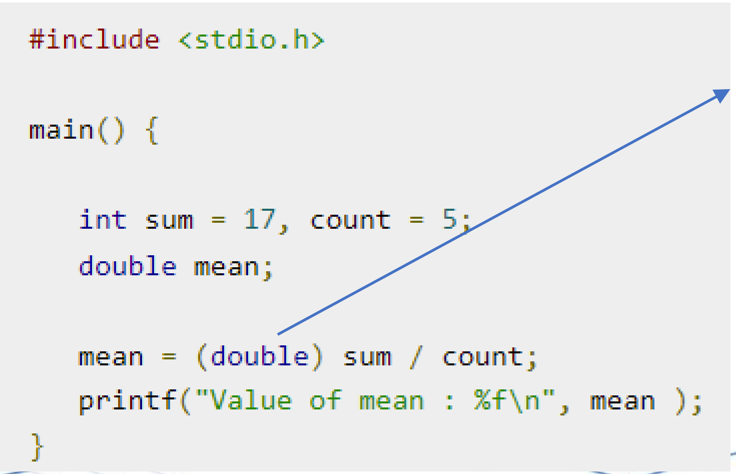
1==2!=3; first 1==2 is executed. Equivalent to **(1==2) != 3.**
(Note that == and =! Have same priority)

Basic Concepts

Type Conversions - Casting

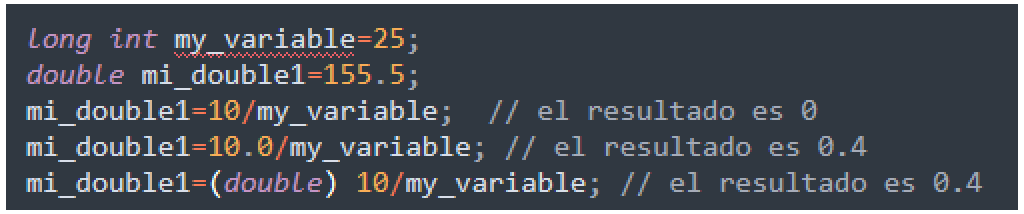
- Occurs when applying an operation to different types of variables.
  - Example **int variable1; float variable2;** What is variable1+variable2?
- If no data is losing, the conversion is automatic. In previous example result is float and integer is added as float. (from narrower to wider variable)
- In case information can be loss, then a warning message is given by compiler. Only warning!
- To convert one datatype into another is known as ***casting.*** For doing this we use the cast operator as:

(type_name) expression   e.g. To converter the int variable to double, we do: (double) variable



```
#include <stdio.h>

main() {

   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

This casting allow that the results of the division of two integers is made as a floating-point division, and the result stored in "mean" is 3.4. Otherwise, it would be 3. As cast has precedence over division, first "sum" is converted to double and then divided.



```
long int my_variable=25;
double mi_double1=155.5;
mi_double1=10/my_variable;  // el resultado es 0
mi_double1=10.0/my_variable; // el resultado es 0.4
mi_double1=(double) 10/my_variable; // el resultado es 0.4
```

Be careful when you use numbers!
Writing 10 is not the same as 10.0!

Basic Concepts

Type Conversions - Casting

- The compiler performs automatic promotion of variables (upgrade somehow) to make them match in type.
- The first promotion is known as “integer promotion” by which a char in converted into int.
- Then, the following rule is used to promote the variable to the next highest level:

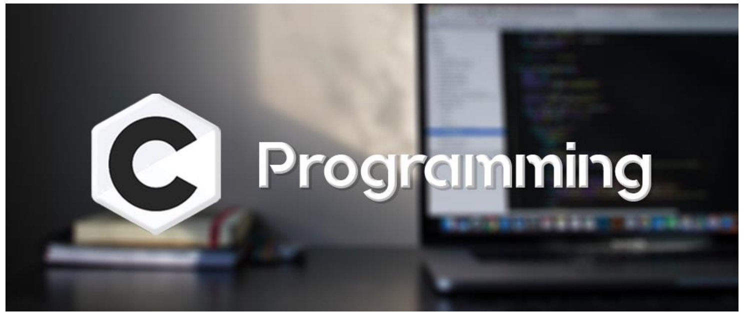
$$\begin{tikzcd}
\text{long double} \\
\text{double} \arrow[u] \\
\text{float} \arrow[u] \\
\text{unsigned long long} \arrow[u] \\
\text{long long} \arrow[u] \\
\text{unsigned long} \arrow[u] \\
\text{long} \arrow[u] \\
\text{unsigned int} \arrow[u] \\
\text{int} \arrow[u]
\end{tikzcd}$$

```c
#include <stdio.h>

main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum );
}
```

Is *sum* equal to 116?
Process of sum=i+c:
First the char c is converted to integer
Then i and c are added
The result in converted to float.
Result is 116.000000

Control Flow
C Programming Language

If, else, switch, case, for, while.

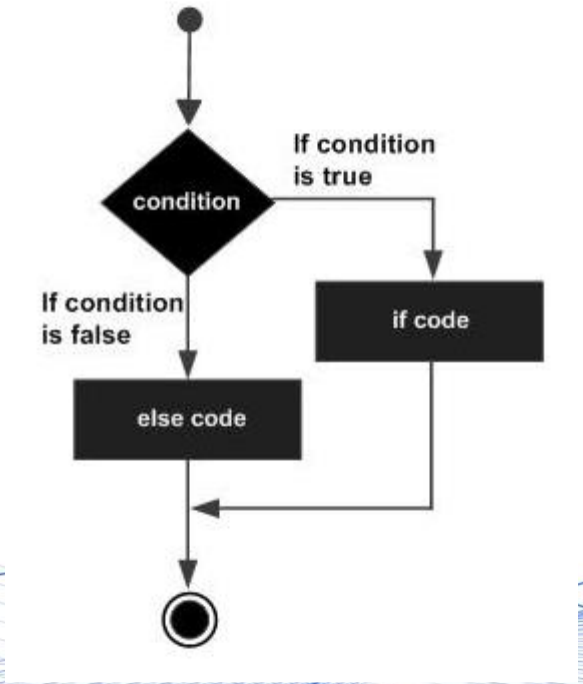


Control Flow

Decision Making – “if-else” and “switch”

- **It requires conditions to be evaluated.**
- **It requires statements to be executed in case of this condition is true and (optionally) in case it is false.**
  - Note: A statement is an expression (e.g x=0 or printf(…) or any other) followed by ;

- C assumes that any **non-zero** or **nun-null** value is **true**.
- C assumes that **zero** or **null** is **false**.



Simple if-else condition:

```
if(boolean_expression 1) {
/* Executes when the boolean expression 1 is true */
 }
else {
/* executes when the none of the above condition is true */
}
```

Note: else part is optional

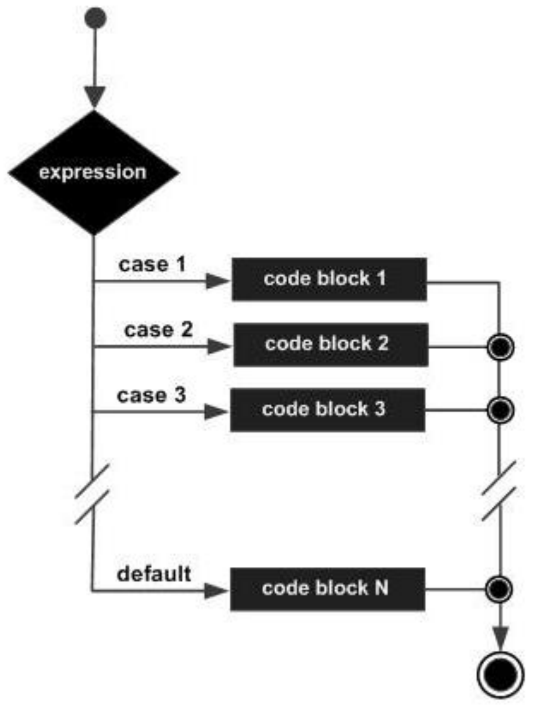
Multiple if-else condition:

```
if(boolean_expression 1) {
/* Executes when the boolean expression 1 is true */
 }
else if( boolean_expression 2) {
 /* Executes when the boolean expression 2 is true */
}
 else if( boolean_expression 3) {
 /* Executes when the boolean expression 3 is true */
}
else {
/* executes when the none of the above condition is true */
}
```

Control Flow

Decision Making – “if-else” and “switch”

- The **switch** expression evaluates an integral (very common) or enumerated (not that common) type of variable.
- You can have as many cases as you want. The program enter to the *case*, which contant-expression matches the switch expression.
- The *case constant-expression* must be same type as switch expression. Must be constant or literal.
- Using the **break** makes that the program-counter jumps to the next instruction and the switch is terminated.
- **No using break** after finishing a *case* makes that the program-counter checks all next *cases* after executing one.
- **Default** *case* can be used to perform a task in case none of the cases is true.

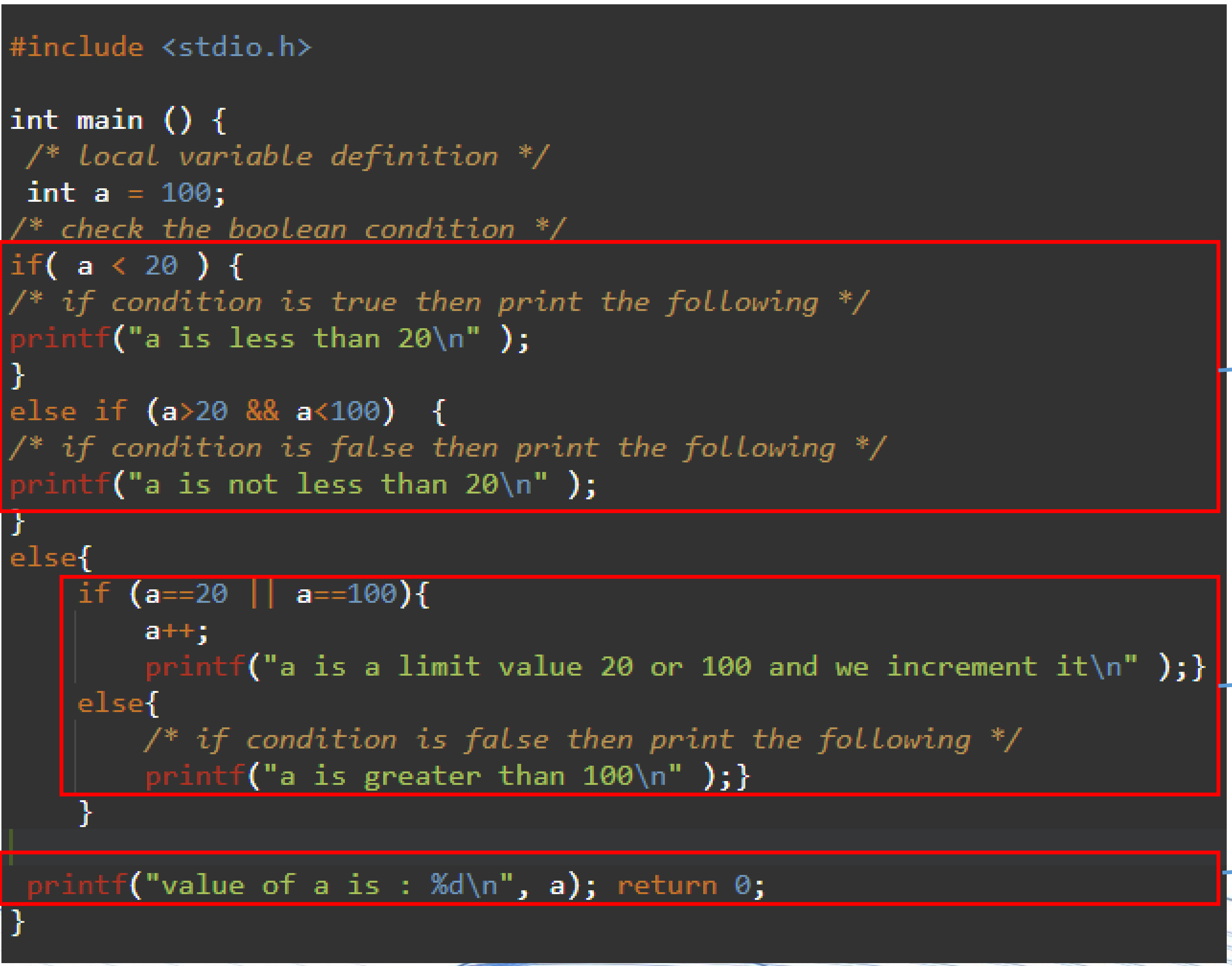


Switch condition

```
switch(expression) {
case constant-expression : statement(s);
break; /* optional */
case constant-expression : statement(s);
break; /* optional */
/* you can have any number of case statements */
default : /* Optional */
statement(s); }
```

Control Flow

Decision Making – “if-else” and “switch”-Examples



```c
#include <stdio.h>

int main () {
 /* local variable definition */
 int a = 100;
/* check the boolean condition */
if( a < 20 ) {
/* if condition is true then print the following */
printf("a is less than 20\n" );
}
else if (a>20 && a<100)  {
/* if condition is false then print the following */
printf("a is not less than 20\n" );
}
else{
    if (a==20 || a==100){
        a++;
        printf("a is a limit value 20 or 100 and we increment it\n" );}
    else{
        /* if condition is false then print the following */
        printf("a is greater than 100\n" );}
    }

 printf("value of a is : %d\n", a); return 0;
}
```

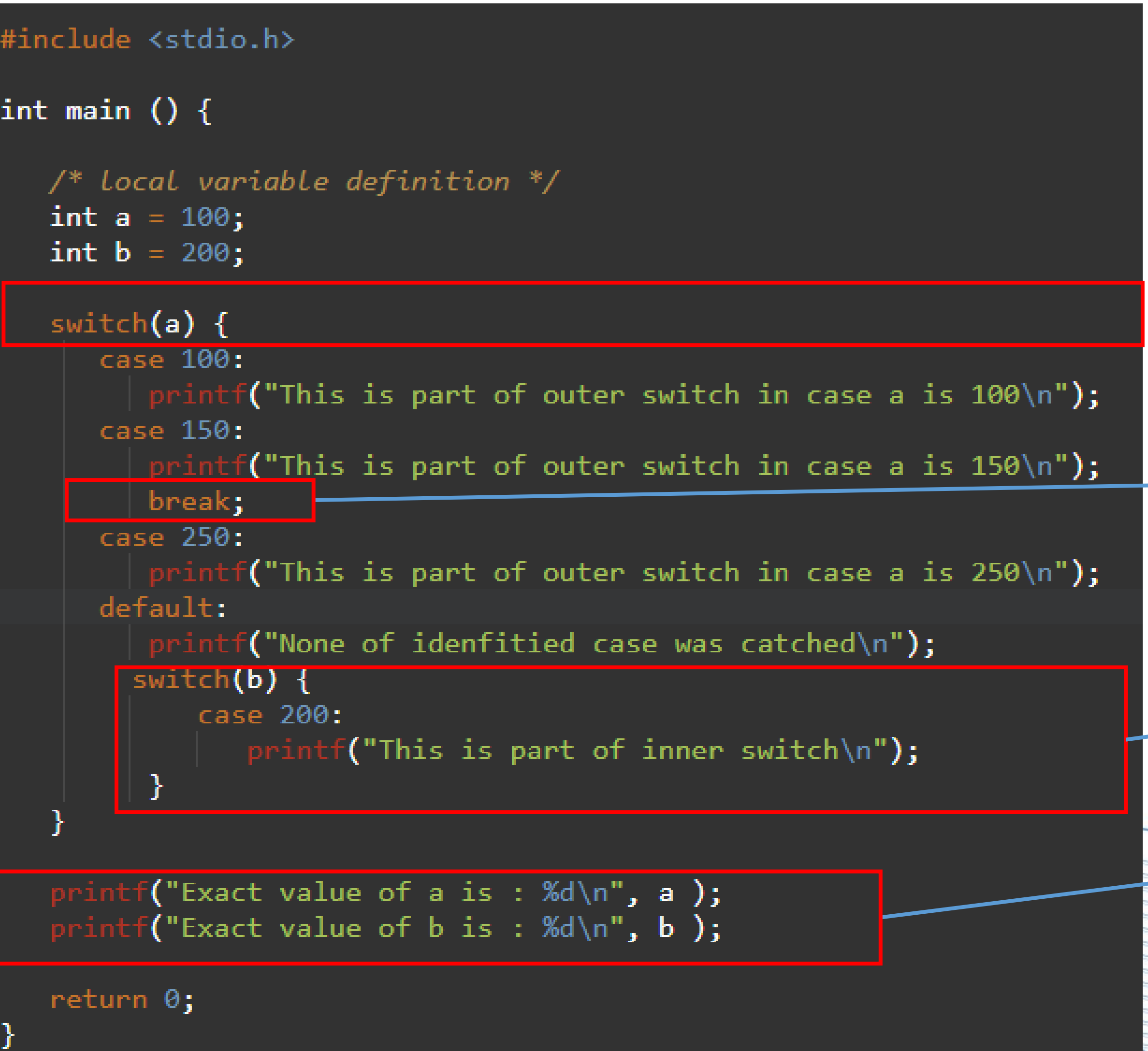
Use of if and else if conditions

This is a nested if.
Make any difference remove this else sentence?

If a=100, the printed value is **101**.

Control Flow

Decision Making – “if-else” and “switch”-Examples



```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   switch(a) {
      case 100:
         printf("This is part of outer switch in case a is 100\n");
      case 150:
         printf("This is part of outer switch in case a is 150\n");
         break;
      case 250:
         printf("This is part of outer switch in case a is 250\n");
      default:
         printf("None of idenfitied case was catched\n");
        switch(b) {
            case 200:
               printf("This is part of inner switch\n");
         }
   }

   printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );

   return 0;
}
```

Main switch

What difference makes this break?

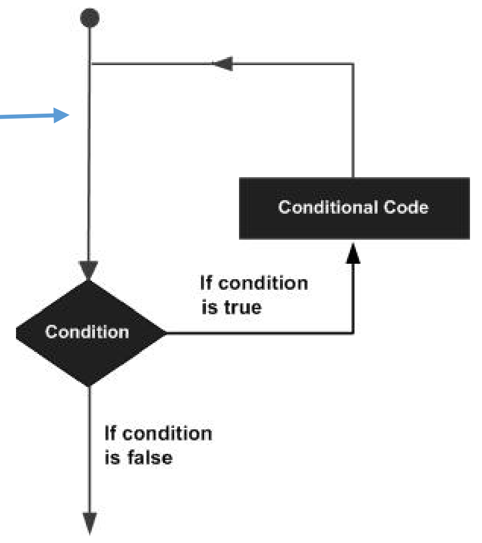
This is a nested switch.
If none of the cases for “a” is caught, then we ask for switch b.

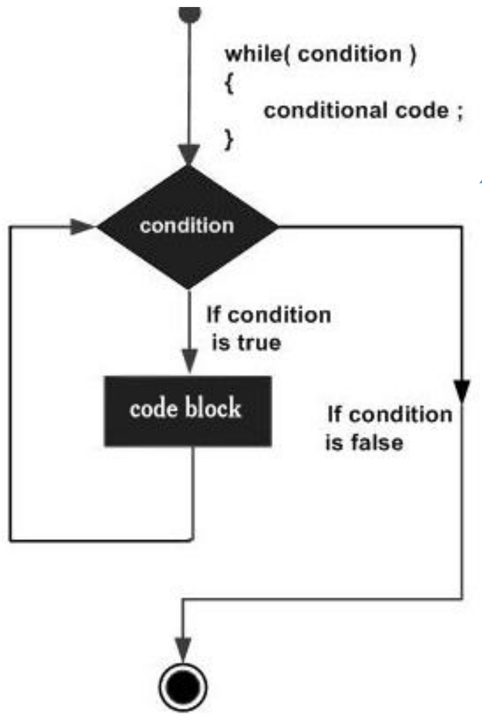
If a=100, the printed value is **101**.

Control Flow

Loops Structure– “while” and “for”

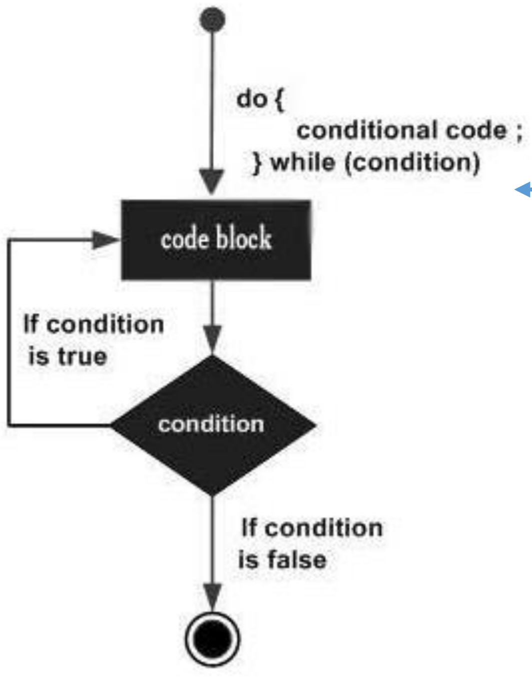
- Loops are useful to execute a piece of code several times.
- C provides **while** , **do…while** and **for** types of loops to handle loops.
- **while** repeatedly executes statements if a conditions is **true** (non-zero non-null).
- **do.. While** condition checks the condition at the end of the loop.
- A **do..while** executes at he statements at least once!. Then check for keep repeating.





Syntax for **While:**

while(condition) {
 statement(s); }



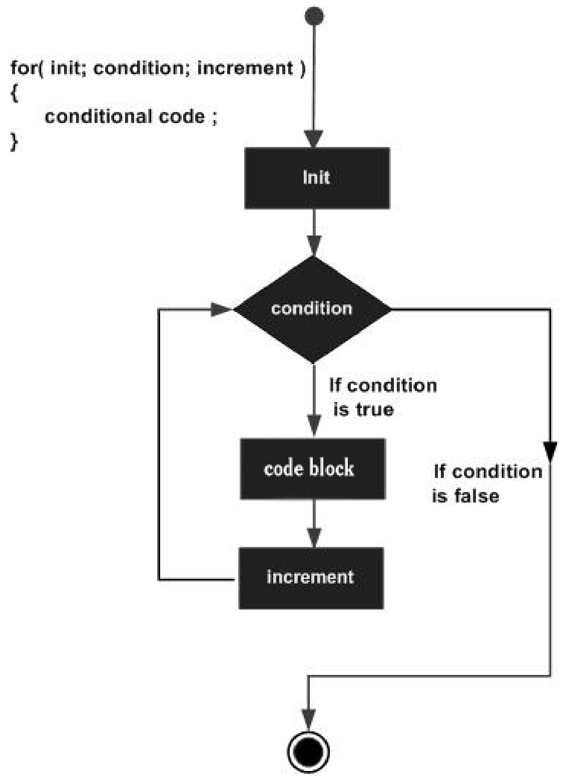
Syntax for **do while:**

do { statement(s); }
while(condition);

Control Flow

Decision Making – “while” and “for”

- **for** loop is a very useful and implemented control structure.
- It executes a block of code a specific number of times. Execution steps are:
    - The **init** step is executed first and once!. You can declare and initialize any loop control variable.
    - Then Condition is evaluated. If true, statements are executed. If false end of **for** loop.
    - After statements are executed, the **increment** statement allow you to update the control variables.
    - After **incrementing** the control variables, the **condition** is evaluated again, and loop is repeated.



The **syntaxis** of **for:**

```
for ( init; condition; increment )
{ statement(s);
 }
```

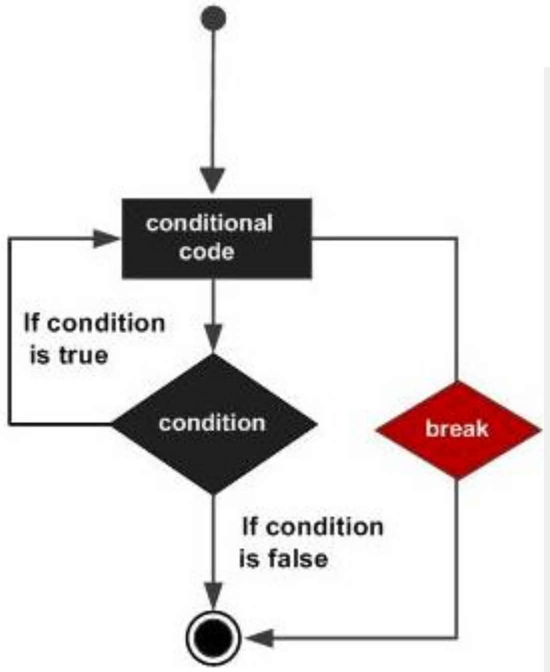
```
#include <stdio.h>

int main () {

   int a;

   /* for loop execution */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }

   return 0;
}
```

Note: init and increment are optional. Try not writting them and see what happend. Can I make a kind of while forever?

Control Flow

Loops Structure– Control Statements

- Loops control statements change the normal sequence of execution of out loop structures
- **C provides the three control statements (break, continue, goto):**
  - **break:** Terminates the loop or switch and program continue with next instruction

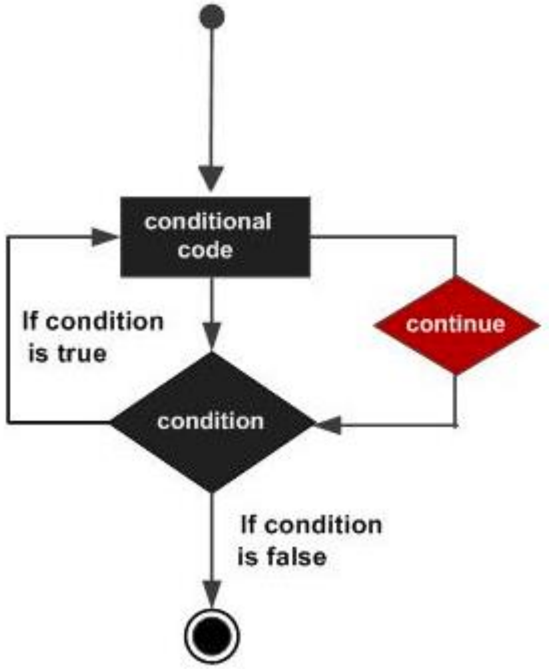


```
#include <stdio.h>
 int main () {
int a = 10;

 /* while loop execution */
while( a < 20 ) {
 printf("value of a: %d\n", a);
 a++;
 if( a > 15) {
 /* terminate the loop using break statement */
 break; }
 } return 0; }
```

Control Flow

Loops Structure– Control Statements

- Loops control statements change the normal sequence of execution of out loop structures
- **C provides the three control statements (break, continue, goto):**
    - **continue:** the loop immediately retest its condition, skipping the remainder of its body.



```c
#include <stdio.h>

int main () {

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

   /* do loop execution */
   do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         continue;
      }

      printf("value of a: %d\n", a);
      a++;

   } while( a < 20 );

   return 0;
}
```

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

Number 15 is skipped!

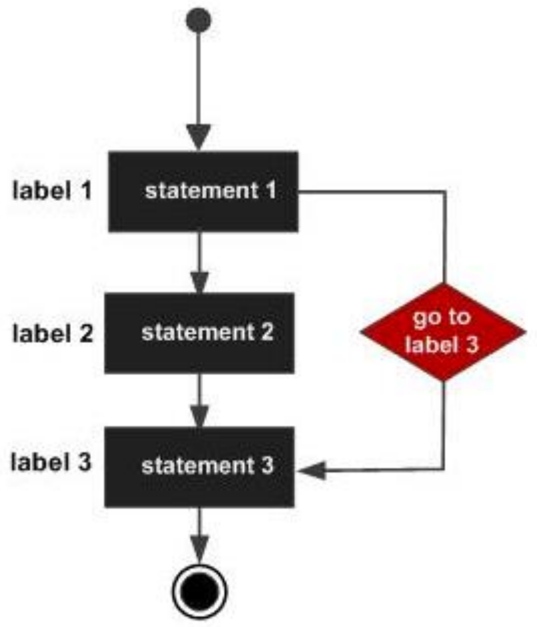
Note that:
For the **for** loop, **continue** statement causes the **conditional** test and **increment** portions of the loop to execute.

For the **while** and **do...while** loops, continue statement causes the program control to pass to the **conditional** tests.

Control Flow

Loops Structure– Control Statements

- Loops control statements change the nomal sequence of execution of out loop structures
- **C provides the three control statements (break, continue, goto):**
  - **goto:** redirect the execution of statements to the labeled statement.



```
goto label;
 .. .
label: statement;
```

```
#include <stdio.h>
int main () {
/* local variable definition */
int a = 10;
/* do loop execution */
 LOOP: do {
if( a == 15) {
 /* skip the iteration */
 a = a + 1;
 goto LOOP; }

 printf("value of a: %d\n", a);
 a++;
}
while( a < 20 );

 return 0; }
```

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

Number 15 is skipped

Control Flow

Excersice

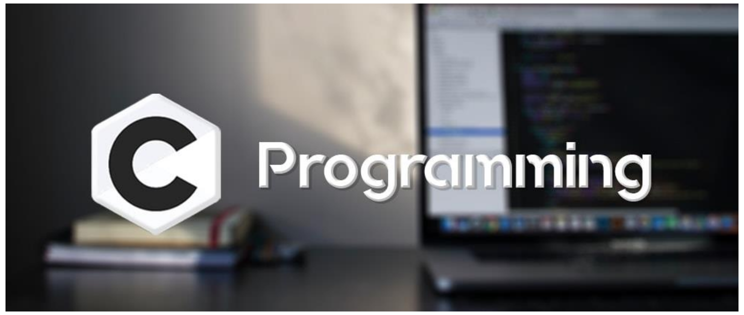
1. Create a program that read one character from console and:

   i) Transform the character to lower case (in case it is upper case) and transform it in upper case in case it is lower case. (read and write the characters from console).
   ii) The system should write in console “this is not a valid character” and ask for a new character in case you write a character which is not a-z or A-Z
   iii) Use switch to identify if the character written by user is S , s , E, e or P, p . In this *case* write into console “the letter XX is a special character!”
   iv) The program should always ask for a new character unless the word END is written. Which finalizes the program.
   v) Recognize if a user write the hidden sequence of characters CONF. Thereby, if the user write C, you should ask for a new second letter, otherwise you say “you couldn’t find the hidden word. Best luck for next time”. Use nested structure to do this task. If the user wrote CONF, you write the message “you are in configuration mode”
   vi) Modify your program using **while**, so that only 10 characters can be incorporated.
   vii) The system now only accepts the keyword CONF. In this case, you can enter a number which restart the number of characters you can enter now!
   viii) GO!

HINT: characters are numbers in ASCII code!
Note validate the use of cast and automatic integer promotion.
Check the functions prinft and scanf from the stdio.h library

Functions
C Programming Language

Definition, declaration, call and arguments of a function.



Functions

Overview

- A function is a **group of statements**. The most elemental function in C is the **main** function.
- A **function** usually written to perform a specific task.
- **Function declaration:** This tells the compiler the functions **name, return type** and **parameters**.
- **Function definition:** is the body of the function (the group of statements).

- A library is nothing but a group of functinos. Usually all functions are related to the same topic.

- The **stdio.h library** contains functions as **strcat()**, which concatenate two strings and **memcpy()** to copy one memory location to another location.

Functions
Function Definition

```
return_type function_name( parameter list ) {

body of the function

}
```

Function header

Return Type: A function may return a value. The *return_type* is the data type of the value the function returns. Return_type is the keyword ***void*** is nothing is returned.

Function Name: Name chosen by the user. This name is later used to invoke the function.

Parameters: The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters, **void** is used. These parameters acts as local variables for the function.

Function Body: The function body contains a collection of statements that define what the function does.

Functions
Function Declaration

A **function declaration** inform the compiler about the function that will be used in the main function.

Function declaration is required when you define the function in a different file where you call (invoke) it.

To declare a function just write:

```
return_type function_name( parameter list );
```

For example, a function named "max" which return an int and takes as parameters two integer named num1 and num2 shall be declared as:

```
int max(int num1, int num2);
```

Functions

Function Calling

To use a function, we have to call it or make use of it. See the example:

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);

int main () {
/* local variable definition */
int a = 100;
int b = 200;
int ret;
/* calling a function to get max value */
ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0; }

 /* function returning the max between two numbers */
 int max(int num1, int num2) {
 /* local variable declaration */
int result;
 if (num1 > num2) result = num1;
 else result = num2;
return result;
```

Function declaration

Function call (or use)

Function definition

Note: As we defined the function ***max*** in the same file as the main function, the declaration could be neglected, because an **implicit declaration** is made (you will get a warning from the compiler). However, a good practice is always to declare it first.

If ***max*** is in a different file, you must declare the function.

Functions

Function Arguments

Call by Value: When we call a function and provide the function with an argument, the function copy the value of the argument into its own local variables. All changes made to this value, provided by the argument, does not change the value of the argument. This is known as calling a function by value **call by value.**

```
#include <stdio.h>
/* function declaration */
 void swap(int x, int y);

 int main () {
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
 printf("Before swap, value of b : %d\n", b );
 /* calling a function to swap the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
 printf("After swap, value of b : %d\n", b );
return 0; }
void swap(int x, int y) {
 int temp;
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put temp into y */
return; }
```

After calling the function swap, the local variable **x** takes the value **100** and **y** takes **200**. Variables **a** and **b** are **never touched**. Therefore, after execution we should see:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

Functions

Function Arguments

Call by Reference: Instead of transferring the value of an argument to the local variable of a function, we can give as parameter of a function the memory **address of the argument.** This address is used inside the function to look for the value of the argument. It means that changes are made direct to the argument itself. This is known as **value by reference.**

```
#include <stdio.h>
/* function declaration */
 void swap(int x, int y);

 int main () {
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
 printf("Before swap, value of b : %d\n", b );
 /* calling a function to swap the values */
swap(&a,&b);
printf("After swap, value of a : %d\n", a );
 printf("After swap, value of b : %d\n", b );
return 0; }
void swap(int *x, int *y) {
 int temp;
temp = *x; /* save the value of x */
*x = *y; /* put y into x */
*y = temp; /* put temp into y */
return; }
```

&a returns the memory **address** of the variable **a.** We provide the memory addresses of variables a and b to the function.

***x** returns the value stored in the memory address **x.** Thereby:

temp = *x; saves in temp the value of a, i.e. 100.

*x=*y: saves the content of the memory address y into the memory address x. This is known as **pointers**.

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Functions

Scope Rules

A **scope** of a variable define a region where it can exist and can be access.

There are only three places where a variable can be defined:

- Inside a function -> **local variable**
- Outside of all functions -> **global variables**
- In the definition of a function parameters -> **formal parameter**

Local Variables:

- Not known to functions outside their own.
- Can be Access and used only by statements inside the function.
- It is not initialized automatically. Be careful.

Global Variables:

- Usually defined at the top of the program.
- Can be Access inside any function defined in the program.
- It can occur that a local and a global variable have the same name. In this case, local variable prevail.
- Initialized automatically as zero see table:

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

Formal Parameters:

- Taken as local variables within a function.
- Have precedence over global variables.

Functions

Recursion

It is the process **of calling a function inside the same function**. It is supported by C.

```
void recursion() {
 recursion(); /* function calls
itself */
}

int main() {
recursion();
}
```

To avoid an infinite loop, it is required to include an exit condition, under certain condition.

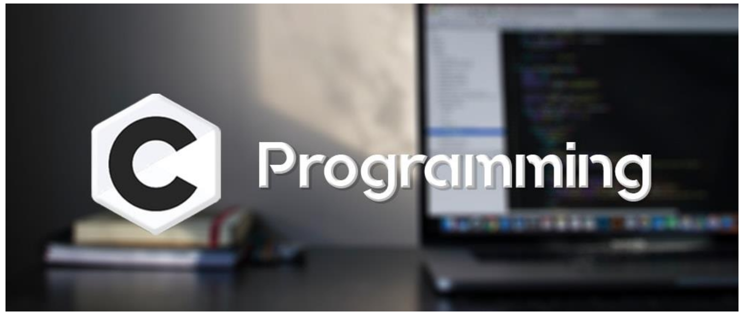
```
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {
 if(i <= 1) {
 return 1;
}
return i * factorial(i - 1);
}

 int main() {
 int i = 12;
printf("Factorial of %d is %d\n", i, factorial(i));
return 0;
 }
```

Which is the exist condition?

What should return this function?

Macro Definitions
Preprocessors and Header Files

Definition, operators, header files.



Program Structure

Preprocessors

- **Preprocessor** is simply a **text substitution** tool. For C, C-Preprocessor is known as **CPP.**
- This tool ask the compiler to do this text replacement before the compilation starts.
- **Therefore, preprocessor directives are not part of the compiler!.**
- All **preprocessor commands start** with **#.**
- With **preprocessor commands** we define **macros**.

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

CPP tells the compiler to:

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

Copy the text of the *system header **stdio.h*** and the *user defined* file ***myheader.h*** into the **source file.**

```
#define MAX_ARRAY_LENGTH 20
```

Replace MAX_ARRAY_LENGTH with 20.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

Undefine the existing FILE_SIZE and **define** it to 42.

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

Define MESSAGE as “you wish!”, only if **is not defined** yet.

```
int main()
{

#ifdef MAX_ARRAY_LENGTH
   /* Your debugging statements here */
    printf("My constante es _%d", MAX_ARRAY_LENGTH);
#endif
```

If MAX_ARRAY_LENGTH is defined, we execute a piece of code. **#ifdef…#endif is written within the main.**

Program Structure

Predefined Macros

- **A Macros** is a simply **#define** but it can make logic decisions or arithmetic functions.
- For example:

```
#define WRONG(A) A*A*A       /* fails for A=2+3 */
#define CUBE(A) (A)*(A)*(A) /* Do not fail for A=(2+3)*/
#define SQUR(A) (A)*(A)      /* Correct Macro */
#define MAX(A,B)  ((A)>(B)?(A):(B))  /* Macro for Max between two numbers*/
#define MIN(A,B)  ((A)>(B)?(B):(A))  /* Macro for Min between two numbers */
```

Macros do not define type of variables; macros simply replace data to program in a more elegant form. **Known as Parameterized Macro**

Predefined Macros:

Sr.No.	Macro & Description
1	**__DATE__** The current date as a character literal in "MMM DD YYYY" format.
2	**__TIME__** The current time as a character literal in "HH:MM:SS" format.
3	**__FILE__** This contains the current filename as a string literal.
4	**__LINE__** This contains the current line number as a decimal constant.
5	**__STDC__** Defined as 1 when the compiler complies with the ANSI standard.

Using Macros:

```
int main() {

int index,mn,mx;

int count = 5;

    mx = MAX(index,count);

    mn = MIN(index,count);
    printf("Max es %d y min es %d\n",mx,mn);
// Macros defined by the system
    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );

}
```

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Program Structure

Preprocessor Operators - Help to create complex Macros

- Macro **Continuation** Operator \:

```
#define message_for(a, b)  \
 printf(#a " and " #b ": Practice Programming in C!\n")
```

- **Stringsize** Operator **#**: Converts a macro parameter into a string constant.

```
#include <stdio.h>
#define message_for(a, b)  \
 printf(#a " and " #b ": Practice Programming in C!\n")

 int main(void) {
 message_for(Juan, Ana);
return 0; }
```

```
Juan and Ana: Practice Programming in C!
```

- **Token Pasting** Operator **##**: Convert two tokens into a single token.

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
 int token34 = 40;
tokenpaster(34);
return 0;
 }
```

```
token34 = 40
```

Macro print the value of *token##n,* where n is the input tokenpaster. In this case the variable token##n is token34, which has the value of 40.

Program Structure

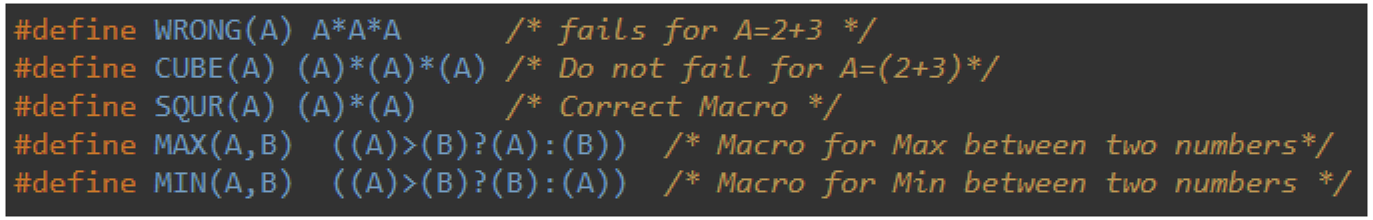
Preprocessor Operators

- **Defined** Operator **defined()**: check if an identifier, constant expression, has been defined with #define. Result is true or false.

```
#include <stdio.h>
#if !defined (MESSAGE)
#define MESSAGE “We define it now!" #endif
int main(void) {
 printf("Here is the message: %s\n", MESSAGE);
return 0;
}
```

```
Here is the message: We define it now!
```

- **Parameterized Macros**: Provide arguments to the macro, as reviewed earlier.



```
#define WRONG(A) A*A*A       /* fails for A=2+3 */
#define CUBE(A) (A)*(A)*(A) /* Do not fail for A=(2+3)*/
#define SQUR(A) (A)*(A)     /* Correct Macro */
#define MAX(A,B)  ((A)>(B)?(A):(B))  /* Macro for Max between two numbers*/
#define MIN(A,B)  ((A)>(B)?(B):(A))  /* Macro for Min between two numbers */
```

Functions

Header Files .h

- File with **extension .h** and contains C **functions declarations and macro definitions** to be shared by several sources.
- You have to use the CPP command #include for including headers.
- A good practice (almost **mandatory** for this course!) is to keep all **constants**, **macros**, system **global variables** and **functions** in **header files.** We include those header files wherever they are required.
- If you include **a header twice c**ompiler makes an **error**. To avoid that we write a **wrapped #ifndef**:

```
#ifndef HEADER_FILE
#define HEADER_FILE
the entire header file file
 #endif
```

- Sometimes you need to select a header o group of header for different cases of a system, in this case:

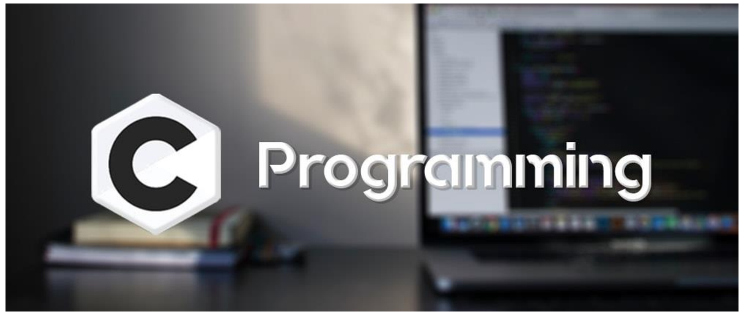
```
#if SYSTEM_1
                # include "system_1.h"
#elif SYSTEM_2
                # include "system_2.h"
#elif SYSTEM_3
...
#endif
```

- We can also put a macro as argument of #include, known as **computed include:**

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

Arrays
C Programming Language

Definition, declaration, call and arguments of a function.



Arrays
Overview

- An array is a **collection of data of a specific length and specific type**, both defined by the user.

- An array is useful instead of defining hundreds of different variables of the same type.

- An array is **declared** as: *type arrayName [arraySize];*
  Example: double labels[3]; This is a 3 elements array, which contains 3 numbers of type doubles.

- *To **initialize** the array, we just write: double labels[3] = {15.2, 5784.25, 15.0} ;*

- If you do not write the size of the array, it crates its minimum size automatically:
  double labels[] = {15.2, 5784.25, 15.0} ; is the same as : double labels[3] = {15.2, 5784.25, 15.0} ;

- *To assign only one value of the array, you write:*
  double labels[2] = 25.0 ; In this case the second element takes the value of 25, the other initiate in 0.0.

- To **access** one element of the array, we just write:
  my_variable=labels[1]; This provides the first element of variable *labels* to variable *my_variable*

Arrays
Multidimensional Array

- Arrays can be defined in any dimension, as: : *type arrayName [arraySize1] [arraySize2]…. [arraySizeN];*
- *Two-dimensional array is the simplest form of multidimensional array and can be seen as a matrix. Three dimensional is also easy to see, as a three-dimensional matrix, but they can take any size.*
- *A 3x4 array is initialized as:*

```
int a[3][4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

Nested brackets are optional, this is the same:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Arrays

Array as Functions Arguments

- To pass an array as argument of a function, there are three alternative. **For all cases C recognize the argument as a pointer!**:

1.- As pointer:
*void thefunction(int *param){ …. };*

*The pointer *param points to the first element of the array. **This is basically the only form to use array as argument.***

2.-As sized Array:
void thefunction(int param[10]){ …. };

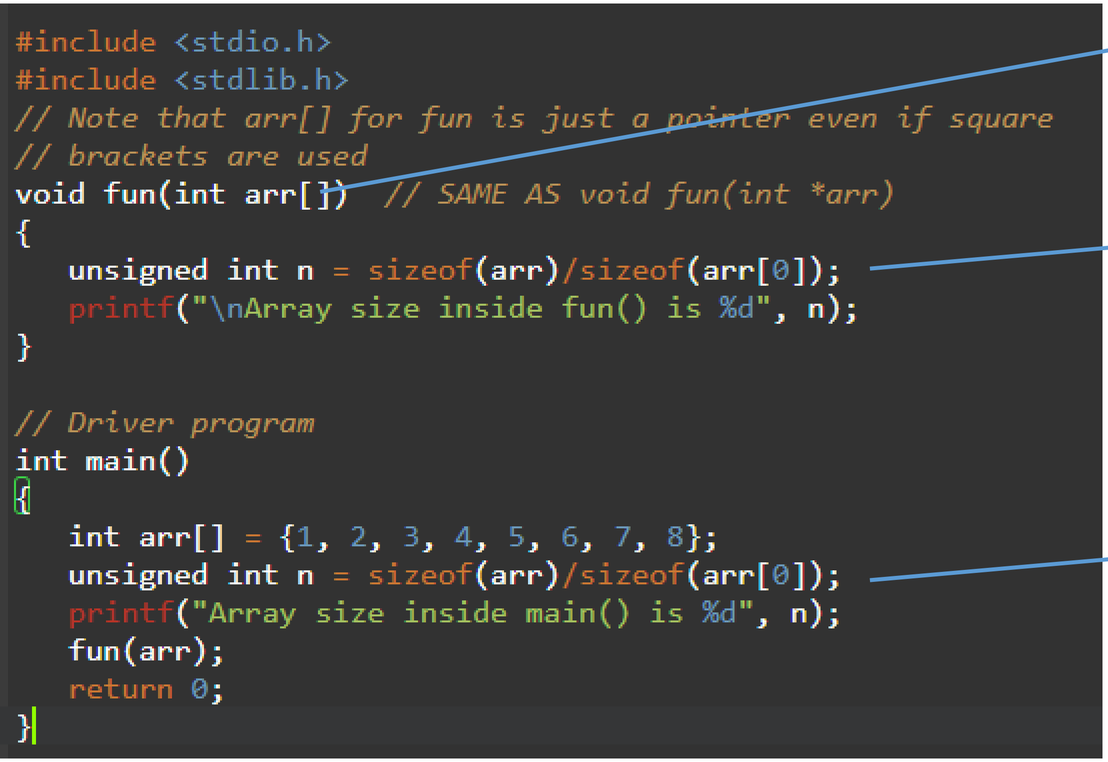
A predefined size is useful when you known in advance the size of your array. However, function takes it as pointer to its first element.

3.-As unsized Array:
void thefunction(int param[]){ …. };

An unsized array is useful when you don't know the size of the input array of the function. However, function takes it as pointer to its first element.

Arrays

Array as Functions Arguments-Example



```c
#include <stdio.h>
#include <stdlib.h>
// Note that arr[] for fun is just a pointer even if square
// brackets are used
void fun(int arr[])  // SAME AS void fun(int *arr)
{
   unsigned int n = sizeof(arr)/sizeof(arr[0]);
   printf("\nArray size inside fun() is %d", n);
}

// Driver program
int main()
{
   int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
   unsigned int n = sizeof(arr)/sizeof(arr[0]);
   printf("Array size inside main() is %d", n);
   fun(arr);
   return 0;
}
```

We pass the **unsized** array **arr[]** as argument to the **function fun.**

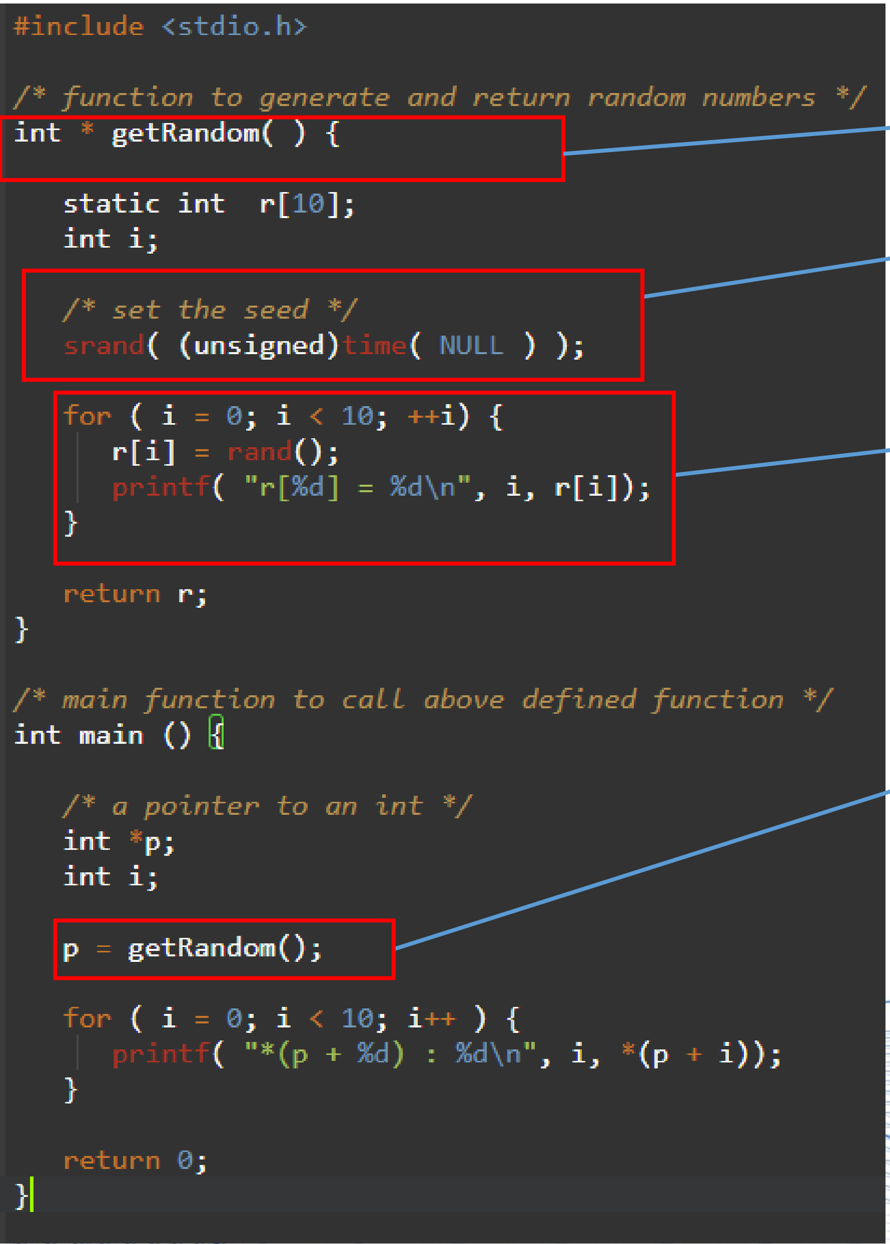
We calculate the number of elements of the array arr[] when function is called.
As arr[] is taken as a pointer to its first element, this size is 1.

We calculate the number of elements of the array arr[] within main.
As arr[] is not a function argument, and therefore is not a pointer, here the size of n is 8.

As C recognizes the argument as a pointer, you can not calculate the size of the array inside the function. You must calculate it previously and give the size of the array to the function as a second parameterA

Arrays

Returning Array from a Function-Example



```c
#include <stdio.h>

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

   static int  r[10];
   int i;

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

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

   return r;
}

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

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

   p = getRandom();

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

   return 0;
}
```

Functions can not return arrays. But they can return a pointer to array. **Here we indicate the return of the function will be a pointer.**

Used to initialize random values. See here

We complete an array with random valuies.

p is a pointer, will take the value that return the function. In this case is the **pointer to the array r**

r[0] = numero1	*(p + 0) : numero1
r[1] = numero2	*(p + 1) : numero2
r[2] = numero3	*(p + 2) : numero3
r[3] = numero4	*(p + 3) : numero4
r[4] = numero5	*(p + 4) : numero5
r[5] = numero6	*(p + 5) : numero6
r[6] = numero7	*(p + 6) : numero7
r[7] = numero8	*(p + 7) : numero8
r[8] = numero9	*(p + 8) : numero9
r[9] = numero10	*(p + 9) : numero10

String
Definition

- A string is simply a **one-dimensional array** terminated by a **null** character **\0**.

- For instance, the following two **declaration and initialization** are equivalent :

```c
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char greeting[] = "Hello";
```

- An array saves each character in consecutive memory space:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

To assign value to a string, either you must do it **character by character** or use the following functions:

Important functions to deal with strings from **string.h**
strcpy(s1, s2); Copies string s2 into string s1.
strcat(s1, s2); Concatenates string s2 onto the end of string s1.
strlen(s1); Returns the length of string s1.
strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

Array

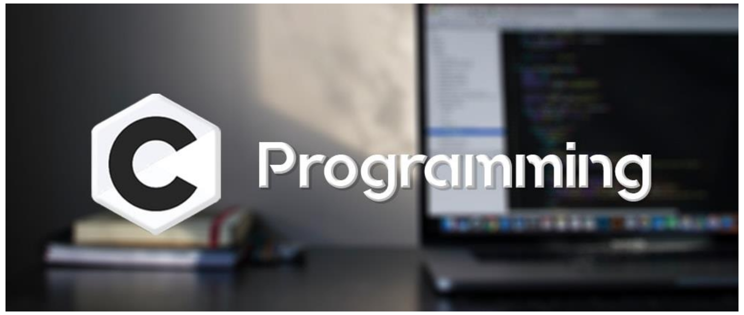
Excersice

1. Create a program that read the date day and month and gives you the temperature of that day, for that:

   i) Saves with random values the temperature of one year.
   ii) Define a function that transform the temperature from calculus to Fahrenheit and show both values.
   iii) Use header files to incorporate the function that change temperature, and also create a function that calculates the average temperature of the month and also show it on console.
   iv) Use macros to define your constants and include your headers files.

Pointers
C Programming Language

Definition, use, arithmetic and its use in functions.



Pointers

Overview

- A **pointer is** a variable whose value **is the address of another variable**.
- The use of **pointers are mandatory for memory management tasks**. So, we must learn it!
- Remember that every variable uses a piece of memory, which has its own address using & operator.
- There are three important steps to understand pointers (pointer definition, adress assignment and data access):
  - **1.- Pointer definition:** type *variable

```
int *ip;        /* pointer to an integer */
double *dp;   /* pointer to a double */
float *fp;   /* pointer to a float */
char *ch      /* pointer to a character */
```

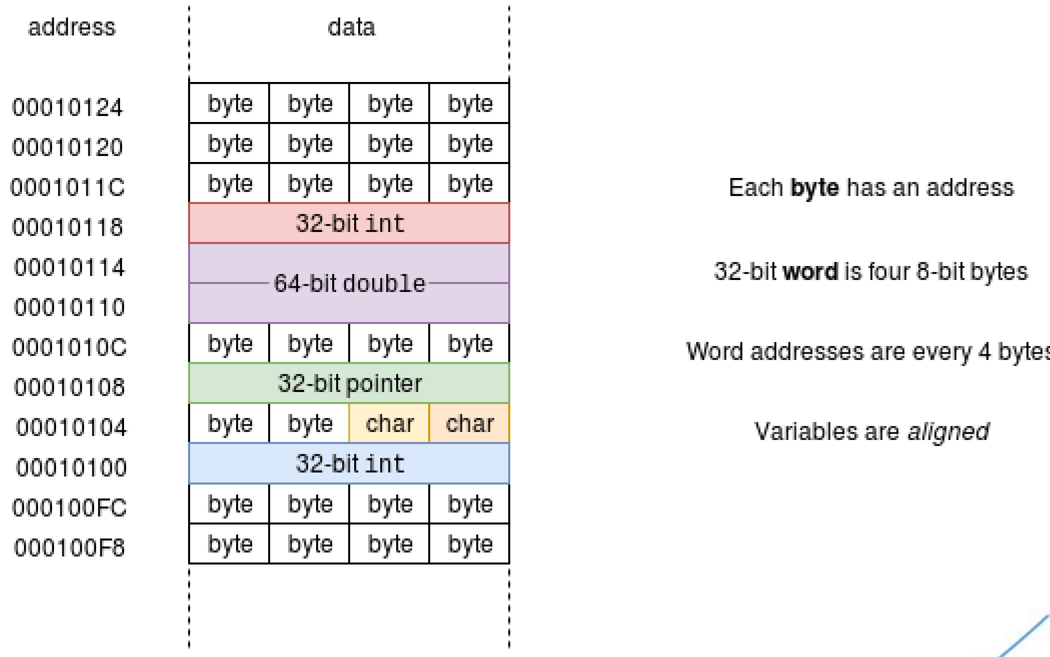
    - **Unary operator *** denotes that we are defining a pointer.
    - type defines the type of the variable located at the address stored in variable (without *)
    - Variables ip, dp, fp or ch are all pointers and content an address in a **long hexadecimal** type.
  - **2.- Address Assignment:**
    - Using **ip=&a**; gives to the pointer **ip**, the **address** where the **integer a** is stored.
  - **3.- Data Access:** Using the operator *, returns the content of address sotred in the pointer.
    - Using ***ip=20;** write the number 20 in &a, i.e. now a is equal to 20.

Pointers
Overview

- In modern systems pointers are byte addressable. This means that 1 byte of data has its own address.
  - Suppose your pointer *ptr* points to the memory address 284b0h (165040 decimal) and suppose that your pointer points to an integer. As an integer is 4bytes. When you increment your pointer *ptr++;* it will be incremented in 4, 284b4h.

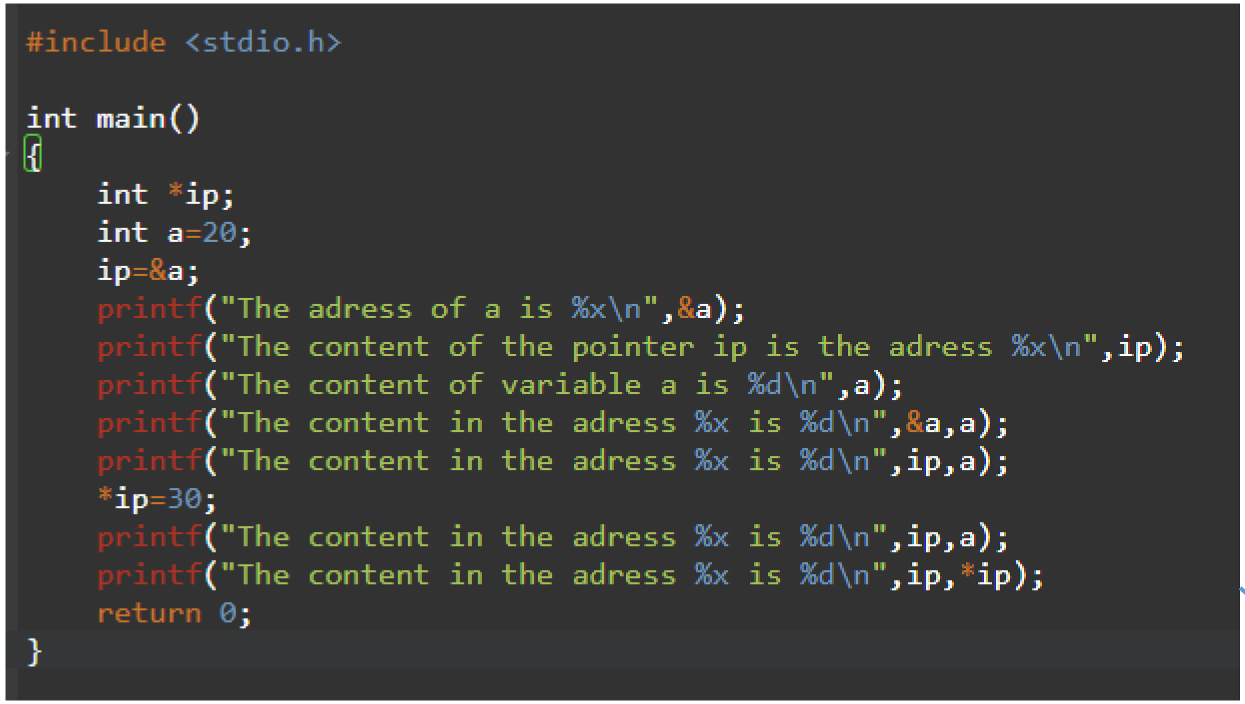
- The size of a pointer only indicates how many address I can save
  - A 1-byte size pointer can point only to 256 possible addresses.
  - A 4-byte size pointer enables 4billions of addresses.

'Realistic' 32-bit memory map



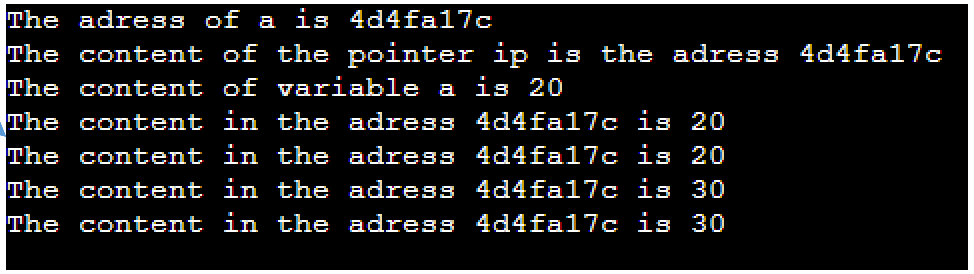
Pointers

Example



```
#include <stdio.h>

int main()
{
    int *ip;
    int a=20;
    ip=&a;
    printf("The adress of a is %x\n",&a);
    printf("The content of the pointer ip is the adress %x\n",ip);
    printf("The content of variable a is %d\n",a);
    printf("The content in the adress %x is %d\n",&a,a);
    printf("The content in the adress %x is %d\n",ip,a);
    *ip=30;
    printf("The content in the adress %x is %d\n",ip,a);
    printf("The content in the adress %x is %d\n",ip,*ip);
    return 0;
}
```

This example show as the difference between **ip**,***ip** and the adress of variable a (**&a**) with variable **a**.



```
The adress of a is 4d4fa17c
The content of the pointer ip is the adress 4d4fa17c
The content of variable a is 20
The content in the adress 4d4fa17c is 20
The content in the adress 4d4fa17c is 20
The content in the adress 4d4fa17c is 30
The content in the adress 4d4fa17c is 30
```

Pointers

NULL Pointers

- To avoid undefinitions, a good practice is always to initialize a pointer. When you don’t know the pointer address in advance, you can **initialize it as NULL** value.
  - Example in ***ip =NULL;**
- This definition means that the pointer is not pointing anything as the memory address 0 is restricted and not accessible. To check is pointer is NMULL or not , we use:
  - **If(ptr)**   /*true is the pointer is not null , in other words it has a valid address */
  - **If(!ptr)**   /*true is the pointer is null , in other words it is not pointing anything */

Pointers Arithmetic – Incrementing/Decrementing

- **Pointers are** address and therefore **numbers.**
- There are four arithmetic operations are valid to pointers **++, --, + , -**
- Each time the pointer is **incremented** by one, **it points to the next variable <u>of its type!.</u>**
  - Suppose a pointer to integer ip. If ip=1000 , then, after ip++ its value is 1004 (with integer length of 4byte)
  - Suppose a pointer to char ch. If ch=1000 , then, after ch++ its value is 1001 (char length is 1byte)

```
#include <stdio.h>
const int MAX = 3;
int main () {
int var[] = {10, 100, 200};
int i, *ptr;

 /* let us have array address in pointer */
ptr = var;  /* Equivalent to ptr = &var[0]*/

for ( i = 0; i < MAX; i++) {
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */ ptr++;
}
return 0; }
```

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Pointers

Pointers Arithmetic - Comparisons

- Relational operators, e.g **==, <, >, >=, <=, etc** can be also used to compare pointers (i.e. addresses) .

```
#include <stdio.h>
const int MAX = 3;
int main () {
int var[] = {10, 100, 200};
int i, *ptr;

/* let us have address of the first element in pointer */
ptr = &var[0];

i = 0;
while ( ptr <= &var[MAX - 1] ) {
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
 /* point to the next location */
ptr++; i++;
}

return 0;
}
```

Using this example, we can print the array starting at the pointer address and up to the end of the array.

Pointers

Array of pointers

- Useful if we want to create a group of pointer that point to the same type of variable

```c
#include <stdio.h>
const int MAX = 3;
int main () {
int var[] = {10, 100, 200};
int i, *ptr[MAX];

for ( i = 0; i < MAX; i++) {
ptr[i] = &var[i];
/* assign the address of integer. */
}
for ( i = 0; i < MAX; i++) {
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0; }
```

What do we get from this code?

10, 100 , 200? Or three different addresses?

Pointers

Pointers to Pointers

- Known as **multiple indirection** or chain of pointers. It is like having nested addresses to a data.
- According to ANSI C, you can have **12 layers** of nested pointers.

```c
#include <stdio.h>
int main () {
int var;
int *ptr;
int **pptr;
var = 3000;
 /* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

Using ** defines a second layer pointer

First, we assign the address of var to ptr.

Then we assign the **address** of the pointer **ptr** (which also contain an address as value) to the pointer **pptr**.

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Pointers

Pointer in Function Arguments and return

```
#include <stdio.h>
/* function declaration */

double getAverage(int *arr, int size);

int main () {
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
/* pass pointer to the array as an argument */
avg = getAverage( balance, 5 ) ;
/* output the returned value */
printf("Average value is: %f\n", avg );
return 0;
}

double getAverage(int *arr, int size) {
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i) {
sum += arr[i];
}
avg = (double)sum / size;
return avg;
}
```

Formal definition:

type * function(type *variable_1,…, type *variable_n){…};

A function can receive a pointer to array as argument.

A function can receive a pointer as argument.

A function can return a pointer.

```
#include<stdio.h>
int *larger(int *, int *);
int main() {
int a = 10, b = 15;
int *greater;
// passing address of variables to function

greater = larger(&a, &b);
printf("Larger value = %d", *greater);
return 0;
}


int *larger(int *a, int *b) {
if (*a > *b) {
return a;
}
// returning address of greater value
return b;
}
```

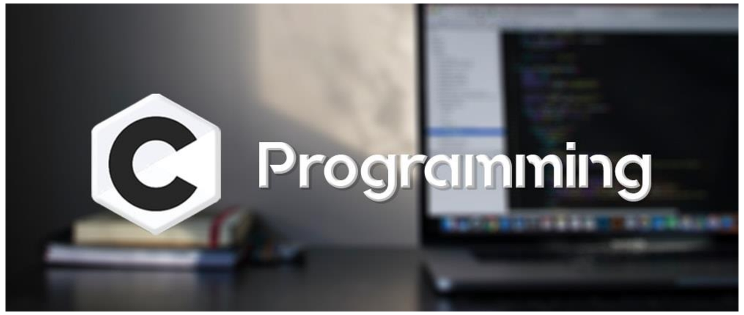
Exercises

Pointers

1. Write a C program to copy one two-dimensional array to another using pointers. Each array is 10x10 elements.
2. Write a C program to swap two two-dimensional arrays using pointers. Array 10x10
3. Write a C program to transpose a two-dimensional array using pointers. Array 10x10
4. Write a C program to search an element in a two-dimensional array using pointers. Refurn the row and column of the single or multiple elements.
5. Write a C program to add and multiply two matrix using pointers. One matrix 10x10 and other 10x2.
6. Write a C program to copy one string to another using pointers.
7. Write a C program to concatenate two strings using pointers.
8. Write a C program to compare two strings using pointers.
9. Write a C program to find reverse of a string using pointers.
10. Write a C program to sort array using pointers.

Structures
C Programming Language

Definition, declaration, call and arguments of a function.



Structures

Definition

- Like array, **structures** allow to store data items of different types.
- A structure is defined as:

```
struct [structure tag] {
member definition;
 member definition;
... member definition; }
[one or more structure variables];
```

Example:

```
struct  BookStore{
char titles[50];
char authors[50];
char subjects[100];
int book_id;
} book;
```

- Each member of the structure is a normal variable definition.
- Structure tag and structure variables are optional.

Once the structure is defined, we can define several variables with that structure!

Structures

Example

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

struct Books {
 char title[50];
 char author[50];
 char subject[100];
 int book_id;
 };

int main( ) {

struct Books Book1;
struct Books Book2;

strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0; }
```

Structure definition

Structure declaration: Two structures (Book1 and Book2) of the same kind (Books) are declared.

Structure specification:
We use **the member access operator .** to access to one member of the structure.
We use function **strcpy** to assign the string to structure members.

Structure console print.

Structures

Structure as argument of function.

```
void printBook( struct Books book ) {
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
 }
```

Giving a structure as argument to a function.

The function prints in console the different member of the structure given to the function as argument.

```
printBook( Book1 );
printBook( Book2 );
```

Calling the function within a main.

Structures

Pointers to Structure

- A pointer to structure is defined the same a any pointer.
  - struct structure_tag *struct_pointer;
- To assign the address of a structure variable we use &
  - struct_pointer = &structure_tag;
- We define a new operator, **-> ,** to access one member of the structure.
  - struct_pointer -> member1;

```
void printBook( struct Books *book ) {
printf( "Book title : %s\n", book -> title);
printf( "Book author : %s\n", book -> author);
printf( "Book subject : %s\n", book -> subject);
printf( "Book book_id : %d\n", book -> book_id);
 }
```

Giving a **pointer to structure** as argument to a function.

Using the operator **-> we access** to the **content of the different members** of the structure book. The function prints in console the different member of the structure given to the function as argument.

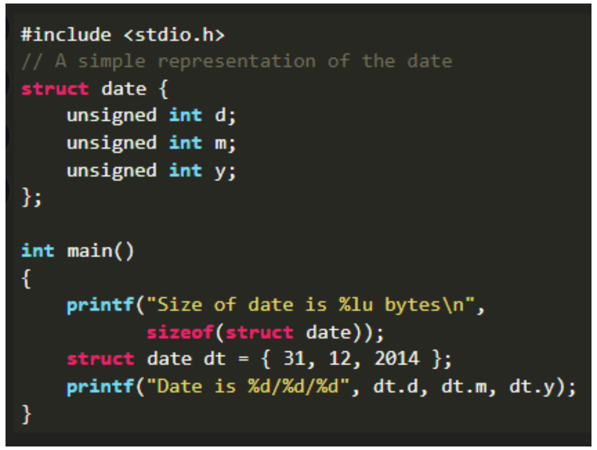
```
printBook( &Book1 );
printBook( &Book2 );
```

Calling the function within a main.
We provide the address of the structure.

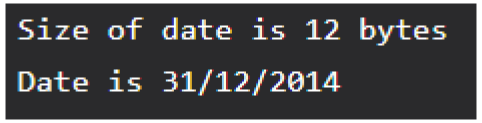
Structures

Bit Fields - Structure for Memory Optimization

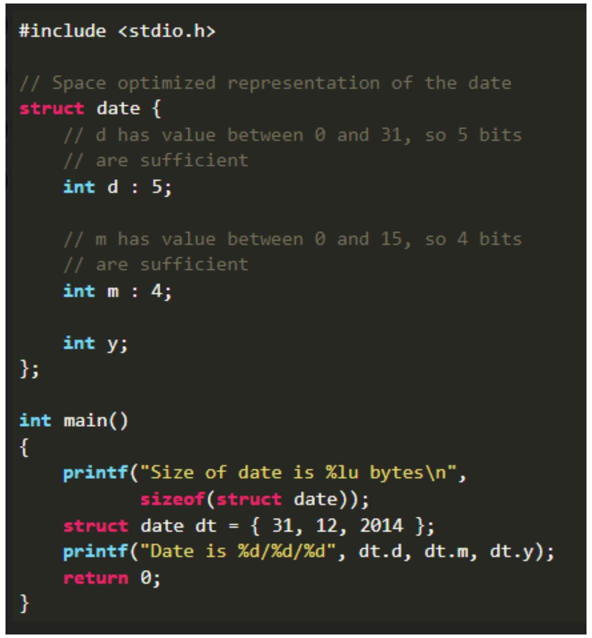
- Bit Fields allow to use the 8/16/32 or 64bits of one word of the machine independently. This optimize the use of memory.
- For making use of bits, we define the number of bits we need after variable declaration using **:**



```c
#include <stdio.h>
// A simple representation of the date
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};

int main()
{
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```



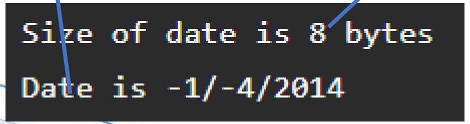
```
Size of date is 12 bytes
Date is 31/12/2014
```



```c
#include <stdio.h>

// Space optimized representation of the date
struct date {
    // d has value between 0 and 31, so 5 bits
    // are sufficient
    int d : 5;

    // m has value between 0 and 15, so 4 bits
    // are sufficient
    int m : 4;

    int y;
};

int main()
{
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}
```

signed int of 5 bits is used to represent 31. Thereby, the number is **1**1111. MSB means negative number. Using twos complement the negative number is -1.

Why size is 8 bytes? Lets see next example.

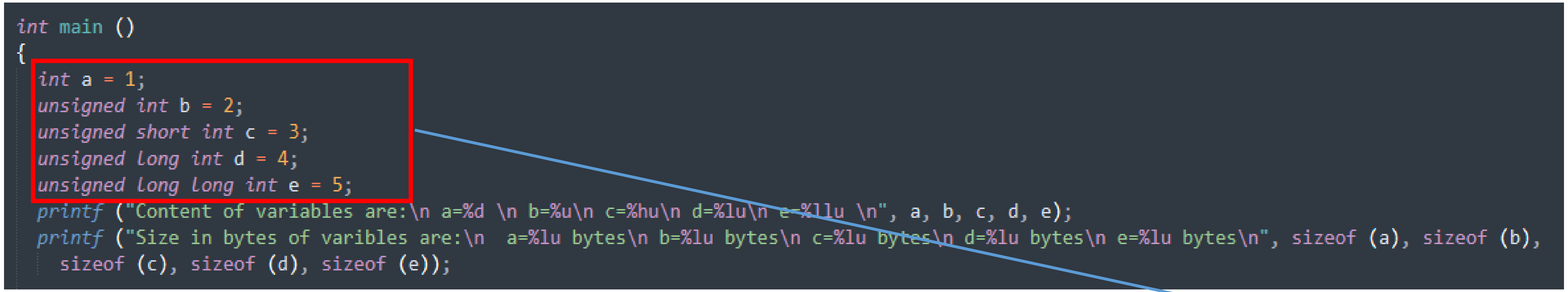


```
Size of date is 8 bytes
Date is -1/-4/2014
```

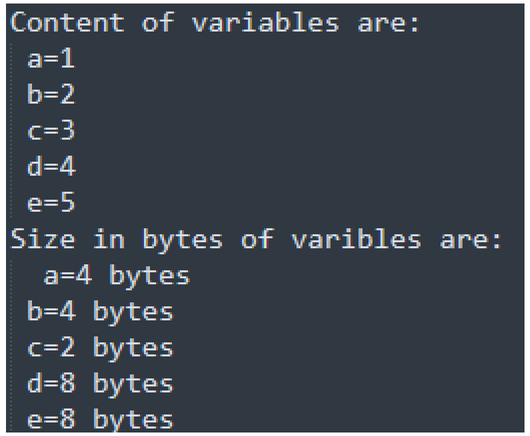
Structures

Bit Fields - Structure for Memory Optimization

- The value of the bit field will not exceed the maximum value of its type. Extra bit are not used.
  - Example: *unsigned int b:55;* This is a wrong definition as unsigned int has 2 bytes.
    unsigned long long int b:55; This is a correct definition.



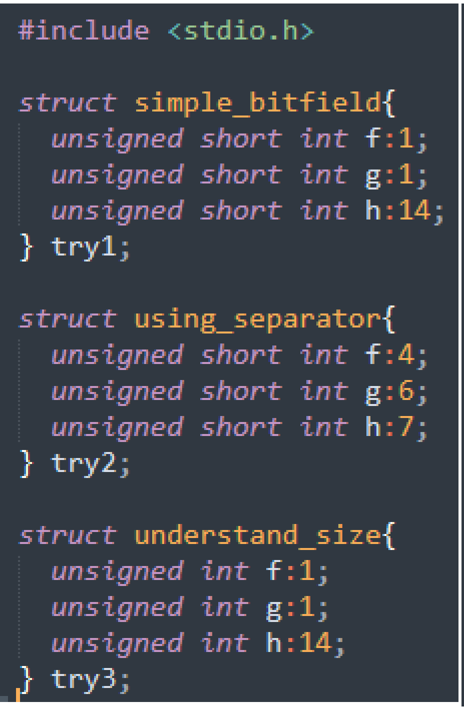
```
int main ()
{
  int a = 1;
  unsigned int b = 2;
  unsigned short int c = 3;
  unsigned long int d = 4;
  unsigned long long int e = 5;
  printf ("Content of variables are:\n a=%d \n b=%u\n c=%hu\n d=%lu\n e=%llu \n", a, b, c, d, e);
  printf ("Size in bytes of varibles are:\n  a=%lu bytes\n b=%lu bytes\n c=%lu bytes\n d=%lu bytes\n e=%lu bytes\n", sizeof (a), sizeof (b),
    sizeof (c), sizeof (d), sizeof (e));
```



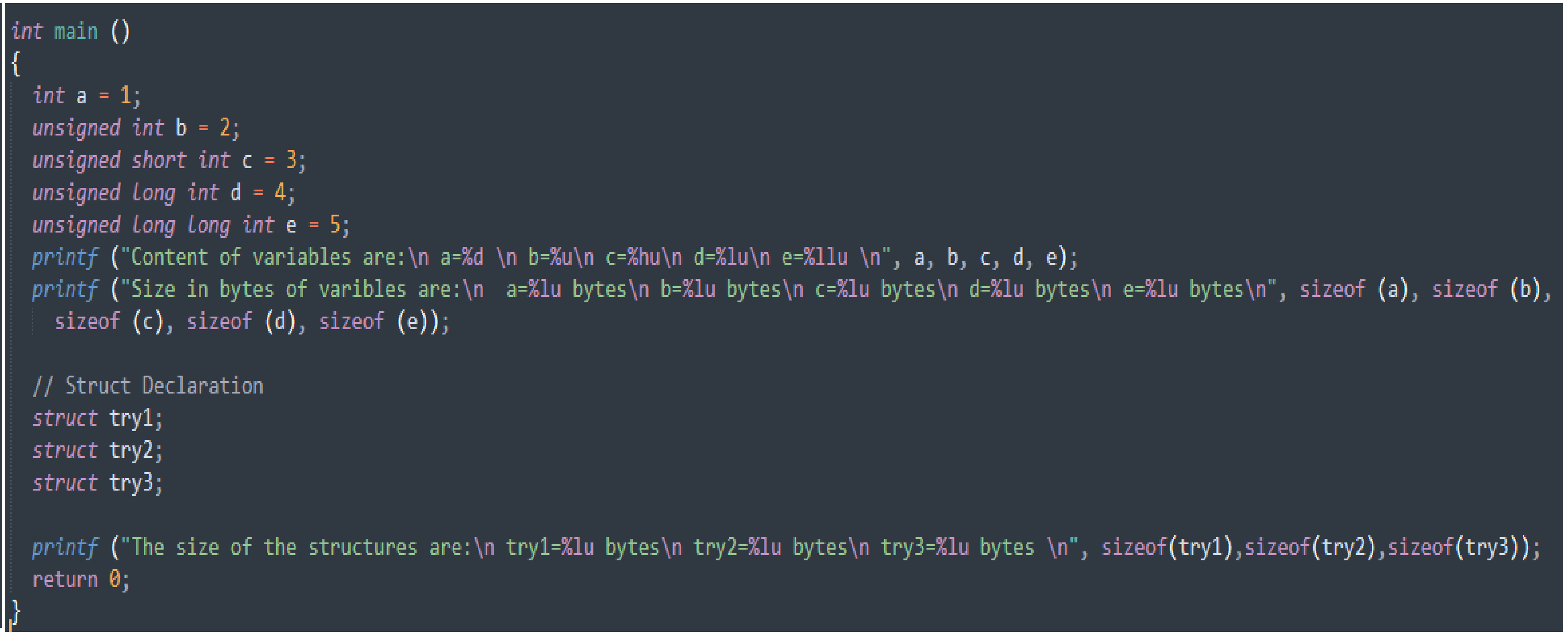
```
Content of variables are:
 a=1
 b=2
 c=3
 d=4
 e=5
Size in bytes of varibles are:
  a=4 bytes
 b=4 bytes
 c=2 bytes
 d=8 bytes
 e=8 bytes
```

Structures

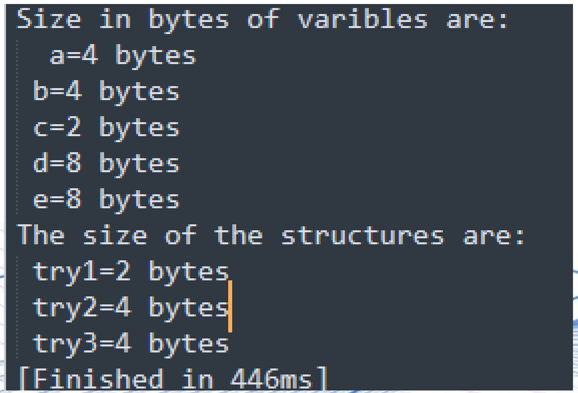
Bit Fields - Structure for Memory Optimization



```c
#include <stdio.h>

struct simple_bitfield{
  unsigned short int f:1;
  unsigned short int g:1;
  unsigned short int h:14;
} try1;

struct using_separator{
  unsigned short int f:4;
  unsigned short int g:6;
  unsigned short int h:7;
} try2;

struct understand_size{
  unsigned int f:1;
  unsigned int g:1;
  unsigned int h:14;
} try3;
```



```c
int main ()
{
  int a = 1;
  unsigned int b = 2;
  unsigned short int c = 3;
  unsigned long int d = 4;
  unsigned long long int e = 5;
  printf ("Content of variables are:\n a=%d \n b=%u\n c=%hu\n d=%lu\n e=%llu \n", a, b, c, d, e);
  printf ("Size in bytes of varibles are:\n  a=%lu bytes\n b=%lu bytes\n c=%lu bytes\n d=%lu bytes\n e=%lu bytes\n", sizeof (a), sizeof (b),
    sizeof (c), sizeof (d), sizeof (e));

  // Struct Declaration
  struct try1;
  struct try2;
  struct try3;

  printf ("The size of the structures are:\n try1=%lu bytes\n try2=%lu bytes\n try3=%lu bytes \n", sizeof(try1),sizeof(try2),sizeof(try3));
  return 0;
}
```

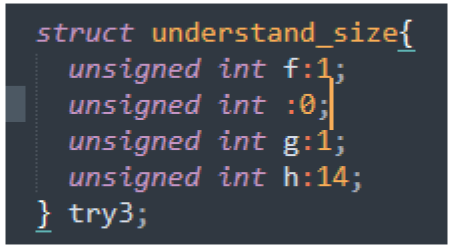


```
Size in bytes of varibles are:
  a=4 bytes
 b=4 bytes
 c=2 bytes
 d=8 bytes
 e=8 bytes
The size of the structures are:
 try1=2 bytes
 try2=4 bytes
 try3=4 bytes
[Finished in 446ms]
```

- For try1, it is possible to keep all bits in 2 bytes.
- For try2 we need 17bits, therefore the system add another block of memory of 2 bytes
- For try3, the minimum block size is 4bytes (u int). As we need 16bits, they are saved in 4bytes.

Structures

Bit Fields - Structure for Memory Optimization

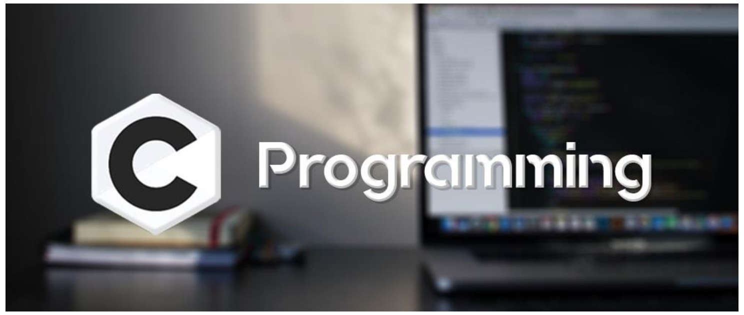


```
struct understand_size{
  unsigned int f:1;
  unsigned int :0;
  unsigned int g:1;
  unsigned int h:14;
} try3;
```

- Using zero-bit width field as **:0,** forces the system to fill the first boundary and start with the next memory bock.
- **In this example the structure size is 8bytes.** First 4 bytes are used only by bit "f".

Memory Management
C Programming Language

Allocating, resizing, releasing

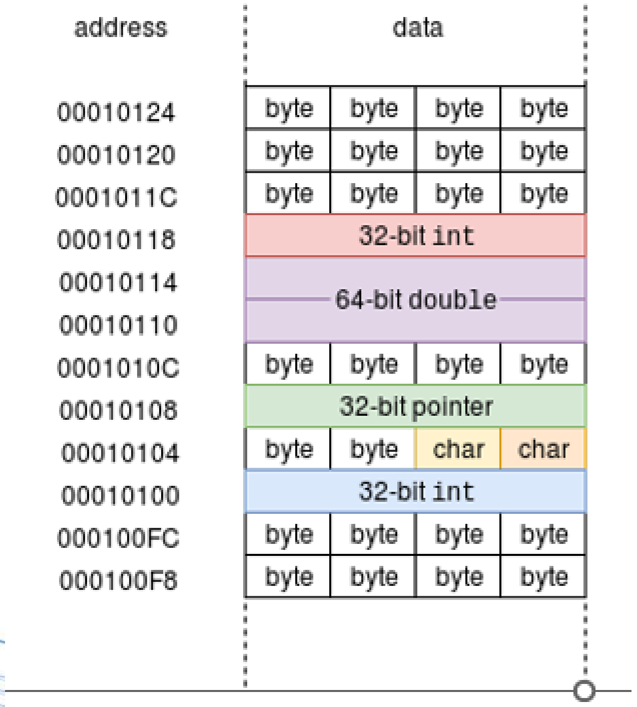


Memory Management

Functions for Memory Management

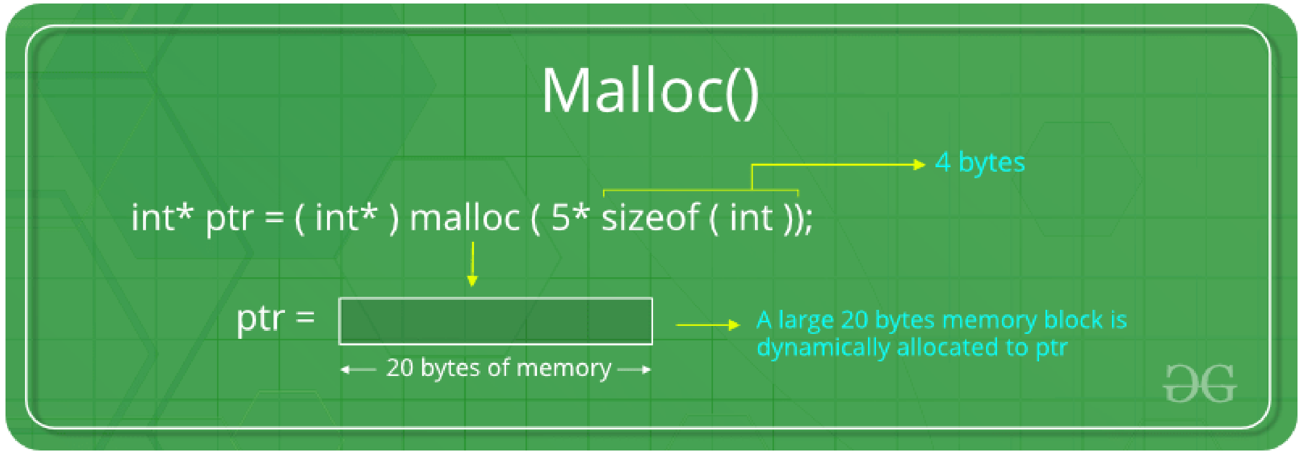
Function & Description
void *calloc(int num, int size); This function allocates an array of **num** elements each of which size in bytes will be **size**.
void free(void *address); This function releases a block of memory block specified by address.
void *malloc(size_t size); This function allocates an array of **num** bytes and leave them uninitialized.
void *realloc(void *address, int newsize); This function re-allocates memory extending it upto **newsize**.

- This is an example of a 32bit data memory.
- The memory address is directed for each byte.
- How to assign this memory dynamically is a key task for modern computers



Memory Management

Memory Allocation Method - MALLOC


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

- Dynamically allocate a block of memory with the specified size.
- It returns a pointer of type void to the first address of the allocated memory.
- We can "cast" the returned pointer to the variable we like.
  - ✓ Ptr = (cast_type*) malloc(byte_size);
- The block of memory is not initialized. It contains previous random data.
- If there is no enough space left in memory, malloc returns a NULL pointer.

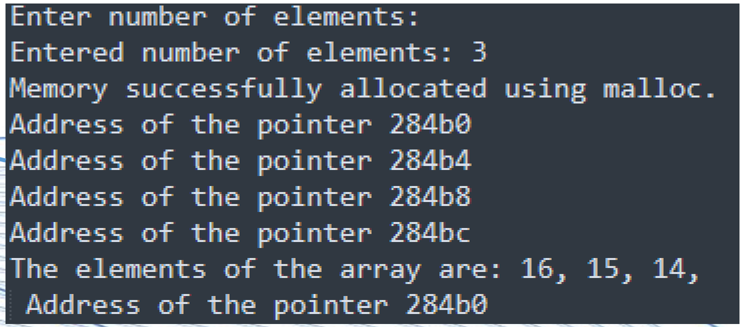
Memory Management

Memory Allocation - MALLOC

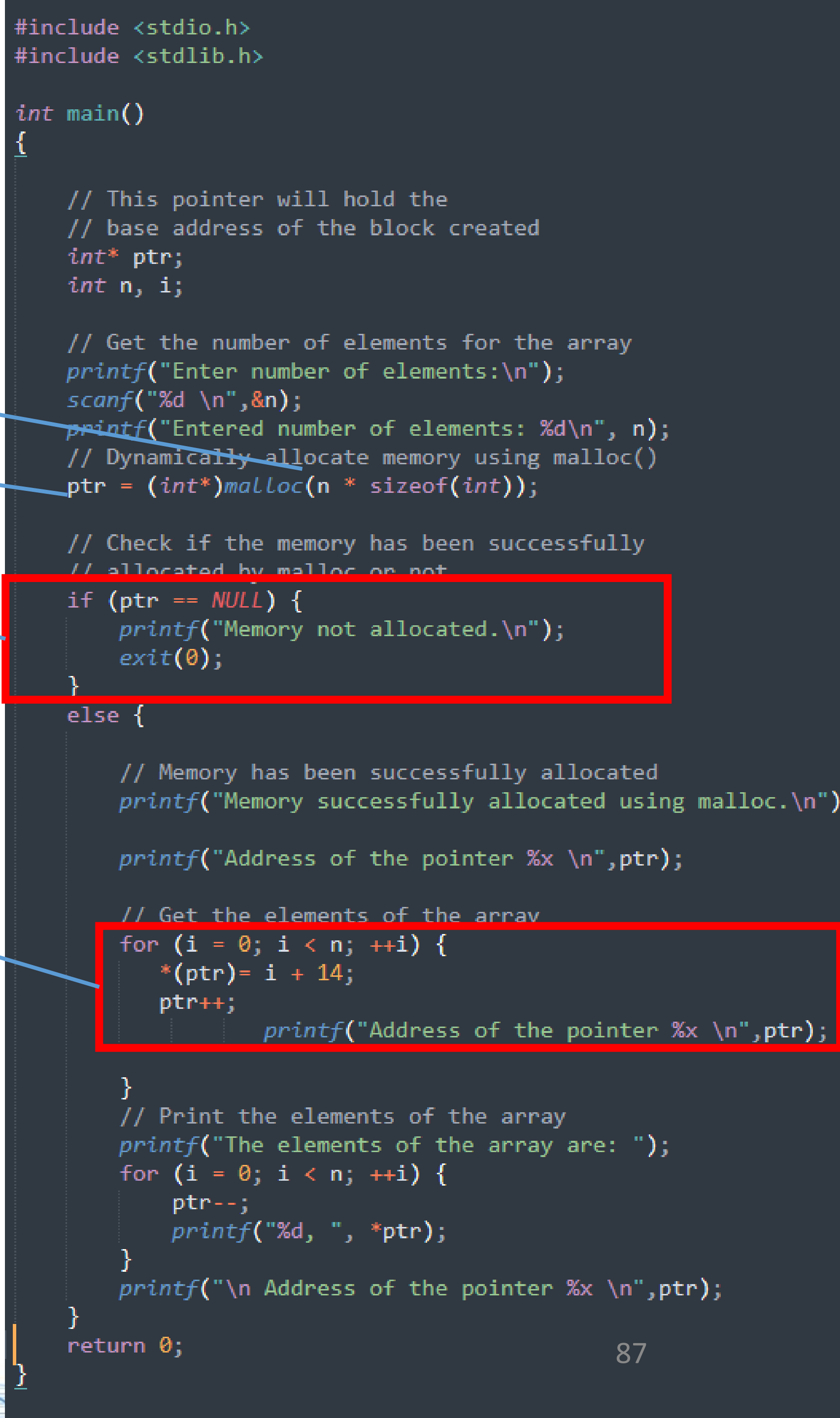
- The reserved memory space u n times 4 bytes. *4n bytes.*
- *ptr* points to the first element of the reserved block of memory of 4n bytes.
- In case *ptr* is NULL, malloc was not able to allocate the memory block.

- We fill with data the memory block. Using ptr[i] as array is equivalent as using is as *(ptr+1)=i+1;

  - Clearly the pointer increase 4-byte by 4-bytes. We print the number from backwards.



```
Enter number of elements:
Entered number of elements: 3
Memory successfully allocated using malloc.
Address of the pointer 284b0
Address of the pointer 284b4
Address of the pointer 284b8
Address of the pointer 284bc
The elements of the array are: 16, 15, 14,
 Address of the pointer 284b0
```



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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:\n");
    scanf("%d \n",&n);
    printf("Entered number of elements: %d\n", n);
    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n")

        printf("Address of the pointer %x \n",ptr);

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            *(ptr)= i + 14;
            ptr++;
                    printf("Address of the pointer %x \n",ptr);

        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            ptr--;
            printf("%d, ", *ptr);
        }
        printf("\n Address of the pointer %x \n",ptr);
    }
    return 0;
}
```

Memory Management

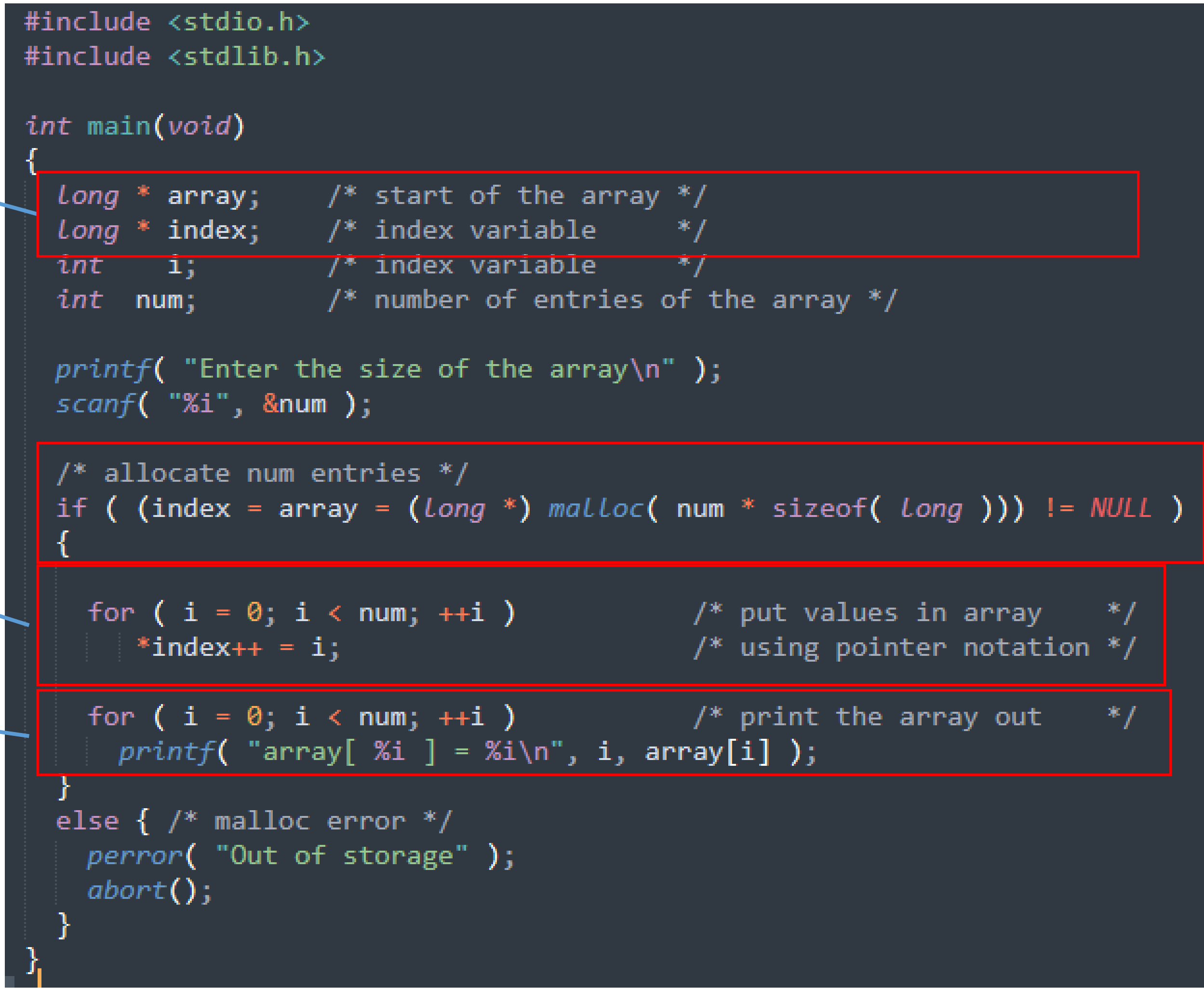
Memory Allocation - MALLOC

We define two pointers to long *array* and *index*.

If allocation is successful, the return pointer from malloc is given to *array* and *index*

We write the allocated values of memory using pointer notation

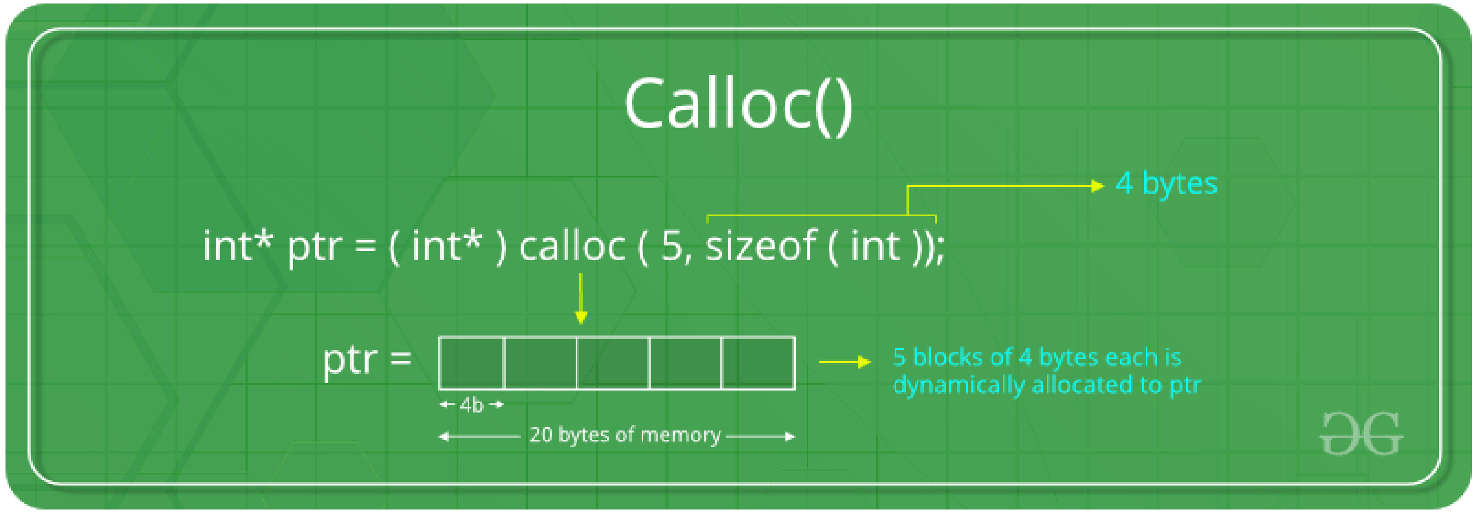
We print the values using an array defined by pointer.



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

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable     */
  int    i;        /* index variable     */
  int  num;        /* number of entries of the array */

  printf( "Enter the size of the array\n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long *) malloc( num * sizeof( long ))) != NULL )
  {

    for ( i = 0; i < num; ++i )           /* put values in array    */
      *index++ = i;                       /* using pointer notation */

    for ( i = 0; i < num; ++i )           /* print the array out    */
      printf( "array[ %i ] = %i\n", i, array[i] );
  }
  else { /* malloc error */
    perror( "Out of storage" );
    abort();
  }
}
```

Memory Management

Contigous Allocation Method - CALLOC



```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

- Very similar to malloc, but it:
  - Initialize each block to the value 0
  - Is uses two parameters as arguments.

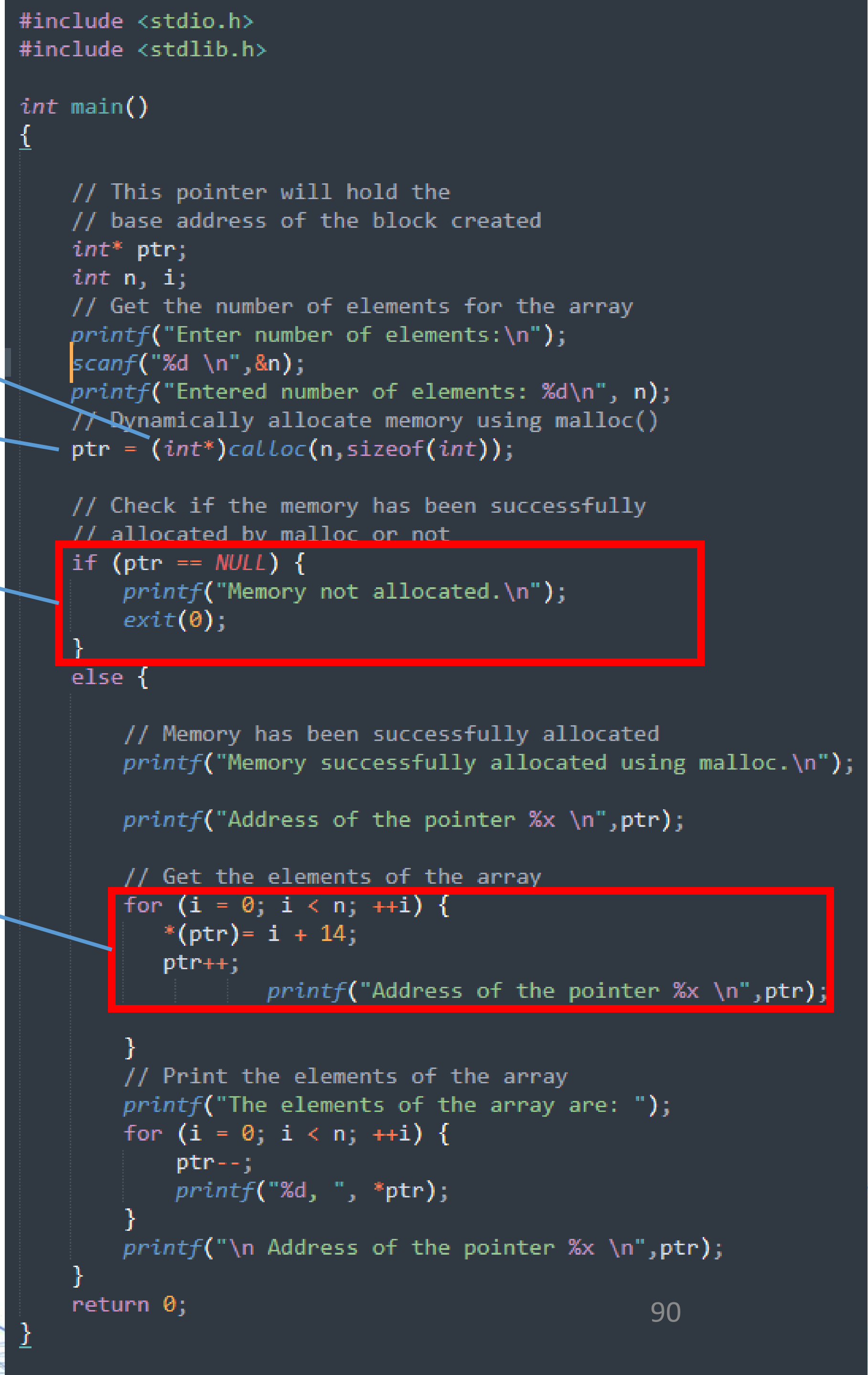
Memory Management

Memory Allocation - MALLOC

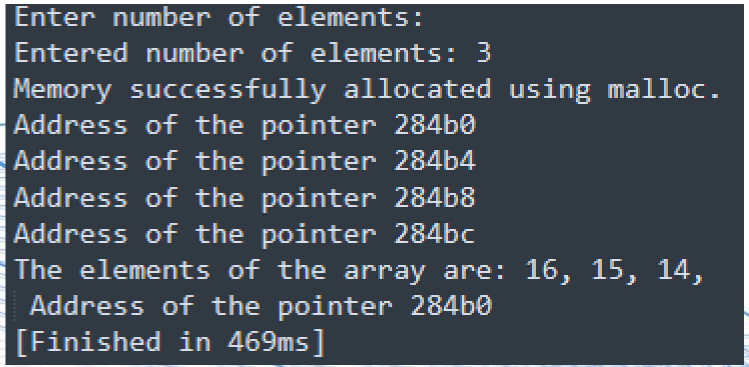
- The reserved memory space u n times 4 bytes. *4n bytes. NOW USING CALLOC*
- *ptr* points to the first element of the reserved block of memory of 4n bytes.
- In case *ptr* is NULL, malloc was not able to allocate the memory block.

- We fill with data the memory block. Using ptr[i] as array is equivalent as using is as *(ptr+1)=i+1;

    - Clearly the pointer increase 4-byte by 4-bytes. We print the number from backwards.



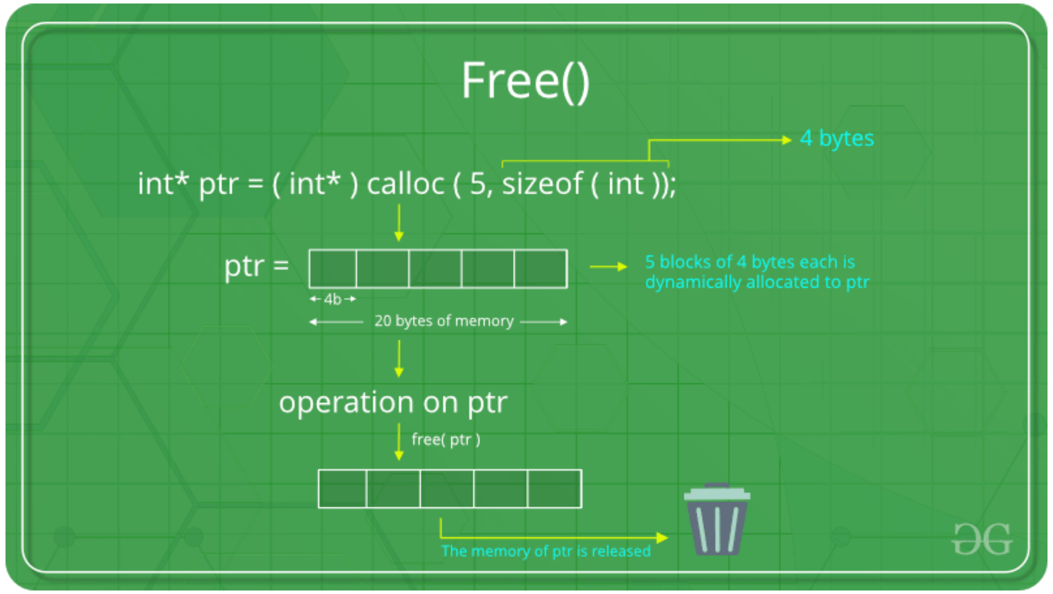
```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    printf("Enter number of elements:\n");
    scanf("%d \n",&n);
    printf("Entered number of elements: %d\n", n);
    // Dynamically allocate memory using malloc()
    ptr = (int*)calloc(n,sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        printf("Address of the pointer %x \n",ptr);

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            *(ptr)= i + 14;
            ptr++;
                        printf("Address of the pointer %x \n",ptr);

        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            ptr--;
            printf("%d, ", *ptr);
        }
        printf("\n Address of the pointer %x \n",ptr);
    }
    return 0;
}
```



```
Enter number of elements:
Entered number of elements: 3
Memory successfully allocated using malloc.
Address of the pointer 284b0
Address of the pointer 284b4
Address of the pointer 284b8
Address of the pointer 284bc
The elements of the array are: 16, 15, 14,
 Address of the pointer 284b0
[Finished in 469ms]
```

Memory Management

Free method for Memory de-allocation.


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

- Dynamic Memory allocation defined by CALLOC or MALLOC is not de-allocated on their own until the program finishes.
- Free release the memory allocated by CALLOC or MALLOC and can be used for other purpose.
- We just need to use as argument the pointer to the block of memory, it means the return value from CALLOC or MALLOC.

Memory Management

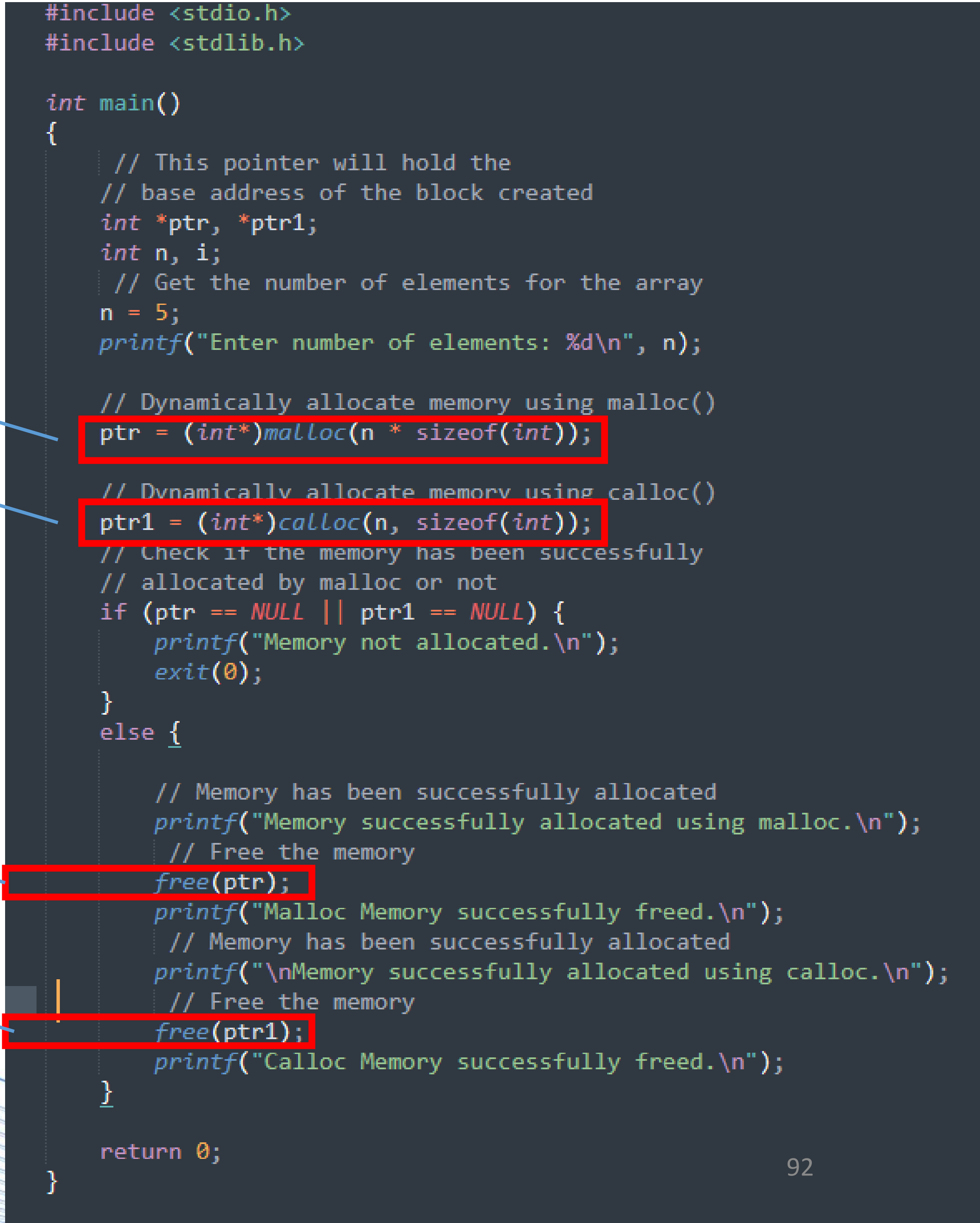
Free method for Memory de-allocation.

ptr points to the memory allocated by MALLOC of size n*4.

ptr points to the memory allocated by CALLOC of size n*4.

Free the memory reserved by malloc, pointed by ptr, using function free.

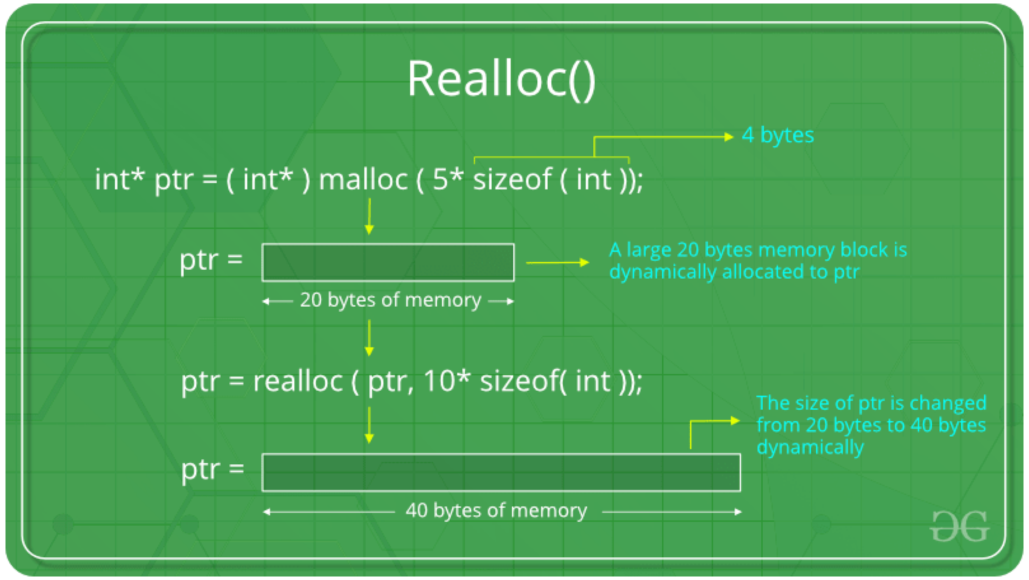
Free the memory reserved by malloc, pointed by ptr1, using function free.



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

int main()
{
     // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;
     // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
         // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");
         // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");
         // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Memory Management

Re-Allocation Method - REALLOC



Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

- Used to dynamically reallocate memory.
- This is useful if we want to enlarge or shrink the allocated memory.
- Can be used to release memory block that is not being used or to increase the reserved memory.
- It copies the old block to the new block of memory.
- We need to use the same pointer provided previously by MALLOC, CALLOC or REALLOC.

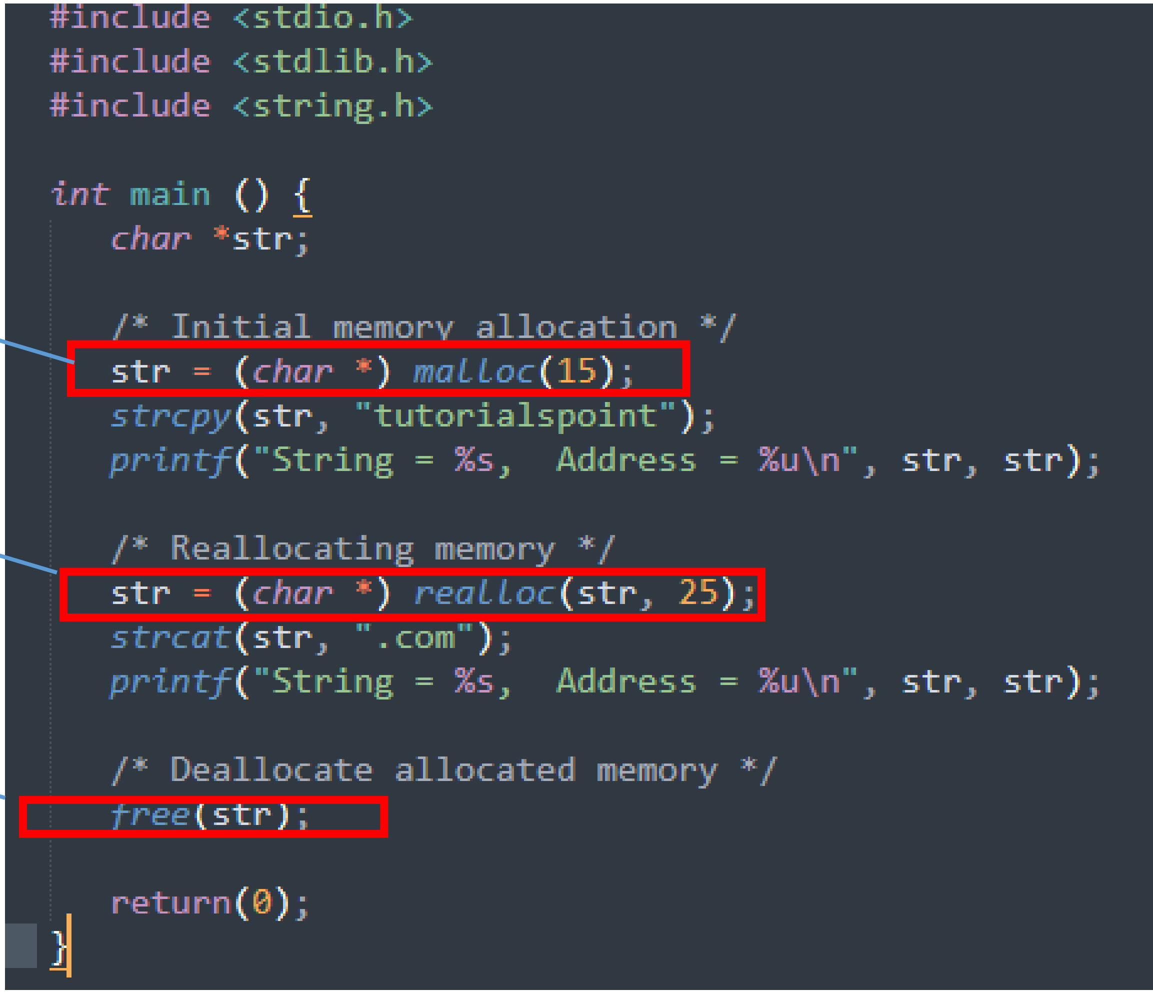
Memory Management

Free method for Memory de-allocation.

Assign 15 bytes using MALLOC

Reallocate the originally 15 bytes pointed by str to 25bytes.

Deallocate the memory allocated by *realloc*.



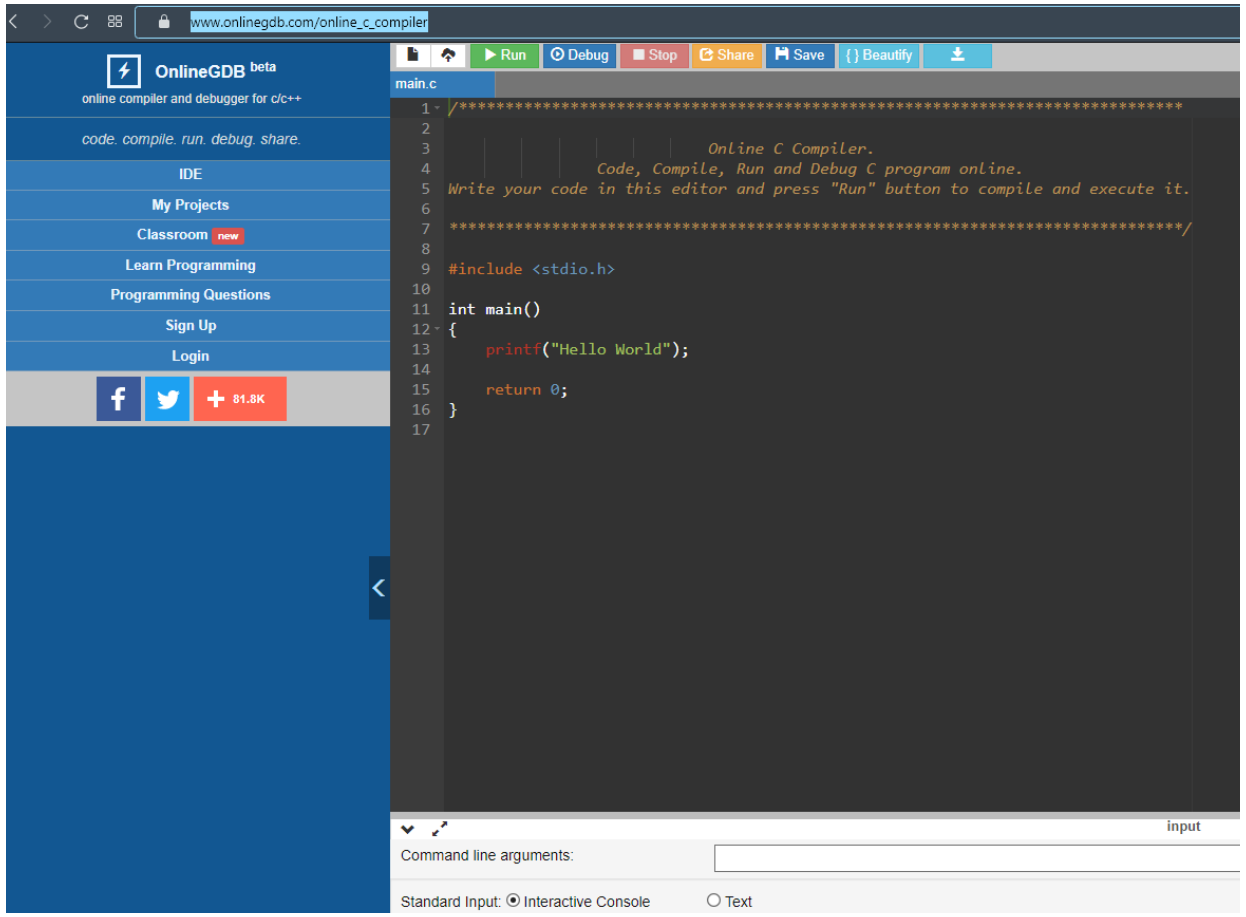
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   /* Deallocate allocated memory */
   free(str);

   return(0);
}
```

Appendix A
Where to Program?

This section provides a brief guideline to install a compiler (or not) into your PC.

Appendix A

Where to program?- Online alternative



```
/******************************************************************************

                            Online C Compiler.
                Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

Command line arguments:

Standard Input: [✓ Interactive Console] Text

An online alternative to try your programming is onlinegdb.

Advantages:

- Do not require text editor
- Do not require C compiler .gcc
- Ready-to-use system

Disadvantages:

- Only online available
- Compilation can be slow for large prgramms.
- Your programs are not private, as GDB owns them.
- You might loss data in case of GDB failure. (very unlikely)

We strongly recommend you practice with this environment before starting with you Zybo Z7.

Appendix A

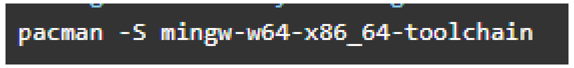
Where to program?- Offline alternative

1.- First, we need to installa texteditor. This will help us to identify program error and we can see our code in colors and in a friendly manner. Some popular text editors are:

- Sublime **(recommended for this course)**
- Eclipse
- A full list of text editors can be found here.

2.- Install the C compiler **gcc** (remember this is the *translator* from our C code to the processor set of instructions and lately a binary code).
There are several alternatives to install a compiler, many of them come with an associated software to program in C code. This is usually not desires, as software require license. To install only the compiler in its ligthest form, we recommend:

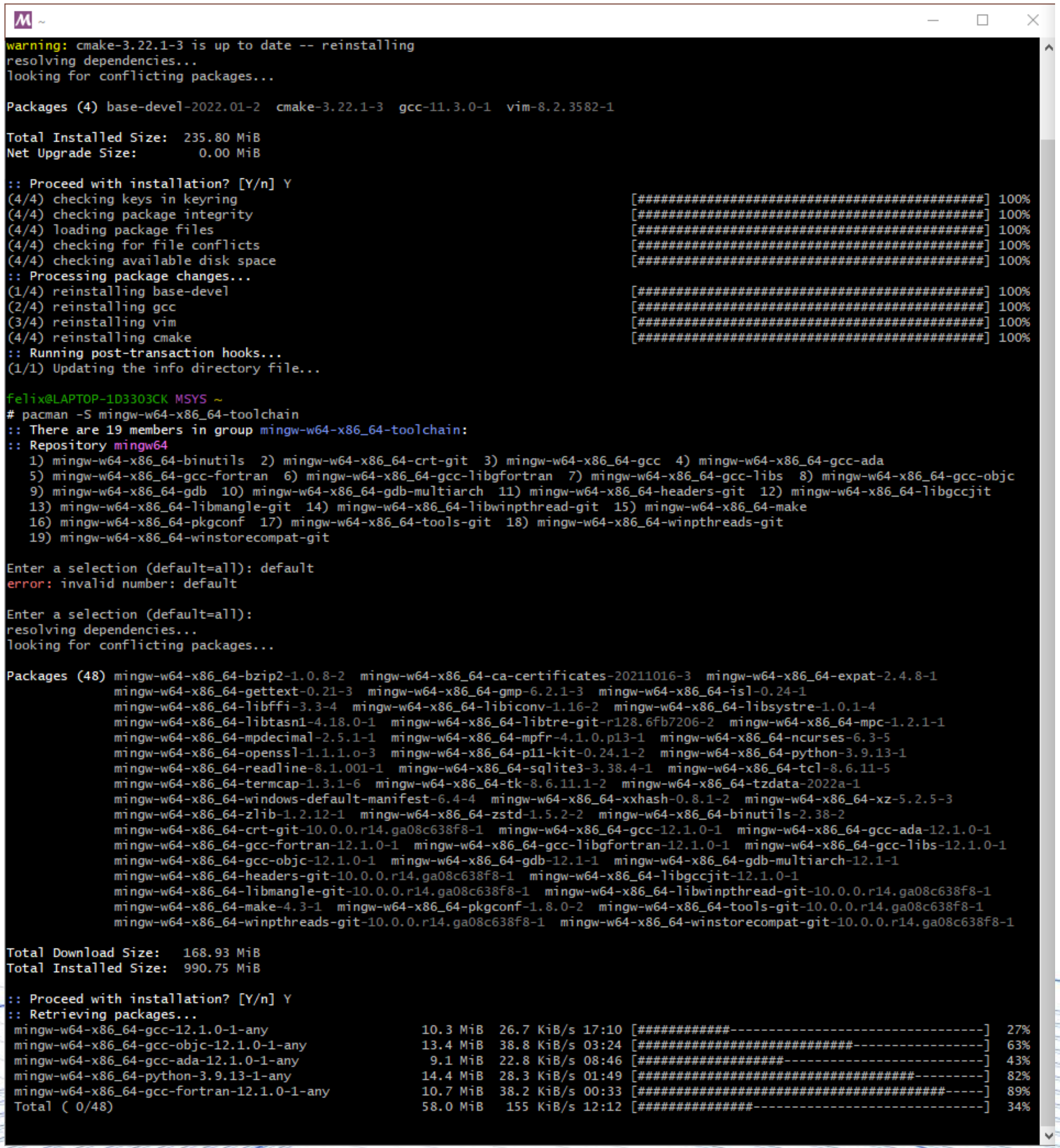
- Install MSYS2 (see here for instructions).
- Alternatively, you can install mingw64. It uses even less compilers, enough for running only C. For that you need to run the following command in the MSYS2 console (read the tutorial above):



```
pacman -S mingw-w64-x86_64-toolchain
```

Appendix A

Where to program?- Offline alternative



```
warning: cmake-3.22.1-3 is up to date -- reinstalling
resolving dependencies...
looking for conflicting packages...

Packages (4) base-devel-2022.01-2  cmake-3.22.1-3  gcc-11.3.0-1  vim-8.2.3582-1

Total Installed Size:  235.80 MiB
Net Upgrade Size:        0.00 MiB

:: Proceed with installation? [Y/n] Y
(4/4) checking keys in keyring                                                   [#############################################] 100%
(4/4) checking package integrity                                                 [#############################################] 100%
(4/4) loading package files                                                      [#############################################] 100%
(4/4) checking for file conflicts                                                [#############################################] 100%
(4/4) checking available disk space                                              [#############################################] 100%
:: Processing package changes...
(1/4) reinstalling base-devel                                                    [#############################################] 100%
(2/4) reinstalling gcc                                                           [#############################################] 100%
(3/4) reinstalling vim                                                           [#############################################] 100%
(4/4) reinstalling cmake                                                         [#############################################] 100%
:: Running post-transaction hooks...
(1/1) Updating the info directory file...

felix@LAPTOP-1D3303CK MSYS ~
# pacman -S mingw-w64-x86_64-toolchain
:: There are 19 members in group mingw-w64-x86_64-toolchain:
:: Repository mingw64
   1) mingw-w64-x86_64-binutils  2) mingw-w64-x86_64-crt-git  3) mingw-w64-x86_64-gcc  4) mingw-w64-x86_64-gcc-ada
   5) mingw-w64-x86_64-gcc-fortran  6) mingw-w64-x86_64-gcc-libgfortran  7) mingw-w64-x86_64-gcc-libs  8) mingw-w64-x86_64-gcc-objc
   9) mingw-w64-x86_64-gdb  10) mingw-w64-x86_64-gdb-multiarch  11) mingw-w64-x86_64-headers-git  12) mingw-w64-x86_64-libgccjit
   13) mingw-w64-x86_64-libmangle-git  14) mingw-w64-x86_64-libwinpthread-git  15) mingw-w64-x86_64-make
   16) mingw-w64-x86_64-pkgconf  17) mingw-w64-x86_64-tools-git  18) mingw-w64-x86_64-winpthreads-git
   19) mingw-w64-x86_64-winstorecompat-git

Enter a selection (default=all): default
error: invalid number: default

Enter a selection (default=all):
resolving dependencies...
looking for conflicting packages...

Packages (48) mingw-w64-x86_64-bzip2-1.0.8-2  mingw-w64-x86_64-ca-certificates-20211016-3  mingw-w64-x86_64-expat-2.4.8-1
              mingw-w64-x86_64-gettext-0.21-3  mingw-w64-x86_64-gmp-6.2.1-3  mingw-w64-x86_64-isl-0.24-1
              mingw-w64-x86_64-libffi-3.3-4  mingw-w64-x86_64-libiconv-1.16-2  mingw-w64-x86_64-libsystre-1.0.1-4
              mingw-w64-x86_64-libtasn1-4.18.0-1  mingw-w64-x86_64-libtre-git-r128.6fb7206-2  mingw-w64-x86_64-mpc-1.2.1-1
              mingw-w64-x86_64-mpdecimal-2.5.1-1  mingw-w64-x86_64-mpfr-4.1.0.p13-1  mingw-w64-x86_64-ncurses-6.3-5
              mingw-w64-x86_64-openssl-1.1.1.o-3  mingw-w64-x86_64-p11-kit-0.24.1-2  mingw-w64-x86_64-python-3.9.13-1
              mingw-w64-x86_64-readline-8.1.001-1  mingw-w64-x86_64-sqlite3-3.38.4-1  mingw-w64-x86_64-tcl-8.6.11-5
              mingw-w64-x86_64-termcap-1.3.1-6  mingw-w64-x86_64-tk-8.6.11.1-2  mingw-w64-x86_64-tzdata-2022a-1
              mingw-w64-x86_64-windows-default-manifest-6.4-4  mingw-w64-x86_64-xxhash-0.8.1-2  mingw-w64-x86_64-xz-5.2.5-3
              mingw-w64-x86_64-zlib-1.2.12-1  mingw-w64-x86_64-zstd-1.5.2-2  mingw-w64-x86_64-binutils-2.38-2
              mingw-w64-x86_64-crt-git-10.0.0.r14.ga08c638f8-1  mingw-w64-x86_64-gcc-12.1.0-1  mingw-w64-x86_64-gcc-ada-12.1.0-1
              mingw-w64-x86_64-gcc-fortran-12.1.0-1  mingw-w64-x86_64-gcc-libgfortran-12.1.0-1  mingw-w64-x86_64-gcc-libs-12.1.0-1
              mingw-w64-x86_64-gcc-objc-12.1.0-1  mingw-w64-x86_64-gdb-12.1-1  mingw-w64-x86_64-gdb-multiarch-12.1-1
              mingw-w64-x86_64-headers-git-10.0.0.r14.ga08c638f8-1  mingw-w64-x86_64-libgccjit-12.1.0-1
              mingw-w64-x86_64-libmangle-git-10.0.0.r14.ga08c638f8-1  mingw-w64-x86_64-libwinpthread-git-10.0.0.r14.ga08c638f8-1
              mingw-w64-x86_64-make-4.3-1  mingw-w64-x86_64-pkgconf-1.8.0-2  mingw-w64-x86_64-tools-git-10.0.0.r14.ga08c638f8-1
              mingw-w64-x86_64-winpthreads-git-10.0.0.r14.ga08c638f8-1  mingw-w64-x86_64-winstorecompat-git-10.0.0.r14.ga08c638f8-1

Total Download Size:   168.93 MiB
Total Installed Size:  990.75 MiB

:: Proceed with installation? [Y/n] Y
:: Retrieving packages...
 mingw-w64-x86_64-gcc-12.1.0-1-any                  10.3 MiB  26.7 KiB/s 17:10 [############---------------------------------]  27%
 mingw-w64-x86_64-gcc-objc-12.1.0-1-any             13.4 MiB  38.8 KiB/s 03:24 [############################-----------------]  63%
 mingw-w64-x86_64-gcc-ada-12.1.0-1-any               9.1 MiB  22.8 KiB/s 08:46 [###################--------------------------]  43%
 mingw-w64-x86_64-python-3.9.13-1-any               14.4 MiB  28.3 KiB/s 01:49 [#####################################--------]  82%
 mingw-w64-x86_64-gcc-fortran-12.1.0-1-any          10.7 MiB  38.2 KiB/s 00:33 [########################################-----]  89%
 Total ( 0/48)                                      58.0 MiB   155 KiB/s 12:12 [###############------------------------------]  34%
```

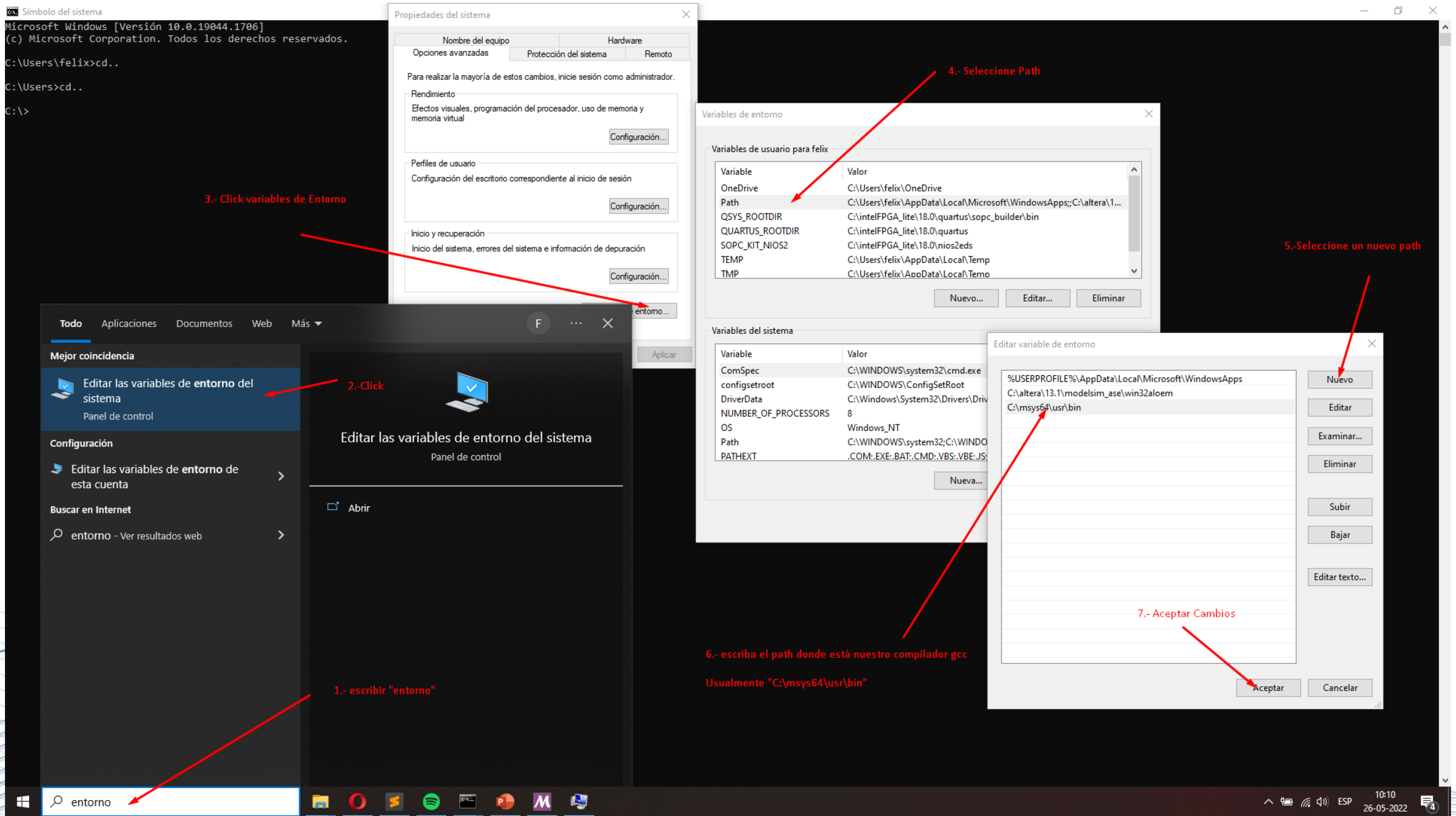
Installing gcc from MSYS

Appendix A

Where to program?- Offline alternative

3.- **Link the gcc compiler to our text editor**. Thereby, the text editor know where the compiler is and our code can be compiled.

3.1.- First we must say to windows that recognizes gcc. See this link from minute 8:45. The configure the system as showed in the picture.



Símbolo del sistema

```
Microsoft Windows [Versión 10.0.19044.1706]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\felix>cd..

C:\Users>cd..

C:\>
```

Propiedades del sistema

Para realizar la mayoría de estos cambios, inicie sesión como administrador.

Rendimiento: Efectos visuales, programación del procesador, uso de memoria y memoria virtual

Perfiles de usuario: Configuración del escritorio correspondiente al inicio de sesión

Inicio y recuperación: Inicio del sistema, errores del sistema e información de depuración

Variables de entorno

Variables de usuario para felix

Variable	Valor
OneDrive	C:\Users\felix\OneDrive
Path	C:\Users\felix\AppData\Local\Microsoft\WindowsApps;;C:\altera\1...
QSYS_ROOTDIR	C:\intelFPGA_lite\18.0\quartus\sopc_builder\bin
QUARTUS_ROOTDIR	C:\intelFPGA_lite\18.0\quartus
SOPC_KIT_NIOS2	C:\intelFPGA_lite\18.0\nios2eds
TEMP	C:\Users\felix\AppData\Local\Temp
TMP	C:\Users\felix\AppData\Local\Temp

Variables del sistema

Variable	Valor
ComSpec	C:\WINDOWS\system32\cmd.exe
configsetroot	C:\WINDOWS\ConfigSetRoot
DriverData	C:\Windows\System32\Drivers\Driv
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDO
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;

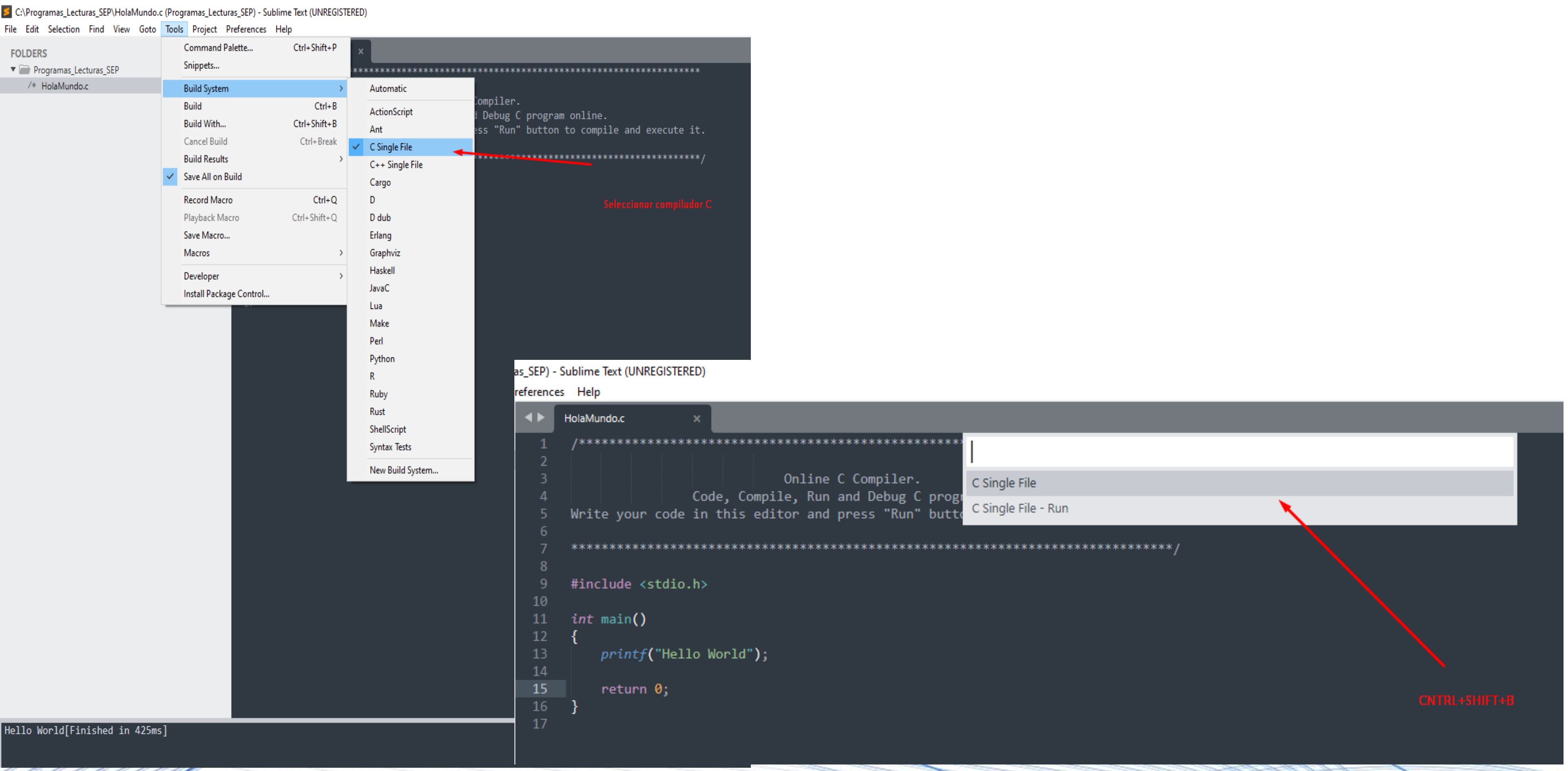
Editar variable de entorno

```
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\altera\13.1\modelsim_ase\win32aloem
C:\msys64\usr\bin
```

Appendix A

Where to program?- Offline alternative

3.2.- Now We are able to link our gcc to the text editor. For this, opens a new file named “HolaMundo.c” can be downloaded from our repository.
3.3.- Go to tools->build System-> C single File. (see picture)
3.4.- Execute cntrl+shif+B to choose between compile + ejectute or cntrl+B to only compile. Done! (see picture)



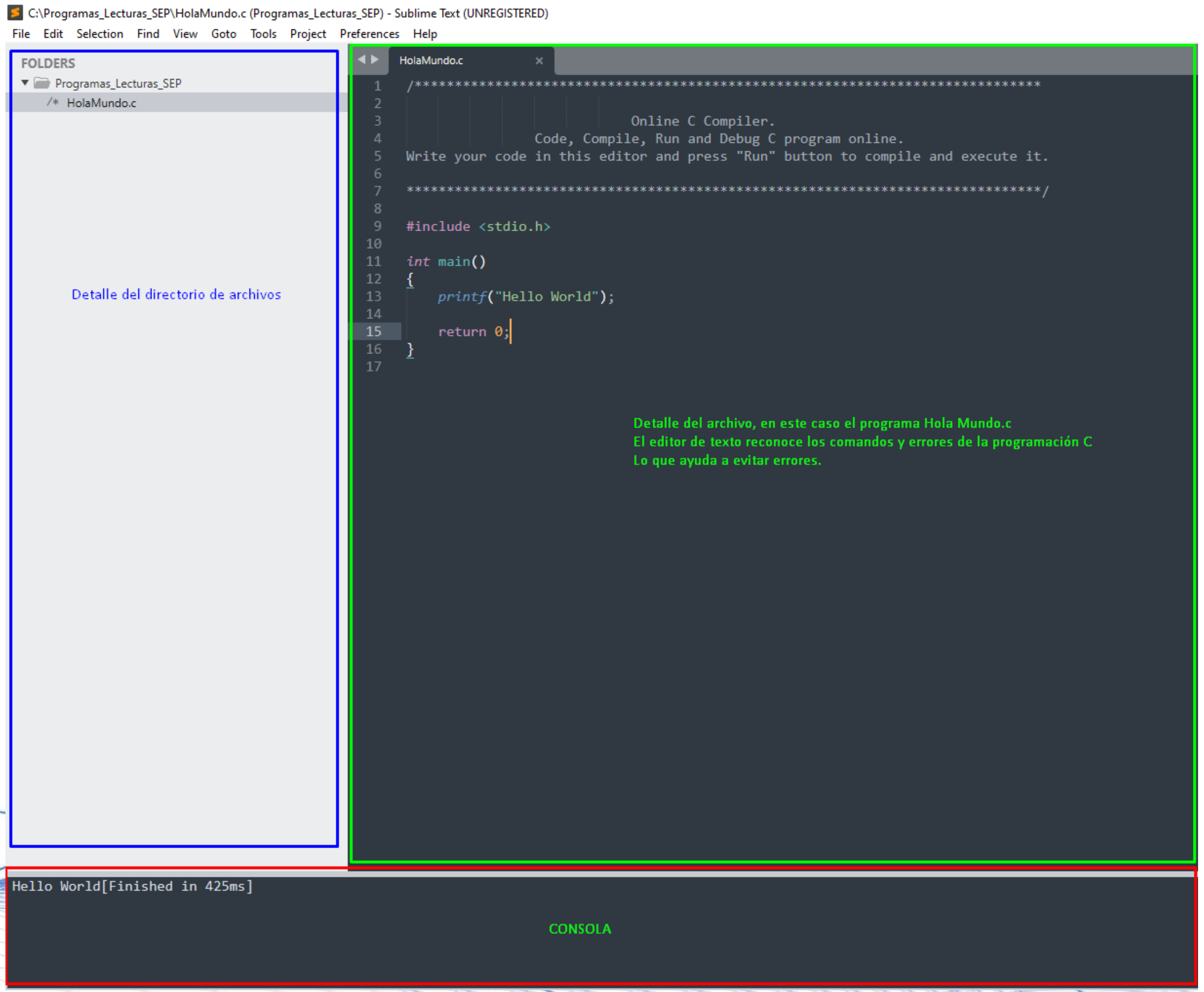
```
/*****************************************************
                              Online C Compiler.
                Code, Compile, Run and Debug C prog
Write your code in this editor and press "Run" butt

*******************************************************************************/

#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

```
Hello World[Finished in 425ms]
```

Appendix A

Where to program?- Offline alternative

4.- Once the code is executed, we see the outputs of the code. We can also write inputs through console.



C:\Programas_Lecturas_SEP\HolaMundo.c (Programas_Lecturas_SEP) - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
Programas_Lecturas_SEP
HolaMundo.c
HolaMundo.c

Detalle del directorio de archivos

```c
/*******************************************************************************

                              Online C Compiler.
                Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}

```

Detalle del archivo, en este caso el programa Hola Mundo.c
El editor de texto reconoce los comandos y errores de la programación C
Lo que ayuda a evitar errores.

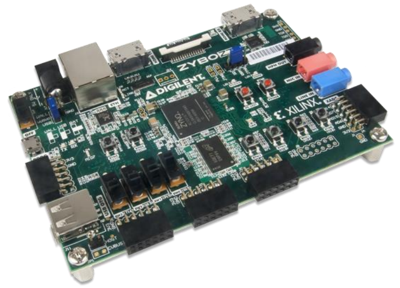
```
Hello World[Finished in 425ms]
```

CONSOLA

- libraries

End of Lecture 01 Architecture of Programmable Systems

Next Lecture – C Programming



peclab

POWER & ENERGY CONVERSION LABORATORY

Electrical Engineering Department

Pontificia Universidad Católica de Chile

peclab.ing.uc.cl