# SISTEMAS OPERATIVOS

3004610 - 1

**German Sánchez Torres, I.S., M.Sc., Ph.D.**

Profesor, Facultad de Ingeniería - Programa de Sistemas

Universidad del Magdalena, Santa Marta.

Phone: +57 (5) 4214079 Ext 1138 - 301-683 6593

Edificio Docente, Cub 3D401.

Email: sanchez.gt@gmail.com –gsanchez@unimagdalena.edu.co


# Escribiendo programas en lenguaje C

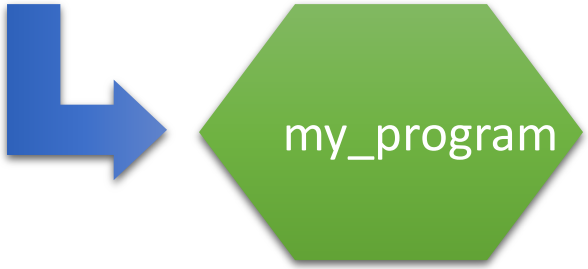
SISTEMAS OPERATIVOS

3004610 - 1

# Escribiendo y Ejecutando Programas

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

```
$ gcc -Wall -g my_program.c -o my_program
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and separately
tt.c:8: `x' undeclared (first use in this function)
tt.c:8: (Each undeclared identifier is reported only once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-void function
tt.c: At top level:
tt.c:11: parse error before `return'
```




1. Escriba el texto del programa(*source code*) usando un editor de textos, guarde el archivo con extensión .c, e.g. **my_program.c**

2. Ejecute el compilador para convertir el código fuente en un "executable" or "binario":
**gcc my_program.c –o my_program**

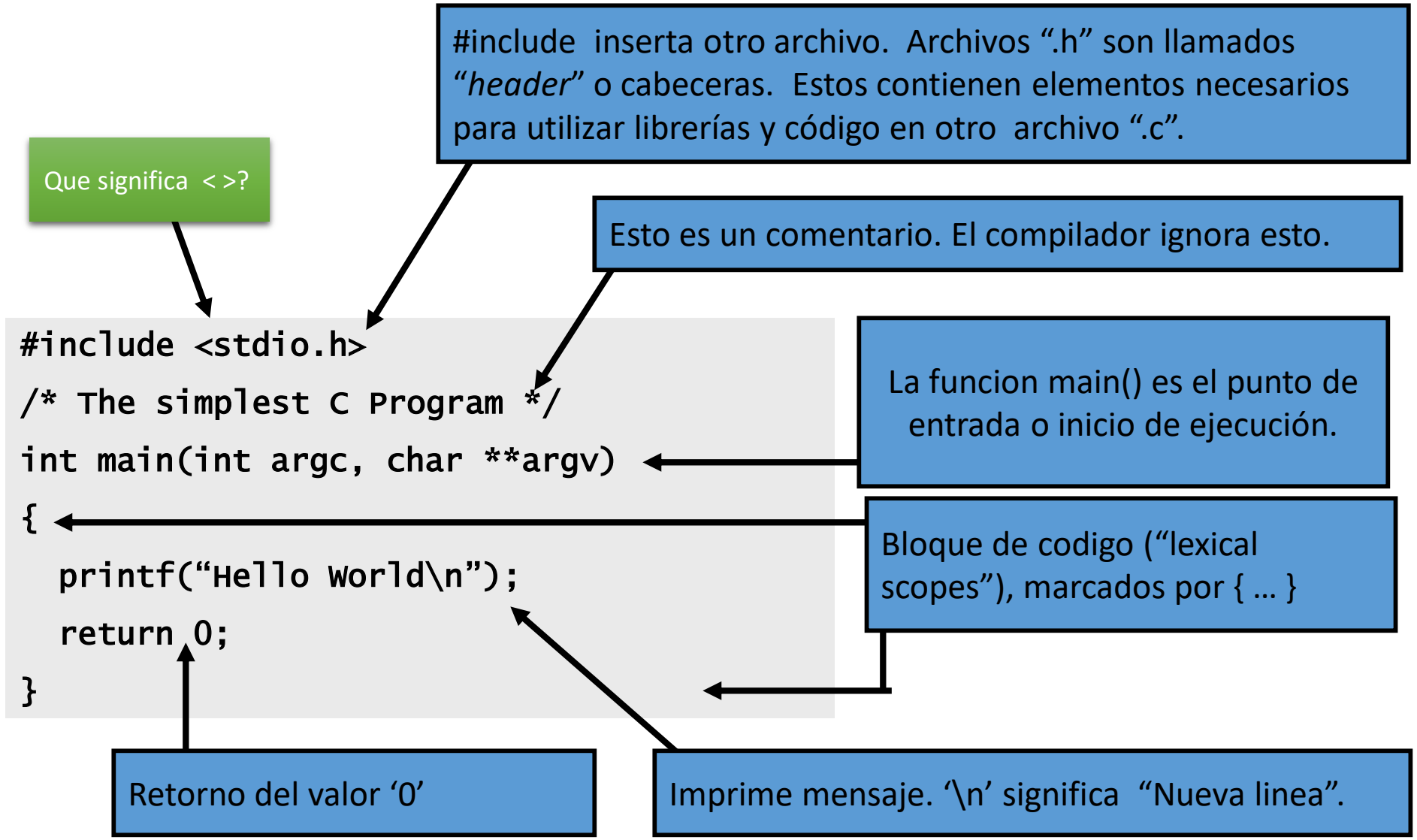
3-N. El compilador genera errores y advertencias; edite el archivo fuente, corrija los errores, y re-compile

N+1. Ejecute y observe si éste funciona ☺

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

./ ?

# Sintaxis C y “Hello World”



# About the Compiler

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

**Preprocess**

```
__extension__ typedef  unsigned long long int   __dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int   __ino_t;
__extension__ typedef  unsigned long long int   __ino64_t;
__extension__ typedef  unsigned int   __nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int   __off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

**Compile**

my_program

Compilation occurs in two steps:
"Preprocessing" and "Compiling"

In Preprocessing, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the Preprocessor.

- Include files are "pasted in" (#include)
- Macros are "expanded" (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \ )

\ ?

The compiler then converts the resulting text into binary code the CPU can run directly.

# Variables?

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

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

declare vs define?

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

Initial value of x is undefined

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?

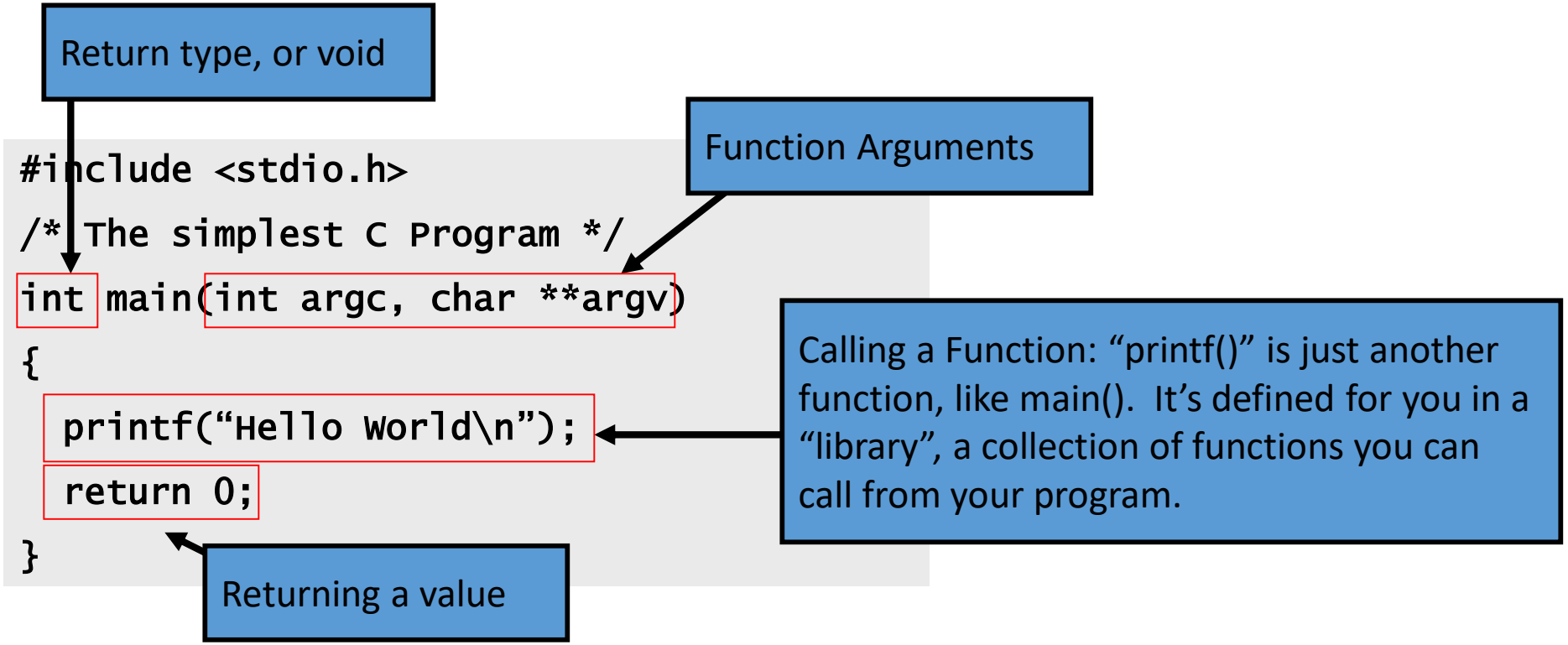
| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | ? |
| y | 5 | 'e' (101) |
| | 6 | |
| | 7 | |
| | 8 | |
| | 9 | |
| | 10 | |
| | 11 | |
| | 12 | |

The compiler puts them somewhere in memory.

# Functions

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

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



# Recursión

"if" statement

```
/* if evaluated expression is not 0 */
if (expression) {
  /* then execute this block */
}
else {
  /* otherwise execute this block */
}
```

Need braces?

X ? Y : Z

Short-circuit eval?

detecting brace errors

Tracing "pow()":

- What does pow(5,0) do?
- What about pow(5,1)?
- "Induction"

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

float pow(float x, uint32_t exp)
{
  /* base case */
  if (exp == 0) {
    return 1.0;
  }

  /* "recursive" case */
  return x*pow(x, exp - 1);
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf("p = %f\n", p);
  return 0;
}
```

# Comparison and Mathematical Operators

```
== equal to
<  less than
<= less than or equal
>  greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
!  logical not
```

```
+  plus
-  minus
*  mult
/  divide
%  modulo
```

```
&  bitwise and
|  bitwise or
^  bitwise xor
~  bitwise not
<< shift left
>> shift right
```

Beware division:

- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause a FPE (float point exception)

Don't confuse & and &&..
1 & 2 → 0 *whereas* 1 && 2 → <true>

# Assignment Operators

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

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

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

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

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

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

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

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

# Los ciclos

Solution: “while” loop.

```
loop:
  if (condition) {
    statements;
    goto loop;
  }
```

```
while (condition) {
  statements;
}
```

```
For (init; condition; incre)
{
  statements;
}
```

```
do{
 statements;
}
while (condition);
```

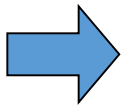
```
float pow(float x, uint exp)
{
  int i=0;
  float result=1.0;
  while (i < exp) {
    result = result * x;
    i++;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf(“p = %f\n”, p);
  return 0;
}
```

# The “for” loop

```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  i=0;
  while (i < exp) {
    result = result * x;
    i++;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf(“p = %f\n”, p);
  return 0;
}
```



```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf(“p = %f\n”, p);
  return 0;
}
```

# “Pointers”

This is exactly how “pointers” work.

```
“address of” or reference operator:  &
“memory_at” or dereference operator: *
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

A “pointer type”: pointer to char

```
void f(char * p)
{
  *p = *p - 32;
}
```

```
char y = 101;       /* y is 101 */
f(&y);              /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

# Estructuras

```
#include <sys/time.h>

/* declare the struct */
struct my_struct {
  int counter;
  float average;
  struct X x_element;
};

/* define an instance of my_struct */
struct my_struct x ;

x.counter = 1;
x.average = sum / (float)(x.counter);

struct my_struct * ptr = &x;
ptr->counter = 2;
(*ptr).counter = 3;  /* equiv. */
```

# Dynamic Memory Allocation

So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires dynamic allocation.

sizeof() reports the size of a type in bytes

For details:
$ man calloc

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf("can't allocate %d ints: %m\n", requested_count);
    return NULL;
  }

  /* now big_array[0] .. big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can't alloc

%m ?

Emstar tips

It's OK to return this pointer. It will remain valid until it is freed with free()