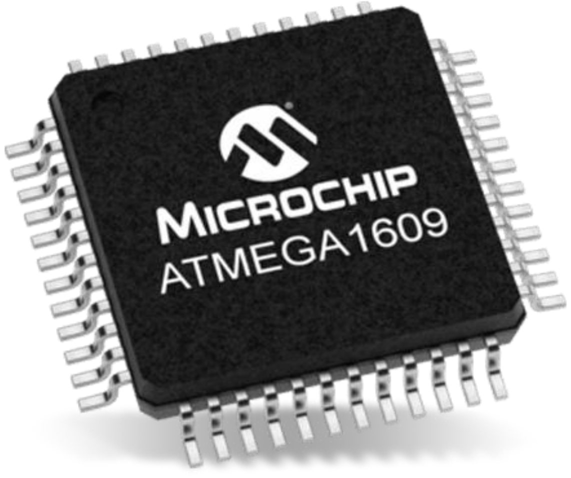
IEE 2463
Programmable Electronic Systems

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

```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.

Important details for the syntaxis:

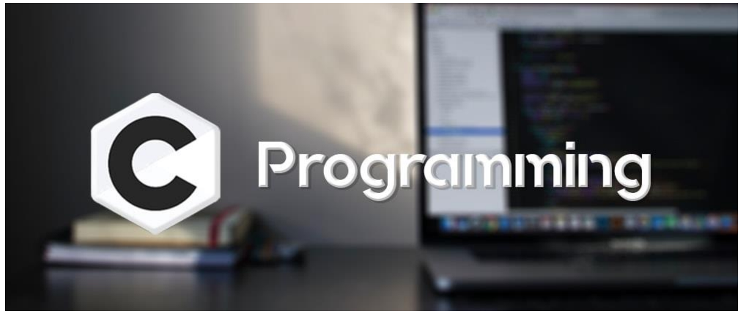
1. To end one instruction the symbol **;** is **mandatory.**
2. Functions content are delimited by brackets **{}**. These brakers are needed inly when we define/describe/create the function, but not when we invoke it.
3. Functions arguments are delimited by brackets **().**

POWER & ENERGY CONVERSION LABORATORY

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

Basic Concepts
C Programming Language

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



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 see this later.

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.
```

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

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
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.

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

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.

HolaMundo.c   definiciones.h

```c
}
*/

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

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

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

```c
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 );
```

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

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

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

```c
#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.

```c
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:

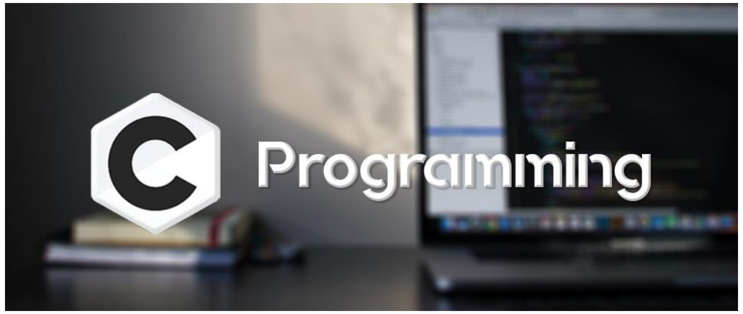
long double
↑
double
↑
float
↑
unsigned long long
↑
long long
↑
unsigned long
↑
long
↑
unsigned int
↑
int

```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

Macro Definitions
Preprocessors and Header Files

Definition, operators, header files.



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:

```c
#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.**

```c
#define MAX_ARRAY_LENGTH 20
```

Replace MAX_ARRAY_LENGTH with 20.

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

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

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

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

```c
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__ );

}
```

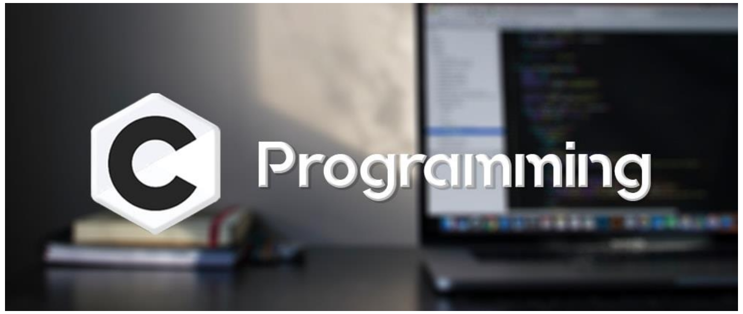
```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

POWER & ENERGY CONVERSION LABORATORY

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

Pointers
C Programming Language

Definition, use, arithmetic and its use in functions.



Pointers

Overview

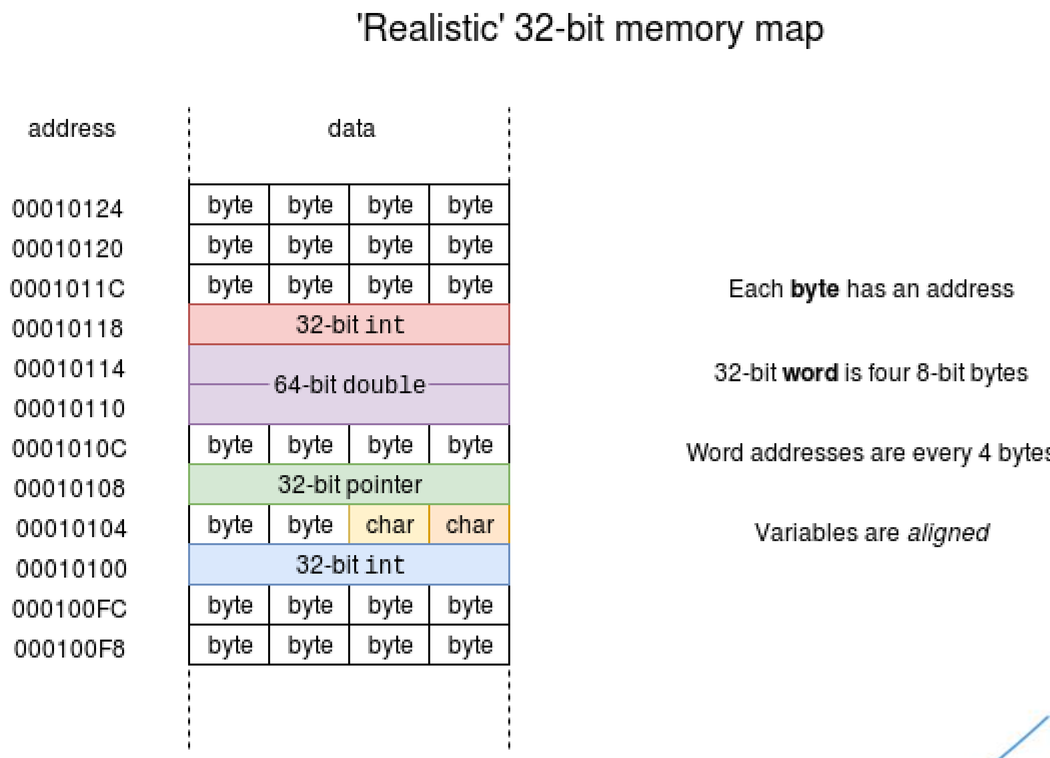
- A **pointer is** a variable whose value **is the address of another variable**.
- The use of **pointers are mandatory for memory management tasks**. So, we must learn it!
- Remember that every variable uses a piece of memory, which has its own address using & operator.
- There are three important steps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

Pointers

Example

```c
#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
```

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 NULL 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 */

Void Pointers

- A pointer in a program that isn’t associated with a data type is known as a void pointer in C. The void pointer points to the data location.
- **Pointer definition:** type *variable
  - void *ptr;
- The **void pointer in C is a pointer that is not associated with any data types**. It points to some data location in the storage. This means that it points to the address of variables. It is also called the general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers.
- A **void Pointer is capable of holding any data type address in the program**. Then, these void pointers that have addresses, can be further typecast into other data types very easily.
- The pointer to void **can be used in generic functions in C because it is capable of pointing to any data type**. One can assign the void pointer with any data type’s address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting**. So, it reduces complications in a code.**
-

link

Pointers

Void Pointers-Examples

```
#include<stdlib.h>
    int main() {
    int v = 7;
    float w = 7.6;
    void *u;
    u = &v;
    printf("The Integer variable in the program is equal to = %d", *( (int*) u) );
    u = &w;
    printf("\nThe Float variable in the program is equal to = %f", *( (float*) u) );
    return 0;
    }
```

The output obtained out of the program would be:

The Integer variable in the program is equal to = 7

The Float variable in the program is equal to = 7.600000

Pointers

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
```

These operations fails in void pointers, as they dont have any type!. We need to csat them

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

```
#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;
}
```

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

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.

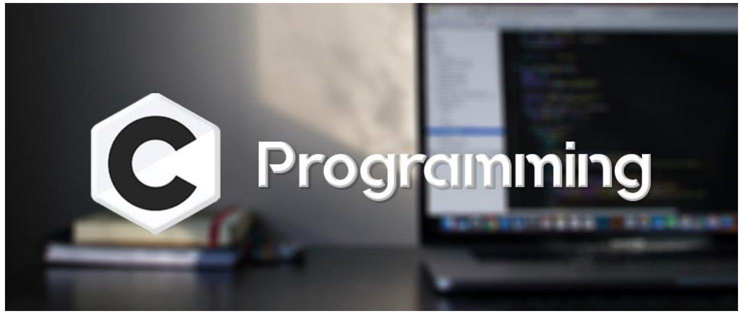
Exercises

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.

Functions
C Programming Language

Definition, declaration, call and arguments of a function.



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.

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.

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
```

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
```

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?

POWER & ENERGY CONVERSION LABORATORY

Electrical Engineering Department
Pontificia Universidad Católica de Chile
peclab.ing.uc.cl