Práctica 4. Introducción a *C*

1.- Objetivos

- Aprender la sintaxis de *C*.
- Escribir un programa sencillo en *C*.
- Compilar y ejecutar un programa en *C*.

2.- Programación en *C*

Estructura de un programa *C*

```
/* Esto es un comentario */

/* Directrices para el preprocesador */
#include <stdio.h>

/* Declaración de variables globales */
int year = 2011;

/* Declaración de funciones */
int main(void){
   printf("¡Hola mundo!\n");
   printf("¡Feliz año %d!\n", year);
   return 0;
}
```

Tipos de datos fundamentales

Tipos enteros:

Nombre	Descripción	Ejemplo
`char`	Carácter	`'a'`
`int`	Entero	`27`
`enum`	Tipo enumerado	

Tipos reales:

Nombre	Descripción	Ejemplo
`float`	Precisión simple	`3.1416`
`double`	Precisión doble	`3.141592653589`

Otros:

Nombre	Descripción
`void`	Conjunto vacío

Tipos derivados

- Arrays.
- Estructuras.
- Uniones.
- Punteros.

Declaración de variables

Sintaxis: `tipo identificador[, identificador];`

Ejemplo:

```
int contador;

int main(void){
   int day, month, year;
   float suma, precio;
   /* ... */
}
```

También podemos inicializar el valor de las variables. Por ejemplo:

```
int contador = 1;

int main(void){
   int day = 20, month = 2, year = 2011;
   float suma = 0.0, precio;
   precio = 10.0;
   /* ... */
}
```

Funciones

Definición y declaración. Sintaxis:

```
tipo_resultado nombre_funcion([parametros]) {
   /* Declaración de variables locales */
   /* Sentencias */
   [return [expresión]];
}
```

Paso de argumentos. Ejemplo:

```
#include <stdio.h>

void intercambio(int x, int y){
   int z = x;
   x = y;
```

```
   y = z;
}

int main(void){
   int a = 20, b = 30;
   intercambio(a, b); // Paso de parámetros por valor
   printf("a es %d y b %d\n", a, b);
   return 0;
}
```

printf()

Para poder utilizar *printf()* es necesario introducir la directiva `#include <stdio.h>`.

Sintaxis: `printf(`*`formato`*`, `*`arg`*`, `*`arg`*`...);`

`formato` es una cadena de caracteres que puede contener:

`\n`	Fin de línea.
`%d`	El siguiente argumento (numérico (`int`)).
`%c`	El siguiente argumento (carácter (`char`)).
`%s`	El siguiente argumento (cadena de caracteres).

Ejemplo:

```
printf("Me llamo %s y tengo %d años\n", "Aurelio", 27);
```

scanf()

Para poder utilizar *scanf()* es necesario introducir la directiva `#include <stdio.h>`.

Sintaxis: `scanf(tipo, &var);`

Ejemplo:

```
// Este ejemplo guarda un número en n.
int n;
printf("Introduce un numero: ");
scanf("%d",&n);

// Este ejemplo guarda un caracter en m.
char m;
printf("Introduce un caracter: ");
scanf("%c",&m);

// Este ejemplo guarda una cadena de caracteres (solamente una palabra) en cad.
```

```
// Notese la ausencia de &
char cad[20];
printf("Introduce una palabra: ");
scanf("%s",cad);

printf("Introduce una palabra: ");
scanf("%10s",cad);  // lee máximo 10 caracteres
```

Operadores

Operadores aritméticos:

`+`	Suma. Operandos enteros o reales.
`-`	Resta. Operandos enteros o reales.
`*`	Multiplicación. Operandos enteros o reales.
`/`	División. Operandos enteros o reales.
`%`	Módulo. Operandos enteros.

Ejemplo

```
int main(void) {
   int a = 10, b = 3, c;
   float x = 2.0, y;
   y = x + a; // 12.0, float
   c = a / b; // 3, int
   c = a % b; // 1, int
   y = a / b; // 3, int. Conversión a float
   c = x / y; // 0,666667, float. Conversión a int
   return 0;
}
```

Operadores lógicos:

`a && b`	AND	1 si “a” y “b” son distintos de 0 .
`a || b`	OR	0 si “a” y “b” son iguales a 0 .
`!a`	NOT	1 si “a” es 0, 0 si es distinto de 0 .

En *C* el resultado de una operación lógica es de tipo *int* . Cualquier valor distinto de*0* se considera verdadero. El valor *0* se considera falso.

Ejemplo:

```
int main(void) {
   int p = 10, q = 0, r = 0;
   r = p && q;                /* 0, falso */
```

```
   r = p || q;                /* 1, verdadero */
   r = !p;                    /* 0, falso */
   return 0;
}
```

Operadores de relación:

`a < b`	"a" menor que "b" .
`a > b`	"a" mayor que "b" .
`a <= b`	"a" menor o igual que "b" .
`a >= b`	"a" mayor o igual que "b" .
`a != b`	"a" distinto que "b" .
`a == b`	"a" igual que "b" .

Ejemplo:

```
int main(void) {
   int x = 10, y = 0, r = 0;
   r = (x == y);  /* 0, falso */
   r = (x > y);   /* 1, verdadero */
   r = (x != y);  /* 1, verdadero */
   return 0;
}
```

Operadores de asignación:

`++`	Incremento.
`--`	Decremento.
`=`	Asignación simple.
`*=`	Multiplicación y asignación.
`/=`	División y asignación.
`+=`	Suma y asignación.
`-=`	Resta y asignación.

Ejemplo:

```
int main(void) {
  int x = 10, n = 0, i = 0;
  x++;      // Incrementa el valor de x en 1
  ++x;      // Incrementa el valor de x en 1
  x = --n;  // Primero decrementa y después asigna */
  x = n--;  // Primero asigna y después decrementa */
  i += 2;   // i = i + 2
  return 0;
```

```
}
```

Otros operadores:

`&`	*Dirección-de*. Da la dirección en memoria de su operando .
`*`	*Indirección*. Acceso a un valor, teniendo su dirección en memoria.

Precedencia de operadores:

`() [] -> .`	Izquierda a derecha.
`! ++ -- * &`	(Unarios) derecha a izquierda.
`* / %`	Izquierda a derecha.
`+ -`	Izquierda a derecha.
`<< >>`	Izquierda a derecha.
`< <= > >=`	Izquierda a derecha.
`== !=`	Izquierda a derecha.
`&&`	Izquierda a derecha.
`||`	Izquierda a derecha.
`= *= /= %= += -=`	Derecha a izquierda.
`,`	Izquierda a derecha.

Sentencias de control

Sentencia *if*:

```
if (EXPRESIÓN) {
   SENTENCIAS1
} else {
   SENTENCIAS2
}
```

Si *EXPRESIÓN* se evalúa como verdadera (esto es, su valor es distinto de 0), se ejecuta *SENTENCIAS1*. En otro caso, se ejecuta *SENTENCIAS2*.

Ejemplo:

```
int a = 10, b = 20, c = 0, menor;
if (a < b) {
   if (a < c) {
      menor = a;
   } else {
      menor = c;
   }
} else if (b < c) {
```

```
   menor = b;
} else {
   menor = c;
}
printf("Menor = %d", menor);
```

Sentencia *switch*:

```
switch (EXPRESIÓN) {
   case VALOR1:
      SENTENCIAS1;
      break;
   case VALOR2:
      SENTENCIAS2;
      break;
   /* ... */
   default:
      SENTENCIAS-DEFAULT;
}
```

Si *EXPRESIÓN* es igual a:

- *VALOR1*: Ejecuta *SENTENCIAS1*.
- *VALOR2*: Ejecuta *SENTENCIAS2*.
- …
- Ninguno de los anteriores: Ejecuta *SENTENCIAS-DEFAULT*.

Ejemplo:

```
switch (mes) {
   case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      dias = 31;
      break;
   case 4: case 6: case 9: case 11:
      dias = 30;
      break;
   case 2:
      dias = 28;
      break;
   default:
      printf("El mes no es válido\n");
      break;
}
```

Sentencia *while*:

```
while (EXPRESIÓN) {
   SENTENCIAS
}
```

Se evalúa *EXPRESIÓN*, si es verdadera (distinto de 0) se ejecuta *SENTENCIAS*. Se evalúa de nuevo *EXPRESIÓN*, si es verdadera se ejecuta *SENTENCIAS*. Así sucesivamente hasta que *EXPRESIÓN* sea falsa.

Podemos conseguir un bucle infinito con `while (1)`. Podemos usar la orden *break* en cualquier

momento para salir del bucle.

Sentencia *do-while*:

```
do  {
   SENTENCIAS
} while (EXPRESIÓN)
```

El bucle *do-while* es idéntico al bucle *while*, salvo por que en este las sentencias se ejecutan siempre al menos una vez, antes de evaluar la condición.

Ejemplo *while*, *do-while*:

```
/* ... */
while (1) {
   int op;
   do {
      printf("\t1. Sumar\n");
      printf("\t2. Restar\n");
      printf("\t3. Salir\n");
      scanf("%d", &op);
   } while (op < 1 || op > 3);
   /* ... */
}
```

Sentencia *for*:

```
for ( INICIALIZACIÓN ; CONDICIÓN ; ACTUALIZACIÓN ) {
   SENTENCIAS;
}
```

Primero se ejecuta *INICIALIZACIÓN*. Después se evalúa *CONDICIÓN* y si es verdadera se ejecuta *SENTENCIAS*. Por último, se ejecuta *ACTUALIZACIÓN*. Se evalúa de nuevo *CONDICIÓN* y repetimos el ciclo sucesivamente hasta que deje de cumplirse *CONDICIÓN*.

Un ejemplo de bucle que se repite 10 veces, usando la variable *i* (desde *i=0* hasta *i=9*) sería:

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

Arrays

Arrays unidimensionales:

```
tipo nombre[tamaño];
```

En los *arrays* de *C* el primer subíndice es 0. Ejemplo:

```
int lista[100]; /* Array de 100 elementos */

int i;
```

```
for (i=0; i<100; i++) {
   lista[i] = i+1;
   printf("lista[%d]=%d\n", i, lista[i]);
}
```

Arrays multidimensionales:

```
tipo nombre[tamaño1][tamaño2]...[tamañoN];
```

Ejemplo:

```
int matriz[10][10]; /* Array de 10x10 elementos */

int sumafila;
int i, j;
for (i=0; i<10; i++) {
   sumafila = 0;
   for (j=0; j<10; j++) {
      sumafila += matriz[i][j];
   }
   printf("Fila %d, suma =%d\n", i, sumafila);
}
```

Cadenas de caracteres:

```
char nombre[tamaño];
```

Ejemplo:

```
char asignatura[100];

strcpy(asignatura, "sistemas operativos");

asignatura[0] = "S";
printf("El nombre empieza por %c\n", asignatura[0]);
printf("Nombre de la asignatura: %s\n", asignatura);

/* .... */
```

Arrays de cadenas de caracteres:

```
char nombre[tamaño1][tamaño2]...[tamañoN];
```

Ejemplo:

```
char profesores[6][60];

/* .... */

printf("Profesores de la asignatura:\n");
int i;
```

```
for (i=0; i < 6; i++) {
   printf("\t%s\n", profesores[i]);
}
```

Estructuras

Sintaxis:

```
struct nombre_estructura {
   /* Declaración de campos */
}
```

Ejemplo:

```
struct ficha_alumno {
   char nombre[60];
   int nota;
};

struct ficha_alumno var1, var2;
struct ficha_alumno alumnos[100];

int i;
for (i=0; i<100; i++) {
   printf("%s\t\t%d\n", alumnos[i].nombre, alumnos[i].nota);
}
```

Punteros

Todas las variables se almacenan en memoria principal y tienen, por tanto, una dirección de memoria donde reside su valor. Podemos acceder a esa dirección con el operador unario *&*. Por ejemplo:

- Con `int a` declaramos una variable de tipo entero.
- Con `&a` tenemos la dirección dentro de la memoria donde reside el calor de `a`.

Una dirección de memoria se puede almacenar en un *puntero*, que se define de la siguiente manera:

- `int *p`: `p` es un *puntero* a una variable de tipo entero.
- `P = &a`: Le asignamos la dirección de la variable `a`.

El operador unario `*` se usa para acceder al contenido de una dirección de memoria. `*p` es el contendido de la dirección `p`, esto es, `a`.

En *C* podemos utilizar *punteros* para conseguir pasar argumentos por referencia.Por ejemplo:

```
#include <stdio.h>

void intercambio(int * x, int * y) {
```

```
   int z = *x;
   *x = *y;
   *y = z;
}

/* ... */

int a = 20, b = 30;
intercambio(&a, &b); /* Paso por referencia */

/* ... */
```

Un puntero es la manera más cómoda de pasarle a una función el contenido de una estructura. Por ejemplo:

```
struct ficha_alumno {
   char nombre[60];
   int nota;
};

struct ficha_alumno var1, var2;

void funcion(struct ficha_alumno * ficha) {
   printf("%s\t\t%d\n", ficha->nombre, ficha->nota);
}

/* ... */
funcion(&var1);
/* ... */
```

Reserva dinámica de memoria

La memoria en un programa en C se puede reservar de dos formas:

- *Estática*: memoria que se reserva durante la compilación de un programa.
- *Dinámica*: memoria que se reserva durante la ejecución de un programa.

Hasta ahora, en los ejemplo vistos, la memoria que hemos reservado para las variables ha sido estática. Por ejemplo, una instrucción como la siguiente reserva memoria estática para una *array* de tres números enteros:

```
int datos [3];
```

La memoria dinámica es más flexible que la estática pues se adapta a las necesidades de cada ejecución. Para reservar memoria durante la ejecución de un programa hemos de utilizar *punteros*. Si queremos reservar una tabla de enteros con *n* elementos, hemos de comenzar declarando un puntero:

```
int *p;
```

Este puntero se utiliza para guardar el valor devuelto por la función de reserva de memoria *malloc* (es necesario incluir la cabecera *stdlib.h*). Esta función recibe como argumento el número de bytes que deseamos reservar y devuelve la dirección de comienzo de la porción de memoria reservada:

```
p = malloc (3 * 4); // Reservamos memoria para almacenar 3 enteros
```

Es mejor utilizar el operador *sizeof* en lugar de 4 (tamaño en bytes de un *int*) pues un *int* puede ocupar un número distinto de bytes en otras implementaciones:

```
p = malloc (3 * sizeof (int));
```

Además, el puntero devuelto por *malloc* se ha de convertir mediante un *cast* (conversión de tipos) al tipo adecuado:

```
p = (int *) malloc (3 * sizeof (int));
```

Una vez reservada la memoria, podemos tratar a *p* como si fuera una tabla estática. Cuando no necesitemos la memoria reservada, podemos liberarla con la función *free*:

```
free(p);
```

3.- Desarrollo de la práctica

Reescribe el *script* de *shell* de la práctica 3 para convertirlo en un programa *C* sintácticamente correcto. Puedes basarte en el siguiente pseudocódigo:

1. Capturar y comprobar argumentos introducidos en la línea de comandos:
    - Si no recibimos ningún argumento trabajaremos sobre el directorio actual.
    - Si recibimos un argumento comprobamos que es un subdirectorio válido.
        - Si es un subdirectorio válido trabajaremos sobre él.
        - Si no lo es finalizamos el programa.
    - Si recibimos dos o más argumentos finalizamos el programa.
2. Mostrar menú de opciones al usuario
3. Leer opción introducida por el usuario.
    - Si se ha seleccionado la opción *salir* finalizamos el programa.
    - En cualquier otro caso calculamos la estadística seleccionada y volvemos al punto 2.

Además de lo que has visto en el guión de la práctica, las siguientes funciones de *C* te serán útiles:

`int system(const char *COMANDO);`	Invoca al sistema operativo y ejecuta `COMANDO`.
`strcpy(cadena_destino, cadena_origen);`	Copia el valor de `cadena_origen` en `cadena_destino`. Hay que incluir la directriz `#include <string.h>`.