FAQ Laboratorio de Fundamentos de Programación

Blasco Planesas, Jordi; Muñoz Baracco, Cesco

2020-07-07

Contents

Introducción

El presente recurso es un recopilatorio de cuestiones planteadas en el **Laboratorio de Fundamentos de Programación** durante los últimos semestres.

Las **FAQ** pueden dar respuesta a dudas y errores habituales en el momento de iniciarse en el mundo de la programación. También se pueden utilizar como material complementario a la teoría explicada en la **XWiki** de **Fundamentos de Programación**.

La lectura de las FAQ no debe realizarse secuencialmente: cada apartado responde a una consulta o conjunto de consultas, sin relación necesaria entre ellas.

El hilo conductor son las **PEC**: las diferentes respuestas dadas en el **Laboratorio de Fundamentos de Programación** se han agrupado en PEC según el momento en que se produjeron. Esto no significa que una respuesta de una PEC anterior o posterior a la que estemos tratando no nos pueda ser de utilidad.

Dada esta independencia de contenido es recomendable utilizar el buscador del portal: por ejemplo, si queremos buscar consultas donde se haya tratado `strlen`, simplemente ponemos esta instrucción en el buscador y obtendremos la relación de entradas que la contienen.

Es posible descargar una versión en PDF/EPUB de todas las FAQ desde el botón de descarga situado en la parte superior.

La documentación utiliza los siguientes iconos para referenciar los bloques:

 Indica que el código comentado está en **lenguaje algorítmico** y es **correcto**.

 Indica que el código comentado está en **lenguaje algorítmico** y es **incorrecto**, contiene algún error.

 Indica que el código comentado está en **lenguaje C** y es **correcto**.

 Indica que el código comentado está en **lenguaje C** y es **incorrecto**, contiene algún error.

 Muestra la ejecución de un programa en **lenguaje C**.

Chapter 1

PEC01

1.1 Lenguaje algorítmico

El lenguaje algorítmico debemos entenderlo como una aproximación al mundo real, el cual utiliza unas normas definidas por nosotros mismos. En este punto todavía no hablamos de programas escritos en **C**, en **Java**, en **Python** o en **PHP**, por decir algunos lenguajes de programación.

Por ejemplo, en el lenguaje algorítmico que utilizamos en la asignatura definimos un bloque de variables de la siguiente forma:

```
var
    edad: integer;
    peso: real;
end var
```

Que se trate de un lenguaje más cercano al mundo real no significa que no tengan que cumplir unas determinadas reglas. Como se puede ver en este ejemplo, una de estas reglas es que cuando definimos variables lo precedente con `var` y lo finalizamos con`end var`.

Hemos decidido utilizar esta forma de lenguaje algorítmico, aunque también lo podríamos haber planteado de la siguiente forma:

```
variable
```

```
    enter edad
    decimal peso
fvariable
```

Remarcar que este segundo ejemplo **es incorrecto**, no sigue la nomenclatura del lenguaje algorítmico definido en la asignatura. El correcto es el primer ejemplo.

El lenguaje algorítmico es como hacer una aproximación formal a la realidad, no es un lenguaje de programación en sí como es **C**, **Java** o similares. Por lo tanto no es un lenguaje que se pueda compilar y ejecutar con el IDE utilizado en la asignatura, el cual está preparado únicamente para interpretar y ejecutar código programado en lenguaje C.

En este punto biene la gran pregunta: ¿y por qué es necesario primero diseñar el algoritmo, si puedo directamente programarlo en C?

Un algoritmo nos permite diseñar un programa sin tener presentes las particularidades de cada lenguaje de programación. Esta aproximación formal a la realidad de los algoritmos nos facilitan poder hacer posteriormente una traducción rápida a cualquier lenguaje de programación simplemente conociendo las equivalencias correspondientes. Por ejemplo, el primer caso si lo programamos en C equivale a:

```c
int edad;
float peso;
```

El código en C no lo podemos cambiar, ya que si en vez de poner `int` utilizamos `entero`, el compilador de C no comprende la palabra y nos dará un error de código.

Si nunca hemos programado es normal que este planteamiento sorprenda al principio, pero es importante que poco a poco se vaya viendo las diferencias entre lenguaje algorítmico y lenguaje C.

1.2 Lenguaje algorítmico vs lenguaje C

En general:

- **Lenguaje algorítmico**: cercano al lenguaje natural, se trata de una convención que adoptamos nosotros mismos para definir el un programa formalmente. Los algoritmos tienen una serie de normas y sentencias que nosotros definimos (**Nomenclátor**), pero que no son de ninguna forma interpretables por un ordenador. Por lo tanto un algoritmo **no puede ser compilado ni ejecutado**.

- **Lenguaje C**: se trata de un lenguaje de programación que sí comprende un ordenador. Esto significa que únicamente podemos utilizar sus pedidos y sus normas para que el código pueda ser compilado y ejecutado sin problemas.

El lenguaje algorítmico es un pseudocódigo que nos ayuda a definir cómo funciona un programa. No está ligado a ningún lenguaje de programación, con lo que las acciones que realizará, la forma de definir variables, etc. es genérica. Funciones como `writeString()`, `readInteger()` o `writeChar()` forman parte del lenguaje algorítmico: indican una acción genérica a realizar, como es escribir una cadena de caracteres, leer un entero o escribir un carácter. Cuando se quiera codificar este algoritmo en un lenguaje de programación concreto como es C, sólo será necesario saber los pedidos propias de C que nos permiten implementar el algoritmo.

La programación en C funciona exclusivamente con la sintaxis definida por este lenguaje de programación. Instrucciones como `scanf()` y `print ()` son propias de C.

A modo de ejemplo:

Algoritmo: queremos introducir la lectura de la luz de nuestra casa; una posible implementación es:

```
algorithm lecturaLuz
    var
        lecturaMensual: integer;
    end var

    writeString("Introduce la lectura mensual de la luz (kWh): ");
    lecturaMensual := readInteger();
end algorithm
```

Lenguaje C: en este lenguaje no existen las funciones algorítmicas `writeString()` ni `readInteger()`, pero en cambio sí tenemos varias funciones propias de C que nos permiten leer un valor por teclado y asignarlo a una variable de entorno. Por tanto, las acciones algorítmicas anteriores tendrán la siguiente correspondencia en C:

```c
#include <stdio.h>

int main(int argc, char **argv) {
```

```
    int lecturaMensual;

    printf("Introduce la lectura mensual de la luz (kWh): ");
    scanf("%d", &lecturaMensual);
    return 0;
}
```

Es muy importante que se vea claramente qué es un **algoritmo** y que es un **programa en C**.

1.3 Equivalencias lenguaje algorítmico vs lenguaje C

A continuación se indican algunas de las equivalencias existentes entre lenguaje algorítmico y el lenguaje de programación C:

	Lenguaje algorítmico	Lenguaje C
Sigue unas normas?	sí	sí
Se puede compilar?	no	sí
Se puede ejecutar?	no	sí
Asignación de valores a variables	`=`	`=`
Tipo booleano	`boolean`	`bool`
Tipo entero	`integer`	`int`
Tipo decimal	`real`	`float`
Tipo carácter	`char`	`char`
Operador igual	`=`	`==`
Operador diferente		`! =`
Operador mayor	`>`	`>`
Operador mayor o igual		`> =`
Operador menor	`<`	`<`
Operador menor o igual		`<=`
Operador lógico de conjunción	`and`	`&&`
Operador lógico de disyunción	`or`	`||`
Operador lógico de negación	`not`	`!`

1.4 Impresión de valores incorrecta

Cuando se muestra por pantalla el contenido de alguna variable con `printf()` es importante eliminar el prefijo `&` de la variable. Por ejemplo, si no lo hacemos tenemos que:

```c
#include <stdio.h>

int main(int argc, char **argv){
    int idAvion;

    printf("Introduce el identificador del avión : ");
    scanf("%d", &idAvion);

    printf(">> Has elegido el avión con id %d \n", &idAvion);
    return 0;
}
```

El resultado de la ejecución es:

```
Introduce el identificador del avión : 9
>> Has elegido el avión con id -1078693464
```

¿Por qué obtenemos el valor extraño en el identificador de avión? Cuando hacemos referencia a `&idAvion` estamos obteniendo realmente la posición de memoria donde reside la variable `idAvion`, no el valor de la variable. Para obtener su valor necesario eliminar dentro `printf()` el prefijo `&` de la variable `idAvion`:

```c
#include <stdio.h>

int main(int argc, char **argv){
    int idAvion;

    printf("Introduce el identificador del avión : ");
    scanf("%d", &idAvion);

    printf(">> Has elegido el avión con id %d \n", idAvion);
    return 0;
}
```

La salida generada ahora sí es correcta:

```
Introduce el identificador del avión : 9
>> Has elegido el avión con id 9
```

1.5 Cómo definir un enumerativo

La definición de un tipo enumerativo en lenguaje algorítmico se hace de la siguiente forma:

```
type
    typeName = {VALUE1, VALUE2, VALUE3, ... , VALUEn};
end type
```

Los elementos `VALUE1`,`VALUE2`, `VALUE3` … acaban siendo constantes, y el valor que de cada uno es:

```
VALUE1 = 0
VALUE2 = 1
VALUE3 = 2
{ ... }
VALUEn = n-1
```

Posteriormente no es posible hacer un cambio de valor de estos elementos de tipo enumerativo.

1.6 Cómo utilizar un enumerativo

Una enumeración es una asignación de un valor entero a la serie de elementos que se ha definido, empezando por 0 y incrementándose en 1 en cada elemento.

Por ejemplo, podemos tener la siguiente definición:

```
typedef enum {MALE, FEMALE} tGender;
```

Esto significa que `MALE == 0` y`FEMALE == 1`. Si la orden de la definición se hubiera hecho al revés, `{FEMALE, MALE}`, tendríamos que `FEMALE == 0` y`MALE == 1`.

Una posible forma de utilizar los enumerativos se leer un entero y compararlo con el elemento correspondiente definido dentro del `enum`, para realizar una acción u otra. Una posible implementación en lenguaje C sería:

```c
#include <stdio.h>

typedef enum {MALE, FEMALE} tGender;

int main(int argc, char **argv) {
    tGender gender;

    printf("Type patient gender: 0 for MALE, 1 for FEMALE\n");
    scanf("%u", &gender);

    if (gender == MALE) {
        printf("Patient gender MALE\n");
    } else {
        if (gender == FEMALE) {
            printf("Patient gender FEMALE\n");
        } else {
            printf("Incorrect option\n");
        }
    }
    return 0;
}
```

1.7 Especificador de un enumerativo

Los enumerativos en lenguaje C, `enum` utilizan el especificador `%u`.

Ejemplo:

```c
#include <stdio.h>

typedef enum {PRIVAT, PUBLIC} tTransporte;

int main(int argc, char **argv) {
    tTransporte tipoTransporte;

    printf("¿Con qué tipo de transporte te desplazas al trabajo (0 = privado, 1 = públi
    scanf("%u", &tipoTransporte);
    printf("Te desplazas al trabajo en transporte (0 = privado, 1 = público):");
    printf("%u\n", tipoTransporte);
    return 0;
}
```

1.8 Lectura de caracteres en C

En el lenguaje C la lectura de un `char` puede comportarse de forma inadecuada si previamente el buffer de entrada contiene algún carácter previo.

Imaginemos que queremos crear un programa muy sencillo que dado un número de DNI y su letra, nos concatenar los dos valores y lo muestre por pantalla. Una posible forma de implementar este programa en C sería:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int dniNum;    /* número del DNI */
    char dniChar;  /* letra del DNI */

    printf("Introduce el número del DNI: ");
    scanf("%d", &dniNum);
    printf("Introduce la letra del DNI: ");
    scanf("%c", &dniChar);

    printf("\nEl DNI introducido es: %d-%c\n", dniNum, dniChar);
    return 0;
}
```

¿Qué pasa si ejecutamos este código? Que vemos que se comporta de forma incorrecta, ya que no nos llega a pedir la letra del DNI, mostrando directamente el resultado:

```
Introduce el número del DNI: 12345678
Introduce la letra del DNI:
El DNI introducido es: 12345678-
```

Cuando tecleamos el primer entero lo que hacemos realmente es introducir un número + un `intro` al final de todo. El número queda asignado a la variable `dni_num`, y el`intro` es leído como un carácter y se asigna a la variable `dni_char`. Por este motivo C interpreta que las dos variables ya tienen valor y finaliza el programa.

Como podemos solucionar este comportamiento? Vaciando el `intro` del buffer de entrada antes de leer el carácter, y una posible forma de hacerlo es mediante el comando `getchar()`. Este comando lee un carácter del buffer de entrada y el vacío del buffer.

Por lo tanto se puede corregir el programa anterior de la siguiente forma:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int dni_num;    /* número del DNI */
    char dni_char;  /* letra del DNI */

    printf("Introduce el número del DNI: ");
    scanf("%d", &dni_num);
    getchar();
    printf("Introduce la letra del DNI: ");
    scanf("%c", &dni_char);

    printf("\nEl DNI introducido es: %d-%c\n", dni_num, dni_char);
    return 0;
}
```

Si ahora ejecutamos ya funcionará como deseamos:

```
Introduce el número del DNI: 12345678
Introduce la letra del DNI: B
```

```
El DNI introducido es: 12345678-B
```

En caso de necesidad, con `getchar()` se puede guardar el carácter del buffer en una variable para tratarlo posteriormente:

```
char nombreVariable;
nombreVariable = getChar();
```

1.9 Lectura de float en C

El separador de valores decimales (tipo `float`) en C **es el punto**, no la coma. De ahí que cuando se introduce un valor decimal desde teclado siempre lo haremos con un punto:

Ejemplo:

```
#include <stdio.h>

int main(int argc, char **argv) {}
    /* Variable que contendrá el peso de una persona */
    float peso;

    /* Lectura del dato por teclado (el separador decimal es un . ) */
    printf("Introduce el peso (kg) de una persona : ");
    scanf("%f", &peso);

    /* Se muestra el valor decimal por pantalla */
    printf("Has introducido el peso = %.1f kg.\n", peso);
    return 0;
}
```

La ejecución será:

```
Introduce el peso (kg) de una persona : 79.440
Has introducido el peso = 79.4 kg.
```

1.10 Frequently Made Mistakes

1.10.1 Definición de tipos: tipo booleano

En lenguaje algorítmico, el tipo `boolean` es un tipo básico, y como tal no es necesario definirlo en un bloque `type`.

Pseudocódigo incorrecto:

```
type
    boolean = {FALSE, TRUE};
type
var
    myNum: integer;
    myBool: boolean;
end var
```

Se pueden declarar variables de tipo booleano directamente, al igual que si fuera un entero, un real o un carácter.

Pseudocódigo correcto:

```
var
    myNum: integer;
    myBool: boolean;
end var
```

1.10.2 Estilo y formato: ausencia de estilo y formato en lenguaje algorítmico

Este no es un error sintáctico o semántico, sino una mala práctica de diseño y programación muy frecuente.

Pseudocódigo incorrecto:

```
action readHotel(out room: integer, out price: real)
var
```

```
i: integer;
end var
writeString("Enter room:");
readInteger(room);
writeString("Enter price:");
readReal(totalPrice);
if(price>MAXPRICE) then
writeString("Invalid price");
end if
for i=1 to rooms do
writeString("Room: ");
writeInteger(i);
end for
end action
```

En el caso del lenguaje algorítmico, las reglas de formato y de estilo son arbitrarias y fijadas por convenio, pero hay que seguir para que sea fácil de leer y revisar, de la misma forma que se hace cuando se programa en C u otros lenguajes. En el ejemplo anterior, no hay aplicada ninguna indentación y el pseudocódigo es muy difícil de leer. Fijaos cómo cambia cuando aplicamos correctamente unas mínimas reglas.

Pseudocódigo correcto:

```
action readHotel(out room: integer, out price: real)
    var
        i: integer;
    end var

    writeString("Enter room:");
    readInteger(room);
    writeString("Enter price:");
    readReal(totalPrice);

    if (price > MAXPRICE) then
        writeString("Invalid price");
    end if

    for i:=1 to rooms do
        writeString("Room: ");
    writeInteger(i);
    end for
end action
```

Cabe mencionar que el **sangrado** del texto (tabulaciones) es especialmente importante para la lectura de los programas, ya que permiten identificar rápidamente los bloques de código, las funciones y acciones, las estructuras iterativas y alternativas, y su dependencia jerárquica. Por este motivo, el uso de sangrado en el pseudocódigo es absolutamente necesario.

1.10.3 Declaración de variables: identificadores no permitidos

Pseudocódigo incorrecto:

```
var
    1Hotel_ID: integer;
    2Hotel_ID: integer;
end var
```

El nombre de las variables puede contener números siempre que no estén en la primera posición. Utilizaremos el modelo camelCase para definir el nombre de las variables.

Pseudocódigo correcto:

```
var
    hotelId1: integer;
    hotelId2: integer;
end var
```

1.10.4 Declaración de variables: operador de declaración

Pseudocódigo incorrecto:

```
var
    id:= integer;
    brand:= string;
    name:= string;
end var
```

El siguiente error podría parecer un error leve, pero es importante respetar el **Nomenclátor** y utilizar los operadores correctamente. En lenguaje algorítmico, el operador de declaración de tipo es `:` y no `=`, que es el operador de asignación de valor.

Pseudocódigo correcto:

```
var
    id: integer;
    brand: string;
    name: string;
end var
```

Chapter 2

PEC02

2.1 Booleanos en C

Algunos puntos a considerar con los booleanos en C:

- Cuando utilizamos el tipo `bool` de C necesitamos importar la librería `<stdbool.h>`, ya que el tipo`bool` no se definió a las primeras versiones del lenguaje C.
- Los valores que puede tomar una variable booleana en C son `false` y`true`. El lenguaje C trata internamente estos valores como enteros: `false` corresponde a `0` y `true` a `1`.
- Cuando queramos introducir el valor de un booleano para teclado o bien mostrarlo por pantalla, utilizaremos el entero `0` para referirnos a `false` y `1` por `true`.
- El especificador de tipo de los booleanos es `%d`.
- Para mostrar el valor de una variable booleana en C lo podemos hacer de la siguiente forma:

```c
bool isVocal;
printf("La letra %c es una vocal (0=false, 1=true) ? %d\n", letra, isVocal);
```

- Para leer un booleano desde teclado, lo haríamos de la siguiente forma:

```c
bool variable;
scanf("%d", &variable);
```

- La lectura para teclado de un booleano, hecha como se indica en el punto anterior, generará un *warning* del siguiente tipo: `warning: formato '% de expects argumento of type' int * ', but argumento 2 has type '_Bool *' [-Wformat =]`. Este aviso significa que estamos utilizando un especificador de tipo (`%d`) distinto del que le correspondería al tipo `bool`, el cual trabaja únicamente con 1 bit. Como C no dispone de ningún especificador de tipo que trabaje sólo con 1 bit, podemos hacer dos cosas:
  - Utilizar una variable auxiliar que nos ayude a hacer una conversión intermedia a `int`, a fin de transformar posteriormente el valor a `bool`:

```
bool variable;
int aux;
scanf("%d", &aux);
variable = aux;
```

  - Ignorar el warning de esta situación específica: a pesar del aviso, el programa se puede compilar y ejecutar.

2.2 Booleanos definidos como enumerativos

En semestres anteriores de la asignatura de **Fundamentos de Programación**, se utilizaba un **enumerativo** para definir el tipo booleano:

```
typedef enum {FALSE, TRUE} boolean;
```

Esta forma de definir el tipo booleano es **obsoleta** y **no utiliza** este semestre; tal y como se ha comentado en el apartado Booleanos en C, los booleanos los definiremos mediante la librería `<stdbool.h>`. Tenedlo presente cuando consulte PAC[1], PR[2] y PS[3] de semestres anteriores, en los que se utilizaba la nomenclatura ahora obsoleta.

[1] PAC: Prueba de Evaluación Continua
[2] PR: Práctica
[3] PS: Prueba de Síntesis

2.3 Constantes: define vs const

La definición de constantes se puede hacer tanto con `define` como con `const`. Sin embargo, la forma de comportarse de estas dos opciones es completamente diferente, aunque el resultado final acabe siendo el mismo:

- `define`: cuando utilizamos esta opción no se guarda en ninguna posición de memoria el valor de la constante. Lo que se hace realmente es que en los pasos previos a la propia compilación del programa, el preprocesador sustituye todas las referencias del define por el valor indicado.

Por ejemplo, si tenemos el siguiente programa con una constante creada con `define`:

```c
#include <stdio.h>
#define MEDIDA 8

char letras[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
int vertical, horizontal;

int main(int argc, char **argv) {
    /* fuente: https://en.wikipedia.org/wiki/Chess */
    for (vertical=MEDIDA; vertical>=1; vertical--) {
        for (horizontal=0; horizontal<=MEDIDA-1; horizontal++) {
            printf("%c%d ", letras[horizontal], vertical);
        }
        printf("\n");
    }
    return 0;
}
```

Antes de la compilación, el preprocesador entre otras acciones elimina comentarios y sustituye todas las referencias `MEDIDA` por `8`:

```c
#include <stdio.h>

char letras[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
int vertical, horizontal;

int main(int argc, char **argv) {
    for (vertical=8; vertical>=1; vertical--) {
```

```
        for (horizontal=0; horizontal<=8-1; horizontal++) {
            printf("%c%d ", letras[horizontal], vertical);
        }
        printf("\n");
    }
    return 0;
}
```

Por lo tanto la definición de constantes con `define` se comporta como si se tratara de un *"buscar-reemplazar"* de un procesador de textos. No se guarda ninguna constante en memoria, pero por el contrario, el programa ocupará un poco más para la sustitución directa de referencias que hace; la sustitución la hace en todo el programa, no se puede limitar a un ámbito concreto (por ejemplo sólo dentro de una función).

- `const`: en este caso sí que se reserva una posición de memoria. En C se comporta igual como si fuera una variable, pero la que únicamente funciona en modo lectura: no le podemos modificar el valor.

Además, `const` nos permite también decir qué tipo de valor tendrá la constante: si es de tipo `float`, `int`,`char`... con lo que este hecho nos da un punto adicional de control, ya que nos aseguramos de que el tipo de valor asignado será el correcto para el programa.

Con este tipo de definición de constante, el ejemplo anterior quedaría de la siguiente forma:

```
#include <stdio.h>
const int MEDIDA 8

char letras[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
int vertical, horizontal;

int main(int argc, char **argv) {
    /* La constante MEDIDA está guardada en memòria */
    printf("posición en memoria de la constante MEDIDA : %p \n", &MEDIDA);
    for (vertical=MEDIDA; vertical>=1; vertical--) {
        for (horizontal=0; horizontal<=MEDIDA-1; horizontal++) {
            printf("%c%d ", letras[horizontal], vertical);
        }
        printf("\n");
    }
    return 0;
}
```

Como se puede ver, es posible obte la dirección en memoria donde se guarda la constante `MEDIDA`. En este caso, sí que puedes definir una constante con `const` y hacer que sólo afecte a un ámbito determinado (por ejemplo, que la constante esté definida únicamente dentro de una función).

Estas son las principales diferencias entre `define` y `const` a la hora de definir una constante; `define` se creó mucho antes que no la sentencia `const`, con lo que es bastante habitual decantarse por esta opción por temas históricos.

2.4 Cómo mostrar el valor de una constante

En C podemos mostrar por pantalla el valor de una constante definida con `#define` mediante `printf()`. ejemplo:

```c
#include <stdio.h>

#define WORD "world"
#define YEAR 2021
#define EXCLAMATION '!'

int main(int argc, char **argv) {
    printf("hello %s and happy %d %c\n", PALABRA, YEAR, EXCLAMATION);
    return 0;
}
```

El resultado que se mostrará por pantalla será:

```
hello world and happy 2021 !
```

2.5 Precisión en variables float

Hay algunos valores decimales determinados que no se pueden representar de forma precisa en una variable de tipo `float`. La mejor solución para los casos que tratamos es redondear al número de decimales que realmente necesitamos.

Si en cambio queremos sí o sí trabajar con todos los decimales, podemos optar por utilizar un tipo de dato que tenga mayor precisión que `float`: `double`.

Por ejemplo, el siguiente programa devuelve el resultado esperado si se guarda en un `double`, y no así si se hace en un `float`:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    float num1;
    float num2;
    float resultado1;
    double resultado2;

    num1 = 1.3;
    num2 = 17;
    resultado1 = num1 + num2;
    printf("resultado con float : %f\n", resultado1);
    resultado2 = num1 + num2;
    printf("resultado con double: %f\n", resultado2);
    return 0;
}
```

La salida generada es:

```
resultado con float : 18.299999
resultado con double: 18.300000
```

2.6 Semántica de una expresión

Si recordamos lo que se comenta en el punto **3.2. Semántica de una expresión** del módulo de la XWiki **Tipos básicos de datos**, tenemos que conseguir que las expresiones y comparaciones realizadas a los algoritmos sean **semánticamente correctas**.

Con un ejemplo se verá más claro: tenemos el siguiente algoritmo que indica si una persona es mayor de edad. Fijaos que la expresión realiza una comparación entre dos enteros: la variable `edad` y el número `17`.

```
var
    edad: int;
    isMayorEdad: boolean;
end var

algorithm serMayorEdad

    writeString("Introduce edad del conductor :");
    edad := readInteger();

    isMayorEdad := (edad > 17);

    writeString("El conductor es mayor de edad? :");
    writeBoolean(isMayorEdad);

end algorithm
```

Esta expresión **es semánticamente correcta**.

En cambio, imaginemos ahora que nuestro algoritmo acepta decimales por la `edad`; por ejemplo, `19.5` indicaría que la edad es de 19 años y 6 meses. Así tenemos el siguiente planteamiento, donde ahora la variable `edad` es de tipo `real`:

```
var
    edad: real;
    isMayorEdad: boolean;
end var

algorithm serMayorEdad

    writeString("Introduce edad del conductor :");
    edad := readReal();

    isMayorEdad := (edad > 17);

    writeString("El conductor es mayor de edad? :");
    writeBoolean(isMayorEdad);

end algorithm
```

El algoritmo ahora **no es correcto** ya que contiene una expresión **semánticamente incorrecta**, en la que se compara `edad` (real) con`17` (entero). Para solucionarlo, podemos utilizar alguna de las funciones de conversión comentadas

en el apartado **4. Funciones de conversión de tipos del mismo módulo**:

```
{ opción 1: hacemos que ambos valores sean de tipo entero }
isMayorEdad: = (realToInteger(edad) > 17);

{ opción 2: hacemos que ambos valores sean de tipo real }
isMayorEdad: = (edad > integerToReal(17));
```

2.7 Ejemplos de expresiones

2.7.1 Ejemplo 1: esPar

Imaginemos que nos piden un algoritmo que indique si un número es par.

Una posible solución sería:

```
algorithm esPar
    var
        numero: integer;
        isPar: boolean;
    end var

    writeString("Introduce un número : ");
    numero:= readInteger();
    isPar:= (numero mod 2 = 0);

    writeString("El número ");
    writeInteger(numero);
    writeString(" es par? ");
    writeBoolean(isPar);

end algorithm
```

La variable `isPar` tomará el valor `TRUE` si el número es par y `FALSE` lo contrario. No ha sido necesario utilizar ninguna estructura `if-else` para resolver el algoritmo.

Una posible forma de codificar en lenguaje C es:

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

int main(int argc, char **argv) {
    int numero;
    bool isPar;

    printf("Introduce un número : ");
    scanf("%d", &numero);
    isPar = (numero % 2 == 0);

    printf("El número %d es par? (0=FALSE, 1=TRUE) : %d \n", numero, isPar);
    return 0;
}
```

2.7.2 Ejemplo 2: finDeSemana

Imaginemos que queremos hacer un programa muy sencillo que nos diga si hoy es fin de semana o no. Su algoritmo sería el siguiente:

```
type
    dias = {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};
end type

algorithm finDeSemana

    var
        isFinDeSemana: boolean;
        diaSemana: dias;
    end var

    writeString("Qué día de la semana es hoy ?\n");
    writeString("Para LUNES teclea 0\n");
    writeString("Para MARTES teclea 1\n");
    writeString("Para MIÉRCOLES teclea 2\n");
    writeString("Para JUEVES teclea 3\n");
    writeString("Para VIERNES teclea 4\n");
    writeString("Para SÁBADO teclea 5\n");
```

```
    writeString("Para DOMINGO teclea 6\n");

    diaSemana:= readInteger();
    isFinDeSemana:= (diaSemana = SABADO or diaSemana = DOMINGO);

    writeString("Hoy es fin de semana?");
    writeBool(isFinDeSemana);

end algorithm
```

La variable `boolean esFinDeSemana` tomará el valor de `true` o`false` en función del resultado de evaluar la expresión. No es necesario la utilización de estructuras condicionales `if-else` que veremos más adelante en el curso.

Una posible forma de codificarlo en lenguaje C es:

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

typedef enum {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO} dias;

int main(int argc, char **argv) {
    bool isFinDeSemana;
    dias diaSemana;

    printf("\nQué día de la semana es hoy ?\n");
    printf("Para LUNES teclea 0\n");
    printf("Para MARTES teclea 1\n");
    printf("Para MIERCOLES teclea 2\n");
    printf("Para JUEVES teclea 3\n");
    printf("Para VIERNES teclea 4\n");
    printf("Para SABADO teclea 5\n");
    printf("Para DOMINGO teclea 6\n");

    scanf("%u", &diaSemana);
    isFinDeSemana = (diaSemana == SABADO || diaSemana == DOMINGO);

    printf("Hoy es fin de semana (0 == false, 1 == true) ? %d\n", isFinDeSemana);
    return 0;
}
```

Varios puntos a considerar:

- Recordemos que inicialmente en C no existía el tipo booleano. Para poder

utilizar `bool`, y los valores `true` y `false` necesitamos importar previamente la librería `<stdbool.h>`.
- El especificador de tipo de un `bool` es `%d`.
- Cuando definimos una variable de tipo `enum` utilizamos el especificador de tipo `%u`. Lo podríamos hacer como `%d`, pero devolverá un warning aunque el resultado sea correcto. El tipo `%u` es igual que un entero `%d` pero sin signo: esto significa que con `%d` podemos tratar valores negativos como -12 y con `%u` no es posible, pero como sabemos que los valores que puede tomar un `enum` siempre serán >= 0 nos conviene utilizar `%u`.
- Para las particularidades de los `bool` en el lenguaje de programación C que ya hemos comentado anteriormente, la entrada y salida de valores de un `boolean` será numérica. Para facilitar la comprensión podemos mostrar por pantalla un literal que nos indique que `0` equivale a `false` y `1` a `true`.

2.7.3 Ejemplo 3: esVocal

Ejemplo: queremos hacer un programa que, entrado un carácter por el canal de entrada, nos indique si se trata o no de una vocal.

Una posible solución para el algoritmo es la siguiente:

```
var
    letra: char;
    isVocal: boolean;
end var

algorithm esVocal

    writeString("Teclea una letra :");
    letra := readChar();

    { en este ejemplo únicamente tratamos las vocales minúsculas }
    isVocal := letra = 'a' or letra = 'e' or letra = 'i' or letra = 'o' or letra = 'u';

    writeString("La letra ");
    writeChar(letra);
    writeString(" es una vocal? ");
    writeBoolean(isVocal);

end algorithm
```

Como se puede ver el planteamiento del algoritmo es:

- Leemos un carácter desde el canal de entrada.
- Comparamos el carácter con `a`, `e`, `i`, `o`, `u`.
  - Si coincide con alguna de estas vocales, la variable `isVocal: = true`.
  - Si no coincide con ninguna de las vocales, la variable `isVocal: = false`.
- Se muestra el resultado por pantalla.

Cómo lo podemos traducir en lenguaje C? Una posible opción es:

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

int main(int argc, char **argv) {
    char letra;
    bool isVocal;

    printf("Introduce una letra : ");
    scanf("%c", &letra);

    { en este ejemplo únicamente tratamos las vocales minúsculas }
    isVocal = letra == 'a' || letra == 'e' || letra == 'i' || letra == 'o' || letra ==

    printf("La letra %c es una vocal (0=FALSE, 1=TRUE) ? %d\n", letra, isVocal);
    return 0;
}
```

2.7.4 Ejemplo 4: votaciones

Imaginemos que nos piden un programa que valide si una persona puede ir a votar o no; la condición que nos dicen que hay que cumplir es que la persona sea mayor de edad y además esté en el censo electoral de la localidad donde está votando.

El algoritmo podría ser el siguiente:

```
algorithm votaciones
    var
        isMayorEdad: boolean;
        isCensado: boolean;
        isVotar: boolean;
```

```
    end var

    writeString("Eres mayor de edad (0=FALSE, 1=TRUE) ? : ");
    isMayorEdad := readBoolean();
    writeString("Estás en el censo electoral (0=FALSE, 1=TRUE) ? : ");
    isCensado := readBoolean();

    { expresión }
    isVotar := isMayorEdad and isCensado;

    writeString("Puedes votar (0=FALSE, 1=TRUE) :");
    writeBoolean(isVotar);

end algorithm
```

Una posible codificación en C sería:

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

int main(int argc, char **argv) {
    bool isMayorEdad;
    bool isCensado;
    bool isVotar;

    printf("Eres mayor de edad (0=false, 1=true) ? : ");
    scanf("%d", &isMayorEdad);

    printf("Estás en el censo electoral (0=false, 1=true) ? : ");
    scanf("%d", &isCensado);

    { expresión }
    isVotar = isMayorEdad && isCensado;

    printf("Puedes votar (0=false, 1=true) : %d\n", isVotar);
    return 0;
}
```

La ejecución de este ejemplo sería:

```
Eres mayor de edad (0=false, 1=true) ? : 1
Estás en el censo electoral (0=false, 1=true) ? : 0
Puedes votar (0=false, 1=true) : 0
```

2.7.5 Ejemplo 5: ginTonicPreparation

Imaginemos que queremos preparar un gin tonic. Sabemos el volumen que usaremos de ginebra y de tónica, y cuál es la capacidad de la copa de balón que lo contendrá.

Hemos visto una oferta por internet y hemos comprado cubitos metálicos de acero inoxidable… pero se nos ha ido un poco la cabeza y hemos comprado un total de 20 unidades.

Queremos hacer un programa que, utilizando únicamente expresiones, nos diga si podemos preparar o no el gintonic en función del número de cubitos que le queremos poner:

- Si el número de cubitos caben dentro de la copa, devolverá `true`.
- En caso contrario, devolverá `false`.

Por tanto lo que tiene que hacer nuestro programa básicamente es validar si el volumen de ginebra + tónica + (hielo) * número de cubitos supera o no el volumen de la copa.

El algoritmo podría ser el siguiente:

```
const
    GIN: real = 50.0;              { in ml }
    TONIC: real = 200.0;           { in ml }
    GLASS: real = 620.0;           { in ml }
    METAL_ICE_CUBE: real = 42.875; { in ml }
end const

algorithm ginTonicPreparation

    var
        numMetalIceCubes: integer;
        isPossible: boolean;
    end var

    writeString("Number of metal ice cubes ? (integer) : ");
    numMetalIceCubes:= readInteger();
```

```
    isPossible:= (GLASS   (GIN + TONIC + METAL_ICE_CUBE * numMetalIceCubes));

    writeString("Can you make a gin & tonic? : ");
    writeBoolean(isPossible);

end algorithm
```

La expresión que da valor a `isPossible` se encarga de evaluar el volumen de la copa respecto el resultante de ginebra, tónica y cubitos.

Su traducción a C podría ser:

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

#define GIN 50.0              /* in ml */
#define TONIC 200.0           /* in ml */
#define GLASS 620.0           /* in ml */
#define METAL_ICE_CUBE 42.875 /* in ml */

int main(int argc, char **argv) {
    int numMetalIceCubes;
    bool isPossible;

    printf("Number of metal ice cubes ? (integer) : ");
    scanf("%d", &numMetalIceCubes);

    isPossible = GLASS >= (GIN + TONIC + METAL_ICE_CUBE * numMetalIceCubes);

    printf("Can you make a gin & tonic? (0=FALSE, 1=TRUE) : %d\n", isPossible);
    return 0;
}
```

Para realizar el cálculo en C también se ha utilizado una expresión.

2.7.6 Ejemplo 6: ginTonicFreeMl

Vamos a evolucionar el ejemplo anterior del gintónic: imaginemos ahora que queremos que nuestro programa nos diga el volumen (en mililitros) que queda libre en la copa una vez puesto un determinado número de cubitos.

El algoritmo quedaría de la siguiente forma:

```
const
    GIN: real = 50.0;              { in ml }
    TONIC: real = 200.0;           { in ml }
    GLASS: real = 620.0;           { in ml }
    METAL_ICE_CUBE: real = 42.875; { in ml }
end const

algorithm ginTonicFreeMl

    var
        numMetalIceCubes: integer;
        volumeFree: real;
    end var

    writeString("Number of metal ice cubes ? (integer) : ");
    numMetalIceCubes:= readInteger();

    volumeFree:= GLASS - (GIN + TONIC + METAL_ICE_CUBE * numMetalIceCubes);

    writeString("How many free ml in the glass? : ");
    writeReal(volumeFree);

end algorithm
```

Y la codificación en C :

```c
#include <stdio.h>

#define GIN 50.0              /* in ml */
#define TONIC 200.0           /* in ml */
#define GLASS 620.0           /* in ml */
#define METAL_ICE_CUBE 42.875 /* in ml */

int main(int argc, char **argv) {
    int numMetalIceCubes;
    float volumeFree;

    printf("Number of metal ice cubes ? (integer) : ");
    scanf("%d", &numMetalIceCubes);
```

```
    volumeFree = GLASS - (GIN + TONIC + METAL_ICE_CUBE * numMetalIceCubes);

    printf("How many free ml in the glass ? : %.3f ml \n", volumeFree);
    return 0;
}
```

2.7.7 Ejemplo 7: scoutingBasquet

Imaginemos que hacemos tareas de scouting para las secciones de baloncesto femenino y masculino de nuestro club, y nos han encargado cubrir alguna de las tres plazas siguientes:

- Para el equipo femenino: una pívot que como mínimo haga 195cm de altura.
- Para el equipo femenino: una base, la altura sea inferior a 170cm.
- Para el equipo masculino: un base que sea más alto de 175cm pero a la vez que no supere los 190cm.

Nuestro programa pedirá por teclado si se trata de una jugadora o un jugador, y cuál es su altura. A continuación con expresiones evaluará las condiciones introducidas y si las cumple por alguna de las tres plazas disponibles, lo escogerá (`isDrafted`).

Una posible forma de codificar en C este programa sería:

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

typedef enum {MALE, FEMALE} tGender;

int main(int argc, char **argv) {
    bool isPointGuard; /* Point Guard = base */
    bool isCenter;     /* Center = pívot */
    bool isDrafted;
    int height;
    tGender gender;

    printf("Gender (0=MALE, 1=FEMALE) : ");
    scanf("%u", &gender);
    printf("Heigth (integer value) : ");
    scanf("%d", &height);

    /* Primero miramos si podemos cubrir la posición de base, ya sea femenino o masculino */
```

```
    isPointGuard =
        (height < 170 && (gender == FEMALE)) ||
        (height < 190 && height > 175 && (gender == MALE));

    /* A continuación comprobamos si se trata de la pívot femenina que buscamos */
    isCenter = (height >= 195 && (gender == FEMALE));

    /* Únicamente si se cumple alguna de las dos expresiones anteriores
       (que isPointGuard sea TRUE o que isCenter sea TRUE), el jugador/a
       será elegido/a para formar parte de nuestras secciones de baloncesto */
    isDrafted = isPointGuard || isCenter;

    printf("\nIs drafted (0=FALSE, 1=TRUE) ? : ");
    printf("%d\n", isDrafted);
    return 0;
}
```

Fijaos que `isPointGuard` y`isCenter` son variables de tipo `bool`, ya que la evaluación de las expresiones también será de tipo`bool`.

Puede haber otras codificaciones igual de válidas, esta no es la única solución posible.

2.8 Frequently Made Mistakes

2.8.1 Interfaz de usuario: ausencia de textos informativos

El siguiente no es un error sintáctico o semántico, sino un error de diseño muy frecuente.

Pseudocódigo incorrecto:

```
writeInteger(room);
writeInteger(totalPrice);
```

En el ejemplo mostrado, parece que la intención es mostrar el canal estándar dos variables (`room` y `totalPrice`), y efectivamente las sentencias para imprimirlas son correctas. El problema es que el usuario no tiene ninguna información sobre el significado de los valores que se le muestran. Hay siempre complementar los datos con mensajes informativos, indicando también las unidades cuando sea necesario.

Pseudocódigo correcto:

```
writeString("Room number: ");
writeInteger(room);
writeString("Total price [€]: ");
writeInteger(totalPrice);
```

Aquest error de disseny és més greu en el cas (també real), de no posar cap text informatiu a l'hora de demanar a l'usuari que introdueixi dades per teclat.

Pseudocodi incorrecte:

```
readInteger(room);
readReal(totalPrice);
```

L'usuari no té cap informació sobre els valors a introduir (tipus de dades, intervals vàlids, etc.).

Pseudocodi correcte:

```
writeString("Enter room number [1-100]: ");
readInteger(room);
writeString("Enter total price [€]: ");
readInteger(totalPrice);
```

2.8.2 Caracteres: uso de comillas simples

Pseudocódigo incorrecto:

```
var
    fastpass: char;
    areaMap: char;
    allowsFastPass: boolean;
end var
{...}
fastPass := (allowsFastPass = y or allowsFastPass = Y) and (areaMap = B or areaMap = C);
{...}
```

En el algoritmo anterior, parece que se quiere comparar la variable `allowsFastPass` con los caracteres `y`, `Y`, y la variable `areaMap` con los caracteres `B`,`C`. El problema es que faltan las comillas simples para indicar que se trata de caracteres, y por lo tanto lo que hace la expresión se compararon con las variables `y`,`Y`, `B` y`C` respectivamente (que además además, en este algoritmo no existen. Este es un error semántico grave, que puede no provocar ningún error de compilación en lenguaje C, y en cambio conllevar comportamientos inesperados.

Pseudocódigo correcto:

```
var
    fastpass: char;
    areaMap: char;
    allowsFastPass: boolean;
end var
{...}
fastPass := (allowsFastPass = 'y' or allowsFastPass = 'Y') and (areaMap = 'B' or areaMa
{...}
```

2.8.3 Sintaxis propia de C: funciones de lectura / escritura

Pseudocódigo incorrecto:

```
writeString("Code value: %d\n", codeValue);
```

Se ha intentado aplicar a la función `writeString()` en lenguaje algorítmico la sintaxis propia de la función `printf()` de C, lo que evidentemente no es correcta. Recuerde que el lenguaje algorítmico es un lenguaje de diseño de propósito general, que debe permitir posteriormente codificar en cualquier lenguaje de programación

Pseudocódigo correcto:

```
writeString("Code value");
writeInteger(codeValue);
```

2.8.4 Conversión de tipos: funciones inexistentes

Pseudocódigo incorrecto:

```
b1:= characterToCode(myChar);
```

Las funciones de conversión de tipo están definidas en el **Nomenclátor** de la asignatura, y sirven para asegurar la coherencia semántica entre los tipos de variables de una expresión. En este caso, la función `characterToCode()` no existe, ya que la que hemos declarado es `characterToInteger()`. Es habitual encontrar errores de este tipo, con diversas variantes de nombres de función. Hay evitarlo y usar el **Nomenclátor** como guía.

Pseudocódigo correcto:

```
b1:= characterToInteger(myChar);
```

2.8.5 Expresiones: variables intermedias

Este no es un error sintáctico o semántico, sino una mala práctica de diseño.

Pseudocódigo incorrecto:

```
var
    bool1: boolean;
    bool2: boolean;
    bool3: boolean;
    bool4: boolean;
end var
bool1:= hasGym or hasPool;
bool2:= closToSubway or distanceFromCityCentre < 5;
bool3:= priceDouble <= 100;
boolFin:= bool1 and bool2 and bool3;
```

En el código anterior, lo que queremos obtener al final es una variable de tipo booleano que nos indique si un objeto (suponemos que un hotel), tiene piscina o gimnasio, y además está cerca del metro o menos de 5 km del centro de la ciudad, y además el precio es inferior a 100 (euros). Queremos reunir toda

esta información en una variable booleana, porque seguramente esto nos indica alguna cualidad del objeto (por ejemplo, que es un buen hotel). Ahora bien, fíjese como para llegar al resultado final, se han declarado hasta tres variables auxiliares, que no tienen otra finalidad que construir la expresión final.

Declarar variables auxiliares o temporales no es una mala práctica siempre que se haga con medida, y aplicando el sentido común (algo que sólo se adquiere con la práctica del diseño y la programación). Recordemos que las variables ocupan espacio de memoria, y por lo tanto hay que ser cuidadosos a la hora de declararlas. Una alternativa sería la que presentamos a continuación.

Pseudocódigo correcto:

```
var
    bool4: boolean;
end var
boolFin:= hasGym or hasPool and
          closToSubway or distanceFromCityCentre < 5 and
          priceDouble <= 100;
```

Recordemos que el delimitador de las sentencias es el punto y coma ;, y que hay flexibilidad a la hora de escribir las sentencias en varias líneas (también en C). Finalmente, recordar una vez más que no hay ninguna norma que nos indique cuantas variables auxiliares debemos utilizar. En la solución alternativa hemos optado por no utilizar ninguna y dar un formato comprensible a la expresión, pero esto habrá que decidirlo en cada caso, en base a la práctica y la experiencia.

2.8.6 Declaración de variables: declaración mal ubicada (bloque central de código)

Pseudocódigo incorrecto:

```
for i:= 1 to selectedHotels->nHotels do
    if selectedHotels[i].city = city then
        var
            scorePoints: integer;
        end var
        scorePoints := scorePoints + 1;
    end if
end for
```

Las variables deben declararse **al principio** del **bloque de pseudocódigo**, ya sea una función, una acción o un algoritmo directamente. No es correcto abrir un bloque de declaración de variables dentro de un bloque central de pseudocódigo (y menos abrir varios bloques de declaración a medida que necesitamos variables). Lo mismo ocurre en C: las variables deben declararse siempre al bloque inicial de código (ya sea una función, acción o algoritmo).

Pseudocódigo correcto:

```
var
    scorePoints: integer;
end var

for i:= 1 to selectedHotels->nHotels do
    if selectedHotels[i].city = city then
       scorePoints := scorePoints + 1;
    end if
end for
```

2.8.7 Declaración de variables: declaración mal ubicada (variables globales)

Este no es un error sintáctico o semántico, sino una mala práctica de porgramació que hay que evitar.

Código incorrecto:

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

float maxPrice;
int bestHotel;

int main(int argc, char **argv) {
    /* ... */
    return 0;
}
```

Las variables se declararán siempre dentro de una función o acción (ya sea el main o cualquier otra), y **al principio** de su **bloque de código**. No es correcto

declarar variables fuera de cualquier función, ya que se convierten en **variables globales**, y su uso no se considera una buena práctica de programación. El motivo es que las variables globales dificultan la lectura y la comprensión del código, y pueden provocar errores de ejecución inesperados o actualizaciones involuntarias, al no estar protegidas dentro de funciones o acciones.

Código correcto:

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

int main(int argc, char **argv) {
    float maxPrice;
    int bestHotel;
    /* ... */
    return 0;
}
```

2.8.8 Tipo booleano: valores numéricos

Código incorrecto:

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

int main(int argc, char **argv) {
    bool myBool1;
    bool myBool2;
    bool resultBool;
    /* ... */
    resultBool = (myBool == 1) && (myBool2 == 0);
}
```

Si disponemos de la librería `<stdbool.h>` para el tratamiento de booleanos en C, no tiene ningún sentido que utilizamos sus equivalentes numéricos al código, especialmente en el caso de las expresiones. En su lugar se ha de utilizar las palabras reservadas `true` y`false`.

Como regla general, debemos procurar usar el mínimo número posible de valores numéricos, y utilizar en su lugar variables y constantes.

Código correcto:

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

int main(int argc, char **argv) {
    bool myBool1;
    bool myBool2;
    bool resultBool;
    /* ... */
    resultBool = (myBool == true) && (myBool2 == false);
    return 0;
}
```

2.8.9 Tipo booleano: cadenas de caracteres

Código incorrecto:

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

int main(int argc, char **argv) {
    bool myBool1;
    bool myBool2;
    bool resultBool;
    resultBool = (myBool =="true") && (myBool2 == "false");
    return 0;
}
```

Si disponemos de la librería `<stdbool.h>` para el tratamiento de booleanos en C, podemos utilizar las palabras reservadas `true` y`false`, sin ningún tipo de modificador ni carácter. Si añadimos las comillas, `"true"` y `"false"` se convierten en cadenas de caracteres, que son algo muy diferente y no tienen nada que ver con los booleanos (ni con sus posibles valores). Este error muy grave también se da en lenguaje algorítmico.

Hay que añadir además, que en C cualquier valor que no sea 0 es interpretado como `true`, por lo que se producirán errores de ejecución, ya que las cadenas de caracteres serán interpretadas siempre como un valor cierto.

Código correcto:

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

int main(int argc, char **argv) {
    bool myBool1;
    bool myBool2;
    bool resultBool;
    resultBool = (myBool == true) && (myBool2 == false);
    return 0;
}
```

2.8.10 Sintaxis propia de C: operadores

Pseudocódigo incorrecto:

```
if (acceptable == 1) or (acceptable == 2) then
    writeString("Hotels cannot be compared");
end if
```

El operador `==` es propio del lenguaje C, y su uso lenguaje algorítmico no es correcto. El operador correcto en este caso es `=`.

Pseudocódigo correcto:

```
if (acceptable) = 1 or (acceptable = 2) then
    writeString("Hotels cannot be compared");
end if
```

Chapter 3

PEC03

3.1 Cómo declarar un vector en lenguaje algorítmico

Cuando se declara un vector en lenguaje algorítmico no hay que declarar una variable para cada una de las posiciones de este vector.

Por ejemplo, si queremos un algoritmo que sume tres enteros contenidos dentro de un vector podemos hacer lo siguiente:

```
const
    NUM_ENTEROS: integer = 3;
end const

algorithm sumaEnteros

    var
        vectorEnteros: vector[NUM_ENTERS] of integer;
        sumaEnteros: integer;
    end var

    vectorEnteros[1] := 13;
    vectorEnteros[2] := 24;
    vectorEnteros[3] := 2;

    { Como se puede ver, accedemos directamente a las posiciones del vector }
```

```
    { En vez de definir una variable para cada posición. }

    sumaEnteros := vectorEnteros[1] + vectorEnteros[2] + vectorEnteros[3];

    { Hay que recordar también que en lenguaje algorítmico, en un vector de tamaño n }
    { La primera posición del vector es la 1 y la última la n; en cambio en lenguaje C
    { La primera siempre es la 0 y la última la n-1. }

    writeString("La suma de los 3 enteros del vector es : ");
    writeInteger(sumaEnteros);

end algorithm
```

3.2 Significado de los argumentos del main

La principal diferencia entre la definición `main(int argc, char **argv)` y `main()` es que la primera opción está preparada para recibir argumentos cuando se ejecuta el programa y no así la segunda.

Por ejemplo, si tenemos el siguiente programa compilado en C y le pasamos una serie de argumentos desde la línea de comandos:

```
$> programa a1 a2 a3
```

Con el main definido como `main(int argc, char **argv)` podemos acceder desde dentro del programa a todos los argumentos pasados; así tendremos:

```
argc = 4

argv[0] = "programa"
argv[1] = "a1"
argv[2] = "a2"
argv[3] = "a3"
```

El propio sistema operativo ocupa de darle el valor al argumento int `argc` (número total de argumentos incluido el nombre del programa), con lo que únicamente tienes que preocuparte de pasar los argumentos. Por otra parte, `argv` es un array de punteros donde cada uno de ellos apunta a un argumento

formado por una cadena de caracteres; así `argv` contendrá cada una de sus posiciones los argumentos pasados desde línea de comandos, y en la posición 0 el propio nombre del programa.

Si en cambio tienes definido el programa como `main ()`, simplemente no tienes forma de acceder a los argumentos que le puedas llegar a pasar. Hay muchas veces que los datos los puedes tener ya definidas dentro del propio programa o las vayas a consultar a una fuente externa, con lo que no tener la capacidad de procesar argumentos no supone ningún impedimento a la hora de ejecutar tu programa.

3.3 Asignar valores a un vector

La lectura y asignación de valores a un vector se realiza de la siguiente forma en lenguaje algorítmico:

```
const
    MAX_TEMP: integer = 2;
end const

algorithm lecturaTemperaturas

    var
        vTemperaturas: vector[MAX_TEMP] of float;
    end var

    writeString("Introduce la lectura 1 : ");
    vTemperaturas[1] := readReal();

    writeString("Introduce la lectura 2 : ");
    vTemperaturas[2] := readReal();

    writeString("Los valores introducidos han sido : ");

    writeString("> Valor de la posición ");
    writeInteger(1);
    writeString(" : ");
    writeReal(vTemperaturas[1]);

    writeString("> Valor de la posición ");
    writeInteger(2);
    writeString(" : ");
```

```
    writeReal(vTemperaturas[2]);

end algorithm
```

Y en lenguaje C:

```c
#include <stdio.h>

#define MAX_TEMP 2

int main(int argc, char **argv) {
    float vTemperaturas[MAX_TEMP];
    int i;

    printf("Introduce la lectura 1 : ");
    scanf("%f", &vTemperaturas[0]);

    printf("Introduce la lectura 2 : ");
    scanf("%f", &vTemperaturas[1]);

    printf("> Valor de la posición %d : %.1f \n", 0, vTemperaturas[0]);
    printf("> Valor de la posición %d : %.1f \n", 1, vTemperaturas[1]);
    return 0;
}
```

Hay que remarcar una diferencia importante:

- En **lenguaje algorítmico** las posiciones del vector van **desde la 1 hasta la N**.
- En **lenguaje C**, van desde **de la 0 hasta la N-1**.

En ambos casos **N** hace referencia al número total de elementos del vector.

3.4 Stack smashing detected

Este mensaje de error se produce cuando se intenta acceder / operar con una posición de un vector que no lo hemos definido previamente. Se puede dar por diferentes situaciones que acaban generando el mismo problema.

- **Caso 1**: se define un vector de N-posiciones, pero en vez de comenzar por la posición 0 lo hacemos por la 1. Esto es incorrecto: recordemos que en C la posición inicial de un vector siempre es la 0, y la final siempre es N-1. Ejemplo, para un vector de 3 posiciones tendremos:

```c
int vector1[3];
vector1[1] = 13;  /* Posición del vector1 válida */
vector1[2] = 24;  /* Posición del vector1 válida */
vector1[3] = 48;  /* Posición del vector1 no válida! */
/* En cambio la posición 0 del vector,
 * que tenemos disponible, no la hemos utilizado!
 */
```

- **Caso 2**: se define un vector con menos posiciones de las que necesitamos. Por ejemplo, si tenemos:

```c
int vector2[2];
```

Significa que las posiciones reservadas en memoria para este vector son:

```c
vector2[0] = 10;  /* Posición del vector2 válida */
vector2[1] = 13;  /* Posición del vector2 válida */
vector2[2] = 24;  /* Posición del vector2 no válida! */
```

Por lo tanto cualquier operación con `vector2[2]` nos generará el error indicado. Si queremos que el vector contenga 3 elementos sólo hay que definir correctamente su tamaño:

```c
int vector2[3];
```

3.5 Concatenación en lenguaje algorítmico

A diferencia del lenguaje C, en notación algorítmica no está contemplado el uso de especificadores que permitan hacer concatenaciones entre cadenas de caracteres, enteros, decimales, etc.

Por lo tanto en lenguaje algorítmico hay que romper los strings con fragmentos más pequeños y que hagan referencia únicamente a un tipo de datos. Por

ejemplo, si se quiere mostrar por pantalla el mensaje *"El empleado que más cobra es Marta, y su nómina es 4675.30 €."*, Lo haremos de la siguiente forma:

```
writeString("El empleado que más cobra es ");
writeString(nombreEmpleado);
writeString(", y su nómina es ");
writeReal(nomina);
writeString(" €.");
```

En lenguaje C equivaldría a:

```
printf("El empleado que más cobra es %s, y su nómina es %.2f €.", nombreEmpleado, nomi
```

3.6 Importancia de los tipos utilizados en lenguaje C

El resultado de las operaciones en lenguaje C depende del tipo de variable definido y de los tipos de valores utilizados. A continuación se exponen tres casos que, según el tipo que se haya definido en las variables utilizadas, dará un resultado u otro:

Caso 1:

```
#include <stdio.h>

int main(int argc, char **argv) {
    int a;
    int b;
    int c;
    int m;

    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);

    m=(a+b+c)/3;
```

```
    printf("%d", m);
    return 0;
}
```

En este caso si damos los valores `a = 1`, `b = 3`,`c = 4`, el resultado es `m = 2`. El resultado de la división será un entero, ya que tanto numerador como denominador están formados por enteros. El resultado entero se guarda en una variable entera, y por pantalla obtendremos: `2`.

Caso 2:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int a;
    int b;
    int c;
    float m;

    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);

    m=(a+b+c)/3;

    printf("%f", m);
    return 0;
}
```

Al igual que en el caso anterior, el numerador y el denominador de la división están formados por enteros, con lo que el resultado será un entero. En este caso el resultado entero lo guardamos en una variable de tipo `float`, con lo que C mostrará el resultado con decimales: `2.000000`

Caso 3:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int a;
    int b;
```

```
    int c;
    float m;

    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);

    m=(a+b+c)/3.0;

    printf("%f", m);
    return 0;
}
```

En este caso el resultado de la división será un decimal, ya que el denominador contiene un decimal (en este caso `3.0`). El resultado con decimales se guarda en una variable de tipo `float`, y por pantalla se mostrará: `2.666667`.

3.7 Ejemplo: notaFinal

En esta PAC03 se empiezan a tratar dos nuevos aspectos: los **condicionales** y los **vectores**, dentro de los cuales contemplamos también los strings.

El siguiente ejemplo contempla bastantes puntos de los que se comentan en los módulos de teoría por la PAC03, con lo que seguramente este ejemplo será bastante más denso que no la propia PAC03. Por lo tanto, si algún aspecto cuesta entender inicialmente no se preocupe, es normal.

He añadido comentarios detallados dentro del propio ejemplo, para que sea lo más comprensible posible. El siguiente ejemplo calcula la nota final de una asignatura en función de una serie de condicionales y de operaciones:

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

/* Ejemplo:
 *
 * Queremos un programa que calcule la nota final
 * de una asignatura. La nota final se calcula a partir
 * de la EC (Evaluación continua) y la nota de la Práctica:
 *
 * Nota final = 30% AC + 70% Práctica
 *
```

```
 * Una vez entradas todas las notas, si se
 * detecta alguna que sea incorrecta (fuera del
 * rango [0.0 a 10.0]), se mostrará un mensaje
 * informativo por pantalla y no se realizará ningún
 * más operación.
 *
 * La EC está formada por 3 PEC: PEC1, PEC2, PAC3.
 * La nota de la EC se calcula mediante la
 * media de las 3 PEC.
 *
 * Si la nota de la EC es inferior a 4, no es necesario
 * realizar ningún cálculo: la asignatura queda suspendida.
 *
 * Si la nota de la EC es superior o igual a 4, se
 * calcula la nota final con la nota de
 * la Práctica.
 *
 * La nota final se mostrará en formato "grade letters",
 * según la siguiente relación:
 *
 * MH: 10
 * A: de 9.0 a 9.9
 * B: de 7.0 a 8.9
 * C+: de 5.0 a 6.9
 * C-: de 3.0 a 4.9
 * D: de 0.0 a 2.9
 *
 * Los puntos que trata este ejemplo:
 * - definición de vectores
 * - utilización del condicional if-else
 * - condicionales if-else anidados
 * - asignación de un string a una variable con strcpy
 * - reserva espacio para el finalizador '\0'
 */

#define PEC1 0
#define PEC2 1
#define PEC3 2
#define PRA 3
#define MAX_ACTIVITADES 4
#define MAX_CHARS 2+1     /* el +1 corresponde al finalizador '\0' */
#define PESO_EC 0.3       /* EC 30% peso en la nota final */
#define PESO_PRA 0.7      /* PRA 70% peso en la nota final */

int main(int argc, char **argv) {
```

```
    /* Vector que contiene las notas de
     * todas las actividades del curso
     */
    float notas[MAX_ACTIVITADES];

    /* Nota de evaluación continua: equivale
     * a la media de las 3 PEC
     */
    float notaEC;
    float notaFinalNumerica;

    /* String que contiene la nota final
     * de la asignatura (MH, A, B...)
     */
    char notaFinal[MAX_CHARS];

    /* Se pide por teclado las notas
     * de las 3 PEC y de la PRA
     */
    printf("Nota PEC1 : ");

    /* La asignación del valor se puede
     * hacer directamente sobre una posición
     * del vector
     */
    scanf("%f", &notas[PEC1]);

    /* Ídem para el resto de actividades */
    printf("Nota PEC2 : ");
    scanf("%f", &notas[PEC2]);
    printf("Nota PEC3 : ");
    scanf("%f", &notas[PEC3]);
    printf("Nota PRA : ");
    scanf("%f", &notas[PRA]);

    /* En primer lugar, comprobamos que
     * todas las notas del vector estén
     * dentro del rango [0.0 .. 10.0]
     */
    if (notas[PEC1] > 10.0 || notas[PEC2] > 10.0 ||
        notas[PEC3] > 10.0 || notas[PRA] > 10.0 ||
        notas[PEC1] < 0.0 || notas[PEC2] < 0.0 ||
        notas[PEC3] < 0.0 || notas[PRA] < 0.0) {

        printf("\n>> Error detectado en una o más notas:");
```

```
        printf("\n>> Se detiene el cálculo de la nota final.\n");

    } else {

        /* En este punto sabemos que las notas
         * están dentro del rango [0.0 .. 10.0]
         */

        /* Comprobamos ahora que la media de las 3 PEC
         * no es inferior a 4
         */
        notaEC = (notas[PEC1] + notas[PEC2] + notas[PEC3]) / 3;

        if (notaEC < 4) {

           /* Para dar mejor visibilidad, mostramos únicamente
            * el primer decimal de las notas numéricas
            */
           printf("\n>> Nota mínima de EC insuficiente: %.1f", notaEC);
           printf("\n>> Se detiene el cálculo de la nota final.\n");

        } else {

            /* En este punto todas las notas son correctas,
             * por lo que se realiza el cálculo de la
             * nota final
             */
            notaFinalNumerica = notaEC * PESO_EC + notas[PRA] * PESO_PRA;

            /* Ahora falta saber qué "grade letter" corresponde
             * a la notaFinalNumerica calculada; lo solucionamos
             * con nuevos if-else anidados
             */
            if (notaFinalNumerica <= 2.9) {

                /* Para asignar un string a una variable
                 * de tipo string usamos el comando
                 * strcpy en vez de '='
                 */
                strcpy(notaFinal, "D");

            } else {
                if (notaFinalNumerica <= 4.9) {
                    strcpy(notaFinal, "C-");
```

```
                } else {
                    if (notaFinalNumerica <= 6.9) {
                        strcpy(notaFinal, "C+");

                    } else {
                        if (notaFinalNumerica <= 8.9) {
                            strcpy(notaFinal, "B");

                        } else {
                            if (notaFinalNumerica <= 9.9) {
                                strcpy(notaFinal, "A");

                            } else {
                                strcpy(notaFinal, "MH");
                            }
                        }
                    }
                }

                /* Para finalizar, mostramos todos los resultados
                 * calculados por pantalla
                 */
                printf("\n>> Nota EC: %.1f", notaEC);
                printf("\n>> Nota PRA: %.1f", notas[PRA]);
                printf("\n>> Nota final: %s (%.1f)\n", notaFinal, notaFinalNumerica);
            }
        }
        return 0;
}
```

3.8 Frequently Made Mistakes

3.8.1 Strings: declaración como vectores de caracteres en lenguaje algorítmico

Pseudocódigo incorrecto:

```
var
    name: vector[MAX_CHAR] of char;
```

```
end var
```

En el algoritmo anterior, se intenta emular el lenguaje C a la hora de declarar una variable de tipo `string`, declarándola como un vector de caracteres. Esto es incorrecto y absolutamente innecesario, porque en lenguaje algorítmico **sí** que existe el tipo `string`, y por tanto su declaración es mucho más sencilla.

Pseudocódigo correcto:

```
var
    name: string;
end var
```

3.8.2 Sintaxis propia de C: funciones complejas

Pseudocódigo incorrecto:

```
function hotelCmp(hotel1: tHotel, hotel2: tHotel): boolean;
    if strcmp(hotel1.brand, hotel2.brand) = 0 then
        {...}
    end if
end function
```

La intención del algoritmo es comparar dos campos de tipo string de las variables `hotel1` y`hotel2`. Sin embargo se hace uso de la función `strcmp()`, propia de C e inexistente en lenguaje algorítmico.

En lenguaje algorítmico, la comparación de dos cadenas de caracteres es mucho más sencilla: se usa directamente `=`.

Pseudocódigo correcto:

```
function hotelCmp(hotel1: tHotel, hotel2: tHotel): boolean;
    if (hotel1.brand = hotel2.brand) then
        { ... }
    end if
```

3.8.3 Estructura alternativa: if s consecutivos

Este no es un error sintáctico o semántico, sino una mala práctica de diseño.

Pseudocódigo incorrecto:

```
if discountHotel >= 0 and discountHotel <= 10 then
    writeString("Invalid data");
end if

if discountHotel > 10 and discountHotel <= 20 then
    writeString("Not bad");
end if

if discountHotel > 20 and discountHotel <= 50 then
    writeString("Good!");
end if
```

El algoritmo anterior quiere mostrar un mensaje en pantalla en función del valor de la variable `discountData`. Para este propósito nada mejor que una estructura alternativa, pero no de la forma en que está diseñada.

Tened en cuenta que se han construido tres bloques `if…end if` independientes y consecutivos. Esto significa que durante la ejecución todos los bloques se evaluarán de forma consecutiva para decidir si hay que ejecutar el código interior o no. Esto no es necesario ni deseable, ya que aumenta el tiempo de ejecución.

Siempre que podamos, hay que construir estructuras alternativas anidadas y excluyentes, de forma que sólo se evalúan las condiciones estrictamente imprescindibles.

Pseudocódigo correcto:

```
if discountHotel >= 0 and discountHotel <= 10 then
    writeString("Invalid data");
    else if discountHotel > 10 and discountHotel <= 20 then
        writeString("Not bad");
        else if discountHotel > 20 and discountHotel <= 50 then
            writeString("Good!");
        end if
    end if
end if
```

3.8.4 Estructura alternativa: if s vacíos

Este no es un error sintáctico o semántico, sino una mala práctica de diseño.

Pseudocódigo incorrecto:

```
if discountHotel >= 0 then
else
    writeString("Invalid data");
end if
```

El algoritmo anterior quiere mostrar un mensaje de error en pantalla si el valor de la variable discountHotel es negativo, pero la estructura para hacerlo no es nada apropiada. En caso de que se cumpla la condición `discountHotel >= 0`, la estructura alternativa no ejecuta nada. Recordemos que es posible una estructura `if` sin `else`, y de hecho parece que en este caso haya añadido únicamente porque se creía lo contrario.

Pseudocódigo correcto:

```
if discountHotel < 0 then
    writeString("Invalid data");
end if
```

3.8.5 Estructura alternativa: interrumpir un algoritmo (I)

Pseudocódigo incorrecto:

```
algorithm nameAlgorithm
    if discountHotel < 0 then
        writeString("Invalid data");
        end algorithm;
    else
        writeString("Continue...");
        { ... }
    end if
```

A menudo nos piden que interrumpir la ejecución de un algoritmo en caso de que se dé alguna situación, por ejemplo, un error en la entrada de datos. A nivel de diseño algorítmico, no hay ninguna función ni sentencia pensada para efectuar esta acción de forma explícita. En el agorismo del ejemplo, se intenta hacerlo utiltzant la sentencia `end algorithm`, pero esto no es correcto.

Sencillamente hay que montar la estructura alternativa de forma que cuando se produzca el error no se ejecute otra sentencia, por ejemplo poniendo **todo** el código a ejecutar dentro del bloque `else`. Hay que asegurar, eso sí, que una vez salimos del bloque `if…else`, el algoritmo no tiene nada pendiente por ejecutar.

Pseudocódigo correcto:

```
algorithm nameAlgorithm
    if discountHotel < 0 then
        writeString("Invalid data");
    else
        writeString("Continue...");
        { ... }
    end if
end algorithm;
```

3.8.6 Estructura alternativa: interrumpir un algoritmo (II)

Pseudocódigo incorrecto:

```
algorithm nameAlgorithm
    if discountHotel < 0 then
        writeString("Invalid data");
        exit();
    else
        writeString("Continue...");
        { ... }
    end if
```

Hay veces que también se quiere emular incorrectamente en lenguaje algorítmico algunas funciones de C que interrumpen la ejecución del código, como por ejemplo `exit()`. Es incorrecto debido a que esta función no existe en lenguaje

algorítmico; de hecho, en lenguaje C, aunque exista, también hay que evitarla, ya que su uso no es una buena práctica de programación.

Pseudocódigo correcto:

```
algorithm nameAlgorithm
    if discountHotel < 0 then
        writeString("Invalid data");
    else
        writeString("Continue...");
        { ... }
    end if
end algorithm;
```

3.8.7 Constantes y números: Valores numéricos al código (hardcode)

Pseudocódigo incorrecto:

```
const
    NUM_SEATS1: integer = 34;
    NUM_RIDES: integer = 3;
    A1: integer = 1;
    A2: integer = 2;
    A3: integer = 3;
end const

var
    emptySeats[3]: vector of integer;
end var

{ input values }
writeString("EMPTY SEATS");
writeString(NAME_RIDE1);
writeString(" >> ");
emptySeats[1] := readInteger();
```

En el algoritmo anterior, se declara un vector de enteros, `emptySeats` con una longitud igual a 3 posiciones. Posteriormente, se lee un entero y se guarda en la primera posición del vector. En el enunciado del ejercicio se daban unas

constantes ya declaradas, y se pedía explícitamente utilizarlas para operar con los vectores.

El motivo no es otro que introducir la (buena) costumbre de utilizar constantes y evitar al máximo los valores numéricos directos al código (técnica conocida como *hardcode*). De esta manera, es mucho más fácil mantener el código posteriormente y realizar modificaciones.

En caso de que quisiéramos modificar la longitud del vector, o guardar los valores en posiciones diferentes, sólo deberíamos modificar la constante al principio del código, en vez de tener que modificar todas las líneas donde aparecen estos valores numéricos.

Pseudocódigo correcto:

```
const
    NUM_SEATS1: integer = 34;
    NUM_RIDES: integer = 3;
    A1: integer = 1;
    A2: integer = 2;
    A3: integer = 3;
end const

var
    emptySeats[NUM_RIDES]: vector of integer;
end var

{input values}
writeString("EMPTY SEATS");
writeString(NAME_RIDE1);
writeString(" >> ");
emptySeats[A1] := readInteger();
```

3.8.8 Constantes: declaración de constantes mal ubicada

Código incorrecto:

```
#include <stdio.h>

int main(int argc, char **argv) {
```

```
    #define MAX_LEN 15
    char brand[MAX_LEN];
    return 0;
}
```

Las constantes deben declararse **al principio del bloque de código**, y **fuera** de cualquier función (sea el `main` u otra), ya que al igual que los tipos de datos, las constantes entienden globales en todo el programa. No es correcto abrir un bloque de declaración de constantes dentro de un bloque central de código, y menos abrir varios bloques de constantes a medida que las necesitamos.

Código correcto:

```c
#include <stdio.h>
#define MAX_LEN 15

int main(int argc, char **argv) {
    char brand[MAX_LEN];
    /* ... */
    return 0;
}
```

3.8.9 Strings: comparación directa

Código incorrecto:

```c
#include <stdio.h>
#define MAX_LEN 15

int main(int argc, char **argv) {
    char name1[MAX_LEN];
    char name2[MAX_LEN];
    if (name1 == name2) {
        /* ... */
    }
    return 0;
}
```

En C las cadenas de caracteres (así como el resto de vectores) **no** se pueden comparar directamente con el operador `==`. En su lugar, hay dos opciones:

- En el caso de los vectores, se pueden comparar elemento a elemento, de forma individual
- En el caso de los strings, C dispone de funciones específicas tales como `strcmp()`.

Código correcto:

```c
#include <stdio.h>
#define MAX_LEN 15

int main(int argc, char **argv) {
    char name1[MAX_LEN];
    char name2[MAX_LEN];
    if (strcmp(name1, name2) == 0) {
        /* ... */
    }
    return 0;
}
```

Chapter 4

PEC04

4.1 Como tratar elementos de un vector con un bucle

Imaginemos que nos piden un programa que realice dos acciones:

1. Leer desde el canal estándar de entrada (teclado) 5 números e introducirlos en un vector de enteros.
2. Mostrar por el canal estándar de salida (pantalla) los 5 números del vector de enteros del punto anterior.

El algoritmo podría ser el siguiente:

```
const
   MAX_NUMS: integer = 5;
end const

algorithm vectorDeNumeros
   var
      i: integer;
      vectorNumeros: vector[MAX_NUMS] of integer;
   end var

   { Asignar valor a cada posición del vector desde teclado }
   for i := 1 to MAX_NUMS do
      writeString("Introduce número : ");
      vectorNumeros[i] := readInteger();
```

```
   end for

   { Mostrar por pantalla qué valor hay en cada posición del vector: }
   { se podría haber usado un bucle for, pero lo implementamos con un while }
   { para que se vea que también es posible hacerlo con este bucle }
   i := 1;

   while i   MAX_NUMS do
      writeString("La posición ");
      writeInteger(i);
      writeString(" del vector contiene el número ");
      writeInteger(vectorNumeros[i]);

      { En un bucle while es muy importante incrementar la variable usada como índice
      { antes de finalizar el bloque de instrucciones que ejecuta, ya que en caso cont
      { su valor sempre sería i == 1 }
      i := i+1;
   end while

end algorithm
```

Como se puede ver, con el fin de insertar/leer los elementos de un vector aprovechamos la iteración de un bucle para recorrerlos todos, uno a uno, mediante una variable que utilizamos de índice (en estos casos, la variable `i`).

La traducción a C del algoritmo podría ser así:

```c
#include <stdio.h>
#define MAX_NUMS 5

int main(int argc, char **argv) {
    int vectorNumeros [MAX_NUMS];
    int i;

    /* Asignar valor a cada posición del vector desde teclado */
    for (i = 0; i < MAX_NUMS; i++) {
        printf("Introduce número : ");
        scanf("%d", &vectorNumeros[i]);
    }

    /* Mostrar por pantalla qué valor hay en cada posición del vector:
     * se podría haber usado un bucle for, pero lo implementamos con un while
     * para que se vea que también es posible hacerlo con este bucle
     */
```

```
    i = 0;

    while (i < MAX_NUMS) {
        printf("\nLa posició %d del vector conté el número %d", i, vectorNumeros[i]);
        /* En un bucle while es muy importante incrementar la variable usada como índice
         * antes de finalizar el bloque de instrucciones que ejecuta, ya que en caso contrario
         * su valor sempre sería i == 1
         */
        i = i+1;
    }
    return 0;
}
```

4.2 Entrada continua de valores con un bucle

Imaginemos que tenemos que hacer un programa que vaya pidiendo números indefinidamente y que finalice únicamente en el caso de que el número ingresado sea par.

Una posible forma de hacerlo usando un único `while` sería la siguiente:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int numero;

    /* Pedimos una primera vez el número validar
     * justo antes de entrar en el bucle
     */
    printf("Teclea un número par : ");
    scanf("%d", &numero);

    while ((numero % 2) != 0) {
        /* Entra en el bucle en el caso de que el
         * resto de la división por 2 sea
         * diferente de 0 (equivale a no ser par)
         */
        printf("El número %d no es par !!\n", numero);

        /* Volvemos a pedir un número, ahora ya
         * dentro del bucle
```

```
         */
        printf("Teclea un número par : ");
        scanf("%d", &numero);
    }

    printf("El número %d és par\n", numero);
    return 0;
}
```

Antes de entrar en el bucle pedimos el valor de la variable `numero`. A continuación se utiliza la condición de bucle para validar si se trata de un número par o impar:

- Si el número es par, no se entra al bucle.
- Si el número es impar se cumple la condición del bucle y se entra; dentro del bucle se vuelve a pedir un valor para la variable numero y se vuelve a actuar igual que antes:
  - Si es impar, no se sale del bucle.
  - En caso contrario, se sale del bucle.

Finalmente se muestra por pantalla el mensaje *"El número X es par"*.

4.3 Como tratar valores en múltiples vectores

Imaginemos que queremos introducir por teclado una serie de datos de los trabajadores de nuestra empresa: DNI, días de antigüedad y sueldo bruto anual. Estos datos los introduciremos en tres vectores diferentes: uno para los DNI, el otro por la antigüedad y el último por el sueldo bruto anual. El valor del sueldo neto menual se calculará a partir del bruto y se guardará también en un vector.

El programa debe pedir por teclado los datos de 5 empleados, y al finalizar mostrará los valores por pantalla de la siguiente forma:

```
>> empleado: 39284019x
   antigüedad (días): 784
   bruto anual (€): 36874.78
   neto mensual (€): 1659.36

>> empleado: 31214557m
   antigüedad (días): 128
   bruto anual (€): 20015.30
   neto mensual (€): 1086.54
```

El sueldo neto mensual se calcula aplicando la siguiente retención y posteriormente dividiendo por 14 pagas:

sueldo bruto	retención
sueldo < 12450.0€	19.0%
12450.0€ <= sueldo < 20200.0€	24.0%
20200.0€ <= sueldo < 35200.0€	30.0%
35200.0€ <= sueldo < 60000.0€	37.0%
sueldo > 60000.0€	45.0%

El algoritmo podría ser el siguiente:

```
const
    MAX_ELEMS: integer = 5;
    TRAMO1: float = 12450.0;
    RETENCION1: float = 19.0;
    TRAMO2: float = 20200.0;
    RETENCION2: float = 24.0;
    TRAMO3: float = 35200.0;
    RETENCION3: float = 30.0;
    TRAMO4: float = 60000.0;
    RETENCION4: float = 37.0;
    RETENCION5: float = 45.0;
    NUM_PAGAS: integer = 14;
end const

algorithm datosSueldos
    var
        vDni: vector[MAX_ELEMS] of string;
        vAntiguedad: vector[MAX_ELEMS] of integer;
        vBrutoAnual: vector[MAX_ELEMS] of real;
        vNetoMensual: vector[MAX_ELEMS] of real;
        i: integer;
    end var

    { Lectura de datos desde el canal de entrada }
    { Se usa un único índice, 'i', para recorrer todos los vectores }
    for i := 1 to MAX_ELEMS do
        writeString("datos empleado num. ");
        writeInteger(i);
        writeString(":");
```

```
        writeString(">> dni : ");
        vDni[i] := readString();
        writeString(">> antigüedad : ");
        vAntiguedad[i] := readInteger();
        writeString(">> bruto anual : ");
        vBrutoAnual[i] := readReal();

        { Cálculo del sueldo neto mensual }
        if (vBrutoAnual[i] < TRAMO1) then
            vNetoMensual[i] := (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION1/100)) / NU
        else
            if (vBrutoAnual[i] < TRAMO2) then
                vNetoMensual[i] := (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION2/100)) /
            else
                if (vBrutoAnual[i] < TRAMO3) then
                    vNetoMensual[i] := (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION3/10
                else
                    if (vBrutoAnual[i] < TRAMO4) then
                        vNetoMensual[i] := (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION4
                    else
                        vNetoMensual[i] := (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION5
                    end if
                end if
            end if
        end if
    end for

    { Se muestran los datos por el canal de salida }
    { Se usa un único índice, 'i', para recorrer todos los vectores }
    for i := 1 to MAX_ELEMS do
        writeString(">> empleado: ");
        writeString(vDni[i]);
        writeString("   antigüedad (días): ");
        writeInteger(vAntiguedad[i]);
        writeString("   bruto anual (€): ");
        writeInteger(vBrutoAnual[i]);
        writeString("   neto mensual (€): ");
        writeInteger(vNetoMensual[i]);
    end for

end algorithm
```

Como lo podemos implementar en C? Una posible solución sería la siguiente:

```c
#include <stdio.h>

#define MAX_ELEMS 5
#define TRAMO1 12450.0
#define RETENCION1 19.0
#define TRAMO2 20200.0
#define RETENCION2 24.0
#define TRAMO3 35200.0
#define RETENCION3 30.0
#define TRAMO4 60000.0
#define RETENCION4 37.0
#define RETENCION5 45.0
#define NUM_PAGAS 14
#define MAX_DNI 9+1

typedef char tDni[MAX_DNI];

int main(int argc, char **argv) {
    tDni vDni[MAX_ELEMS];
    int vAntiguedad[MAX_ELEMS];
    float vBrutoAnual[MAX_ELEMS];
    float vNetoMensual[MAX_ELEMS];
    int i;

    /* Lectura de datos desde el canal de entrada.
       Se usa un único índice, 'i', para recorrer todos los vectores */
    for (i = 0; i < MAX_ELEMS; i++) {
        printf("datos empleado núm. %d : \n", i);
        printf(">> dni : ");
        scanf("%s", vDni[i]);
        printf(">> antigüedad : ");
        scanf("%d", &vAntiguedad[i]);
        printf(">> brut anual : ");
        scanf("%f", &vBrutoAnual[i]);

        /* Cálculo del sueldo neto mensual */
        if (vBrutoAnual[i] < TRAMO1) {
            vNetoMensual[i] = (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION1/100)) / NUM_PAGAS;
        } else {
            if (vBrutoAnual[i] < TRAMO2) {
                vNetoMensual[i] = (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION2/100)) / NUM_PAGAS;
            } else {
                if (vBrutoAnual[i] < TRAMO3) {
                    vNetoMensual[i] = (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION3/100)) / NUM_PA
                } else {
```

```
                    if (vBrutoAnual[i] < TRAMO4) {
                        vNetoMensual[i] = (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION4/
                    } else {
                        vNetoMensual[i] = (vBrutoAnual[i] - (vBrutoAnual[i]*RETENCION5/
                    }
                }
            }
        }
    }

    /* Se muestran los datos por el canal de salida.
       Se usa un único índice, 'i', para recorrer todos los vectores */
    for (i = 0; i < MAX_ELEMS; i++) {
        printf("\n>> empleado: %s \n", vDni[i]);
        printf("   antigüedad (días): %d \n", vAntiguedad[i]);
        printf("   bruto anual (€): %.2f \n", vBrutoAnual[i]);
        printf("   neto mensual (€): %.2f \n", vNetoMensual[i]);
    }
    return 0;
}
```

Dentro del bucle utilizamos la variable `i` como índice para ir recorriendo todos los vectores a la vez.

En ambos casos la inserción de los valores en los vectores la hacemos de la misma forma: utilizamos el índice `i` para determinar la posición del vector donde ubicaremos los valores:

```
/* ... */
scanf("%s", vDni[i]);
/* ... */
scanf("%d", &vAntiguedad[i]);
/* ... */
scanf("%f", &vBrutoAnual[i]);
```

Al final del ejemplo mostramos todos los empleados introducidos mediante un segundo bucle, mostrando todos los datos introducidos anteriormente y las calculadas.

4.4 Definición chars vs strings

En el lenguaje C, los caracteres se definen siempre con comilla simple ', mientras que por los string se utiliza la comilla doble ".

Ejemplo:

```c
#include <stdio.h>

int main(int argc, char **argv) {

    /* Asignación de valor a un string con comilla doble */
    char salutacion[]= "Hi World";
    /* Asignación de valor a un char con comilla simple */
    char exclamacion = '!';

    printf("%s %c\n", salutacion, exclamacion);
    return 0;
}
```

4.5 Ejemplo: pesoPromedio

Imaginemos que queremos hacer un programa que nos ayude a calcular nuestro peso promedio semanal. Añadiremos por teclado las lecturas diarias de nuestro peso en el programa y, en caso de encontrar algún valor incoherente la obviará y lo volverá a pedir.

En lenguaje algorítmico lo podemos implementar de la siguiente forma:

```
const
    NUM_DIAS: integer = 7;
    PESO_MIN: real = 50.0;
    PESO_MAX: real = 110.0;
end const

algorithm pesoPromedio

    var
        y: integer;   { Contador que utilizará nuestro programa }
```

```
    aux: real;   { Variable auxiliar para la lectura de pesos }
    vectorPesos: vector[NUM_DIAS] of real;   { Peso en Kg: 79.5, ... }
    sumaPesos: real;
end var

y := 1;
sumaPesos := 0;

{ Se lee desde teclado los pesos diarios, uno a uno }
while y   NUM_DIAS do

    { Leemos un valor desde el canal standard de entrada }
    writeString( "Introduce peso (Kg.):");
    aux: = readReal();

    { A continuación hay que revisar que este valor sea coherente. Tomamos como val
    "posibles" aquellos que estén entre PESO_MIN y PESO_MAX. Fijaros que la lectura
    se guarda temporalmente en la variable aux: cuando hayamos validado que conteng
    válido, lo añadiremos dentro del vector de pesos }

    if (PES_MIN   aux) and (aux   PES_MAX) then
        { En este punto, la variable aux contiene un valor correcto, con lo que ya
        añadir al vector de pesos }
        vectorPesos[i] := aux;

        { El siguiente punto es muy importante: como ya hemos añadido el peso corre
        incrementaremos la variable 'i', la cual nos sirve para acceder a una nueva
        del vector a la siguiente iteración del bucle }
        i = i + 1;

    else
        { En caso contrario, el peso es incorrecto con el que se muestra el corresp
        mensaje de error }
        writeString("Peso incorrecto!");

        { Importante: aquí no incrementamos la variable 'i', ya que el valor no es
        que en la siguiente iteración del bucle, se continúe intentando añadir el v
        de la posición 'i' del bucle}

    end if

    { En este punto tenemos el vectorPesos que tiene 7 (=NUM_DIES) pesos válidos }

    { Ahora utilizaremos un segundo bucle para recorrer todos los valores del vecto
    pedido: la media de todos ellos. Si hubiéramos querido, habríamos podido hacer
    en un único bucle, pero he preferido separarlo para que quede más claro que se
```

```
        dentro de cada uno de los dos bucles}

        for i := 1 to NUM_DIAS do

            { Dentro de la variable sumaPesos vamos sumando cada uno de los pesos
            de las posiciones del vector }
            sumaPesos: = sumaPesos + vectorPesos[i];

        end for

        { Para calcular la media, únicamente nos falta dividir el valor sumaPesos por
        número total de pesos introducidos o, lo que es lo mismo, NUM_DIAS }
        writeString( "El peso promedio semanal es:");
        writeReal(sumaPesos / NUM_DIAS);

end algorithm
```

Una posible implementación en lenguaje C de este algoritmo puede ser la siguiente:

```c
#include <stdio.h>

#define NUM_DIAS 7
#define PESO_MIN 50.0
#define PESO_MAX 110.0

int main(int argc, char **argv) {
    float vectorPesos[NUM_DIAS];
    int i;
    float aux;
    float sumaPesos;

    i = 0; /* Importante! en lenguaje C los vectores empiezan por 0 */
    sumaPesos = 0;

    /* Primer bucle: introducción y validación de datos */
    while (i<NUM_DIAS) {

        printf("Introduce peso (Kg.) : ");
        scanf("%f", &aux);

        if (PESO_MIN <= aux && aux <= PESO_MAX) {
            vectorPesos[i] = aux;
```

```
            i = i+1;
        } else {
            printf("Peso incorrecto!\n");
        }
    }

    /* Segundo bucle: cálculo intermedio para el promedio de pesos */
    for (i=0; i<NUM_DIAS; i++) {
        sumaPesos = sumaPesos + vectorPesos[i];
    }
    printf("El peso promedio semanal es : %.2f", sumaPesos/NUM_DIAS);
    return 0;
}
```

4.6 Frequently Made Mistakes

4.6.1 Estructura iterativa: for vs while

Este no es un error sintáctico o semántico, sino un error de diseño frecuente.

Código incorrecto:

```
#include <stdio.h>
#include <stdbool.h>
#define NUM 10

int main(int argc, char **argv) {
    int i;
    char password[NUM] = {'a', '1', 'h', '\0'};
    for (i = 0; password[i] != '\0'; i++){
        /* ... */
    }
    return 0;
}
```

En el código anterior, parece que queremos recorrer un vector de caracteres elemento a elemento, para ejecutar alguna acción. Para ello, decidimos utilizar un bucle y una condición final: las iteraciones se detendrán en el momento en que nos encontramos con el carácter especial `'\0'`, indicador de fin de cadena de caracteres.

El problema es que el bucle `for` no es la estructura más indicada para resolver este algoritmo, ya que está pensado para repetir un bloque de código un número de veces predeterminado, basado casi siempre en un índice (`i`) que se ha de incrementar/decrementar desde un valor inicial a uno final. En este caso, la condición final no tiene nada que ver con el valor numérico del índice, por lo tanto es mucho mejor utilizar el bucle `while`.

Código correcto:

```c
#include <stdio.h>
#include <stdbool.h>
#define NUM 10

int main(int argc, char **argv) {
    int i;
    char password[NUM] = {'a', '1', 'h', '\0'};
    i = 0;
    while (password[i] != '\0'){
        /* ... */
    i++;
    }
    return 0;
}
```

Es importante recordar que en el caso de la estructura `while`, el índice `i` **no se actualiza automáticamente**: se debe hacer manualmente antes de cerrar el bloque.

Chapter 5

PEC05

5.1 strcmp()

¿En qué se basa `strcmp()` para decidir que, por ejemplo, la letra `'O'` es mayor que la letra `'A'`?

La respuesta la tenemos en el código ASCII (numérico) que tiene asociado cada carácter. Por este motivo es normal que interprete diferente una `'A'` y una `'a'`, ya que son caracteres diferentes; de hecho según los valores ASCII, tenemos que `'A'` < `'a'`.

Por si queremos consultar la tabla ASCII por internet, la podemos generar nosotros mismos de la siguiente forma:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int i = 0;
    /* Relación de caracteres ASCII (únicamente se trata de un subconjunto!)
       ordenados de menor a mayor */
    for (i=33; i<=126; i++) {
        printf("%d : %c\n", i, i);
    }
    return 0;
}
```

5.2 scanf()

Cuando utilizamos `scanf()` hasta ahora siempre le hemos pasado el nombre de la variable precedido por `&`. Este carácter significa que le estamos pasando **la posición de la memoria** donde reside la variable facilitada.

Así, por ejemplo, cuando hacemos la siguiente operación `scanf("%d", &numero);` estamos pasando el valor que introducimos por teclado directamente a la posición de memoria donde tenemos guardada la variable `numero`. De ahí utilizar `&numero` en vez de `numero`. El mismo comportamiento tenemos los tipos primitivos `char`,`float`, etc.

Los vectores de caracteres en lenguaje C tienen una característica: el nombre del array contiene la dirección de memoria donde se guarda la primera posición del array.

Por ejemplo, cuando ejecutamos `scanf("%s", cadena);` el valor de cadena es la dirección de memoria inicial donde está ubicado el array. Dicho de otro modo, cadena contiene el mismo valor que `&cadena[0]` (es otra forma que tenemos para referirnos a la posición inicial en memoria del array).

A continuación se adjunta un ejemplo con todos estos conceptos:

```c
#include <stdio.h>
#define MAXIMO 10

int main(int argc, char **argv) {
    char cadena[MAXIMO];
    int numero;

    printf("Reservada la posición de memoria %p para la variable numero\n", &numero);
    printf("Reservada la posición de memoria %p para la variable cadena\n", cadena);
    printf("Reservada la posición de memoria %p para la variable cadena\n", &cadena[0]);

    printf("\nIntroduce un número entero: ");
    scanf("%d", &numero);
    printf("\nIntroduce una cadena: ");
    scanf("%s",cadena);

    printf("\nHas asignado los siguientes valores :\n");
    printf("numero = %d \n", numero);
    printf("cadena = %s \n", cadena);
    return 0;
}
```

5.3 El finalizador '\0' y strcmp()

Como se vio en el módulo **Cadenas de caracteres en C** de la XWiki, *"una cadena de caracteres o string es una secuencia de caracteres finalizada por el carácter '\0'"*. Por lo tanto tenemos que tener en cuenta que el finalizador `'\0'` quepa en nuestra variable, ya que ésta es la forma que tiene C de saber dónde termina un string en memoria.

Imaginemos que tenemos tres cadenas, con el mismo contenido pero de tamaño diferente (podemos tener posiciones vacías). ¿Qué pasa si las comparamos? Y si comparamos con una cadena de mismo contenido pero sin finalizador `'\0'`?

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

int main(int argc, char **argv) {
    char ciudad1[7] = "Girona";
    char ciudad2[8] = "Girona";
    char ciudad3[6] = "Girona"; /* no contiene el '\0' final */

    /* si strcmp devuelve 0 significa que las dos cadenas son iguales */
    printf ("¿Las variables ciudad1 y ciudad2 son iguales? %d\n", strcmp(ciudad1, ciudad2));
    printf ("¿Las variables ciudad1 y ciudad3 són iguales? %d\n", strcmp(ciudad1, ciudad3));
}
```

La forma que tenemos para forzar que una cadena no contenga el finalizador es limitando su tamaño a los caracteres que contendrá, sin tener en cuenta reservar uno por `'\0'`. En este caso lo hacemos con `char ciudad3[6] = "Girona"`.

La salida generada es la siguiente:

```
¿Las variables ciudad1 y ciudad2 son iguales? 0
¿Las variables ciudad1 y ciudad3 són iguales? -1
```

De ahí la importancia del finalizador de cadenas de caracteres. Por lo tanto, si por ejemplo tenemos una variable `x` de tipo string y de tamaño máximo 15, realmente en nuestro programa la definiremos con longitud **15+1**, para que quepa el finalizador `'\0'` en caso de que se ocupen los 15 caracteres anteriores.

5.4 El finalizador '\0' y strlen()

A continuación se expone un ejemplo en el que se muestra la importancia del finalizador `'\0'` en la función `strlen()` de C, la que nos devuelve el tamaño de una cadena de caracteres:

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

#define MAX_LETRAS 8+1

int main(int argc, char **argv) {
    char nombre[MAX_LETRAS];
    int numLetras;

    printf("Introduce un nombre: ");
    scanf("%s", nombre);

    /* Ejemplo: si en este punto hemos introducido el
     * nombre Quim, dentro de la cadena de caracteres nombre[]
     * tendremos los siguientes datos:
     *
     * nombre[0] = 'Q'
     * nombre[1] = 'u'
     * nombre[2] = 'i'
     * nombre[3] = 'm'
     * nombre[4] = '\0' (finalizador del string)
     * nombre[5] = valor aleatorio
     * nombre[6] = valor aleatorio
     * nombre[7] = valor aleatorio
     * nombre[8] = valor aleatorio
     *
     * El finalizador '\0' se añade automáticamente
     * al leer un string desde teclado mendiante scanf.
     *
     * El comando strlen(...) recorre el string posición
     * a posición para conocer su longitud. ¿Cuando finaliza
     * este recorrido? hay dos posibles opciones:
     *
     * - cuando encuentra el finalizador '\0'
     * - cuando llega a la última posición del string
     *
```

```
     * Por lo tanto, si hemos introducido Quim, el valor que
     * devolverá strlen(...) será 4.
     */

    numLetras = strlen(nombre);

    printf("El nombre \"%s\" tiene %d letras\n", nombre, numLetras);

    /* ¿Qué pasa si sobrescribimos el finalizador '\0' con
     * caracter cualquiera? por ejemplo 'X'
     */
    printf("\nSobrescribimos el finalizador '\0'.");
    nombre[numLetras] = 'X';

    /* Volvemos a calcular la longitud del string */
    numLetras = strlen(nombre);

    /* ¿Por qué motivo ahora ha cambiado la longitud de la
     * variable nombre, si no la hemos vuelto a definir?
     */
    printf("\nAhora el nombre \"%s\" tiene %d letras\n", nombre, numLetras);

    return 0;
}
```

Si se ejecuta el programa, la salida obtenida será similar a la siguiente:

```
Introduce un nombre: Quim
El nombre "Quim" tiene 4 letras

Sobrescribimos el finalizador '\0'.
Ahora el nombre "QuimX ! " tiene 9 letras
```

5.5 Ejemplo: comparacionStrings

La forma como comparamos strings en lenguaje algorítmico es diferente que la usada en lenguaje C:

- Lenguaje algorítmico: la comparación entre strings se hace con `=`.
- Lenguaje C: la comparación entre strings se hace con la función `strcmp()`, la cual realiza una comparación carácter a carácter de las dos cadenas y

como resultado:
- Devuelve `0`: si ambas cadenas son **iguales**.
- Devuelve `-1`: si la primera cadena < segunda cadena.
- Devuelve `1`: si la primera cadena > segunda cadena.

Un ejemplo donde realiza una comparación de dos strings puede ser el siguiente:

```
algorithm comparacioStrings

    var
        cadena1: string;
        cadena2: string;
    end var

    cadena1:= "UOC";
    cadena2:= "UAB";

    if (cadena1 = cadena2) then
        writeString(cadena1);
        writeString(" = ");
        writeString(cadena2);
    else
        if (cadena1 > cadena2) then
            writeString(cadena1);
            writeString(" > ");
            writeString(cadena2);
        else
            writeString(cadena1);
            writeString(" < ");
            writeString(cadena2);
        end if
    end if

end algorithm
```

En lenguaje C, la comparación carácter a carácter entre los string `"UOC"` y `"UAB"` que realiza la función `strcmp()` es la siguiente:

- Caracteres de la posición 0 de los dos string: **U**OC vs **U**AB. Son iguales, con lo que pasa a comparar el siguiente carácter.
- Caracteres de la posición 1 de los dos string: U**O**C vs U**A**B. Son diferentes (`'O'` > `'A'`), finaliza la comparación y la función `strcmp()` devuelve el valor 1.

********** continuar a partir de este punto

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

#define MAX_STRING 3+1

int main(int argc, char **argv) {
    char cadena1[MAX_STRING] = "UOC";
    char cadena2[MAX_STRING] = "UAB";
      int resultadoComparacion = 0;

      resultadoComparacion = strcmp(cadena1, cadena2);

      printf("Comparación strings \"%s\" y \"%s\" = %d\n", cadena1, cadena2, resultadoComparacion

      if (resultadoComparacion == 0) {
            printf("El resultado %d significa que el string \"%s\" == string \"%s\"\n", resultado
      } else if (resultadoComparacion == -1) {
            printf("El resultado %d significa que el string \"%s\" < string \"%s\"\n", resultadoC
      } else if (resultadoComparacion == 1) {
            printf("El resultado %d significa que el string \"%s\" > string \"%s\"\n", resultadoC
      }
      return 0;
}
```

La salida de la ejecución del programa C es:

```
Comparación strings "UOC" y "UAB" = 1
El resultado 1 significa que el string "UOC" > string "UAB"
```

5.6 Ejemplo: nóminas

Imaginemos que queremos un programa que nos permita entrar las nóminas de todos los empleados de nuestra empresa. Un empleado lo definimos como nombre (cadena de caracteres) + nómina (real). El programa debe mostrar al final de todo la relación de nóminas de todos los empleados, la media de todas las nóminas de la empresa, y quien cobra más y menos en la empresa.

Una posible forma de programa esto sería la siguiente:

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

#define MAX_EMPLEADOS 5
#define MAX_NOMBRE 20+1

typedef struct {
    char nom[MAX_NOMBRE];
    float nomina;
} tEmpleado;

int main(int argc, char **argv) {
    tEmpleado vEmpleados[MAX_EMPLEADOS];
    int i = 0;
    int maxNomina = 0;
    int minNomina = 0;
    float sumaNominas = 0;

    for (i=0; i<MAX_EMPLEADOS; i++) {
        printf("\nNombre empleado : ");
        scanf("%s", vEmpleados[i].nom);
        printf("Nómina : ");
        scanf("%f", &vEmpleados[i].nomina);
    }

    printf("\nListado de nóminas de empleados : \n\n");

    for (i=0; i<MAX_EMPLEADOS; i++) {
        sumaNominas = sumaNominas + vEmpleados[i].nomina;
        if (vEmpleados[i].nomina > vEmpleados[maxNomina].nomina) {
            maxNomina = i;
        }
        if (vEmpleados[i].nomina < vEmpleados[minNomina].nomina) {
            minNomina = i;
        }
        printf("%s --> %.2f €\n", vEmpleados[i].nom, vEmpleados[i].nomina);
    }

    printf("\nPromedio nóminas : %.2f €", sumaNominas/MAX_EMPLEADOS);
    printf("\nNómina mayor : %.2f € (%s)", vEmpleados[maxNomina].nomina, vEmpleados[ma
    printf("\nNómina menor : %.2f € (%s)", vEmpleados[minNomina].nomina, vEmpleados[mi
}
```

Como se puede ver, se utiliza un vector de `tEmpleat` de manera que, dada una longitud máxima del vector, iremos introduciendo los `tEmpleat` uno a uno dentro de él. Una vez hecho, volvemos a recorrer el vector de `tEmpleat` y realizar todos los cálculos que nos piden, así como mostrar por pantalla las nóminas de todos los empleados.

En este ejemplo la variable y hace de índice para recorrer el vector, y las variables `maxNomina` y `minNomina` también son índices: indican en qué posición están los empleados con la nómina más alta y más baja respectivamente.

5.7 Ejemplo: brisca

Ejemplo con diferentes recorridos realizado sobre tuplas guardadas en un vector, utilizando el juego de cartas de la brisca. Para explicar mejor el planteamiento de cada implementación, se han añadido comentarios detallados dentro del código.

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

/* Punto de partida: sabemos jugar perfectamente
 * a la brisca (https://es.wikipedia.org/wiki/Brisca),
 * pero en cambio somos un desastre contando
 * la puntuación de todas las cartas ganadas. Por
 * este motivo queremos hacer un programa que nos ayude
 * en esta tarea (y también para practicar diferentes
 * iteraciones con bucles). El programa pedirá por
 * teclado carta a carta y finalizará cuando se introduzca
 * un tipo de palo distinto de los definidos.
 * Deberá generar las siguientes salidas:
 * 1. Listar todas las cartas introducidas.
 * 2. Mostrar la carta con una puntuación más alta.
 * 3. Mostrar la puntuación total alcanzada.
 * 4. Comparar parejas de cartas (1a vs 2a, 2a vs 3a,
 * 3a vs 4a, ...) e indicar si son del mismo palo.
 */

#define MAX_NUM_CARTAS 48
#define NUM_VALORES 13
#define MAX_NUM_PALOS 4

typedef enum {OROS, COPAS, BASTOS, ESPADAS, FINALIZAR} tPalo;
```

```c
typedef struct {
    tPalo palo;
    int numero;
    int valor;
} tCarta;

int main(int argc, char **argv) {
    tCarta cartaAux;
    tCarta cartaDeMayorValor;
    tCarta vCartas[MAX_NUM_CARTAS];
    bool isFinalizado;
    int puntuacion;
    int i, j;
    bool isMismoPalo;

    /* Se inicializa el vector vValores con las
     * puntuaciones de todas las cartas. El nombre
     * de la carta hace de índice del vector, y el
     * valor guardado en aquella posición es su
     * puntuación. Ejemplo:
     * - una carta con un 1 devolverá 11 puntos,
     * correspondientes al valor de la posición 1.
     * - una carta con un 5 devolverá 0 puntos,
     * correspondientes al valor de la posición 5.
     */
    int vValores[NUM_VALORES] = {0, 11, 0, 10, 0, 0, 0, 0, 0, 0, 2, 3, 4};

    /* El siguiente vector nos ayudará a mostrar
     * por pantalla de forma automática la descripción
     * asociada a cada uno de los valores definidos en
     * el enumerativo tPalo
     */
    char* vNombres[MAX_NUM_PALOS] = {"Oros", "Copas", "Bastos", "Espadas"};

    /* La variable booleana se utilizará para
     * finalizar el primer bucle de introducción
     * de cartas desde teclado
     */
    isFinalizado = false;

    i = 0;
    j = 0;
    puntuacion = 0;

    printf("Tipo de carta: 0=OROS, 1=COPAS, 2=BASTOS, 3=ESPADAS\n");
```

```
    while (!isFinalizado) {
        printf("\nIntroduce tipo de carta: ");
        scanf("%u", &cartaAux.palo);

        /* Si el palo introducido no corresponde a ninguno de los
         * cuatro definidos, significa que no se quiere
         * entrar otra carta desde teclado
         */
        if (cartaAux.palo != OROS && cartaAux.palo != COPAS
            && cartaAux.palo != BASTOS && cartaAux.palo != ESPADAS) {
            isFinalizado = true;

        } else {
            printf("Introduce número de carta: ");
            scanf("%d", &cartaAux.numero);

            /* Se valida que el número de la carta
             * introducida sea válido: valor comprendido
             * entre 1 y 12
             */
            if (cartaAux.numero < 1 || cartaAux.numero > 12) {
                printf("Error: número de la carta incorrecto!\n");

            } else {
                /* Se calcula el valor de la carta y
                 * asigna al campo valor de la tupla
                 * cartaAux (de tipo tCarta)
                 */
                cartaAux.valor = vValores[cartaAux.numero];
                vCartas[i] = cartaAux;

                /* La variable 'i' contendrá el número
                 * de cartas válidas introducidas por
                 * teclado.
                 */
                i = i + 1;
            }
        }
    }

    /* Salida 1:
     * Recorremos el vector de cartas para mostrar
     * por pantalla todos los elementos (cartas)
     * que contiene.
     */
```

```
    printf("\nRelación de cartas introducidas: \n");
    for (j = 0; j < i; j++) {
        printf("\t%d de %s (%d puntos) \n", vCartas[j].numero, vNombres[vCartas[j].pal
    }

    /* Salida 2:
     * Recorremos el vector de cartas y buscamos
     * la que tiene una mayor puntuación. En caso
     * de empate, mostramos la última introducida.
     */

    /* Inicializamos el valor de cartaDeMayorValor
     * a valor = 0
     */
    cartaDeMayorValor.valor = 0;
    printf("\nCarta de mayor puntuación: \n");
    for (j = 0; j < i; j++) {
        /* En cada iteració del bucle ens assegurem
         * que cartaDeMajorValor sigui la carta
         * amb un valor més gran. Comparem la carta
         * que estem tractant actualment amb la carta
         * que fins ara hem trobat de major valor.
         */
        if (vCartas[j].valor >= cartaDeMayorValor.valor) {
            cartaDeMayorValor.palo = vCartas[j].palo;
            cartaDeMayorValor.numero = vCartas[j].numero;
            cartaDeMayorValor.valor = vCartas[j].valor;
        }
    }
    printf("\t%d de %s (%d puntos) \n", cartaDeMayorValor.numero, vNombres[cartaDeMayo

    /* Salida 3:
     * Recorremos el vector de cartas y vamos sumando
     * todos los valores, a fin de obtener la
     * puntuación total.
     */
    printf("\nPuntuación total: \n");
    for (j = 0; j < i; j++) {
        puntuacion = puntuacion + vCartas[j].valor;
    }
    printf("\t%d puntos \n", puntuacion);

    /* Salida 4:
     * Recorremos el vector de cartas y vamos comparando
     * parejas de carta: la que estamos tratando con
```

```
     * la que viene a continuación. Mostramos por pantalla
     * si son del mismo palo o no.
     */
    printf("\nComparación de cartas: \n");

    /* Importante: en este recorrido del vector
     * de cartas vamos comparando la carta de la
     * posición actual j con la carta que está
     * en la siguiente posición j+1. Esto significa
     * que el límite de las iteraciones del
     * bucle pasa a ser i-1 (un elemento antes
     * del final), ya que cuando tratamos este
     * elemento lo compararemos con el siguiente,
     * que es el último del vector. Si en vez
     * tener i-1 tuviéramos sólo i, en
     * la última iteración daría error, ya
     * que se estaría accediendo a una posición
     * incorrecta del vector de cartas (i+1).
     */
    for (j = 0; j < i-1; j++) {

        /* Para comparar la carta de la posición actual
         * con la que hay en la siguiente posición,
         * utilizamos los índices j y j+1 respectivamente.
         */
        isMismoPalo = (vCartas[j].palo == vCartas[j+1].palo);
        printf("\t%d de %s vs %d de %s: ", vCartas[j].numero, vNombres[vCartas[j].palo], vCartas[
        if (isMismoPalo) {
            printf("son del mismo palo!\n");
        } else {
            printf("son de palos distintos!\n");
        }
    }
    return 0;
}
```

Un ejemplo de ejecución sería:

```
Tipo de carta: 0=OROS, 1=COPAS, 2=BASTOS, 3=ESPADAS

Introduce tipo de carta: 0
Introduce número de carta: 1
```

```
Introduce tipo de carta: 0
Introduce número de carta: 10

Introduce tipo de carta: 0
Introduce número de carta: 5

Introduce tipo de carta: 2
Introduce número de carta: 10

Introduce tipo de carta: 2
Introduce número de carta: 1

Introduce tipo de carta: 3
Introduce número de carta: 4

Introduce tipo de carta: 9

Relación de cartas introducidas:
    1 de Oros (11 puntos)
    10 de Oros (2 puntos)
    5 de Oros (0 puntos)
    10 de Bastos (2 puntos)
    1 de Bastos (11 puntos)
    4 de Espadas (0 puntos)

Carta de mayor puntuación:
    1 de Bastos (11 puntos)

Puntuación total:
    26 puntos

Comparación de cartas:
    1 de Oros vs 10 de Oros: son del mismo palo!
    10 de Oros vs 5 de Oros: son del mismo palo!
    5 de Oros vs 10 de Bastos: son de palos distintos!
    10 de Bastos vs 1 de Bastos: son del mismo palo!
    1 de Bastos vs 4 de Espadas: son de palos distintos!
```

5.8 Frequently Made Mistakes

5.8.1 Definición de tipo: definición de tuplas mal ubicada

Código incorrecto:

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

int main(int argc, char **argv) {
    typedef struct {
        int id;
        char brand[MAX_LEN];
        float price;
    } tHotel;

    tHotel myHotel;
    /* ... */
    return 0;
}
```

Los tipos de datos se declararán **al principio del bloque de código**, y fuera de cualquier función (sea el `main` u otra), ya que los tipos son globales en todo el programa, y las variables asociadas a los tipos se declaran posteriormente en cada una de las funciones que se necesitan.

Las tuplas (`structs`) no son una excepción, y por tanto no es correcto abrir un bloque de definición de tuplas en un bloque central de código (y menos abrir varios bloques de definición de tuplas a medida que los necesitamos).

Código correcto:

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

typedef struct{
    int id;
    char brand[MAX_LEN];
    float price;
} tHotel;

int main(int argc, char **argv) {
    tHotel myHotel;
    /* ... */
    return 0;
}
```

5.8.2 Definición de tipo: definición mal ubicada

Código incorrecto:

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

int main(int argc, char **argv) {
    typedef enum {BUDGET, INN, RESORT} tTypeHotel;
    typedef enum {STANDARD, SUITE} tTypeRoom;

    tTypeHotel myHotelType = BUDGET;
    tTypeRoom myRoomType = STANDARD;
    /* ... */
    return 0;
}
```

Los tipos de datos se declararán **al principio del bloque de código**, y fuera de cualquier función (sea el `main` u otra), ya que los tipos son globales en todo el programa, y las variables asociadas a los tipos se declaran posteriormente en cada una de las funciones que se necesitan. No es correcto abrir un bloque de definición de tipo dentro de un bloque central de código (y menos abrir varios bloques de definición de tipo a medida que los necesitamos).

Código correcto:

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

typedef enum {BUDGET, INN, RESORT} tTypeHotel;
typedef enum {STANDARD, SUITE} tTypeRoom;

int main(int argc, char **argv) {
    tTypeHotel myHotelType = BUDGET;
    tTypeRoom myRoomType = STANDARD;
    /* ... */
    return 0;
}
```

Chapter 6

PEC06

6.1 Diferencias entre funciones y acciones

A continuación se explican las diferencias entre una **función** y una **acción**, qué tipos de parámetros utilizan, cómo se implementan en lenguaje C, y finalmente qué pasa cuando un parámetro es una tupla. Es importante que se vayan consolidando todos estos conceptos.

Las principales diferencias son:

	funciones	acciones
Devuelven un valor?	sí	no
Tipo de parámetros	entrada (in)	entrada (in), salida (out), entrada / salida (inout)

El retorno de valor de las funciones permite que este se pueda asignar a una variable, hecho que no permiten las acciones.

Por ejemplo, imaginemos que queremos implementar en lenguaje C la función `suma()`; una posible implementación podría ser:

```c
#include <stdio.h>

/* Predeclaración de la función */
```

```c
int suma(int n1, int n2);

int main(int argc, char **argv) {
    int num1 = 3;
    int num2 = 2;
    int resultado = 0;
    printf("Valor de num1 = %d\n", num1);
    printf("Valor de num2 = %d\n", num2);
    printf("Valor de resultado = %d\n", resultado);
    printf(">> Inicio ejecución función\n");
    resultado = suma(num1, num2);
    printf("Suma = %d\n", resultado);
    printf(">> Fin ejecución función\n");
    printf("Valor de resultado = %d\n", resultado);
    return 0;
}

/* Implementación de la función */
int suma(int n1, int n2) {
    return (n1+n2);
}
```

La ejecución generará la siguiente salida:

```
Valor de num1 = 3
Valor de num2 = 2
Valor de resultado = 0
>> Inicio ejecución función
Suma = 5
>> Fin ejecución función
Valor de resultado = 5
```

Como se puede ver, el valor de retorno de la función `suma()` la asignamos a la variable `resultado`.

En una función, los parámetros pasados siempre serán **de entrada** (in): esto significa que dentro de la función únicamente serán valores de consulta, no los modificaremos para nada.

Para entender bien cómo se implementa una acción, relacionaremos ejemplos similares con los diferentes tipos de parámetros que puede tener una acción: **entrada** (in), **salida** (out) y **entrada/salida** (inout).

6.1.1 Parámetros de entrada (in)

Son aquellos parámetros que se pasan a una **acción** y de los que únicamente utilizaremos su contenido. Esto significa que vamos a trabajar con ellos en *modo lectura*: obtendremos sus valores para realizar cálculos, pero nunca modificaremos su contenido. Ejemplo:

```c
#include <stdio.h>

/* Predeclaración de la acción */
void suma(int n1, int n2);

int main(int argc, char **argv) {
    int num1 = 3;
    int num2 = 2;
    printf("Valor de num1 = %d\n", num1);
    printf("Valor de num2 = %d\n", num2);
    printf(">> Inicio ejecución acción\n");
    suma(num1, num2);
    printf(">> Fin ejecución acción\n");
    printf("Valor de num1 = %d\n", num1);
    printf("Valor de num2 = %d\n", num2);
    return 0;
}

/* Implementación de la acción */
void suma(int n1, int n2) {
    int resultado = n1 + n2;
    printf("Suma = %d\n", resultado);
}
```

La ejecución generará la siguiente salida:

```
Valor de num1 = 3
Valor de num2 = 2
>> Inicio ejecución acción
Suma = 5
>> Fin ejecución acción
Valor de num1 = 3
Valor de num2 = 2
```

Como se puede observar, ni `num1` ni `num2` han modificado su valor después de la ejecución de la acción: son **parámetros de entrada**.

Este tipo de parámetro también se referencia como **parámetro por valor**, o **paso por valor**, ya que lo que estamos pasando es un **valor**, no un puntero a un valor.

6.1.2 Parámetros de salida (out)

A diferencia de los parámetros de entrada, los de salida se utilizan únicamente para guardar valores. Pueden contener cualquier valor inicial, que este no será utilizado dentro de la acción. Una vez realizados todos los cálculos de la acción, el resultado final se guardará en el parámetro de salida. Ejemplo:

```c
#include <stdio.h>

/* Predeclaración de la acción */
void suma(int n1, int n2, int *res);

int main(int argc, char **argv) {
    int num1 = 3;
    int num2 = 2;
    int resultado = 0;
    int *pResultado = &resultado;
    printf("Valor de num1 = %d\n", num1);
    printf("Valor de num2 = %d\n", num2);
    printf("Valor de resultado = %d\n", resultado);
    printf(">> Inicio ejecución acción\n");
    suma(num1, num2, pResultado);
    printf("Suma = %d\n", resultado);
    printf(">> Fin ejecución acción\n");
    printf("Valor de resultado = %d\n", resultado);
    return 0;
}

/* Implementación de la acción */
void suma(int n1, int n2, int *res) {
    *res = n1 + n2;
}
```

La ejecución generará la siguiente salida:

```
Valor de num1 = 3
Valor de num2 = 2
Valor de resultado = 0
>> Inicio ejecución acción
Suma = 5
>> Fin ejecución acción
Valor de resultado = 5
```

De momento ignoraremos que se trata de un puntero: independientemente del valor que tenga nada al pasar por parámetro, cuando finalice la acción contendrá la suma de los otros dos parámetros de entrada `num1` y `num2`.

El valor resultado ha cambiado, de `0` a`5`. Se considera un **parámetro de salida** porque, independientemente del valor inicial que tenga este, no se utiliza para nada y al finalizar la acción contendrá el resultado de la operación `suma()`.

Ahora sí comentamos el hecho de utilizar el puntero: el modo que tenemos en C para modificar una variable definida fuera de una acción desde dentro de la misma es trabajando precisamente con su dirección de memoria.

Este tipo de parámetro también se referencia como **parámetro por referencia**, o **paso por referencia**, ya que pasamos un puntero a un valor, no el valor en sí mismo.

6.1.3 Parámetros de entrada/salida (inout)

Este tipo de parámetro es una suma de los dos comportamientos anteriores: por un lado su valor importa a la hora de realizar los cálculos de la acción, ya la vez al finalizar la ejecución de la acción tendrá un valor diferente, también significativo por tratarse del resultado final del cálculo. Ejemplo:

```c
#include <stdio.h>

/* Predeclaración de la acción */
void suma(int *pN1, int n2);

int main(int argc, char **argv) {
    int num1 = 3;
    int num2 = 2;
    int *pNum1 = &num1;
```

```
    printf("Valor de num1 = %d\n", num1);
    printf("Valor de num2 = %d\n", num2);
    printf(">> Inicio ejecución acción\n");
    suma(pNum1, num2);
    printf("Suma = %d\n", num1);
    printf(">> Fin ejecución acción\n");
    printf("Valor de num1 = %d\n", num1);
    return 0;
}

/* Implementación de la acción */
void suma(int *pN1, int n2) {
    *pN1 = *pN1 + n2;
}
```

La ejecución generará la siguiente salida:

```
Valor de num1 = 3
Valor de num2 = 2
>> Inicio ejecución acción
Suma = 5
>> Fin ejecución acción
Valor de num1 = 5
```

En este caso se ha realizado la suma como `num1 = num1 + num2`: el resultado de la suma de las dos variables se guarda en la primera de ellas. Así pues el valor de la variable `num1` importa tanto a la entrada como a la salida de la acción, con lo que se trata de un **parámetro de entrada/salida**.

De igual forma que en el caso anterior, este tipo de parámetro también se referencia como **parámetro por referencia**, o **paso por referencia**, ya que pasamos un puntero.

6.1.4 Tuplas como parámetros

Cuando un parámetro de una acción/función es una **tupla**, la forma como accederemos a sus atributos dentro de la acción/función variará en función del tipo de parámetro:

- Parámetros de **entrada**: el accesor los atributos es el `.` (un punto). Por lo tanto dentro de la función/acción, accederemos a los atributos de la tupla pasada por parámetro de la siguiente forma: `nombreTupla.nombreAtributo`.

- Parámetros de **salida** y de **entrada/salida**: en este caso el accesor los atributos de la tupla es `->`. Así, dentro de la acción podremos hacer referencia a los atributos de la tupla pasada por parámetro de la forma: `nombreTupla-> nombreAtributo`.

6.2 Ejemplo: uso de acciones

A continuación se adjunta un ejemplo inventado de cómo se pueden definir acciones que permitan modificar los atributos de una tupla pasada como puntero. Dentro del código hay comentarios adicionales para que quede lo más claro posible:

```c
#include <stdio.h>

#define MAX_CHAR 10+1

typedef struct {
    char nombre[MAX_CHAR];
    float nomina;
} tEmpleado;

/* Predeclaración de las acciones */

void printEmpleado(tEmpleado empleado);
void setNominaEmpleado(tEmpleado *empleado, float nomina);
void setNombreEmpleado(tEmpleado *empleado, char nombre[MAX_CHAR]);

int main(int argc, char **argv) {
    /* Declaramos nuevoEmpleado de tipo tEmpleado,
     * pero no le daremos ningún valor directamente
     * a sus dos atributos (número y nómina): lo
     * haremos mediante dos acciones
     */
    tEmpleado nuevoEmpleado;

    /* Los valores de los atributos número y nómina los
     * leeremos desde teclado y los guardaremos inicialmente
     * en las siguientes dos variables
     */
    char nombre[MAX_CHAR];
    float nomina;
```

```
    printf("\nNombre empleado : ");
    scanf("%s", nombre);

    printf("Nómina empleado : ");
    scanf("%f", &nomina);

    /* Asignamos el número y la nómina al tEmpleado nuevoEmpleado
     * utilizando las acciones definidas. Fijaos
     * que el parámetro nuevoEmpleado lo pasamos como puntero
     * (pasamos su dirección en memoria, ya que va
     * precedido de &)
     */
    setNombreEmpleado(&nuevoEmpleado, nombre);
    setNominaEmpleado(&nuevoEmpleado, nomina);

    /* Mostramos los datos de la tupla tEmpleado nuevoEmpleado
     * por pantalla
     */
    printEmpleado(nuevoEmpleado);
    return 0;
}

/* Implementación de las acciones */

void printEmpleado(tEmpleado empleado) {
    /* El parámetro empleado es de entrada, por lo
     * que el acceso a sus atributos lo haremos
     * con un punto: empleado.nombre, empleado.nomina
     */
    printf("\nDatos del empleado: \n");
    printf("\tNombre: %s\n", empleado.nombre);
    printf("\tNómina: %.2f €\n", empleado.nomina);
}

void setNominaEmpleado(tEmpleado *empleado, float nomina) {
    /* El parámetro empleado (de tipo inout) es un puntero,
     * para que desde dentro de la acción sea posible
     * modificar el valor (de un atributo) del empleado
     * definido al main de nuestro programa, fuera el ámbito
     * de la acción.
     * El acceso a un atributo de un elemento referenciado con
     * un puntero se hace con '->': empleado->nomina.
     */
    empleado->nomina = nomina;
}
```

```c
void setNombreEmpleado(tEmpleado *empleado, char nombre[MAX_CHAR]) {
    /* Idem que en la acción setNominaEmpleado. En este caso
     * además hay que recordar que la asignación de strings
     * la hacemos con la función strcpy de C, en vez
     * de utilizar la asignación habitual '=' de los tipos
     * primitivos (char, int, float, ...)
     */
    strcpy(empleado->nombre, nombre);
}
```

Un ejemplo de ejecución sería:

```
Nombre empleado : Laura
Nómina empleado : 1850.32

Datos del empleado:
    Nombre: Laura
    Nómina: 1850.32 €
```

6.3 Ejemplo: uso de funciones

Si necesitamos un método que como resultado devuelva un string, en vez de utilizar una función utilizaremos una acción con un parámetro de salida o de entrada/salida. Las funciones las usaremos para devolver valores de **tipo primitivo** (entero, decimal, carácter), dejando los **tipos compuestos** (vectores, tuplas, etc.) para las acciones.

A continuación se expone un ejemplo para que quede más claro:

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

/* Programa que, dada una hora, indica
 * a qué parte del día corresponde.
 * Por "parte del día" se entiende: madrugada,
 * mañana, mediodía, tarde, atardecer.
```

```c
 * La hora se consigue mediante una
 * función, lo devuelve un valor
 * entero en formato HHMM (donde HH = hora y
 * MM = minuto).
 * Por otra parte, el cálculo de la parte
 * del día se realiza con una acción,
 * la que recibe dos parámetros: uno de entrada
 * correspondiente a la hora en formato HHMM,
 * y otro de salida que contendrá
 * la parte del día (string) que corresponde
 * a la hora.
 */

#define MAX_CHARS 8+1

/* Predeclaración de funciones y acciones */
void calcularParteDelDia(int hora, char *parte);
int pedirHora();

/* Programa principal */
int main(int argc, char **argv) {
    int hora;
    char parteDelDia[MAX_CHARS];
    hora = pedirHora();
    calcularParteDelDia(hora, parteDelDia);
    printf("La hora %d corresponde a: %s \n", hora, parteDelDia);
    return 0;
}

/* Implementación de funciones y acciones */

/* Función que devuelve un entero, correspondiente
 * a la hora introducida por teclado, en formato
 * HHMM
 */
int pedirHora() {
    int hora;
    printf("Teclea hora (formato HHMM) : ");
    scanf("%d", &hora);
    return hora;
}

/* Acción que, dada una hora (parámetro
 * de entrada), calcula qué parte del día
 * corresponde (parámetro de salida). La
```

```
 * parte del día es de tipo string. El
 * parámetro de salida debe ser un
 * puntero, para que desde dentro de
 * la acción se pueda modificar el valor
 * de la variable definida fuera del
 * ámbito de la acción, dentro del main.
 */
void calcularParteDelDia(int hora, char *parte) {

    /* Particionado horario extraído de
     * http://www.wikilengua.org/index.php/Hora
     */

    /* 0 corresponde a 0000 */
    /* 600 corresponde a 0600 */
    if (hora >= 0 && hora <= 600) {
        strcpy(parte, "madrugada");
    } else {
        if (hora > 600 && hora <= 1200) {
            strcpy(parte, "mañana");
        } else {
            if (hora > 1200 && hora <= 1800) {
                strcpy(parte, "tarde");
            } else {
                if (hora > 1800 && hora <= 2359) {
                    strcpy(parte, "noche");
                } else {
                    strcpy(parte, "unknown!");
                }
            }
        }
    }
}
```

Un ejemplo de ejecución:

```
Teclea hora (formato HHMM) : 1906
La hora 1906 corresponde a: noche
```

6.4 Ejemplo: nóminas

Imaginemos que trabajamos con los empleados de una empresa. Supongamos que después de introducir n-empleados en nuestro sistema, queremos una función que nos devuelva el empleado que tiene la nómina menor de todas.

Como esperamos un valor de retorno, estamos ante una función. A la función le pasaremos un vector de empleados con todos los empleados que previamente hemos introducido en nuestra empresa.

Sin entrar en la codificación de la función, el main de nuestro programa puede ser similar al siguiente:

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

#define MAX_EMPLEADOS 2
#define MAX_CARACTERES 20+1

typedef struct {
    char nombre[MAX_CARACTERES];
    char apellido[MAX_CARACTERES];
    float nomina;
} tEmpleado;

int main(int argc, char **argv) {
    tEmpleado vEmpleados[MAX_EMPLEADOS];
    int i = 0;

    for (i=0; i<MAX_EMPLEADOS; i++) {
        printf("\nNombre empleado %d : ", i);
        scanf("%s", vEmpleados[i].nombre);
        printf("Apellido empleado %d : ", i);
        scanf("%s", vEmpleados[i].apellido);
        printf("Nómina empleado %d : ", i);
        scanf("%f", &vEmpleados[i].nomina);
    }
    tEmpleado empleado = buscarEmpleadoNominaMenor(vEmpleados);
    printf("\nEl empleado con menor nómina es %s, %s (%.2f €)", empleado.apellido, empl
    return 0;
}
```

Hasta este punto no hay nada nuevo: utilizamos un vector de `tEmpleado` para ir introduciendo los empleados por teclado, y por cada empleado pedimos el

nombre, el apellido y su nómina.

Ahora toca implementar la función `buscarEmpleadoNominaMenor()`, que recibe como parámetro el vector de empleados de la empresa.

De momento olvidemos de que estamos ante una función, simplemente centrémonos en cuál es la acción que queremos hacer. En este caso, queremos hacer un programa que recorra uno a uno todos los elementos de un vector, y encuentre el empleado que cobra menos.

Una posible codificación sería la siguiente:

```c
int i = 0;
int nominaMenor = 0;
for (i=0; i<MAX_EMPLEATS; i++) {
    if (vector[i].nomina < vector[nominaMenor].nomina) {
        nominaMenor = i;
    }
}
```

Este fragmento de código simplemente recorre uno a uno todos los `tEmpleat` de un vector, comparando la nómina menor encontrada hasta el momento con la del empleado que está tratando en cuestión: si esta segunda es inferior, nos quedamos con su posición dentro del vector como empleado con la nómina menor.

Ahora convertimos este fragmento de código en una función:

```c
tEmpleat buscarEmpleadoNominaMenor(tEmpleado vector[MAX_EMPLEADOS]) {
    int i = 0;
    int nominaMenor = 0;

    for (i=0; i<MAX_EMPLEADOS; i++) {
        if (vector[i].nomina < vector[nominaMenor].nomina) {
            nominaMenor = i;
        }
    }
    return vector[nominaMenor];
}
```

Como se puede ver, la única diferencia es que ahora el código tiene la cabecera de la función, la cual nos dice que recibe como parámetro un elemento de tipo

vector de `tEmpleado`, y que devolverá un elemento de tipo `tEmpleado`.

De esta forma el programa completo queda de la siguiente manera:

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

#define MAX_EMPLEADOS 2
#define MAX_CARACTERES 20+1

typedef struct {
    char nombre[MAX_CARACTERES];
    char apellido[MAX_CARACTERES];
    float nomina;
} tEmpleado;

/* Predeclaración de funciones/acciones */
tEmpleat buscarEmpleadoNominaMenor(tEmpleado vector[MAX_EMPLEADOS]);

int main(int argc, char **argv) {
    tEmpleado vEmpleados[MAX_EMPLEADOS];
    int i = 0;

    for (i=0; i<MAX_EMPLEADOS; i++) {
        printf("\nNom empleado %d : ", i);
        scanf("%s", vEmpleados[i].nom);
        printf("Cognom empleado %d : ", i);
        scanf("%s", vEmpleados[i].cognom);
        printf("Nòmina empleado %d : ", i);
        scanf("%f", &vEmpleados[i].nomina);
    }
    tEmpleado empleado = buscarEmpleadoNominaMenor(vEmpleados);
    printf("\nEl empleado con menor nómina es %s, %s (%.2f €)", empleado.apellido, empl
    return 0;
}

/* Implementación de funciones/acciones */
tEmpleado buscarEmpleadoNominaMenor(tEmpleado vector[MAX_EMPLEADOS]) {
    int i = 0;
    int nominaMenor = 0;
    for (i=0; i<MAX_EMPLEADOS; i++) {
        if (vector[i].nomina < vector[nominaMenor].nomina) {
            nominaMenor = i;
```

```
        }
    }
    return vector[nominaMenor];
}
```

Este ejemplo puede parecer sencillo para que el tipo de comparación que estamos haciendo es numérica: comparamos dos campos de tipo `float` (las nóminas de dos empleados).

Ampliamos ahora la funcionalidad de nuestro programa: queremos que pueda hacer una búsqueda por apellido entre nuestros empleados. Para hacer esta búsqueda, habrá comparar una cadena de caracteres con el apellido de cada empleado.

Para no hacer la función más compleja de lo necesario, imaginamos que ningún apellido se puede repetir y que siempre encontraremos un apellido como el que buscamos (el objetivo es ver cómo funciona `strcmp()`). Así pues nuestra función recibirá un vector de empleados y un apellido a buscar, y devolverá el empleado con ese apellido.

Pongo todo el código completo, comentando al detalle la parte del `strcmp()`:

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

#define MAX_EMPLEADOS 2
#define MAX_CARACTERES 20+1

typedef struct {
    char nombre[MAX_CARACTERES];
    char apellido[MAX_CARACTERES];
    float nomina;
} tEmpleado;

/* Predeclaración de funciones/acciones */
tEmpleado buscarEmpleadoNominaMenor(tEmpleado vector[MAX_EMPLEADOS]);
tEmpleado buscarEmpleadoPorApellido(tEmpleado vector[MAX_EMPLEADOS], char apellido[MAX_CARACTERES

int main(int argc, char **argv) {
    tEmpleado vEmpleados[MAX_EMPLEADOS];
    char apellido[MAX_CARACTERES];
    int i = 0;

    for (i=0; i<MAX_EMPLEADOS; i++) {
```

```
        printf("\nNombre empleado %d : ", i);
        scanf("%s", vEmpleados[i].nombre);
        printf("Apellido empleado %d : ", i);
        scanf("%s", vEmpleados[i].apellido);
        printf("Nómina empleado %d : ", i);
        scanf("%f", &vEmpleados[i].nomina);
    }
    tEmpleado empleado = buscarEmpleadoNominaMenor(vEmpleados);
    printf("\nEl empleado con menor nómina es %s, %s (%.2f €)", empleado.apellido, empl

    printf("\nApellido del empleado a buscar : ");
    scanf("%s", apellido);

    tEmpleado empleadoApellido = buscarEmpleadoPorApellido(vEmpleados, apellido);
    printf("\nDatos del empleado: %s, %s -> %f", empleadoApellido.apellido, empleadoApe
    return 0;
}

/* Implementación de funciones/acciones */
tEmpleado buscarEmpleadoNominaMenor(tEmpleado vector[MAX_EMPLEADOS]) {
    int i = 0;
    int nominaMenor = 0;
    for (i=0; i<MAX_EMPLEADOS; i++) {
        if (vector[i].nomina < vector[nominaMenor].nomina) {
            nominaMenor = i;
        }
    }
    return vector[nominaMenor];
}

tEmpleado buscarEmpleadoPorApellido(tEmpleado vector[MAX_EMPLEADOS], char apellido[MAX_
    int i = 0;
    tEmpleado empleado;

    for (i=0; i<MAX_EMPLEADOS; i++) {
        /* Para comparar strings utilizaremos la función
         * strcmp() de C. Esta función compara dos
         * cadenas de caracteres, y devuelve un valor
         * como resultado de la comparación:
         * - si el valor es 0: las dos cadenas son iguales
         * - si el valor es -1: la primera cadena < segunda cadena
         * - si el valor es 1: la primera cadena > segunda cadena
         * En nuestro caso nos interesa detectar que los
         * apellidos sean iguales, con lo que
         * tenemos que controlar que el valor devuelto por
```

```
         * la función strcmp() sea 0.
         */
        if (strcmp(vector[i].apellido, apellido) == 0) {
            return vector[i];
        }
    }
}
```

6.5 Ejemplo: pivoteDefensiva

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

/* Recibimos una petición de un equipo femenino de baloncesto,
 * en la que nos piden un programa que les permita
 * seleccionar la mejor pivote defensiva entre una
 * serie de candidatas.
 * La mejor pivote defensiva es aquella que captura
 * más rebotes; en caso de empate, se elegirá la que
 * haga más tapones.
 * Habrá que implementar 3 acciones y 1 función:
 * - acción leerJugadora(j): lee de teclado y
 *   guarda todos los atributos de la jugadora a la
 *   en la tupla j.
 * - acción mostrarJugadora(j): muestra por pantalla
 *   el valor de los atributos de la tupla j.
 * - acción copiarJugadoras(j1, j2): copia el valor
 *   de todos los atributos de j2 hacia j1.
 * - función compararJugadoras(j1, j2): devuelve -1 en
 *   caso de que la mejor pivote sea j1, y 1 en caso
 *   que la mejor sea j2.
 */
#define MAX_NOMBRE 20+1
#define MAX_APELLIDO 20+1
#define MAX_JUGADORAS 3

typedef struct {
    char nombre[MAX_NOMBRE];
    char apellido[MAX_APELLIDO];
    float rebotes;
```

```
    float tapones;
} tJugadora;

/* Predeclaraciones de funciones/acciones */
void leerJugadora(tJugadora *j);
void mostrarJugadora(tJugadora j);
void copiarJugadora(tJugadora *destino, tJugadora origen);
int compararJugadoras(tJugadora j1, tJugadora j2);

/* Programa principal */
int main(int argc, char **argv) {
    tJugadora vJugadoras[MAX_JUGADORAS];
    int i, resultado;

    /* Se crea la tJugadora ficticia
     * mejorPivote que nos ayudará a encontrar
     * la mejor opción de entre todas las
     * candidatas
     */
    tJugadora mejorPivote;
    mejorPivote.rebotes = 0;
    mejorPivote.tapones = 0;

    /* Leemos todas las jugadoras con
     * la acción leerJugadora(). Esta
     * acción recibe un parámetro de salida
     * (out), el cual contendrá la
     * jugadora leída por teclado. Como
     * se trata de un parámetro de tipo
     * out, se realizará un paso por
     * referencia (= pasaremos un puntero)
     */
    for (i=0; i<MAX_JUGADORAS; i++) {
        tJugadora jugadora;
        leerJugadora(&jugadora);
        vJugadoras[i] = jugadora;
    }

    /* Mostramos por pantalla cuál
     * es la mejor jugadora de perfil
     * pivote defensivo. La idea es ir
     * recorriendo una a una las jugadoras
     * del vector y compararlas con mejorPivote:
     * 1. Si la jugadora del vector es mejor
     *    que mejorPivote, copiaremos los datos
```

```
     *    de la jugadora hacia mejorPivote.
     * 2. Si mejorPivote es mejor que la
     *    jugadora del vector, no haremos nada.
     * Al finalizar el recorrido de todas
     * las jugadoras del vector, tendremos que
     * mejorPivote contendrá la jugadora
     * que estamos buscando.
     */
    for (i=0; i<MAX_JUGADORAS; i++) {
        resultado = compararJugadoras(mejorPivote, vJugadoras[i]);
        if (resultado == 1) {
            copiarJugadora(&mejorPivote, vJugadoras[i]);
        }
    }
    printf("\nMejor opción como pivote defensiva : ");
    mostrarJugadora(mejorPivote);
    return 0;
}

/* Implementación de les funciones/acciones */
void leerJugadora(tJugadora *j) {
    printf("Introduce los datos de la nueva jugadora: \n");
    printf("\tNombre: ");
    scanf("%s", j->nombre);
    printf("\tApellido: ");
    scanf("%s", j->apellido);
    printf("\t>> Promedios por partido:\n");
    printf("\tRebotes: ");
    scanf("%f", &j->rebotes);
    printf("\tTapones: ");
    scanf("%f", &j->tapones);
}

void mostrarJugadora(tJugadora j) {
    printf("\n%s, %s: %.1f rebotes, %.1f tapones \n", j.apellido, j.nombre, j.rebotes, j.tapones)
}

void copiarJugadora(tJugadora *destino, tJugadora origen) {
    /* Recordemos:
     * - si el parámetro es un puntero, el accesor
     *   de atributos será '->'
     * - si el parámetro és un valor, el accesor
     *   de atributos será '.'
     */
    strcpy(destino->nombre, origen.nombre);
```

```
    strcpy(destino->apellido, origen.apellido);
    destino->rebotes = origen.rebotes;
    destino->tapones = origen.tapones;
}

int compararJugadoras(tJugadora j1, tJugadora j2) {
    /* Estamos buscando una jugadora que
     * tenga un perfil de pivote defensivo,
     * Con lo que seleccionaremos:
     * 1. Aquella que tenga más rebotes por partido
     * 2. En caso de empate en rebotes, aquella que
     *    haga más tapones por partido
     */
    if (j1.rebotes > j2.rebotes) {
        return -1;
    } else {
        if (j1.rebotes < j2.rebotes) {
            return 1;
        } else {
            /* En este punto tenemos que
             * j1.rebotes == j2.rebotes,
             * con lo que vamos a comparar el siguiente
             * atributo según la prioridad definida
             * para la posición de pivote defensiva
             */
            if (j1.tapones > j2.tapones) {
                return -1;
            } else {
                if (j1.tapones < j2.tapones) {
                    return 1;
                } else {
                    return 0;
                }
            }
        }
    }
}
```

Ejemplo de ejecución:

```
Introduce los datos de la nueva jugadora:
    Nombre: Julia
```

```
    Apellido: Sanz
    >> Promedios por partido:
    Rebotes: 7.9
    Tapones: 0.9
Introduce los datos de la nueva jugadora:
    Nombre: Nuria
    Apellido: Gutierrez
    >> Promedios por partido:
    Rebotes: 7.9
    Tapones: 1.4
Introduce los datos de la nueva jugadora:
    Nombre: Mireia
    Apellido: Mateu
    >> Promedios por partido:
    Rebotes: 6.8
    Tapones: 2.1

Mejor opción como pivote defensiva :
Gutierrez, Nuria: 7.9 rebotes, 1.4 tapones
```

6.5.1 Explicación sobre la acción copiarJugadora()

La acción `copiarJugadora(tJugadora *destino, tJugadora origen)` del ejemplo recibe dos parámetros:

- El primero de ellos es `destino`, un puntero a una tupla de tipo`tJugadora`; otra forma de decirlo es que el valor `destino` lo **pasamos por referencia**. Este parámetro es de tipo `out`, ya que no utilizamos para nada el valor inicial que tiene y sólo nos interesa el valor final que tendrá en ejecutar la acción.
- El segundo es `origen`, una tupla que **pasamos por valor**. Por lo tanto en este caso no estamos pasando el puntero a una `tJugadora`, sinó directamente una `tJugadora`.

Cuando tenemos un puntero a una tupla accedemos a sus atributos mediante el operador `->`. En cambio, si estamos tratando una tupla accederemos a ellos con el operador `.` (punto).

La codificación de la acción es:

```c
void copiarJugadora(tJugadora *destino, tJugadora origen) {
    /* Recordemos:
     * - si el parámetro es un puntero, el accesor
```

```
     * de atributos será '->'
     * - si el parámetro és un valor, el accesor
     * de atributos será '.'
     */
    strcpy(destino->nombre, origen.nombre);
    strcpy(destino->apellido, origen.apellido);
    destino->rebotes = origen.rebotes;
    destino->tapones = origen.tapones;
}
```

No utilizaremos el prefijo `&` delante de la `tJugadora origen`, de la misma forma que no utilizamos `&`cuando vamos a imprimir por pantalla con `printf()` el valor de una variable: como que estamos tratando con un valor, simplemente accedemos a él y lo utilizamos. Por lo tanto estas acciones serían **incorrectas**:

```
    strcpy(destino->nombre, &origen.nombre);
    strcpy(destino->apellido, &origen.apellido);
    destino->rebotes = &origen.rebotes;
    destino->tapons = &origen.tapons;
```

Sí que tenemos una alternativa posible al operador `->`, según se indica en la XWiki:

```
    strcpy((*desti).nombre, origen.nombre);
    strcpy((*desti).apellido, origen.apellido);
    (*desti).rebotes = origen.rebotes;
    (*desti).tapones = origen.tapones;
```

Por lo tanto este último bloque de código se puede sustituir en el ejemplo anterior y todo seguirá funcionando correctamente, ya que son equivalentes. Se puede utilizar una forma u otra, aunque el operador `->` parece más fácil de entender visualmente.

6.6 Frequently Made Mistakes

6.6.1 Definició d'accions/funcions: noms de paràmetres

Pseudocódigo incorrecto:

```
action hotelCmp(in tHotel, in tHotel)
    { ... }
end action
```

En el ejemplo mostrado parece que la intención es definir dos parámetros, pero falta indicar el identificador (nombre) para cada uno de ellos. Recordemos que en la declaración de la cabecera de una acción / función, hay que indicar, para cada parámetro:

- El tipo de parámetro que recibe la acción o función, (in, out o inout)
- El tipo de datos del parámetro (tHotel en el ejemplo)
- El identificador asociado al parámetro (no definido en el ejemplo).

Pseudocódigo correcto:

```
action hotelCmp(in hotel1: tHotel, in hotel2: tHotel)
    { ... }
end action
```

6.6.2 Definició d'accions/funcions: tipus de paràmetres

Pseudocódigo incorrecto:

```
action hotelCmp(in hotel1: tHotel, out hotel2: tHotel)
    hotel1.id = hotel2.id;
end action
```

En la **declaración de la cabecera** de una acción/función hay que indicar para cada parámetro el tipo de parámetro que recibe la acción/función: `in`,`out` o `inout`. Los parámetros de tipo `in` no se pueden modificar en el interior de la acción, mientras sí podemos hacerlo en el caso de los parámetros de tipo `out` o `inout`. En el ejemplo, se está modificando el parámetro `hotel1` actualizando el valor de uno de sus campos. O bien se trata de una confusión entre ambos parámetros, o bien `hotel1` debe ser de tipo `out`.

Pseudocódigos correctos:

```
action hotelCmp(in hotel1: tHotel, out hotel2: tHotel)
    hotel2.id = hotel1.id;
end action
```

```
action hotelCmp(out hotel1: tHotel, in hotel2: tHotel)
    hotel1.id = hotel2.id;
end action
```

```
action hotelCmp(out hotel1: tHotel, inout hotel2: tHotel)
    hotel1.id = hotel2.id;
end action
```

Fijaros que si el parámetro `hotel1` es de tipo`out` y tenemos que leer el parámetro `hotel2`, este segundo parámetro debe ser forzosamente de tipo `in` o `inout`, por lo que hemos modificado el tipo en la cabecera de la función.

6.6.3 Definición de funciones: tipos de datos de regreso

Pseudocódigo incorrecto:

```
function hotelCmp(h1: tHotel, h2: tHotel)
    { ... }
end function
```

En el ejemplo mostrado, `hotelCmp` está definida como función, con los parámetros de entrada`h1` y `h2`, pero las funciones siempre devuelven un valor, y hay que indicar siempre el tipo de dato en la misma cabecera.

Pseudocódigo correcto:

```
function hotelCmp(h1: tHotel, h2: tHotel): integer
```

```
    { ... }
end function
```

Obsérvese que en este caso, no es necesario indicar el tipo de parámetros de entrada (`in`,`out` o `inout`), porque en las funciones los parámetros siempre son de entrada y no se pueden modificar.

6.6.4 Definición de acciones: retorno de valor

Pseudocódigo incorrecto:

```
action hoteTableInit(inout tHotelTable: tabHotels): void
    { ... }
end action
```

Se ha definido una acción llamada `hotelTableInit`, que recibe un parámetro de entrada/salida (`inout`) de tipo `tHotelTable`, con el identificador`tabHotels`. Las acciones nunca devuelven un valor, pero erróneamente se ha interpretado que hay que indicar este hecho añadiendo `: void` al final de la cabecera de la acción.

Este caso es doblemente incorrecto debido a que `: void` es un identificador propio de C que no existe en lenguaje algorítmico.

Pseudocódigo correcto:

```
action hoteTableInit(inout tHotelTable: tabHotels)
    { ... }
end action
```

Observad que en este caso, al tratarse de una **acción** es necesario indicar el tipo de parámetros de entrada: `in`, `out` o `inout`.

6.6.5 Definición de acciones/funciones: acciones abiertas

Pseudocódigo incorrecto:

```
action hotelCmp(in hotel1: tHotel, out hotel2: tHotel)
    hotel1.id = hotel2.id;
    { ... }
action readHotel(out hotel1: tHotel)
    hotel1.id = readInteger();
end action
```

Las **funciones** y **acciones** constituyen **bloques de código** con unas funcionalidades y características especiales, y como tal deben estar bien delimitadas mediante las palabras reservadas `function` …`end function` y `action` …`end action`. Dejar una acción sin cerrar puede parecer un tema menor, pero en C esto comportará con toda seguridad un error de compilación (a menudo difícil de detectar).

Pseudocódigo correcto:

```
action hotelCmp(in hotel1: tHotel, out hotel2: tHotel)
    hotel1.id = hotel2.id;
    { ... }
end action
action readHotel(out hotel1: tHotel)
    hotel1.id = readInteger();
end action
```

6.6.6 Llamada a funciones/acciones: tipos de parámetros

Pseudocódigo incorrecto:

```
hotelRead(out h1: tHotel);
hotelRead(out h2: tHotel);

action hotelRead(out h: tHotel)
    { ... }
end action
```

El tipo de parámetro que recibe una acción/función, así como el tipo de datos y el identificador del parámetro, hay que indicarlos únicamente en la **definición de la cabecera** de la **acción**. En el caso de las **funciones**, no indicaremos el tipo de datos ya que todos son 'in'.

En la **llamada** a una acción/función únicamente se tienen que indicar los parámetros.

Pseudocódigo correcto:

```
hotelRead(h1);
hotelRead(h2);

action hotelRead(out h: tHotel)
    { ... }
end action
```

6.6.7 Llamada a acciones/funciones: palabras reservadas

Pseudocódigo incorrecto:

```
action hotelRead(h1);
action hotelRead(h2);

action hotelRead(out h: tHotel)
    { ... }
end action
```

La palabra reservada `action` se usará únicamente para la **definición de la cabecera** de la acción.

En la **llamada** a una acción únicamente hay que indicar el nombre de la acción y los parámetros; es incorrecto añadir nuevamente la palabra reservada `action`.

Pseudocódigo correcto:

```
hotelRead(h1);
hotelRead(h2);

action hotelRead(out h: tHotel)
    { ... }
end action
```

6.6.8 Llamada a funciones: llamada vacía

Pseudocódigo incorrecto:

```
bestHotel: integer;

compareHotels(h1,h2);

if (bestHotel = 1) then
    { ... }
end if

function hotelRead(h1: tHotel, h2: tHotel): integer
    { ... }
    return bestHotel;
end function
```

Como ocurre en todas las funciones, la función `hotelRead()` devuelve un valor: en este caso la variable `bestHotel`. Sin embargo, la **llamada** que se hace de la función no es correcta, porque el retorno de la función no se guarda ninguna parte y la variable `bestHotel` del programa principal sigue sin haberse actualizado.

En el caso de C este código no provocaría un error, pero si no hacemos nada más desde el programa principal no tendremos acceso al resultado que devuelve la función (que es lo que se supone que queríamos hacer).

Pseudocódigo correcto:

```
bestHotel: integer;

bestHotel:= compareHotels(h1,h2);

if (bestHotel = 1) then
    {...}
end if

function hotelRead(h1: tHotel, h2: tHotel): integer
    {...}
    return bestHotel;
end function
```

El retorno de una función también se puede utilizar en una expresión, en este caso sin asignarlo a ninguna variable, ya que en la misma expresión el compilador calculará el retorno de la función y lo sustituirá el valor correspondiente a la función:

Pseudocódigo correcto:

```
bestHotel: integer;

if (compareHotels(h1,h2) = 1) then
    { ... }
end if

function hotelRead(h1: tHotel, h2: tHotel): integer
    { ... }
    return bestHotel;
end function
```

6.6.9 Llamada a funciones: ausencia de parámetros

Pseudocódigo incorrecto:

```
var
    myReal: real;
    myInteger: integer;
end var

myInteger:= integerToReal();
```

La función `integerToReal` devuelve un valor y lo queremos guardar en la variable `myInteger`. El problema es que la función necesita que le pasamos un **parámetro** cuando la llamamos: en este caso, el número real a convertir a entero.

En caso de duda hay que consultar siempre la **cabecera de declaración** de la función, ya que nos indicará en cada caso el número y tipo de parámetros que espera.

Pseudocódigo correcto:

```
var
    myReal: real;
    myInteger: integer;
end var

myInteger:= integerToReal(myReal);
```

6.6.10 Llamada a funciones: errores de sintaxis múltiples

Pseudocódigo incorrecto:

```
var
    hotel1: tHotel;
end var

hotelRead(in: &myHotel);

action hotelRead(in myHotel: tHotel)
    { ... }
end action
```

El pseudocódigo anterior contiene varios errores de sintaxis en la **llamada a la función**:

- Se indica el tipo de parámetro `in`.
- Se incluye el operador `:`.
- Se añade sintaxis propia de C con el operador `&`, para pasar el parámetro `myHotel` por referencia. En lenguaje algorítmico el paso de un parámetro por referencia se determina mediante la cabecera de la función.

Pseudocódigo correcto:

```
var
    hotel1: tHotel;
end var

hotelRead(myHotel);
```

```
action hotelRead(in myHotel: tHotel)
    { ... }
end action
```

6.6.11 Definición de funciones: funciones dentro del main

Código incorrecto:

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

#define MAX_LEN 15

int main(int argc, char **argv) {
    int num1;
    int num2;
    bool myBool;

    bool compareNumbers(int n1, int n2){
       return (n1 > n2);
    }

    myBool = compareNumbers(num1, num2);
    return 0;
}
```

Las funciones y acciones se tienen que definir de forma individual y separada. No es correcto definir funciones anidadas o incluidas dentro de otras (como por ejemplo el `main`), entre otras cosas porque estas funciones no estarán disponibles desde el exterior de la función donde son creadas.

Código correcto:

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

#define MAX_LEN 15

int main(int argc, char **argv) {
```

```
    int num1;
    int num2;
    bool myBool;
    myBool = compareNumbers(num1, num2);
    return 0;
}

bool compareNumbers(int n1, int n2){
    return (n1 > n2);
}
```

Chapter 7

PEC07

7.1 Cuándo utilizar & dentro de funciones/acciones

A veces cuando hacemos llamadas a **acciones** dentro de **acciones**, vemos que si pasamos un parámetro con `&` los resultados no son correctos, pero en cambio el parámetro sin el `&` todo funciona . Y al revés.

Para aclarar estas dudas usaremos el siguiente ejemplo: es un programa sencillo que trabaja con tipo `tPec`. Un elemento `tPec` contiene dos atributos: un `nombre` (cadena de caracteres) y una `nota` (decimal). El programa hace la lectura de valores desde teclado con la acción `pecRead(...)`, la modificación del nombre de la PEC con `nombreToUpperCase (...)`, y su posterior impresión por pantalla mediante `pecWrite(...)`.

Con el objetivo de dejarlo todo lo más claro posible se han añadido comentarios extensos dentro del programa, explicando en cada situación qué se hace y por qué.

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

/* Definición de constantes */
#define MAX_NOMBRE 5+1

/* Definición de la tupla tPec */
typedef struct {
```

```
    char nombre[MAX_NOMBRE];
    float nota;
} tPec;

/* Predeclaració de funcions y acciones */

/* Acción que lee por teclado los atributos de un
 * tipo tPec y le asigna los valores. En este
 * caso el parámetro pec es de tipo 'out', dado que
 * su valor antes y después de ejecutar la acción
 * Hhabrá cambiado. Además, como que el valor inicial
 * de pec no nos interesa para nada (recordemos que
 * el objetivo de esta acción es leer de teclado
 * y dar valor a pec, por lo tanto sobrescribir
 * cualquier valor previo que tenga), podemos
 * descartar que sea de tipo 'inout'.
 * Cuando un parámetro es de tipo out/inout, lo
 * pasamos por referencia o, lo que es lo mismo,
 * pasamos un puntero a un tipo de elemento; como
 * se puede ver, aquí eac es un puntero a un
 * Elemento de tipo tPec, ya que va precedido por *.
 * Utilizamos un puntero porque es la única manera
 * que tenemos de modificar un elemento definido
 * fuera del ámbito de la acción, desde dentro de
 * la propia acción.
 */
void pecRead(tPec *pec);

/* Acción que, dado un tipo tPec, muestra su
 * contenido (nombre y nota) por pantalla. Aquí el
 * parámetro pec es de tipo in: su valor
 * antes y después de ejecutar la acción no variará.
 * Un parámetro de tipo in lo pasamos por valor:
 * en este caso es un elemento de tipo tPec.
 * Como se puede ver, no tiene que ir precedido por *.
 */
void pecWrite(tPec pec);

/* Acción que, dado un tipo tPec, coge su
 * nombre y lo pasa a mayúsculas. Por ejemplo, si
 * el nombre es "pEc01" lo modificará por "PEC01".
 * El parámetro es de tipo 'inout': su valor
 * antes y después de ejecutar la acción habrá cambiado
 * y además, el valor inicial que tiene es
 * importante, ya que lo necesitamos para calcular el
```

```
 * valor final ( "pEc01" -> "PEC01"). Al ser un
 * parámetro de tipo 'inout', pasaremos su
 * valor por referencia: necesitamos que pueda ser
 * modificado desde dentro de la acción, con lo
 * es necesario trabajar con un puntero al elemento
 * Pec, de ahí que vaya precedido con *.
 */
void nomToUpperCase(tPec *pec);

/* Programa principal */
int main(int argc, char **argv) {
    /* Definimos las variables */
    tPec pec1, pec2, pec3;

    /* Damos valor a los atributos de cada una
     * de las 3 PEC. Tened en cuenta que estamos pasando
     * punteros a elementos de tipo tPec: el &
     * previo indica que tomamos la dirección de
     * memoria donde reside el elemento de tipo
     * tPec. Como pasamos punteros a memoria,
     * desde dentro de la acción podremos modificar
     * el contenido de pec1, pec2 y pec3, aunque
     * estas tres variables hayan sido
     * definidas fuera del ámbito de la acción.
     */
    pecRead(&pec1);
    pecRead(&pec2);
    pecRead(&pec3);

    /* Mostramos por pantalla los atributos de
     * cada una de las 3 PEC. En este caso
     * el paso de parámetros se hace por valor:
     * pasamos directamente los elementos de tipo
     * tPec, ya que estos no serán modificados
     * desde dentro de la acción (son de tipo 'in').
     */
    pecWrite(pec1);
    pecWrite(pec2);
    pecWrite(pec3);
    return 0;
}

/* Implementación de funciones y acciones */

void pecRead(tPec *pec) {
```

```
    /* Leemos desde teclado el valor
     * correspondiente al nombre de la PEC
     */
    printf("Introduce nombre : ");
    scanf("%s", pec->nombre);

    /* Leemos desde teclado el valor
     * correspondiente a la nota de la PEC
     */
    printf("Introduce nota: ");
    scanf("%f", &pec->nota);

    /* Ahora llegamos en un punto donde, dentro de una
     * acción, llamamos a otra acción. ¿Debemos
     * pasar como atributo pec, o &pec?
     * Ante esta duda, debemos preguntarnos
     * qué tipo de valor contiene ahora
     * mismo (antes de ejecutar la siguiente acción)
     * el parámetro pec. Si recordamos como está
     * definida la acción pecRead (tPec *pec), pec
     * es un puntero a memoria, ya que va precedido
     * de *. Esto significa que en este preciso
     * momento pec sigue siendo un puntero.
     * Por otro lado, si nos fijamos en la definición
     * de la acción nomToUpperCase(tPec *pec), vemos
     * que también espera recibir un puntero. Así como
     * pec es un puntero y nomToUpperCase(...)
     * espera recibir un puntero, simplemente le pasamos
     * pec como parámetro (y no &pec!)
     */
    nomToUpperCase(pec);
    printf("----------------------\n");
}

void pecWrite(tPec pec) {
    /* Mostramos por pantalla los valores
     * de los atributos de la PEC
     */
    printf(">> %s con nota %.1f \n", pec.nombre, pec.nota);
}

void nomToUpperCase(tPec *pec) {
    /* Hay librerías que ya implementan los cambios
     * entre mayúsculas y minúsculas. En este caso
     * hemos optado por no utilizar ninguno e implementarlo
```

```
     * nosotros mismos, tratando el string nombre de pec
     * como un recorrido carácter a carácter.
     */
    int i = 0;
    for (i = 0; pec->nombre[i] != '\0'; i++) {
        if (pec->nombre[i] >= 'a' && pec->nombre[i] <= 'z') {
            pec->nombre[i] = pec->nombre[i] + ('A' - 'a');
        }
    }
}
```

7.2 Ejemplo: cómo modular un programa con CodeLite

A continuación se explicará detalladamente cómo modular el programa típico `HelloWorld` de una forma diferente de como se explica en los vídeos de la xWiki, que puede ser más comprensible. La explicación es un poco extensa para que quede lo más claro posible.

7.2.1 Funcionamiento por defecto de un proyecto a CodeLite:

Cuando en CodeLite creamos un nuevo proyecto, por ejemplo `Modularidad`, con el típico `main.c` inicial que contiene el programa `HelloWorld`, tenemos la siguiente distribución de archivos:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l
total 12
-rw-r--r-- 1 uoc uoc   93 nov 17 23:56 main.c
-rw-rw-r-- 1 uoc uoc 4274 nov 17 23:56 Modularidad.project
```

Si compilamos este programa en CodeLite, vemos que nos ha generado una nueva carpeta llamada `Debug`:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l
total 24
drwxrwxr-x 2 uoc uoc 4096 nov 17 23:58 Debug
-rw-r--r-- 1 uoc uoc   93 nov 17 23:56 main.c
-rw-rw-r-- 1 uoc uoc 3249 nov 17 23:58 Modularidad.mk
-rw-rw-r-- 1 uoc uoc 4274 nov 17 23:56 Modularidad.project
-rw-rw-r-- 1 uoc uoc   17 nov 17 23:58 Modularidad.txt
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$
```

El contenido de la carpeta `Debug` es el programa ejecutable final junto con los archivos objeto intermedios:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ cd Debug
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/Debug$ ls -l
total 24
-rw-rw-r-- 1 uoc uoc 4620 nov 17 23:58 main.c.o
-rw-rw-r-- 1 uoc uoc   23 nov 17 23:58 main.c.o.d
-rwxrwxr-x 1 uoc uoc 9708 nov 17 23:58 Modularidad
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/Debug$
```

Si se quiere, desde un terminal de Lubuntu se puede ejecutar el programa resultante `Modularidad` de la siguiente forma:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/Debug$ ./Modularidad
hello world
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/Debug$
```

7.2.2 Ejecutable dentro de /bin.

Vamos ahora a modificar CodeLite para que el ejecutable nos lo guarde siempre dentro de la carpeta `./bin`.

El primer video de la xWiki indica cómo cambiar la carpeta donde se guarda el archivo ejecutable resultante de compilar nuestro programa. El objetivo es que en vez de utilizar la carpeta `./Debug` el archivo ejecutable se guarde dentro de la nueva carpeta`./bin`

Atención: este apartado no se pide explícitamente en la PEC07 (únicamente se habla de las carpetas `./src` e `./include`), con lo que no hay que aplicarlo a la PEC.

Para hacer este cambio, hacemos clic botón derecho del ratón sobre el nombre del proyecto, **Modularidad** -> **Settings ...** -> **General** -> **Output File**: ponemos el valor `./bin/$(ProjectName)`. Esto significa que el ejecutable se guardará dentro de la carpeta `./bin` y que su nombre será el mismo que el nombre del proyecto (en el caso que nos ocupa, se llamará`Modularidad`)

Además, sin salir de esta misma pantalla, cambiamos el valor del **Working Directory** por: `./bin`. Este valor es el que tiene en consideración CodeLite a la hora de buscar el ejecutable del proyecto, para cuando hacemos el `Run`.

Guardamos los cambios y si ahora compilamos nuevamente el programa, vemos que ha creado la carpeta `./bin` dentro de nuestro proyecto:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l
total 28
drwxrwxr-x 2 uoc uoc 4096 nov 18 00:11 bin
drwxrwxr-x 2 uoc uoc 4096 nov 17 23:58 Debug
```

```
-rw-r--r-- 1 uoc uoc   93 nov 17 23:56 main.c
-rw-rw-r-- 1 uoc uoc 3230 nov 18 00:11 Modularidad.mk
-rw-rw-r-- 1 uoc uoc 4350 nov 18 00:10 Modularidad.project
-rw-rw-r-- 1 uoc uoc   17 nov 18 00:11 Modularidad.txt
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$
```

Al igual que en el caso anterior, si queremos podemos ejecutar el programa desde el propio terminal:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ cd ./bin
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/bin$ ls -l
total 12
-rwxrwxr-x 1 uoc uoc 9708 nov 18 00:11 Modularidad
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/bin$ ./Modularidad
hello world
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad/bin$
```

A partir de ese momento, siempre que compilemos nuestro programa, guardará el ejecutable dentro de la carpeta `./bin`.

7.2.3 Modularización del programa en un archivo de cabeceras y uno de funciones.

Centraremos ahora la explicación con la estructura que se pide a la PEC07 y trabajando sobre el mismo ejemplo `Modularidad`.

Lo que queremos conseguir es la siguiente estructura:

```
proyecto Modularidad
   |
   |-- /include/
   |     |
   |     \-- helloWorld.h
   |
   \-- /src/
         |
         |-- helloWorld.c
         \-- main.c
```

El objetivo es dividir (modularizar) el archivo único `main.c` que tenemos hasta ahora en tres archivos:

- `helloWorld.h`: este archivo contendrá la predeclaración de todas las funciones y acciones de nuestro programa, así como la definición de todos los tipos necesarios (enumerativos, tuplas, etc) y constantes que requiera nuestro programa C. Este archivo lo ubicaremos dentro de la carpeta `./include` de nuestro proyecto

- `helloWorld.c`: dentro de él se implementarán todas las acciones y funciones del programa. Básicamente contendrá el código de todo lo que hemos predeclarado en el archivo `helloWorld.h`.
- `main.c`: contendrá el código del programa principal, identificado por la función `main()`.

El primer paso que podemos hacer es crear las dos carpetas que necesitará nuestro proyecto modularizado, `src` e `include`:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ mkdir src
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ mkdir include
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l
total 36
drwxrwxr-x 2 uoc uoc 4096 nov 18 00:14 bin
drwxrwxr-x 2 uoc uoc 4096 nov 18 00:14 Debug
drwxrwxr-x 2 uoc uoc 4096 nov 18 00:18 include
-rw-r--r-- 1 uoc uoc   93 nov 17 23:56 main.c
-rw-rw-r-- 1 uoc uoc 3230 nov 18 00:14 Modularidad.mk
-rw-rw-r-- 1 uoc uoc 4350 nov 18 00:10 Modularidad.project
-rw-rw-r-- 1 uoc uoc   17 nov 18 00:14 Modularidad.txt
drwxrwxr-x 2 uoc uoc 4096 nov 18 00:18 src
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$
```

El siguiente paso es transformar el programa `HelloWorld` típico en un programa modularizado: por este motivo necesitamos la definición de algunas funciones o acciones que nos permitan separar un único archivo en uno de cabeceras (`helloWorld.h`), uno de implementación de funciones/acciones (`helloWorld.c`) y un principal (`main.c`). Por tanto, el objetivo es pasar del siguiente programa ...:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("hello world\n");
    return 0;
}
```

... el siguiente código modularizado:

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

```c
/* Predeclaración de las funciones/acciones */
void showHelloMessage();

/* Código principal */
int main(int argc, char **argv) {
    showHelloMessage();
    return 0;
}

/* Implementación de las funciones/acciones */
void showHelloMessage() {
    printf("hello world\n");
};
```

Si se ejecuta el nuevo código dentro de un único `main.c`, éste seguirá mostrando correctamente el mensaje de *"hello world"*. El objetivo es terminándolo dividiendo en tres bloques:

```c
#include <stdio.h>

/* Inicio contenido del fichero helloWorld.h */
/* Predeclaración de las funciones/acciones */
void showHelloMessage();
/* Fin contenido del fichero helloWorld.h */

/* Inicio contenido del fichero main.c */
/* Código principal */
int main(int argc, char **argv) {
    showHelloMessage();
    return 0;
}
/* Fin contenido del fichero main.c */

/* Inicio contenido del fichero helloWorld.c */
/* Implementación de las funciones/acciones */
void showHelloMessage() {
    printf("hello world\n");
};
/* Fin contenido del fichero helloWorld.c */
```

El primer paso que haremos será eliminar desde CodeLite el programa antiguo; por tanto, **CodeLite** -> proyecto **Modularidad** -> **src** -> botón derecho sobre el nombre del archivo `main.c` -> **Remove** -> confirmamos el borrado ->

confirmamos el borrado del archivo `main.c` de disco.

En este punto, a CodeLite, dentro del proyecto `Modularidad` únicamente tenemos una carpeta *virtual* llamada `src`. Atención: la carpeta *virtual* llamada `src` de nuestro proyecto de CodeLite en este momento no tiene ninguna relación con la carpeta `./src` que se ha creado hace un momento.

Creamos el programa principal `main.c` haciendo: **CodeLite** -> proyecto **Modularidad** -> botón derecho sobre carpeta **src** -> **Add New File** -> seleccionamos el tipo **C source File (.c)** -> indicamos como **Name**: `main.c`, y como **Location** seleccionamos la carpeta `./src` que hemos creado antes.

Vamos a crear el segundo archivo, `helloWorld.c`, exactamente de la misma forma: **CodeLite** -> proyecto **Modularidad** -> botón derecho sobre carpeta **src** -> **Add New File** -> seleccionamos el tipo **C Source File (.c)** -> indicamos como **Name**: `helloWorld.c`, y como **Location** seleccionamos la carpeta `./src` creada anteriormente.

Creamos ahora una carpeta *virtual* dentro de nuestro proyecto desde **CodeLite** -> botón derecho sobre el proyecto **Modularidad** -> **New Virtual Folder** -> le ponemos por nombre: `include`

Para finalizar, creamos el tercer archivo requerido: `helloWorld.h`. Los pasos serán: **CodeLite** -> proyecto **Modularidad** -> botón derecho sobre carpeta **include** -> **Add New File** -> seleccionamos el tipo **Header File (h)** -> indicamos como **Name**: `helloWorld.h`, y como **Location** seleccionamos la carpeta `./include` creada anteriormente.

En este punto nuestro proyecto `Modularidad` de CodeLite tendrá la estructura deseada:

```
proyecto Modularidad
   |
   |-- /include/
   |     |
   |     \-- helloWorld.h
   |
   \-- /src/
         |
         |-- helloWorld.c
         \-- main.c
```

Ahora sólo hay que dar contenido a los 3 archivos vacíos que hemos creado.

Empezamos por el archivo de cabeceras `helloWorld.h`. El editamos como lo hacemos habitualmente en CodeLite, y le copiamos el fragmento de código que hemos dicho antes que le correspondía:

```c
#include <stdio.h>

/* Predeclaración de las funciones/acciones */
void showHelloMessage();
```

Editamos el fichero `helloWorld.c` y le añadimos el código correspondiente:

```c
/* Implementación de las funciones/acciones */
void showHelloMessage() {
    printf("hello world\n");
};
```

Y para finalizar editamos el contenido del archivo `main.c` con el código comentado anteriormente:

```c
/* Código principal */
int main(int argc, char **argv) {
    showHelloMessage();
    return 0;
}
```

En este punto tenemos el programa modularizado, pero nos falta dos cuestiones:

- Configurar CodeLite para que tenga presentes donde buscar el archivo de cabeceras `.h` cuando hacemos referencia.
- Hacer que los archivos estén enlazados entre ellos: ahora mismo son completamente independientes.

Vamos por el primer punto: para indicar a CodeLite donde encontrará todos los archivos `.h` de un programa, basta con hacer **CodeLite** -> clic botón derecho sobre **Modularidad** -> **Settings ...** -> **Compiler** -> dentro de la opción **Include Paths** añadir el valor `.;./include`

Para el segundo punto: tenemos que el archivo `main.c` en estos momentos hace referencia a una acción llamada `showHelloMessage()`, de la que no sabemos nada. Lo que necesitamos es importar el archivo de cabeceras; como es un archivo que hemos creado nosotros, el `include` va entre comillas dobles:

```c
#include "helloWorld.h"

/* Código principal */
int main(int argc, char **argv) {
    showHelloMessage();
    return 0;
}
```

El siguiente paso será añadir el `include` habitual de la librería `stdio.h` dentro del archivo`helloWorld.c`, ya que en él utilizamos la función `printf()` incluida en esta librería:

```c
#include <stdio.h>

/* Implementación de las funciones/acciones */
void showHelloMessage() {
    printf("hello world\n");
};
```

En este punto ya podemos ejecutar nuestro programa modularizado correctamente.

La estructura resultante y el contenido de cada carpeta es el deseado:

```
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l ./bin
total 12
-rwxrwxr-x 1 uoc uoc 10172 nov 18 01:09 Modularidad
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l ./include/
total 4
-rw-rw-r-- 1 uoc uoc 70 nov 18 00:54 helloWorld.h
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l ./src
total 8
-rw-rw-r-- 1 uoc uoc 122 nov 18 01:09 helloWorld.c
-rw-rw-r-- 1 uoc uoc 113 nov 18 01:07 main.c
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$ ls -l ./Debug/
total 20
-rw-rw-r-- 1 uoc uoc 4544 nov 18 01:09 src_helloWorld.c.o
-rw-rw-r-- 1 uoc uoc   43 nov 18 01:09 src_helloWorld.c.o.d
-rw-rw-r-- 1 uoc uoc 2440 nov 18 01:09 src_main.c.o
-rw-rw-r-- 1 uoc uoc   75 nov 18 01:09 src_main.c.o.d
uoc@fp:~/Documents/codelite/workspaces/Test/Modularidad$
```

Ni para la realización de este ejemplo ni para la PEC07 hay que crear librerías propias, con lo que este punto no lo comento: únicamente trataremos con

archivos de cabecera `.h`, archivos de implementación `.c`, y el archivo correspondiente al código principal `main.c`.

7.3 Tipo de parámetros en acciones y funciones

En las **acciones** los parámetros pueden ser de tipo **in**, **out** o **inout**. Debe especificarse en el momento de definir la acción a nuestro algoritmo.

Ejemplo: tres formas de implementar una suma de dos enteros con diferentes acciones según los tipos **in**, **out** o **inout** de los parámetros:

```
action suma1(in num1: integer, in num2: integer)
    var
        resultado: integer;
    end var

    resultat := num1 + num2;
    writeString("Resultado de la suma = ");
    writeInteger(resultado);
end action

action suma2(in num1: integer, in num2: integer, out resultado: integer)
    resultado := num1 + num2;
end action

action suma3(inout num1: integer, in num2: integer)
    num1 := num1 + num2;
end action
```

A las funciones en cambio todos los parámetros son de entrada, con lo que no es necesario indicar el **in**.

Ejemplo:

```
function suma4(num1: integer, num2: integer): integer
    var
        resultado: integer;
    end var

    resultado := num1 + num2;
```

```
    return resultado;
end function
```

7.4 scanf(): acción o función?

Lo que nos indica la firma `scanf()` es que estamos ante una función:

```c
int scanf(const char *format, type* var1, ...);
```

El segundo argumento no es de tipo **out**, ya que lo que le pasamos no es la variable que guardará el valor leído, sino el puntero a la variable que guardará el valor leído, y este puntero no varía en ningún momento.

```c
/* Por ejemplo, el valor de &numero es 0xbfb86c40 */
scanf("%d", &numero)
/* una vez ejecutada la función scanf(), el valor de &numero continua siendo 0xbfb86c4
```

La función `scanf` devuelve un valor, aunque nosotros no lo utilizamos habitualmente: el número de elementos procesados correctamente. Dicho de otra forma: que una función devuelva un valor no nos obliga a recuperarlo y tratarlo, aunque habitualmente sí lo haremos.

Por otra parte, el hecho de definir un parámetro de una función/acción como `const` significa que este parámetro dentro de la acción/función se comportará como una constante. Por lo tanto dentro del ámbito de la función/acción no se podrá modificar.

Este hecho nos puede interesar o no. Por ejemplo, imaginemos que creamos la siguiente función para sumar dos enteros:

```c
#include <stdio.h>

/* Predeclaración de funcioens/aciones */
int suma(int numA, int numB);

int main(int argc, char **argv) {
    int a = 10;
```

```
    int b = 13;

    int resultado = suma(a, b);
    printf("Resultado: %d + %d = %d\n", a, b, resultado);
    return 0;
}

/* Implementación de funciones/acciones */
int suma(int numA, int numB) {
    numA = numA + numB;
    return (numA);
}
```

Dentro del ámbito de la función, nos interesa por ejemplo que el parámetro `numA` sea constante? No, ya que si lo definimos de esta forma la compilación nos fallará:

```
#include <stdio.h>

/* Predeclaración de funciones/aciones */
int suma(const int numA, int numB);

int main(int argc, char **argv) {
    int a = 10;
    int b = 13;

    int resultado = suma(a, b);
    printf("Resultado: %d + %d = %d\n", a, b, resultado);
    return 0;
}

/* Implementación de funciones/acciones */
int suma(const int numA, int numB) {
    numA = numA + numB;
    return (numA);
}
```

El error que obtendremos al intentar compilar el programa será *"error: assignment of read-only parameter 'numA"*, ya que dentro de la función suma estamos modificando el valor de `numA`.

Muy importante: que dentro de la función modifiquemos el valor de `numA` no significa que fuera del ámbito de la función el valor de la variable a varíe (lo podemos comprobar en el `printf()` que se hace posteriormente).

Por lo tanto el uso de const en un parámetro debería limitarse a aquellos valores que se vayan a tratar realmente como constantes: por ejemplo, si pasamos la constante `PI = 3.14159...` como parámetro, dentro de la función seguro que no tenemos la necesidad de modificar esta constante matemática, con lo que en este caso es más adecuado utilizar `const` en el parámetro de la función.

Como último punto, no hay que confundir el hecho de definir un parámetro de una función como `const`, a hacer que fuera del ámbito de la función el parámetro correspondiente se defina como constante:

```c
#include <stdio.h>

/* Predeclaración de funciones/aciones */
int suma(int numA, int numB);

int main(int argc, char **argv) {
    const int a = 10;
    int b = 13;

    int resultado = suma(a, b);
    printf("Resultado: %d + %d = %d\n", a, b, resultado);
    return 0;
}

/* Implementació de funciones/acciones */
int suma(int numA, int numB) {
    numA = numA + numB;
    return (numA);
}
```

En este caso cuando compilamos no obtendremos ningún error, ya que la variable `a` es una constante fuera de la función `suma`, pero su valor pasado como parámetro dentro de la función no es una constante.

7.5 Paso por valor vs paso por referencia

El clásico **paso por valor** corresponde a los parámetros de tipo **in**, en los que pasamos la variable/valor.

Por otra parte el **paso por referencia** consiste en pasar como parámetro de tipo **out** o **inout** la dirección de memoria de la variable (puntero).

7.6 Ejemplo: invertirPalabra

Imaginemos que tenemos la tupla `tPalabra` que tiene dos campos:

- `cadena`: contiene el string con el valor de la `tPalabra`
- `numeroCaracteresCadena`: contiene el número de caracteres de la cadena

Implementamos la acción `invertir()`, lo hace dos operaciones:

- Invierte (gira) la `cadena` de `tPalabra`; por ejemplo, si entramos *"Fundamentos"* el resultado será *"stnemanoF"*.
- Calcula el valor de `numeroCaracteresCadena`; si tenemos como cadena *"Fundamentos"*, el valor será 9.

Queremos que por teclado se pida el valor para el campo `cadena` de dos `tPalabra`, y en ambas queremos aplicar la acción `invertir()`. Una posible forma de implementarla sería la siguiente:

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

#define MAX_CHAR 20+1
#define MAX_PALABRAS 2

typedef struct {
    char cadena[MAX_CHAR];
    int numeroCaracteresCadena;
} tPalabra;

/* Predeclaración de la acción.
 * Esta acción recibe un parámetro inout de tipo tPalabra,
 * invierte (gira) el campo cadena y calcula el valor
 * correspondiente para el campo numeroCaractersCadena
 */
void invertir(tPalabra *palabra);

int main(int argc, char **argv) {
    int i = 0;

    /* Introduim per teclat un total de MAX_PALABRAS */
    for (i = 0; i < MAX_PALABRAS; i++) {
        tPalabra palabra;
```

```
        printf("Introduce una palabra : ");
        scanf("%s", palabra.cadena);

        invertir(&palabra);
        printf("La palabra invertida es : %s, de %d letras.\n", palabra.cadena, palabra
    }
}

/* Implementación de la acción */
void invertir(tPalabra *palabra) {

    int i;
    tPalabra palabraInvertida;
    int longitudPalabra = strlen(palabra->cadena);

    for (i=0; i<longitudPalabra; i++ ) {
        palabraInvertida.cadena[(longitudPalabra-1)-i] = palabra->cadena[i];
    }

    /* Indicamos el finalizador del string */
    palabraInvertida.cadena[longitudPalabra] ='\0';

    strcpy(palabra->cadena, palabraInvertida.cadena);
    palabra->numeroCaracteresCadena = longitudPalabra;
}
```

La acción sólo recibe un parámetro `tPalabra`, ya que únicamente se ejecuta sobre una `tPalabra`. Como la queremos ejecutar para cada una de las dos `tPalabra`, repetimos la llamada dos veces dentro del bucle (una por cada`tPalabra`).

7.7 Ejemplo: isPar

Ejemplo de función que devuelve un booleano:

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

/* Predeclaración de la función isPar, la cual devuelve un
 * booleano que indica si el número pasado por parámetro es
 * par (true) o impar (false).
```

```c
 */
bool isPar(int numero);

int main(int argc, char **argv) {
    int numero;

    printf("Teclea un número : ");
    scanf("%d", &numero);

    if (!isPar(numero)) {
        printf("El número %d es impar.\n", numero);
    } else {
        printf("El número %d es par.\n", numero);
    }
}

/* Implementación de la función */
bool isPar(int numero) {
    if (numero % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

7.8 Ejemplo: pivoteDefensivaTirosLibres

Se añade al ejemplo **pivoteDefensiva** de la PEC06 un nuevo factor de comparación: en caso de empate de las comparaciones anteriores, escogeremos la que tenga un mejor porcentaje de tiros libres.

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

/* Recibimos una petición de un equipo femenino de baloncesto,
 * en la que nos piden un programa que les permita
 * seleccionar la mejor pivote defensiva entre una
 * serie de candidatas.
 * La mejor pivote defensiva es aquella que captura
```

```
 * más rebotes; en caso de empate, se elegirá la que
 * haga más tapones. En caso de empate nos interesará
 * escoger la que tenga mejor porcentaje en tiros libres.
 * Habrá que implementar 3 acciones y 2 funciones:
 * - acción leerJugadora(j): lee de teclado y
 *   guarda todos los atributos de la jugadora a la
 *   en la tupla j.
 * - acción mostrarJugadora(j): muestra por pantalla
 *   el valor de los atributos de la tupla j.
 * - acción copiarJugadoras(j1, j2): copia el valor
 *   de todos los atributos de j2 hacia j1.
 * - función compararJugadoras(j1, j2): devuelve -1 en
 *   caso de que la mejor pivote sea j1, y 1 en caso
 *   que la mejor sea j2.
 * - función porcentajeTirosLibres(intentados, anotados):
 *   devuelve el porcentaje de acierto en tiros libres
 *   en función de los valores pasados por parámetro.
 */

#define MAX_NOMBRE 20+1
#define MAX_APELLIDO 20+1
#define MAX_JUGADORAS 3

typedef struct {
    char nombre[MAX_NOMBRE];
    char apellido[MAX_APELLIDO];
    float rebotes;
    float tapones;
    float tirosLibres; /* en porcentaje */
} tJugadora;

/* Predeclaraciones de funciones/acciones */
void leerJugadora(tJugadora *j);
void mostrarJugadora(tJugadora j);
void copiarJugadora(tJugadora *destino, tJugadora origen);
int compararJugadoras(tJugadora j1, tJugadora j2);
float porcentajeTirosLibres(int intentados, int anotados);

/* Programa principal */
int main(int argc, char **argv) {
    tJugadora vJugadoras[MAX_JUGADORAS];
    int i, resultado;

    /* Se crea la tJugadora ficticia
     * mejorPivote que nos ayudará a encontrar
```

```
     * la mejor opción de entre todas las
     * candidatas
     */
    tJugadora mejorPivote;
    mejorPivote.rebotes = 0;
    mejorPivote.tapones = 0;
    mejorPivote.tirosLibres = 0.0;

    /* Leemos todas las jugadoras con
     * la acción leerJugadora(). Esta
     * acción recibe un parámetro de salida
     * (out), el cual contendrá la
     * jugadora leída por teclado. Como
     * se trata de un parámetro de tipo
     * out, se realizará un paso por
     * referencia (= pasaremos un puntero)
     */
    for (i=0; i<MAX_JUGADORAS; i++) {
        tJugadora jugadora;
        leerJugadora(&jugadora);
        vJugadoras[i] = jugadora;
    }

    /* Mostramos por pantalla cuál
     * es la mejor jugadora de perfil
     * pivote defensivo. La idea es ir
     * recorriendo una a una las jugadoras
     * del vector y compararlas con mejorPivote:
     * 1. Si la jugadora del vector es mejor
     *    que mejorPivote, copiaremos los datos
     *    de la jugadora hacia mejorPivote.
     * 2. Si mejorPivote es mejor que la
     *    jugadora del vector, no haremos nada.
     * Al finalizar el recorrido de todas
     * las jugadoras del vector, tendremos que
     * mejorPivote contendrá la jugadora
     * que estamos buscando.
     */
    for (i=0; i<MAX_JUGADORAS; i++) {
        resultado = compararJugadoras(mejorPivote, vJugadoras[i]);
        if (resultado == 1) {
            copiarJugadora(&mejorPivote, vJugadoras[i]);
        }
    }
    printf("\nMejor opción como pivote defensiva : ");
```

```
    mostrarJugadora(mejorPivote);
    return 0;
}

/* Implementación de les funciones/acciones */
void leerJugadora(tJugadora *j) {
    int intentados;
    int anotados;
    printf("Introduce los datos de la nueva jugadora: \n");
    printf("\tNombre: ");
    scanf("%s", j->nombre);
    printf("\tApellido: ");
    scanf("%s", j->apellido);
    printf("\t>> Promedios por partido:\n");
    printf("\tRebotes: ");
    scanf("%f", &j->rebotes);
    printf("\tTapones: ");
    scanf("%f", &j->tapones);
    printf("\tTiros libres intentados: ");
    scanf("%d", &intentados);
    printf("\tTiros libres anotados: ");
    scanf("%d", &anotados);
    j->tirosLibres = porcentajeTirosLibres(intentados, anotados);
}

void mostrarJugadora(tJugadora j) {
    printf("\n%s, %s: %.1f rebotes, %.1f tapones, %.1f%% tiros libres \n", j.apellido,
}

void copiarJugadora(tJugadora *destino, tJugadora origen) {
    /* Recordemos:
     * - si el parámetro es un puntero, el accesor
     *   de atributos será '->'
     * - si el parámetro és un valor, el accesor
     *   de atributos será '.'
     */
    strcpy(destino->nombre, origen.nombre);
    strcpy(destino->apellido, origen.apellido);
    destino->rebotes = origen.rebotes;
    destino->tapones = origen.tapones;
    destino->tirosLibres = origen.tirosLibres;
}

int compararJugadoras(tJugadora j1, tJugadora j2) {
    /* Estamos buscando una jugadora que
```

```
     * tenga un perfil de pivote defensivo,
     * Con lo que seleccionaremos:
     * 1. Aquella que tenga más rebotes por partido
     * 2. En caso de empate en rebotes, aquella que
     *    haga más tapones por partido
     * 3. En caso de empate, la que tenga mejor
     *    percentaje de tiros libres
     */
    if (j1.rebotes > j2.rebotes) {
        return -1;
    } else {
        if (j1.rebotes < j2.rebotes) {
            return 1;
        } else {
            /* En este punto tenemos que
             * j1.rebotes == j2.rebotes,
             * con lo que vamos a comparar el siguiente
             * atributo según la prioridad definida
             * para la posición de pivote defensiva
             */
            if (j1.tapones > j2.tapones) {
                return -1;
            } else {
                if (j1.tapones < j2.tapones) {
                    return 1;
                } else {
                    /* Añadimos la variante de valorar
                     * el porcentaje de acierto en tiros libres
                     */
                    if (j1.tirosLibres >= j2.tirosLibres) {
                        return -1;
                    } else {
                        return 1;
                    }
                }
            }
        }
    }
}

float porcentajeTirosLibres(int intentados, int anotados) {
    return ((float)anotados/intentados)*100.0;
}
```

Chapter 8

PEC08

8.1 Cómo inicializar una tabla

Para inicializar/borrar una tabla únicamente necesitamos indicar que el número de elementos que contiene es 0.

La pregunta que nos podemos hacer es *"simplemente inicializando a 0 este atributo es suficiente?"*. La respuesta es afirmativa: el atributo que contiene el número de elementos de una tabla siempre es el utilizado a la hora de recorrer una tabla, ya que nos indica cuál es el último elemento de la tabla. Del mismo modo, cuando insertamos un elemento incrementaremos en 1 su valor.

¿Qué pasa cuando le damos valor 0? Estamos indicando que la tabla tiene 0 elementos, de forma que cuando añadimos uno nuevo lo haremos en la primera posición, sobreescribiendo todo lo que previamente pudiera haber en memoria.

8.2 Ejemplo: calcularNota

Se debe entender una tabla como un conjunto de elementos de los que sabemos en todo momento cuantos tenemos.

Imaginemos que queremos un programa que calcule la nota media de las PEC de una asignatura. Podríamos plantearlo como un simple array de enteros, pero como nos gusta poder ofrecer más funcionalidades en un futuro en nuestro programa, tendremos el siguiente escenario:

- `tPec`: será el tipo de datos básico que tratará nuestro programa. Esta tupla estará formada por un lado por un número descriptivo de la pec, y por otro lado por su nota numérica con decimales.

- `tAsignatura`: tabla que contendrá elementos de tipo tPec. Aparte del array de tPec, también tendrá un contador interno de elementos: numPecs.

Sobre esta base realizaremos dos acciones:

- `insertar_pec()`: se trata de una acción que incluye un elemento de tipo `tPec` dentro de la tabla`tAsignatura`. Es una operación similar a la acción de llenar una tabla de los ejemplos de la XWiki.
- `calcular_nota()`: es una función que revisa todos los elementos `tPec` de la tabla`tAsignatura` y calcula su nota promedio. Se trata de una operación equivalente a la de los recorridos de tabla de los ejemplos de la XWiki, ya que estamos recorriendo uno a uno todos los elementos `tPec` para obtener su nota.

Una posible forma de implementarlo sería la siguiente:

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

/* Definición de constantes */
#define MAX_PECS 5
#define MAX_NOMBRE 5+1

/* Definició de la tupla tPec */
typedef struct {
    char nombre[MAX_NOMBRE];
    float nota;
} tPec;

/* Definición de la tabla tAsignatura */
typedef struct {
    tPec pec[MAX_PECS];
    int numPecs;
} tAsignatura;

/* Definición funciones/acciones */

/* Acción que añade un elemento de tipo tPec en la tabla tAsignatura */
void insertar_pec(tAsignatura *asignatura, tPec pec);

/* Función que calcula la media de todas las tPec que contiene la tabla
 * tAsignatura
 */
float calcular_nota(tAsignatura asignatura);
```

```c
/* Programa principal */
int main(int argc, char **argv) {

    /* Definimos las variables */
    tAsignatura fp;
    tPec pec1, pec2, pec3;
    float nota;

    /* Inicializamos las variables */
    nota = 0;
    strcpy(pec1.nombre,"PEC01");
    pec1.nota = 10;
    strcpy(pec2.nombre,"PEC02");
    pec2.nota = 8.5;
    strcpy(pec3.nombre,"PEC03");
    pec3.nota = 7.5;

    /* La inicialización de la tabla se hace simplemente
     * poniendo a 0 su contador
     */
    fp.numPecs=0;

    /* Añadimos ahora las pec a la asignatura, que es una tabla
     * de elementos de tipo tPec
     */
    insertar_pec(&fp, pec1);
    insertar_pec(&fp, pec2);
    insertar_pec(&fp, pec3);

    /* Calculamos la nota con la función calcular_nota */
    nota = calcular_nota(fp);
    printf("La nota promedio de las %d PEC es %f\n", fp.numPecs, nota);
    return 0;
}

/* Implementación funciones/acciones */

void insertar_pec(tAsignatura *asignatura, tPec pec) {
    /* numPecs contiene el número de elementos de tipo tPec
     * que contiene la tabla en cada momento
     */
    asignatura->pec[asignatura->numPecs] = pec;

    /* Una vez hemos asignado un elemento nuevo tPec en la tabla
     * incrementamos el valor de numPecs
```

```
     */
    asignatura->numPecs = asignatura->numPecs + 1;
}

float calcular_nota(tAsignatura asignatura) {
    /* La variable suma contiene el sumatorio de todas
     * las notas de las tPec que están dentro de la tabla
     * tAsignatura
     */
    float suma = 0;

    /* Se recorren todos los elementos tPec de tAsignatura
     * para obtener su nota y acumularlas a la
     * variable suma
     */
    for (int i = 0; i < asignatura.numPecs; i++) {
        suma = suma + asignatura.pec[i].nota;
    }

    /* Para calcular la media se divide el sumatorio de
     * notas por el total de elementos de la tabla tAsignatura
     */
    return suma/asignatura.numPecs;
}
```

Para facilitar la lectura se ha unido todo el programa en un único bloque de código (un único archivo).

8.3 Ejemplo: calcularNota con introducción iterativa

El siguiente ejemplo adapta el caso anterior para que pida los valores iterativamente:

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

/* Definición de constantes */
#define MAX_PECS 5
#define MAX_NOMBRE 5+1
```

```c
/* Definició de la tupla tPec */
typedef struct {
    char nombre[MAX_NOMBRE];
    float nota;
} tPec;

/* Definición de la tabla tAsignatura */
typedef struct {
    tPec pec[MAX_PECS];
    int numPecs;
} tAsignatura;

/* Definición funciones/acciones */

/* Acción que añade un elemento de tipo tPec en la tabla tAsignatura */
void insertar_pec(tAsignatura *asignatura, tPec pec);

/* Función que calcula la media de todas las tPec que contiene la tabla
 * tAsignatura
 */
float calcular_nota(tAsignatura asignatura);

/* Programa principal */
int main(int argc, char **argv) {

    /* Definimos las variables */
    tAsignatura fp;
    float nota;
    int numPecs, i;

    /* Inicializamos las variables */
    nota = 0;
    numPecs = 0;
    i = 0;

    /* Inicializamos la tabla */
    fp.numPecs=0;

    /* Introducimos los datos de las PEC desde teclado */
    printf("Número de PECs a introduir (<%d): ", MAX_PECS);
    scanf("%d", &numPecs);

    for (i = 0; i < numPecs; i++) {
        tPec pecAux;
```

```
        printf("Datos de la PEC0%d : \n", i+1);
        printf("\tNombre : ");
        scanf("%s", pecAux.nombre);
        printf("\tNota : ");
        scanf("%f", &pecAux.nota);

        /* Añadimos a la tabla la tPec auxiliar utilizada
         * dentro del bucle. En cada iteración se construirá
         * y se añadirá una tPec diferente
         */
        insertar_pec(&fp, pecAux);
    }
    /* Calculamos la nota con la función calcular_nota */
    nota = calcular_nota(fp);
    printf("La nota promedio de las %d PEC es %f\n", fp.numPecs, nota);
    return 0;
}

/* Implementación funciones/acciones */

void insertar_pec(tAsignatura *asignatura, tPec pec) {
    /* numPecs contiene el número de elementos de tipo tPec
     * que contiene la tabla en cada momento
     */
    asignatura->pec[asignatura->numPecs] = pec;

    /* Una vez hemos asignado un elemento nuevo tPec en la tabla
     * incrementamos el valor de numPecs
     */
    asignatura->numPecs = asignatura->numPecs + 1;
}

float calcular_nota(tAsignatura asignatura) {
    /* La variable suma contiene el sumatorio de todas
     * las notas de las tPec que están dentro de la tabla
     * tAsignatura
     */
    float suma = 0;

    /* Se recorren todos los elementos tPec de tAsignatura
     * para obtener su nota y acumularlas a la
     * variable suma
     */
    for (int i = 0; i < asignatura.numPecs; i++) {
        suma = suma + asignatura.pec[i].nota;
```

```
    }

    /* Para calcular la media se divide el sumatorio de
     * notas por el total de elementos de la tabla tAsignatura
     */
    return suma/asignatura.numPecs;
}
```

8.4 Ejemplo: recorrido vs búsqueda

El siguiente ejemplo se tratan los conceptos de **recorrido** y de **búsqueda** de los elementos de una tabla. Dentro del código se han añadido comentarios detallados para que cada paso quede explicado.

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

/* Nos piden un programa que permita registrar
 * jugadoras de baloncesto dentro de una tabla. En una
 * misma tabla tendremos tanto las jugadoras locales
 * como las visitantes. Sobre ellas realizaremos
 * un recorrido, consistente en mostrarlas todas
 * por pantalla, y también una búsqueda, donde a partir
 * del equipo y el dorsal mostraremos por pantalla
 * la jugadora en caso que exista.
 * Necesitaremos implementar las siguientes acciones:
 * - inicializarTabla(...): para inicializar la
 *   tabla de jugadoras.
 * - insertarJugadora(...): para añadir una
 *   jugadora a una tabla.
 * - mostrarJugadoras(...): para mostrar por
 *   pantalla todas las jugadoras de la tabla.
 * - buscarJugadora(...): para buscar dentro de la
 *   tabla la jugadora que tenga un equipo y
 *   un dorsal determinado.
 */

#define MAX_NOMBRE 20+1
#define MAX_APELLIDO 20+1
```

```c
#define MAX_JUGADORAS 10

typedef enum {LOCAL, VISITANTE} tEquipo;

typedef struct {
    char nombre[MAX_NOMBRE];
    char apellido[MAX_APELLIDO];
    int dorsal;
    tEquipo equipo;
} tJugadora;

/* Definición del tipo tabla de
 * jugadoras; como se puede ver, según
 * se indica en las XWiki, únicamente
 * está formada por un vector y un
 * contador con las jugadoras que contiene.
 */
typedef struct {
    tJugadora jugadoras[MAX_JUGADORAS];
    int nJugadoras;
} tTablaJugadoras;

/* Predeclaraciones */
void leerJugadora(tJugadora *j);
void mostrarJugadora(tJugadora j);
void copiarJugadora(tJugadora *destino, tJugadora origen);
void inicializarTabla(tTablaJugadoras *tabla);
void insertarJugadora(tTablaJugadoras *tabla, tJugadora j);
void mostrarJugadoras(tTablaJugadoras tabla);
void buscarJugadora(tTablaJugadoras tabla, int dorsal, tEquipo equipo);

/* Programa principal */
int main(int argc, char **argv) {
    tJugadora jugadora;
    tTablaJugadoras tabla;
    tEquipo equipo;
    int dorsal, i;

    /* Para inicializar una tabla simplemente
     * ponemos el contador de elementos a 0.
     * La inserción de jugadoras en la tabla
     * tiene presente este contador, y sobrescribirá
     * cualquier otro valor que existiera en
     * esta misma posición.
     */
```

```
    inicializarTabla(&tabla);

    /* Pedimos las jugadoras para teclado y todo
     * seguido las vamos añadiendo a la tabla
     */
    for (i=0; i<MAX_JUGADORAS; i++) {
        leerJugadora(&jugadora);
        insertarJugadora(&tabla, jugadora);
    }

    /* Se recorren todos los elementos
     * de la tabla de jugadoras para
     * imprimir los datos por pantalla
     */
    mostrarJugadoras(tabla);

    printf("\nQué jugadora quieres buscar? : ");
    printf("\n>> Equipo (0=LOCAL, 1=VISITANTE) : ");
    scanf("%u", &equipo);
    printf(">> Dorsal : ");
    scanf("%d", &dorsal);

    /* Se hace una búsqueda entre todas las
     * jugadoras de la tabla, de forma que
     * mostrará por pantalla el resultado
     * obtenido; si no se encuentra ninguna,
     * devuelve un mensaje informativo.
     */
    buscarJugadora(tabla, dorsal, equipo);
    return 0;
}

/* Implementación de las acciones */
void leerJugadora(tJugadora *j) {
    printf("Introduce los datos de la nueva jugadora: \n");
    printf("\tNombre: ");
    scanf("%s", j->nombre);
    printf("\tApellido: ");
    scanf("%s", j->apellido);
    printf("\tEquipo (0=LOCAL, 1=VISITANTE): ");
    scanf("%u", &j->equipo);
    printf("\tDorsal: ");
    scanf("%d", &j->dorsal);
}
```

```
void mostrarJugadora(tJugadora j) {
    if (j.equipo == LOCAL) {
        printf("LOCAL    : %d %s,%s\n", j.dorsal, j.apellido, j.nombre);
    } else {
        printf("VISITANTE: %d %s,%s\n", j.dorsal, j.apellido, j.nombre);
    }
}

void copiarJugadora(tJugadora *destino, tJugadora origen) {
    strcpy(destino->nombre, origen.nombre);
    strcpy(destino->apellido, origen.apellido);
    destino->dorsal = origen.dorsal;
    destino->equipo = origen.equipo;
}

void inicializarTabla(tTablaJugadoras *tabla) {
    tabla->nJugadoras= 0;
}

void insertarJugadora(tTablaJugadoras *tabla, tJugadora j) {

    /* Se comprueba primero que la tabla no esté llena! */
    if(tabla->nJugadoras >= MAX_JUGADORAS) {
        printf("Error al insertar jugadora en la tabla\n");

    } else {
        /* Se añade la jugadora en la tabla y se incrementa
           el contador de elementos de la tabla */
        copiarJugadora(&tabla->jugadoras[tabla->nJugadoras], j);

        /* Importante!! después de añadir una jugadora no nos
         * tenemos que olvidar de incrementar el contador! */
        tabla->nJugadoras++;
    }
}

void mostrarJugadoras(tTablaJugadoras tabla) {
    int i;
    i = 0;
    printf("\nRecorrido: jugadoras insertadas en la tabla ...\n");
    while (i < tabla.nJugadoras) {
        mostrarJugadora(tabla.jugadoras[i]);
        i = i+1;
    }
}
```

```
void buscarJugadora(tTablaJugadoras tabla, int dorsal, tEquipo equipo) {
    int i;
    bool encontrada;

    /* El booleano 'encontrada' se utilizará
     * para salir del bucle que recorre todas
     * las jugadoras de la tabla: cuando encuentre
     * una, encontrada = true, la condición de entrada
     * del bucle ya no se cumplirá con lo
     * la búsqueda habrá finalizado.
     */
    encontrada = false;

    if (equipo == LOCAL) {
        printf("\nBúsqueda: jugadora local con dorsal núm. %d ...", dorsal);
    } else {
        printf("\nBúsqueda: jugadora visitant con dorsal núm. %d ...", dorsal);
    }

    for(i=0; i<tabla.nJugadoras && !encontrada; i++) {
        if (tabla.jugadoras[i].dorsal == dorsal &&
            tabla.jugadoras[i].equipo == equipo) {
                encontrada = true;
        }
    }

    /* Se muestra por pantalla los resultados
     * obtenidos
     */
    if (!encontrada) {
        printf("\n>> No se ha encontrado ninguna jugadora. \n");
    } else {
        printf("\n>> Jugadora encontrada : \n");
        mostrarJugadora(tabla.jugadoras[i-1]);
    }
}
```

Chapter 9

PEC09

9.1 Ejemplo: listaCartas

El planteamiento es el siguiente: imaginemos que tenemos una lista de cartas de tipo `DIAMANTES`,`CORAZONES`, `TREVOLES` y `PICAS`; lo que queremos conseguir es filtrar esta lista para un tipo determinado de carta, `DIAMANTES`, y añadir todas estas cartas de `DIAMANTES` a otra lista.

Una posible implementación sería:

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

/* Definición del modelo de cartas según el enlace:
 *  https://es.wikipedia.org/wiki/Baraja_francesa
 */

#define MAX_CARTAS 54+1
#define MAX_DIAMANTES_CARTAS 13+1

/* El término "palo" equivale a "baraja" */
typedef enum {DIAMANTES, PICAS, TREVOLES, CORAZONES} tPalo;

typedef struct {
    char valor;
    tPalo palo;
} tCarta;
```

```
typedef struct {
    tCarta cartas[MAX_CARTAS];
    int nCartas;
} tCartasList;

/* Predeclaración de las funciones y acciones */
void createList();
void insert(tCartasList *lista, tCarta carta, int indice);
void delete(tCartasList *lista, int indice);
tCarta get(tCartasList lista, int indice);
bool end(tCartasList lista, int pos);
bool emptyList(tCartasList lista);
bool fullList(tCartasList lista);
void printList(tCartasList lista);
void getCartasByPalo(tCartasList lista, tPalo palo, tCartasList *listaByPalo);

/* Programa principal */
int main(int argc, char **argv) {

    tCarta carta1, carta2, carta3, carta4, carta5;
    tCartasList listaCartas, listaCartasDiamantes;

    /* Creamos las dos listas */
    createList(&listaCartas);
    createList(&listaCartasDiamantes);

    /* Definimos una serie de cartas */
    carta1.valor = '3';
    carta1.palo = CORAZONES;
    carta2.valor = 'A';
    carta2.palo = DIAMANTES;
    carta3.valor = 'J';
    carta3.palo = TREVOLES;
    carta4.valor = '5';
    carta4.palo = DIAMANTES;
    carta5.valor = 'Q';
    carta5.palo = PICAS;

    /* Y las añadimos a la lista genérica de cartas */
    insert(&listaCartas, carta1, 0);
    insert(&listaCartas, carta2, 1);
    insert(&listaCartas, carta3, 2);
    insert(&listaCartas, carta4, 3);
    insert(&listaCartas, carta5, 4);
```

```
    /* Mostramos el contenido de la lista de cartas
     * por pantalla mediante la acción printList
     */
    printf("Contenido de la lista 'listaCartas' :\n");
    printList(listaCartas);

    /* Ahora queremos filtrar la lista genérica de cartas con
     * uno de los palos posibles. Concretamente queremos separar
     * de la lista genérica de cartas aquellas que sean
     * del palo DIAMANTES; lo hacemos mediante la llamada a
     * la acción getCartasByPalo. Para ver su funcionamiento,
     * revisad el comentario hecho en la implementación de esta
     * acción
     */
    getCartasByPalo(listaCartas, DIAMANTES, &listaCartasDiamantes);

    /* Y mostramos el contenido de la lista que contiene
     * únicamente las cartas del palo DIAMANTES
     */
    printf("Contenido de la lista 'listaCartasDiamantes' :\n");
    printList(listaCartasDiamantes);
    return 0;
}

/* Implementación de los métodos de la lista: se ha hecho un copy/paste
 * de la codificación C del ejemplo 19_12 de la XWiki, cambiando
 * el genérico "elem" por "tCarta", y el genérico "list" por "tCartesList",
 * ya que éstos serán los elementos con los que trabajaremos
 * en este ejemplo
 */
void createList(tCartasList *lista) {
    lista->nCartas = 0;
}

void insert(tCartasList *lista, tCarta carta, int indice) {

    int i = 0;
    if (lista->nCartas == MAX_CARTAS) {
        printf("\n Full list \n");
    } else {
        for (i=lista->nCartas-1; i>=indice; i--) {
            lista->cartas[i+1] = lista->cartas[i];
        }
        lista->nCartas++;
        lista->cartas[indice]=carta;
```

```
    }
}

void delete(tCartasList *lista, int indice) {

    int i;
    if (lista->nCartas == 0) {
        printf("\n Empty list\n");
    } else {
        for (i=indice; i<lista->nCartas-1; i++) {
            lista->cartas[i] = lista->cartas[i+1];
        }
        lista->nCartas--;
    }
}

tCarta get(tCartasList lista, int indice) {

    tCarta carta;

    if (lista.nCartas == 0) {
        printf("\n Empty list \n");
    } else {
        carta=lista.cartas[indice];
    }
    return carta;
}

bool end(tCartasList lista, int pos) {
    return (pos >= lista.nCartas);
}

bool emptyList(tCartasList lista) {
    return (lista.nCartas == 0);
}

bool fullList(tCartasList lista) {
    return (lista.nCartas == MAX_CARTAS);
}

/* A continuación se implementarán dos nuevas acciones que
 * no salen en el ejemplo 19_12. La primera acción,
 * printList(), imprime por pantalla una lista. la segunda
 * acción, getCartesByPalo(), permite hacer un filtrado de
 * cartas sobre una lista.
```

```
 */
void printList(tCartasList lista) {

    int i;
    tCarta cartaAux;

    for(i = 0; i < lista.nCartas; i++) {
        cartaAux = get(lista, i);
        if (cartaAux.palo == DIAMANTES) {
            printf(" [%c] de DIAMANTES\n", cartaAux.valor);
        } else if (cartaAux.palo == PICAS) {
            printf(" [%c] de PICAS\n", cartaAux.valor);
        } else if (cartaAux.palo == TREVOLES) {
            printf(" [%c] de TREVOLES\n", cartaAux.valor);
        } else if (cartaAux.palo == CORAZONES) {
            printf(" [%c] de CORS\n", cartaAux.valor);
        }
    }
}

/* La siguiente acción, getCartesByPalo, recibe tres parámetros:
 * - tCartasList lista (in): lista sobre la que aplicaremos el filtro
 * - tPalo tipo (in): corresponde al tipo de palo que utilizaremos
 *   para hacer el filtrado; por ejemplo, si como cuello indicamos
 *   DIAMANTES significa que el filtrado lo haremos
 *   sobre las cartas de tipo DIAMANTES
 * - tCartasList listaByPalo (out): lista de salida en la
 *   que se incluirán aquellas cartas de la lista de entrada que
 *   son del palo tipo
 */
void getCartasByPalo(tCartasList lista, tPalo tipo, tCartasList *listaByPalo) {

    int i, j;
    tCarta carta;

    createList(listaByPalo);
    i = 0;
    j = 0;

    /* Con un bucle y la función end(), controlamos que no
     * hayamos llegado al final de la lista
     */
   while (end(lista, i) == false) {

        /* Obtenemos la carta de la posición i */
```

```
        carta = get(lista, i);

        /* Si la carta es del palo indicado por tipo,
         * se añade a la lista de salida
         */
        if (carta.palo == tipo) {
            insert(listaByPalo, carta, j);
            j = j + 1;
        }
        i = i + 1;
    }
}
```

Chapter 10

PEC10

Chapter 11

PR1

11.1 Modo menu vs modo test

El workspace de la PR1 tiene habilidades dos modos de funcionamient/ejecución. Para activar un modo u otro hacemos lo siguiente, con el workspace de la PR1 abierto: **CodeLite** -> **Build** -> **Configuration manager ...**

Aquí se muestra un desplegable con dos opciones:

- **Menu**: es el modo estándar de funcionamiento del programa, el cual muestra el menú por pantalla con las acciones que permite realizar.
- **Test**: ejecutan una serie de tests para validar que las acciones que hemos codificado en nuestro programa funcionen como se espera que lo hagan.

Tanto si se elige la opción **Menu** como la opción **Test**, después tenemos que hacer el habitual **CodeLite** -> **Build** -> **Build and Run Project** para ejecutar el programa en el modo que hemos escogido.

11.2 Ejemplo: tupla dentro de tupla

A veces puede costar ver cómo trabajar con atributos de una tupla que a su vez está dentro de otra tupla, si se accede por valor, por referencia (puntero), etc. Por este motivo, se adjunta el siguiente ejemplo inventado, en el que se han añadido comentarios para que se vea claramente cómo trabajar con atributos de una tupla contenida dentro de otra tupla:

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

/* Un parking de un centro comercial nos ha
 * encargado una app que facilite sus
 * clientes a localizar donde ha aparcado su
 * vehículo.
 *
 * El parking ya dispone de tres tipos de
 * cámaras, independientes entre ellas:
 * - de entrada: cámara posicionada en la entrada
 *   del parking que, además de leer la matrícula,
 *   permite saber el tipo de vehículo.
 * - de acceso: cámaras que nos permiten
 *   saber si un vehículo ha subido o ha bajado
 *   una planta.
 * - de planta: cámaras que nos permiten saber
 *   la fila y el número de plaza donde se ha
 *   aparcado un vehículo.
 * El identificador de vehículo es su
 * matrícula.
 * A partir de la información obtenida por los
 * tres conjuntos de cámaras, deberá indicarse
 * al usuario dónde ha aparcado su vehículo.
 */

#define MAX_MATRICULA 7+1

typedef enum {COCHE, MOTO, FURGONETA} tTipo;

/* El lugar donde aparca un vehículo viene dado
 * por tres valores: la planta, la fila y el
 * número de la plaza.
 */
typedef struct {
    int planta;
    char fila;
    int numero;
} tAparcamiento;

/* El tipo tVehiculo contiene su matrícula
 * (identificador único), el tipo de vehículo
 * detectado en la primera cámara (COCHE, MOTO,
 * FURGONETA), y el lugar donde ha aparcado (tAparcamiento)
 */
```

```c
typedef struct {
    char matricula[MAX_MATRICULA];
    tTipo tipo;
    tAparcamiento aparcamiento;
} tVehiculo;

/* Predeclaración de las acciones */
void inicializar(tVehiculo *vehiculo, tTipo tipo, char *matricula);
void subirPlanta(tVehiculo *vehiculo);
void bajarPlanta(tVehiculo *vehiculo);
void aparcar(tVehiculo *vehiculo, char fila, int numero);
void obtenerPosicion(tVehiculo);

/* Programa principal */
int main(int argc, char **argv){

    tVehiculo vehiculo;

    /* Se lee la matrícula del vehiculo con la
     * cámara de entrada del parking y se asigna
     * en el elemento de tipo tVehiculo de nuestro
     * parking
     */
    printf("\n>> Entrada al parking \n");
    inicializar(&vehiculo, COCHE, "7472GZZ");

    printf("\n>> Bajar una planta\n");
    bajarPlanta(&vehiculo);

    printf("\n>> Bajar una planta\n");
    bajarPlanta(&vehiculo);

    printf("\n>> Subir una planta\n");
    subirPlanta(&vehiculo);

    printf("\n>> Bajar una planta\n");
    bajarPlanta(&vehiculo);

    printf("\n>> Bajar una planta\n");
    bajarPlanta(&vehiculo);

    printf("\n>> Aparcar\n");
    aparcar(&vehiculo, 'C', 23);

    printf("\nVamos de compras ...");
```

```
    printf("\n... pasan más de 3 hores ...");
    printf("\n... y olvidamos dónde aparcamos el vehículo !!\n");

    printf("\nSolución: consultamos la posición de nuestro vehículo");
    printf("\nen la app del parquing : \n");
    obtenerPosicion(vehiculo);
    return 0;
}

/* Como la posición viene dada por los atributos planta, fila y
 * número de tAparcamiento, es muy importante inicializar
 * los valores (sobre todo por la planta). La planta inicial del
 * parking es la 0, y el resto de plantas son subterráneas
 */
void inicializar(tVehiculo *vehiculo, tTipo tipo, char *matricula) {
    strcpy(vehiculo->matricula, matricula);
    vehiculo->tipo = tipo;
    /* El acceso al atributo aparcamiento (tupla) lo hacemos con '->'
     * ya que se trata de un puntero, y accedemos
     * los atributos de la tupla aparcamiento con '.'
     */
    vehiculo->aparcamiento.planta = 0;
    vehiculo->aparcamiento.fila = '-';
    vehiculo->aparcamiento.numero = 0;
}

/* Cuando subimos una planta, incrementamos en 1 el atributo
 * planta de la tupla tAparcamiento que contiene la tupla tVehiculo
 */
void subirPlanta(tVehiculo *vehiculo) {
    vehiculo->aparcamiento.planta = vehiculo->aparcamiento.planta + 1;
}

/* Cuando bajamos una planta, decrementamos en 1 el atributo
 * planta de la tupla tAparcamiento que contiene la tupla tVehiculo
 */
void bajarPlanta(tVehiculo *vehiculo) {
    vehiculo->aparcamiento.planta = vehiculo->aparcamiento.planta - 1;
}

/* Cuando aparcamos el vehiculo, damos valor a los atributos
 * fila y número de la tupla tAparcamiento que contiene la
 * tupla tVehiculo
 */
void aparcar(tVehiculo *vehiculo, char fila, int numero) {
```

```
    vehiculo->aparcamiento.fila = fila;
    vehiculo->aparcamiento.numero = numero;
}

/* Mostramos por pantalla la posición del tVehiculo */
void obtenerPosicion(tVehiculo vehiculo) {
    if (vehiculo.tipo == COCHE) {
        printf("\nCoche %s : ", vehiculo.matricula);
    } else {
        if (vehiculo.tipo == MOTO) {
            printf("\nMoto %s : ", vehiculo.matricula);
        } else {
            printf("\nFurgoneta %s : ", vehiculo.matricula);
        }
    }

    /* En este caso el acceso al atributo aparcamiento (tupla)
     * se hace con '.' ya que se ha pasado por valor (no es un puntero)
     * y accedemos a los atributos de la tupla aparcamiento con '.'
     */
    printf("planta %d, ", vehiculo.aparcamiento.planta);
    printf("fila %c, ", vehiculo.aparcamiento.fila);
    printf("número %d \n", vehiculo.aparcamiento.numero);
}
```

El resultat de l'execució d'aquest programa és:

```
>> Entrada al pàrking

>> Baixar una planta

>> Baixar una planta

>> Pujar una planta

>> Baixar una planta

>> Baixar una planta

>> Aparcar

Anem a comprar regals de reis ...
```

```
... passem més de 3 hores ...
... i ens oblidem d'on hem aparcat el vehicle !!

Solució: consultem la posició del nostre vehicle
a l'app del parking :

Cotxe 7472GZZ : planta -3, fila C, número 23
```

11.3 Desplazamiento de elementos en un vector

Los desplazamientos en un vector se pueden producir cuando se añade o elimina un elemento de un vector.

11.3.1 Añadir un elemento

Queremos añadir un elemento dentro del vector de una tabla en una posición que no es la última. Por ejemplo, si queremos que los elementos que añadimos a un vector vayan a la posición inicial, habrá antes desplazar el resto de elementos una posición a la derecha:

Contenido inicial del vector:

`[element2] [element5] [element1] [element7]`

Antes de añadir el nuevo `element6`, habrá que desplazarse todos los elementos una posición hacia la derecha, para hacer un hueco al inicio:

`[........] [element2] [element5] [element1] [element7]`

Ahora ya se puede añadir el `element6` en la primera posición del vector:

`[element6] [element2] [element5] [element1] [element7]`

Importante: como último paso, siempre debemos incrementar en 1 el atributo que contiene el tamaño de la tabla.

11.3.2 Borrar un elemento

Cuando queramos eliminar un elemento de un vector también se puede producir un desplazamiento de elementos. Por ejemplo:

Contenido inicial del vector:

`[element2] [element5] [element1] [element7]`

Para borrar `element5` simplemente desplazamos el resto de elementos que van a continuación una posición hacia la izquierda:

`[element2] [element1] [element7]`

Importante: como último paso, decrementar en 1 el atributo que contiene el tamaño de la tabla.

11.3.3 Ejemplo: diasSemana

A continuación se adjunta un ejemplo inventado para consolidar la explicación anterior: la idea es que se pueda comparar la inserción de elementos en una tabla como hemos hecho habitualmente hasta ahora (añadidos al final de todo), con otra acción que permite añadir de forma ordenada los elementos en la tabla. En esta segunda inserción habrá que ir realizando desplazamientos hacia la derecha de los elementos del vector de la mesa para que vayan quedando ordenados.

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

/* Definición de constantes */
#define MAX_DIAS 7
#define MAX_NOMBRE 9+1

/* Definición de la tupla tDia */
typedef struct {
    char nombre[MAX_NOMBRE];
    int id;
} tDia;

/* Definición de la tabla tSemana */
typedef struct {
    tDia dias[MAX_DIAS];
    int numDias;
} tSemana;

/* Predeclaración de las acciones */

/* Acción que inicializa la tabla tSemana,
 * mediante la asignación de 0 a numDias
 */
void inicializar(tSemana *semana);

/* Acción que añade un elemento de tipo tDia
```

```
 * en la tabla tSemana
 */
void insertarDia(tSemana *semana, tDia dia);

/* Acción que añade un elemento de tipo tDia
 * en la tabla tSemana de forma ordenada
 */
void insertarDiaOrdenado(tSemana *semana, tDia dia);

/* Acción que copia el contenido de un tDia origen (src)
 * a un tDia destino (dst)
 */
void copiarDia(tDia *dst, tDia src);

/* Acción que muestra por pantalla el contenido
 * de la tabla tSemana
 */
void mostrar(tSemana semana);

/* Programa principal */
int main(int argc, char **argv) {
    /* Definimo las variables */
    tSemana semana, semanaOrdenada;
    tDia lun, mar, mie, jue, vie, sab, dom;

    /* Inicialitzem les variables de tipus tDia */
    strcpy(lun.nombre, "lunes");
    lun.id = 1;
    strcpy(mar.nombre, "martes");
    mar.id = 2;
    strcpy(mie.nombre, "miércoles");
    mie.id = 3;
    strcpy(jue.nombre, "jueves");
    jue.id = 4;
    strcpy(vie.nombre, "viernes");
    vie.id = 5;
    strcpy(sab.nombre, "sábado");
    sab.id = 6;
    strcpy(dom.nombre, "domingo");
    dom.id = 7;

    /* Inicializamos las variables de tipo
     * tSetmana
     */
    inicializar(&semana);
```

```
    inicializar(&semanaOrdenada);

    /* Añadimos sin orden las variables
     * tDia dentro de las tablas semana
     * y semanaOrdenada. La diferencia
     * entre ellas será el método utilizado
     * para insertar los tDia: mientras que
     * la inserción por semana será
     * el habitual añadir al último
     * elemento de la tabla, para la
     * semanaOrdenada utilizaremos la acción
     * insertarDiaOrdenado()
     */
    insertarDia(&semana, jue);
    insertarDia(&semana, lun);
    insertarDia(&semana, mie);
    insertarDia(&semana, dom);
    insertarDia(&semana, sab);
    insertarDia(&semana, mar);
    insertarDia(&semana, vie);
    insertarDiaOrdenado(&semanaOrdenada, jue);
    insertarDiaOrdenado(&semanaOrdenada, lun);
    insertarDiaOrdenado(&semanaOrdenada, mie);
    insertarDiaOrdenado(&semanaOrdenada, dom);
    insertarDiaOrdenado(&semanaOrdenada, sab);
    insertarDiaOrdenado(&semanaOrdenada, mar);
    insertarDiaOrdenado(&semanaOrdenada, vie);

    printf("\nSemana sin ordenar : \n");
    mostrar(semana);
    printf("\nSemana ordenada : \n");
    mostrar(semanaOrdenada);
    return 0;
}

/* Implementación de las acciones */

void inicializar(tSemana *semana) {
    semana->numDias = 0;
}

void insertarDia(tSemana *semana, tDia dia) {
    /* numDias indica el número de elementos
     * de tipo tDia que contiene la tabla
     * tSemana en cada momento
```

```
     */
    copiarDia(&semana->dias[semana->numDias], dia);

    /* Una vez insertado un nuevo elemento
     * tDia en la taula, incrementamos el valor
     * de numDias
     */
    semana->numDias = semana->numDias + 1;
}

/* Acción que añade un elemento de tipo tDia
 * en una tabla tSemana de forma ordenada
 */
void insertarDiaOrdenat(tSemana *semana, tDia dia) {

    int i, j;
    bool isInsertado;

    isInsertado = false;
    i = 0;

    /* Para realizar la ordenación se utilizará el campo
     * id de tDia
     */
    for (i = 0; i < semana->numDias && !isInsertado; i++) {

        /* Si el id del tDia de la tSetmana > id del
         * parámetro dia, a partir de este punto
         * habrá que desplazar las tuplas tDia hacia
         * la derecha. El objetivo es conseguir un
         * lugar libre donde se añadirá el tDia dia
         * pasado por parámetro a la acción
         */
        if (semana->dias[i].id > dia.id) {

            /* Desplazamiento de los tDia hacia la derecha, a
             * partir de la posición donde habrá que añadir
             * el parámetro día pasado por valor
             */
            for (j = semana->numDias; j > i; j--) {
                copiarDia(&semana->dias[j], semana->dias[j-1]);
            }

            /* Se añade el parámetro dia a la
             * posición que le corresponde, una vez
```

```
             * desplazados las demás tuplas tDia
             * hacia la derecha
             */
            copiarDia(&semana->dias[j], dia);
            semana->numDias = semana->numDias + 1;
            isInsertado = true;
        }
    }

    /* Si en este punto todavía no se ha
     * añadido el tDia, significa que debe ir
     * al final de todo de la tabla tSemana
     */
    if (!isInsertado) {
        copiarDia(&semana->dias[i], dia);
        semana->numDias = semana->numDias + 1;
    }
}

void copiarDia(tDia *dst, tDia src) {
    dst->id = src.id;
    strcpy(dst->nombre, src.nombre);
}

void mostrar(tSemana semana) {
    int i;
    for (i = 0; i < semana.numDias; i++) {
        printf("%s\n", semana.dias[i].nombre);
    }
}
```

La ejecución del programa generará la siguiente salida:

```
Semana sin ordenar :
jueves
lunes
miércoles
domingo
sábado
martes
viernes
```

```
Semana ordenada :
lunes
martes
miércoles
jueves
viernes
sábado
domingo
```

Chapter 12

PR2

12.1 Ejemplo: pilaCartas

A continuación se expone un ejemplo donde, dada una pila de cartas inicial, lo que queremos es codificar la acción `separarDiamantes` que nos permita separar de la pila todas aquellas que son `DIAMANTES`, añadiéndolas a una nueva pila de cartas `DIAMANTES`.

Ejemplo: si inicialmente tenemos la pila1:

```
[Q] de PICAS
[5] de DIAMANTES
[J] de TREVOLES
[A] de DIAMANTES
[3] de CORAZONES
```

Queremos que por un lado la pila1 contenga todas las cartas que no son `DIAMANTES`:

```
[3] de CORAZONES
[J] de TREVOLES
[Q] de PICAS
```

Y por otro lado una nueva pila2 con todas las cartas `DIAMANTES`:

```
[A] de DIAMANTES
[5] de DIAMANTES
```

Se han añadido comentarios dentro del código para facilitar su comprensión:

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

/* Definición del modelo de cartas según el enlace:
   https://es.wikipedia.org/wiki/Baraja_francesa */

#define MAX_CARTAS 54+1

/* El término "palo" equivale a "baraja" */
typedef enum {DIAMANTES, PICAS, TREVOLES, CORAZONES} tPalo;

typedef struct {
    char valor;
    tPalo palo;
} tCarta;

typedef struct {
    tCarta A[MAX_CARTAS];
    int nelem;
} tStack;

/* Predeclaración de las funciones y acciones */
void createStack(tStack *s);
void push(tStack *s, tCarta e);
void pop(tStack *s);
tCarta top(tStack s);
bool emptyStack(tStack s);
bool fullStack(tStack s);
void printStack(tStack s);
void separarDiamantes(tStack *pilaCartas, tStack *pilaDiamantes);

/* Programa principal */

int main(int argc, char **argv) {
    tCarta carta1, carta2, carta3, carta4, carta5;
    tStack pilaCartas, pilaCartasDiamantes;

    /* Creamos dos pilas (revisad el comentario inicial del bloque
     * de implementación de las acciones/funciones de la pila).
     */
    createStack(&pilaCartas);
    createStack(&pilaCartasDiamantes);

    /* Definimos una serie de cartas */
    carta1.valor = '3';
```

```
    carta1.palo = CORAZONES;
    carta2.valor = 'A';
    carta2.palo = DIAMANTES;
    carta3.valor = 'J';
    carta3.palo = TREVOLES;
    carta4.valor = '5';
    carta4.palo = DIAMANTES;
    carta5.valor = 'Q';
    carta5.palo = PICAS;

    /* Y las añadimos a pilaCartas */
    push(&pilaCartas, carta1);
    push(&pilaCartas, carta2);
    push(&pilaCartas, carta3);
    push(&pilaCartas, carta4);
    push(&pilaCartas, carta5);

    /* Mostramos el contenido de pilaCartas por
     * pantalla, con la acción adicional printStack.
     */
    printf("\nContenido de la pila 'pilaCartas' :\n");
    printStack(pilaCartas);

    /* Ahora queremos separar de pilaCartas todas
     * aquellas cartas que son DIAMANTES, las cuales
     * formarán parte de una nueva pila de cartas.
     */
    printf("\nSeparamos las cartas en dos pilas!!\n");

    /* Explicación detallada dentro de la implementación
     * de la acción.
     */
    separarDiamantes(&pilaCartas, &pilaCartasDiamantes);

    printf("\nContenido de la pila 'pilaCartas' sin DIAMANTES :\n");
    printStack(pilaCartas);
    printf("\nContenido de la pila 'pilaCartasDiamantes' :\n");
    printStack(pilaCartasDiamantes);
    return 0;
}

/* Implementación de las acciones/funciones de la pila: se ha hecho un copy/paste
 * de la codificación C del ejemplo 19_04 de la XWiki, cambiando
 * el genérico "elem" por "tCarta".
 *
```

```
 * !!! Atención !!!: puede haber diferencias con funciones/acciones
 * que piden en la PR2. De cara a la PR2 debe codificar estos
 * métodos según las indicaciones del enunciado.
 */

void createStack(tStack *s) {
    s->nelem=0;
}

void push(tStack *s, tCarta e) {
    if (s->nelem == MAX_CARTAS) {
        printf("\n Full stack \n");
    } else {
        s->A[s->nelem]=e; /* first position in C is 0 */
        s->nelem++;
    }
}

void pop(tStack *s) {
    if (s->nelem == 0) {
        printf("\n Empty stack\n");
    } else {
        s->nelem--;
    }
}

tCarta top(tStack s) {
    tCarta e;
    if (s.nelem == 0) {
        printf("\n Empty stack \n");
    } else {
        e = s.A[s.nelem-1];
    }
    return e;
}

bool emptyStack(tStack s) {
    return (s.nelem == 0);
}

bool fullStack(tStack s) {
    return (s.nelem == MAX_CARTAS);
}

/* Acción adicional que muestra por pantalla todos
```

```
 * los elementos de una pila.
 */
void printStack(tStack s) {

    tCarta cartaAux;

    while (s.nelem > 0) {
        cartaAux = s.A[s.nelem-1];
        if (cartaAux.palo == DIAMANTES) {
            printf("  [%c] de DIAMANTES\n", cartaAux.valor);
        } else if (cartaAux.palo == PICAS) {
            printf("  [%c] de PICAS\n", cartaAux.valor);
        } else if (cartaAux.palo == TREVOLES) {
            printf("  [%c] de TREVOLES\n", cartaAux.valor);
        } else if (cartaAux.palo == CORAZONES) {
            printf("  [%c] de CORAZONES\n", cartaAux.valor);
        }
        s.nelem--;
    }
}

/* Acción que recibe dos parámetros:
 * - pilaCartes (inout)
 * - pilaDiamants (out)
 * Esta acción separa de pilaCartas (inout) aquellas
 * cartas con palo DIAMANTES, y las añade a
 * PilaDiamantes (out). Así una vez ejecutada la acción, tendremos:
 * - pilaCartas: contendrá todas las cartas que no son DIAMANTES.
 * - pilaDiamantes: contendrá todos los DIAMANTES.
 */
void separarDiamantes(tStack *pilaCartas, tStack *pilaDiamantes) {

    tCarta cartaAux;
    tStack pilaNoDiamantes;

    /* Todas las cartas inicialmente están a pilaCartes.
     * Se trata de ir añadiendo los DIAMANTES a la pilaDiamantes,
     * y los que no son DIAMANTES la pila temporal pilaNoDiamantes.
     * Posteriormente asignaremos pilaNoDiamantes a pilaCartas (inout).
     */

    /* Inicializamos la pila auxiliar. */
    createStack(&pilaNoDiamantes);

    /* Tenemos que tratar todos los elementos de la pila. */
```

```
    while (!emptyStack(*pilaCartas)) {
        cartaAux = top(*pilaCartas);
        if (cartaAux.palo == DIAMANTES) {
            push(pilaDiamantes, cartaAux);
        } else {
            push(&pilaNoDiamantes, cartaAux);
        }

        pop(pilaCartas);
    }

    /* Reasignación de la pila auxiliar pilaNoDiamantes
     * a pilaCartas, que al fin y al cabo es el parámetro
     * de tipo inout de la acción.
     */
    *pilaCartas = pilaNoDiamantes;
}
```

Chapter 13

VMWare y CodeLite

13.1 ¿Por qué una máquina virtual?

La máquina virtual se utiliza para tener un entorno homogéneo de programación, tanto por parte de los estudiantes como por parte de los consultores, de forma que cualquier enunciado/solución publicado en las aulas de teoría funcione a todos los estudiantes, y que todos los programas que realice se comporten igual a los entornos que se utilizarán para corregirlos.

Hace unos semestres nos encontramos con unos pocos casos en los que un programa que funcionaba correctamente en el PC de un estudiante, fallaba a la hora de ser corregido. Y también algunos enunciados que en determinados sistemas operativos/versiones de compiladores C, tampoco funcionaban correctamente. Por este motivo se decidió utilizar una máquina virtual.

Nosotros no tenemos forma de controlar que realmente utilice la FP20181 o bien un CodeLite instalado directamente en su PC, pero si no es así se puede dar alguna situación comentada y nada deseable.

13.2 VirtualBox y requerimientos de virtualización

Para poder utilizar VirtualBox es necesario que disponga de acceso a la opción de virtualización. Si no es así, se pueden obtener errores del tipo: ** VT-X está deshabilitado en el BIOS para todos los CPUs **.

El error se puede producir por varias razones:

- La BIOS del PC tiene deshabilitada la opción. Para solucionarlo, hay que entrar dentro de la BIOS del PC y encontrar la opción de activación de la virtualización (varía según el fabricante): puede ser **VT-x**, **Virtualization Technology**, **VTX/AMD-V**, **Intel Virtual Technology**, **Tecnología de virtualización (VTX/VTD)**, etc.
- Tenemos activo un antivirus el cual tiene activada una virtualización propia que entra en conflicto con la requerida para VirtualBox. En este caso, únicamente hay que desactivar la opción de virtualización del antivirus.

13.3 Cómo instalar las Guest Additions

Las **Guest Additions** son una serie de drivers que mejoran la interacción entre host y máquina virtual. Para instalarlas únicamente hay que hacer:

1. Poner en marcha la máquina virtual **FP20181**
2. Ir a la barra de menú superior -> **VirtualBox** -> **Devices** -> **Insert Guest Additions CD Image ...** -> instalará una unidad de CD con las **Guest Additions**.
3. Abrir un terminal desde **Lubuntu** -> **System Tools** -> **LXTerminal**
4. Dentro del terminal **LXTerminal** ejecutar el siguiente (la versión que se esté instalando puede ser más nueva que la 6.0.12):

```
cd /media/uoc/VBox_GAs_6.0.12/
sudo sh ./VBoxLinuxAdditions.run
```

1. Reiniciar la **FP20181** para activar las **Guest Additions**.

13.4 Primeros pasos con CodeLite

A las XWiki encontrará el módulo **Introducción al entorno de programación CodeLite**, en el que se detallan los pasos a realizar para preparar el entorno.

A continuación se resumen una serie de aspectos que se comentan en la XWiki y que son importantes de recordar:

- Un **workspace** de CodeLite es una agrupación de proyectos.
- Únicamente se puede tener un **workspace** abierto dentro de CodeLite.
- Para crear un **workspace**: **CodeLite** -> **Workspace** -> **New Workspace ...** -> **Workspace type:** `C ++` -> **Workspace name**: el que corresponda; **Workspace Path**: `/home/uoc/Documentos/codelite/workspaces/` (o cualquier otro) -> fin
- Como no se puede tener más de un **workspace** abierto, no podrá crear ningún nuevo si previamente no cierre el workspace activo. Si dentro

del menú de CodeLite ve la opción **New Workspace** en gris y que no se puede seleccionar, significa que ya tiene un workspace abierto. Para cerrarlo: **CodeLite** -> **Workspace** -> **Close Workspace**.

- Para añadir un proyecto a un workspace: **CodeLite** -> **File** -> **New** -> **New Project** -> de tipo **Console**: `Simple ejecutable (gcc)`-> **Project name**: el que corresponda -> **Compiler**: `GCC`; **Debugger**: `GNU gdb debugger` -> fin
- El proyecto que acabamos de crear contiene un programa `hello world` de muestra, que si ejecutamos muestra el mensaje *"hello world"* por pantalla. Este programa en C lo podemos editar y añadir/quitar todo lo que queremos. Es aquí dentro donde debemos codificar nuestros programas en C (**no** los algoritmos!).
- Si tenemos más de un proyecto dentro de un workspace, la forma que tenemos para indicar cuál de ellos es el que está activo es haciendo **doble clic** sobre el nombre del proyecto. Verá que el nombre queda remarcado en ***negrita y cursiva***: a partir de este momento, este será el proyecto que compilaremos y ejecutaremos desde las opciones del menú de CodeLite. Aunque estemos visualizando en pantalla el código de otro proyecto, la compilación y ejecución siempre se hará del proyecto activo.
- Para mostrar la barra de herramientas (iconos) en la parte superior: **CodeLite** -> **View** -> **Show toolbar**.
- El icono del **play** de color verde hacia la derecha de la barra de herramientas es el **debugger**, no sirve para compilar.
- Para compilar y ejecutar podemos hacer: **CodeLite** -> **Build** -> **Build and run project**. También se puede compilar con el icono de la barra de herramientas de la flecha blanca abajo con fondo verde, y ejecutar con el icono de las ruedas dentadas grises.
- El resultado de la ejecución del programa se mostrará en una pantalla nueva tipo terminal. Es importante ir cerrando estas ventanas una vez ya hemos comprobado el resultado de la ejecución.

13.5 Cómo activar un proyecto

Dentro de CodeLite, en la parte izquierda se muestran todos los projects que se han creado en el espacio de trabajo. Si nos fijamos en el nombre de todos ellos, veremos que uno está remarcado en negrita; por ejemplo podemos tener:

- PEC01
- ***PEC02***
- PEC03
- PEC04

Esto significa que cuando vamos a **CodeLite** -> **Build** -> **Build and Run Project**, se ejecutará la acción sobre el project ***PEC02***, aunque por pantalla esté mostrando el código de otro project.

Si se hace **doble clic** con el ratón sobre el nombre del project **PEC04**, ahora veremos:

- PAC01
- PAC02
- PAC03
- ***PEC04***

A partir de este momento, el **Build and Run Project** se aplicará sobre ***PEC04***.

13.6 Cambiar idioma del teclado

Para cambiar el idioma del teclado en Lubuntu, pulsad con el botón derecho del ratón sobre la barra gris superior -> **Add/ Remove Panel Items** -> pestaña **Panel Applets** -> botón **Add** -> seleccionad el plugin **Keyboard Layout Handler** -> **Add** -> **Close**.

En estos momentos en la parte superior derecha se mostrará el idioma definido por defecto por el teclado -> marca sobre la bandera con el botón derecho -> **"Keyboard Layout Handler" settings** -> desmarca la opción **Keep system layout** -> se añade el idioma que se desee desde el botón **Add**; se puede priorizar un idioma u otro poniéndolo en primera posición en la lista. Una vez se haya guardado la configuración deseada, ya se habrá cambiado la disposición del teclado al nuevo idioma.

Si se han dejado definidos varios idiomas en la lista, cada vez que se haga click sobre la bandera de la parte superior derecha, se cambiará al otro idioma de la lista.

13.7 Programa por defecto al crear un proyecto

Cada vez que se crea un nuevo proyecto a CodeLite, por defecto siempre contiene el código del programa `hello world`:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("hello world\n");
    return 0;
}
```

Por lo tanto, cuando queremos crear nuestro propio programa, basta con borrar el programa que crea por defecto y empezar a codificar tu programa nuevo.

Si a pesar de todo no se quiere que se muestre este código cada vez, podemos modificar el código por defecto asociado a los nuevos proyectos:

- Editar el archivo `/usr/share/codelite/templates/projects/ejecutable/main.cpp`, que es el que se carga por defecto al crear cada nuevo proyecto.
- Crear un nuevo template de CodeLite y utilizarlo en el momento de crear un nuevo proyecto. La creación de un nuevo templates se hace a partir de un proyecto que tengamos -> botón derecho sobre el nombre del proyecto -> `Save as template ...`.

13.8 Cómo fijar el kernel de inicio con Lubuntu

A pesar de ser una situación nada habitual, puede darse el caso de que Lubuntu sólo nos arranque en un determinado kernel por problemas con determinadas tarjetas gráficas.

Aunque el kernel siempre se puede seleccionar en el momento de inicio de Lubuntu, podemos fijar qué es lo que queremos utilizar a pesar de que éste no sea el más nuevo de todos. Una posible opción para hacerlo con la aplicación **Grub Customizer**; para instalarla hacemos:

```
sudo add-apt-repository ppa:danielrichter2007/grub-customizer
```

Nos identificamos con la password del usuario **uoc**.

A continuación, actualizamos la lista de packages de todos los repositorios:

```
sudo apt-get update
sudo apt-get install grub-customizer
```

En este punto tendremos instalado **Grub Customizer** dentro de la máquina virtual **FP20181**: este programa nos permitirá confirmar exactamente qué hay definido en el arranque grub de la máquina virtual. Para ejecutar el programa: icono de **Lubuntu** -> **System Tools** -> **Grub Customizer** -> poner contraseña **uoc** -> en la pestaña inicial **List configuration** se muestran todos los kernels de que dispone el entorno en estos momentos.

Es recomendable hacer una copia previa de las PAC/PR que podamos tener dentro de la **FP20181** antes de cambiar el kernel de inicio. Para cambiar el kernel de arranque por defecto: **Grub Customizer** -> pestaña **General Settings** -> como **default entry**, seleccionar el kernel que funciona correctamente de entre todos los disponibles. Una vez elegido, se pulsa el botón **Save** de la parte superior izquierda.

Para aplicar el cambio que acaba de guardar en el arranque de Lubuntu abrir un terminal de Lubuntu desde **System Tools** -> **LXTerminal**, y desde el terminal teclear:

```
sudo update-grub
```

Una vez haya finalizado el proceso, hay que realizar un reboot de la **FP20181** para que coja ahora por defecto el kernel elegido.

Chapter 14

Otros

14.1 Bibliografía

Recursos gratuitos:

- *C Notes for Professionals book*: https://books.goalkicker.com/CBook/
- *C Programming*: https://en.wikibooks.org/wiki/C_Programming
- *C Programming Boot Camp*: https://www.gribblelab.org/CBootCamp/
- *The C Book*: https://publications.gbdirect.co.uk//c_book/
- *Programming in C*: http://ee.hawaii.edu/~tep/EE160/Book/PDF/Book.html
- *The ANSI C Programming Language*: https://www.dipmat.univpm.it/~demeio/public/the_c_programming_language_2.pdf