

INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

BREVE INTRODUCCIÓN AL LENGUAJE C

UNIVERSIDAD DE MURCIA

EL LENGUAJE DE PROGRAMACIÓN C

ALGUNAS CARACTERÍSTICAS

- Es un lenguaje para la programación estructurada.
- Es un lenguaje tipificado aunque no tanto como puede ser Pascal.
- Un programa en C es una colección de funciones, que pueden devolver un valor o no (= procedimiento), y que se encuentran distribuidas en varios ficheros o módulos. Entre todas las funciones, debe existir una llamada “main” que constituye la función principal.
- Permite escribir operaciones relativamente complejas de forma sencilla y concisa (aunque a veces esto puede ser un inconveniente pues hay programas en C que son excesivamente crípticos).
- Contiene muy pocas palabras reservadas. No contiene órdenes para trabajar con objetos compuestos (cadenas, arrays o arreglos, registros,...). Tampoco tiene instrucciones propias para la asignación dinámica de memoria ni instrucciones de entrada/salida. Todas estas operaciones de alto nivel pueden ser realizadas por funciones llamadas explícitamente.
- Distingue entre mayúsculas y minúsculas.

TIPOS, OPERADORES Y EXPRESIONES

- Al declarar una variable debemos especificar su tipo y, opcionalmente, su valor inicial.
- Tipos base del lenguajes: *int*, *float*, *double* y *char*. Variaciones aplicables a los tipos base: *short*, *long*, *unsigned* y *register*. Por ejemplo, *short int* se puede utilizar para declarar variables enteras de 2 bytes, *int* para las de 4 bytes y *long int* para declarar variables enteras de 8 bytes (cuidado, esto depende del compilador de C que estemos utilizando).
- Tipos de constantes: *int* (-345), *long* (456L), *float* y *double* (-1.23E-1), entero hexadecimal (0x4F6 o 0x4F6L si es además *long*), entero octal (0457 o 0457L si además es *long*), carácter ('a'), caracteres no imprimibles (\t, \n, \b, \0 (nulo), \\, \', \014...)
- Cadenas: “hola mundo”. Técnicamente es una array de caracteres cuyo último elemento es un carácter nulo (\0). Atención, 'x' <> “x” ya que “x”≡{'x', '\0'} por ser una cadena.
- Operadores aritméticos: +, -, *, / y % (no para *float* o *double*).
- Operadores relacionales y lógicos: >, >=, <, <=, ==, !=, ! (negación), && (AND) y || (OR).

- Cuando en una expresión aparecen varios tipos mezclados, se producen las siguientes conversiones automáticas de tipos:
 1. *char* y *short* se convierten en *int*, y *float* en *double*.
 2. Si algún operando es de tipo *double*, el otro se convierte en *double* y el resultado es de tipo *double*.
 3. Si no se aplica la regla anterior, y si algún operando es de tipo *long*, el otro se convierte en *long* y el resultado es *long*.
 4. Si no se aplica la regla anterior, y si algún operando es de tipo *unsigned*, el otro se convierte en *unsigned* y el resultado es *unsigned*.
 5. Si no se puede aplicar ninguna de las reglas anteriores, los operandos deben ser de tipo *int* y el resultado será *int*.
- Conversiones en asignaciones: el valor de la parte derecha se convierte al tipo de la izquierda, que es el tipo del resultado.
- Conversión explícita (casting): *sqrt ((double) n)*. Si *n* es de tipo *int*, el resultado de la expresión *(double) n* es de tipo *double*, que es el tipo del parámetro que espera la función *sqrt*.
- Operadores de incremento y decremento (sólo para enteros): *x=n++*; *x=++n*; *x=n--*; *x=--n*; La operación *++n* primero incrementa y devuelve ese valor, mientras que la operación *n++* devuelve el valor de *n* y luego lo incrementa.
- Operadores para manejo de bits (no para *float* o *double*): *&* (y), *|* (o), *^* (o exclusivo), *<<* (rotación o desplazamiento a la derecha), *>>* (rotación o desplazamiento a la izquierda) y *~* (complemento a 1: cambiar unos por ceros y viceversa).
- Operadores y expresiones de asignación:
 - $e1\ op = e2 \Rightarrow e1 = (e1)\ op\ (e2)$, donde *op* puede ser *+*, *-*, ***, */*, *%*, *<<*, *>>*, *&*, *|* y *^*.
 - Ejemplos: $i\ +=\ 2 \Rightarrow i = i + 2$; $x\ * =\ y + 1 \Rightarrow x = x * (y + 1)$;
- Operador ternario: *e1 ? e2: e3*. Esta expresión devuelve el resultado de *e2* si el valor de *e1* es distinto de 0; en caso contrario, devuelve el resultado de *e3*.
- Todos estos operadores tienen una precedencia y un orden de evaluación. Por eso, en las expresiones es recomendable utilizar cuantos paréntesis sean necesarios para expresar sin ambigüedad lo que queremos hacer.
- Ejemplos de definición de variables (las definiciones suelen aparecer al principio del cuerpo de la función o fuera de todas las funciones):

```
int un_entero, otro_entero = -540;
float un_float=-5.48E-2;
double un_double=-5.48E-2;
long int un_long=456L;
short int un_short=56;
unsigned long int un_unsigned_long=0X4F6L; /*En hexadecimal*/
unsigned short int un_unsigned_short=0117; /*En octal*/
char un_caracter='2', otro_caracter='t'
unsigned char un_byte;
```

- Ejemplos de expresiones aritméticas:

```
un_entero = un_caracter - '0';
otro_entero = 7/2 + (5%3 - 5)*2;
un_entero = 7/2;
un_float = 7.0/2.0;
un_double=cuadrado((double) un_entero);
```

- Ejemplos de operadores de incremento y decremento:

```
un_entero = 5;
otro_entero = ++un_entero; /*otro_entero = 6 y un_entero = 6 */
otro_entero = un_entero++; /*otro_entero = 6 y un_entero = 7 */
```

- Ejemplos de operaciones a nivel de bits:

```
un_byte = 5;
un_byte = un_byte << 1; /*un_byte=10*/
un_byte = un_byte >> 2; /*un_byte=2*/
un_byte = un_byte | 1; /*un_byte=3*/
un_byte = un_byte & (~2); /*un_byte=1*/
un_byte = un_byte ^ 2; /*un_byte=3*/
```

- Ejemplos de asignaciones y operadores:

```
un_byte ^= 2; /*equivale a un_byte = un_byte ^ 2;*/
un_entero *= 3; /*equivale a un_entero = un_entero * 3;*/
un_entero *= otro_entero + 2; /*equivale a
un_entero = un_entero * (otro_entero + 2);*/
```

- Ejemplo de operador ternario:

```
otro_entero = (un_entero > 10)?10:un_entero;
```

CONTROL DE FLUJO

- Propositiones y bloques: las proposiciones (sentencias) en C terminan en “;”. Con las llaves { y } se agrupan declaraciones y sentencias en una proposición compuesta o bloque. Una expresión se evalúa como “cierta” si su resultado es distinto de 0 y como “falsa” si su valor es 0.

- If-else:

```
if (expresión) ← ¡ Obsérvense los paréntesis !
    proposición_1
else
    proposición_2
```

- Else-if

```

if (expresión)
    proposición_1
else if (expresión)
    proposición_2
else if (expresión)
    proposición_3
.....

else
    proposición_n

```

- *switch* (expresión) {
 - case* "etiqueta":
 - [proposición]
 - case* "etiqueta":
 - [proposición]
 - default*:
 - [proposición]

Cada etiqueta debe ser un entero, constante de carácter o una expresión constante. *default* es opcional. Cuidado, cuando *expresión* coincide con una etiqueta, se ejecutan todas las instrucciones que hay a partir de ese caso y no salimos del *switch* hasta que encontramos una orden *break*, un *return* o llegamos al final del *switch*.

- *while* (expresión)
 - proposición
- *for* (expr1; expr2; expr3)
 - proposición

Esta expresión equivale conceptualmente a

```

expr1;
while (expr2) {
    proposición
    expr3;
}

```

Tanto *expr1*, *expr2* como *expr3* pueden contener varias sentencias separadas por “,”.

- *do*
 - proposición
- while* (expresión);

- *break* y *continue* en bucles: *break* hace abandonar el bucle mientras que *continue* fuerza la siguiente iteración del bucle (saltando a la parte de condición en *while* y *do_while* y a la etapa de actualización (*expr3*) en el *for*).

- Ejemplo de *if-else*:

```
int factorial (int n) {
    if (n < 2)
        return 1;
    else
        return n * factorial(n-1);
}
```

- Ejemplo de *while*:

```
int factorial (int n) {
    int resultado = 1;
    while (n > 1)
        resultado = (n--)*resultado;
    return resultado;
}
```

- Ejemplo de *for*:

```
int factorial (int n) {
    int resultado;
    for(resultado=1; n > 1; n--)
        resultado = n*resultado;
    return resultado;
}
```

- Ejemplo de *switch*:

```
int es_vocal(char c) {
    int resultado;
    switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            resultado = 1;
            break;
        default:
            resultado = 0;
    }
    return resultado;
}
```

FUNCIONES Y MÓDULOS

- Todas las funciones devuelve por defecto un valor entero. Si devuelven otro tipo de valor o no devuelven nada, debemos especificarlo en su definición. Por ejemplo, una función que devuelve un *double* se declararía como

```
double funcion(declaración de argumentos)
{
    cuerpo de la función que debe incluir, al menos, un return
}
```

- Para poder utilizar una función dentro de otra, debemos conocer su prototipo. Por ello, los prototipos de las funciones se suelen escribir al comienzo de un módulo, para que sean conocidos por todas las funciones de ese módulo. Un prototipo especifica el número y tipo de los argumentos de una función y el valor devuelto por dicha función. Ejemplos:

```
int factorial(int);
void imprime(char *);
```

- Entre todas las funciones que componen un programa (repartidas en varios módulos), debe existir una (y sólo una) llamada *main*. Esta es la función principal en la que comienza la ejecución del programa.
- Argumentos de funciones: siempre se realiza paso por valor, nunca por referencia. Para hacer algo similar al paso por referencia \Rightarrow punteros.
- Módulo: es un fichero de texto que contiene la implementación de funciones.
- Ámbito de validez de una variable:

- * Variables definidas dentro de una función \Rightarrow Dentro de esa función.
- * Variables definidas fuera de las funciones (variables globales) \Rightarrow Se conocen en todas las funciones que aparecen después de la definición. Estas variables se pueden utilizar en otros módulos con el uso de *extern*. Por ejemplo, si en un módulo definimos una variable como “*int resultado;*”, en otro módulo podemos utilizar dicha variable declarándola como “*extern int resultado;*”.
- * Variables estáticas. Se definen anteponiendo en su declaración la palabra *static* (por ejemplo, “*static int resultado;*”). Si se definen fuera de las funciones se conocen a partir de ahí en el módulo y sólo en ese módulo (son privadas al módulo). Si se definen dentro de una función, sólo se conocen en esa función y su valor permanece entre ejecuciones de la función.

- También las funciones que se encuentren precedidas por la palabra “*static*” será privadas al módulo donde se definen.

- Todas las variables se pueden inicializar en su declaración. Por ejemplo:

```
int matriz[2][4] = {{4,2,-1,3},{3,6,9,-10}};
int matriz[][4] = {{4,2,-1,3},{3,6,9,-10}}; /* No ponemos el 2.*/
char mensaje[] = “hola mundo”;
```

- En C, podemos hacer llamadas recursivas
- Inclusión de archivos: `#include <fichero>` o `#include "fichero"` (este segundo caso se utiliza cuando el fichero a incluir se encuentra en el mismo directorio que el módulo). Por ejemplo, si se quiere realizar entrada/salida en el programa, debe aparecer una línea como:

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

Antes de la compilación, esta sentencia es sustituida por el contenido del fichero `stdio.h`.

Los ficheros terminados en `.h` se denominan ficheros cabecera y suelen contener declaraciones de variables, estructuras y prototipos (de funciones definidas en otros módulos que queremos utilizar en nuestro módulo). Pueden contener código pero esto puede dar lugar a problemas, por lo que no es recomendable.

- Macros: se definen con `#define`. Antes de realizar la compilación del programa, se sustituyen por su definición. Se suelen utilizar para dar claridad al programa y para declarar "constantes" (aunque no son verdaderas constantes). Por ejemplo:

```
#define TRUE 1
#define FALSE 0
#define max(a,b) ((a) > (b) ? (a):(b))
```

- Asociación de conceptos entre Pascal y C:

PASCAL		C
UNIT nombre_unidad; INTERFACE	⇒	Fichero "Módulo.h" (se incluyen los prototipos de funciones y las definiciones de datos)
IMPLEMENTATION	⇒	Fichero "Módulo.c" (implementación de funciones y de variables)
PROGRAM Nombre_programa; USES Nombre_unidad;		<code>#include <Módulo.h></code>
Declaraciones de funciones	⇒	"prototipos y variables globales"
BEGIN		<code>void main(void) {</code>
END.		<code>}</code> Declaraciones de funciones

Ejemplos de ámbito de validez de variables y funciones:

Módulo.c

```
float un_float; /* "un_float" se conoce fuera y en todo el módulo. */
static int un_entero; /* "un_entero" se conoce en todo este módulo pero no fuera */
char una_func(char *cadena, int posicion) {
.....
}
double un_double= 5.0;
static char un_car; /* "una_func" no sabe de que existe "un_car" ni "un_double". */
static float power(float b, float e) {
.....
} /* La función "power" se conoce en todo el módulo; fuera no. */
```

Módulo.h

```
extern float un_float;
char una_func(char*,int); /*definimos el prototipo. */
extern double un_double;
float power(float,float); /* INCORRECTO. power es una función privada. */
```

Otro-módulo.c

```
#include "módulo.h"
int otra_funcion() {
    char c;

    un_float = 2.3;
    c = una_func("Buenos días",3);
}
```

PUNTEROS Y ARRAYS

- Los punteros se definen anteponiendo uno (o más) "*" al nombre de la variable:

```
int *puntero_a_int, un_entero;
puntero_a_int = &un_entero; /* Asignamos a puntero_a_int la dirección
                             de memoria de la variable un_entero. */
un_entero = 5; /* Equivale a *puntero_a_int = 5; */
```

- Los punteros permiten que las funciones puedan modificar el valor de las variables que se pasan como argumento. Por ejemplo:

<pre>void intercambiar(int x, int y) { int temp; temp = x; x = y; y = temp; }</pre>	MAL	<pre>void intercambiar(int *px, int *py) { int temp; temp = *px; *px = *py; *py = temp; }</pre>	BIEN
--	-----	--	------

y la función se llamaría como “*intercambiar(&a,&b);*”.

- En C los punteros y los arreglos son equivalentes en muchos aspectos. Por ejemplo:

```
int a[10], *pa;
pa = a;          /* “a” es una constante y, por tanto, no se puede
                  modificar su valor */
a[2]=5;          /* equivale a *(pa + 2)=5; y también, aunque abusando
                  de la notación, a *(a + 2) = 5; */
```

```
/* Función que calcula la longitud de una cadena. */
int longcad(char *s) {
    int n = 0;

    while (*s++)
        n++;
    return n;
}
```

Si se define *cadena[]* = “*Hola mundo*” se podría llamar como *longcad(cadena)* y nos devolvería “10”. También se puede pasar parte de un arreglo. Por ejemplo, *longcad(cadena+2)* que equivale a *longcad(&cadena[2]);*

A continuación, mostramos dos versiones de la misma función:

```
void copiar_cadena(char s[], char t[]) {
    int n = 0;
    while (t[n] != '\0') {
        s[n] = t[n];
        n++;
    }
}

void copiar_cadena(char *s, char *t) {
    while (*s++ = *t++);
}
```

- Operaciones con punteros: a los punteros se les puede sumar y restar enteros, se pueden incrementar o decrementar o se pueden restar entre sí (pero no sumar). También se pueden comparar con $<$, $>$, $<=$, $>=$, $=$ y $!=$. No se puede asignar enteros a punteros, aunque el 0 (que se representa por NULL) es un caso especial. Ninguna otra operación está permitida con punteros. Por ejemplo:

```
char v[] = "hola";
char *p = v;
printf("%s", p+1); /* Imprime "ola" */
p++;
printf("%s", p); /* Imprime "ola" */
--p;
printf("%s", p); /* Imprime "hola" */
```

- Arreglos multidimensionales. Se definen así: `int a[10][2]`. En C un arreglo bidimensional como el anterior se interpreta como un arreglo unidimensional donde cada uno de sus elementos es, a su vez, un arreglo unidimensional. Las siguientes definiciones son equivalentes en C

```
void inversa(float matriz[10][20]);
void inversa(float matriz[][20]);
void inversa(float (*matriz)[20]);
```

En este último caso son necesarios los paréntesis pues sin ellos quedaría `float *matriz[20]`; que dice que `matriz` es un arreglo de 20 entradas, siendo cada una de ellas un puntero a un `float`. Este significado es distinto al que pretendemos.

Si tenemos la siguiente declaración de variables

```
float *matriz[20], **pfloat;
```

se puede hacer

```
pfloat = (float **)matriz;
```

- Dos formas de definir un arreglo de cadenas

```
char meses[][20] = {"Enero", "Febrero", ....., "Noviembre", "Diciembre"}
char *meses[] = {"Enero", "Febrero", ....., "Noviembre", "Diciembre"}.
```

Esta segunda es mejor pues ocupa el espacio justo de memoria, ni más, ni menos. (¿Por qué?. Pensar en cómo se reserva memoria en cada declaración.).

- Argumentos en la línea de comandos. Cuando deseamos que nuestro programa admita parámetros al ser ejecutado, debemos declarar la función `main` de la siguiente manera


```
void main(int argc, char *argv[]);
```

`argc` indica el número de parámetros (incluyendo el nombre del programa) y `argv` es una array de cadena (siendo `argv[0]` el nombre del programa).

ESTRUCTURAS

- Posibles formas de utilización:

```
/* Declaramos la estructura. */
struct date {
    int dia;
    int mes;
    int anno;
};
```

```
/* Declaramos variables del nuevo tipo. */
struct date una_fecha, otra_fecha;
```

```
/* Lo mismo, de otra manera. */
struct date {
    int dia;
    int mes;
    int anno;
} una_fecha, otra_fecha;
```

```
/* Aquí, al eliminar el nombre de la estructura, no podremos declarar variables
cuyo tipo sea esta estructura. */
struct {
    int dia;
    int mes;
    int anno;
}una_fecha, otra_fecha;
```

- Podemos hacer operaciones como “*una_fecha.dia = 30;*”.
- Se puede inicializar estructuras externas y estáticas (igual ocurre con los arreglos). Por ejemplo: *struct date una_fecha = {26,1,1996}*.
- Si se define un puntero a una estructura, por ejemplo, *struct date *pfecha*, entonces se accede a cada campo como *(*pfecha).dia;*, aunque la forma más habitual de hacerlo es como *pfecha->dia;*
- Estructuras autoreferenciadas:

```
struct tnode {
    char *palabra;
    int apariciones;
    struct tnode *hijo_izq;
    struct tnode *hijo_der;
}
```

- Uniones. Son similares a las estructuras y se diferencian en que todos los campos comienzan en la misma dirección de memoria (y, por tanto, la comparten). Esto hace que, en un momento determinado, sólo uno de los campos tenga un valor correcto:

```
union u_datos {
    int valor_int;
    float valor_float;
    char *valor_cadena;
}
```

- *Typedef*: permite crear nuevos nombres de tipos de datos (aunque no en el sentido de la orden *Type* de Pascal). Ejemplos:

```
typedef unsigned char BYTE;
typedef char *STRING
```

y luego se pueden hacer declaraciones de variables como

```
BYTE un_byte, otro_byte=64;
STRING nombre, ciudad;
```

RESERVAR Y LIBERAR MEMORIA CON APUNTADES.

Si se definen las variables

```
int *x;
struct date {int dia, mes, anno} *pdate;
```

se les reserva memoria como

```
x = (int *) malloc(10*sizeof(int)); /* Se reserva memoria para 10 enteros */
pdate = (struct date *) malloc(sizeof(struct date)); /* una estructura */
```

y se libera la memoria asignada con

```
free(x)
free(pdate);
```

(Nota: es necesario hacer un `#include <alloc.h>` o `#include <stdlib.h>` para poder utilizar correctamente las funciones `malloc` y `free`)

ENTRADA/SALIDA

- Salida por pantalla: `printf(control, arg1, arg2,...)`. Permite formatear la salida. Por ejemplo:

```
printf("El resultado obtenido ha sido\nX = %d\n", valor)
```

que puede producir una salida como

El resultado obtenido ha sido
X = 345

Las posibles letras que pueden ir con % son:

<i>d</i>	El argumento se convierte a notación decimal.
<i>ld</i>	El argumento se convierte a notación decimal (entero largo).
<i>o, x</i>	El argumento se convierte a notación octal (sin el 0 inicial) o a notación hexadecimal (sin el 0X inicial), respectivamente.
<i>u</i>	El argumento se convierte a notación decimal sin signo.
<i>c</i>	El argumento se toma como un carácter.
<i>s</i>	El argumento es una cadena. Se puede indicar una precisión (%20s: deja un campo de 20 caracteres alineando a la derecha la cadena y rellenando con espacios si la cadena no es suficientemente ancha). Se pasa como argumento un puntero a la cadena o el nombre del array.
<i>e</i>	El argumento se toma como float o double y se escribe en notación científica.
<i>f</i>	El argumento se toma como float o double y se escribe en notación decimal de la forma [-]mmmm.nnnnn.

puts(cadena): muestra una cadena.

putchar(carácter): muestra un carácter.

- Entrada por teclado. Se puede utilizar la función *scanf*, que tiene un funcionamiento parecido a *printf*. Por ejemplo:

```
scanf("%d",&un_entero);
```

Lee un entero y lo almacena en la variable *un_entero*. Atención, si lo que se introduce no es un entero puede dar problemas. Otro ejemplo:

```
scanf("X=%d",&un_entero).
```

Para que lo lea bien es necesario escribir algo como "X=10".

gets(cadena): lee una cadena.

character=getchar(): lee un carácter.

- Ficheros

```
FILE *fp;
fp = fopen("datos.dat", "r+");
```

La segunda línea abre el fichero “*datos.dat*” para actualización (lectura/escritura), “*r+*”. Si se produce un error, *fd* valdrá *NULL*. Se pueden hacer operaciones como:

```
un_caracter = getc(fp);
putc(c,fp);
fgets(linea, MAXLINEA, fp);
fputs(linea,fp);
fprintf(fp, "El resultado es: %d\n", dato);
fscanf(fp, "%s", cadena);
fclose(fp);
```

Para poder utilizar ficheros de esta manera debemos incluir la línea `#include <stdio.h>`. Fijarse que los ficheros se ven aquí como una secuencia de caracteres.

- Otra forma de utilizar ficheros a más bajo nivel es mediante el uso de handlers. En este caso, los ficheros siempre son una serie de bytes, es decir, no tienen estructura interna. Por ejemplo:

```
int fd;

fd = open("datos.dat", O_WRONLY | O_TRUNC | O_CREAT, 0666);
write(fd, &valor, sizeof(valor));
close(fd);
```

La segunda línea abre el fichero “*datos.dat*” sólo para escritura, lo trunca si ya existe (es decir, lo deja vacío) o lo crea si no existe (con los permisos 0666, es decir, rw-rw-rw-; a estos permisos habrá que aplicarles la máscara *umask*).

La segunda línea escribe en el fichero el contenido de una zona de memoria (en este caso, la que comienza en la dirección en la que se encuentra la variable *valor*). Escribe tantos bytes como los indicados por el tercer parámetro (en nuestro caso, el tamaño de la variable *valor*). El resultado final es que se escribe en el fichero el valor de la variable.

También existe la función *lseek*, que permite colocar el puntero de lectura/escritura del fichero en una posición concreta.

Todas estas funciones de manejo de ficheros, junto con otras, permiten manejar ficheros en los que podemos guardar desde enteros hasta estructuras complejas.

EJEMPLO COMPLETO

Para terminar, se incluye un ejemplo completo de un programa en C constituido por 3 ficheros.

- Fichero “potencia.c”

```
#include <math.h>

float potencia(float b, float e) {
    return exp(e*log(b));
}
```

- Fichero “potencia.h”

```
float potencia(float,float);
```

- Fichero “princ.c”

```
#include <stdio.h>
#include <ctype.h>
#include "potencia.h"
void main(void) {
    float base, expon;
    char res;

    do {
        printf("\nIntroduzca la base: ");
        scanf("%f",&base);
        printf("Introduzca el exponente: ");
        scanf("%f",&expon);
        printf("El resultado de %f elevado a %f es: %f\n\n",
            base, expon, potencia(base,expon));
        printf("¿ Desea repetir (S/N) ? ");
        do {
            scanf("%c", &res);
            res = toupper(res);
        } while (res != 'S' && res != 'N');
    } while(res == 'S');
}
```

COMPILACIÓN

- En Linux, la compilación se realiza con la orden `gcc`. La opción “-c” permite compilar un programa (pero sin realizar el enlace final de direcciones, es decir, sin crear un ejecutable). Con esto creamos un módulo objeto. Ejemplo:

```
gcc -c potencia.c (creará el fichero potencia.o)
```

- Para construir un ejecutable, basta con no especificar la opción “-c”. La opción “-o” nos permite indicar el nombre del ejecutable (el nombre de nuestro programa). Si no la especificamos, se generará el fichero “a.out”. Ejemplo:

```
gcc -o miprograma princ.c potencia.c -lm
```

o también

```
gcc -o miprograma princ.c potencia.o -lm
```

si ya hemos compilado el fichero “*potencia.c*”. Esto creará el fichero “*miprograma*” que será nuestro programa, listo para ser ejecutado. La opción “-lm” le indica a gcc que debe incluir la librería matemática (el programa utiliza funciones matemáticas, y la librería matemática no se incluye por defecto).