Chapter 3

Diseño descendente en C

3.1 Diseño descendente de algoritmos

Al hablar de las acciones y funciones se vio como una cierta tarea –el intercambio de valores entre dos variables, la ordenación de los elementos de una tabla o el cálculo de un factorial, por ejemplo– que aparezca, repetida o no, dentro de un algoritmo puede extraerse del mismo, especificarse y reescribirse en función de una serie de parámetros formales para conseguir de ese modo una definición de dicha tarea que sea genérica e independiente del contexto particular en el que aparece (este contexto está formado por las variables del algoritmo sobre las que actúa dicha tarea).

Esa capacidad de abstracción por parametrización es un importante recurso cuando se quiere diseñar algoritmos en los que los objetos a tratar no pertenecen a ninguno de los tipos básicos que proporciona el lenguaje de programación empleado.

Supongamos que se quiere resolver el problema de especificación

```
Entrada f:  fichero de caracter;
Requisitos:  El fichero no esta vacio;
Salida:  l_max (entero) contiene la longitud de la palabra mas larga de f
```

Tras un momento de reflexión, es fácil caer en la cuenta que se trata de un problema de recorrido sobre ficheros[1]. En este caso los elementos que forman el fichero son de tipo carácter, pero en la especificación del problema no se habla de caracteres sino de *palabras* –bloques de caracteres distintos del carácter blanco–, y esa diferencia entre los elementos constitutivos del fichero (caracteres) y los términos en los que se enuncia la especificación (palabras) hace necesario plantear la solución del problema con un enfoque que evite tomar en consideración detalles que, por irrelevantes en un primer momento, podrían entorpecer el diseño del algoritmo.

Así, la solución de este problema se podría presentar como un algoritmo de recorrido sobre ficheros en el que la *unidad lógica de tratamiento* sea la palabra. Este algoritmo requerirá una operación que obtenga la longitud de la primera palabra que se encuentre desde la posición del elemento distinguido[2]. Llamaremos `obtenerLongitud` a esta

[1]El problema es muy parecido al que tiene en la teoría. Sin embargo el enfoque de la solución es diferente. Se ha considerado que una palabra es algo que comienza por blancos (que pueden no existir) seguidos de letras y seguidas por un blanco (que puede no estar)

[2]Operación que en el algoritmo que se da a continuación desempeña un papel totalmente homólogo al desempeñado por la operación *leer* en los recorridos vistos hasta ahora.

operación y le daremos forma de acción con la siguiente especificación[3]:

```
Entrada f:  fichero de caracter;
Requisitos:  El fichero no esta vacio y pd(f) = αM y α ≠ ε;
Modifica:  f avanzando el elemento distinguido hasta el principio de la
        siguiente palabra;
Salida:  long (entero) contiene la longitud de la primera palabra de α;
accion obtenerLongitud(e/s f:  fichero de caracter; sal long:  entero);
```

Si el lenguaje empleado para describir nuestros algoritmos ya dispusiera de la operación `obtenerLongitud`, entonces para obtener la solución correcta del problema bastaría con emplearla del mismo modo en el que se emplea la operación `leer` en los recorridos sobre secuencias vistos hasta ahora.

```
Entrada f:  fichero de caracter;
Requisitos:  el fichero no esta vacio;
Salida:  l_max (entero) contiene la longitud de la palabra mas larga de f
algoritmo longitudDeLaPalabraMasLarga;
    var
        lup:  entero;
    fvar
    abrir_lec(f);
    l_max:= 0;
    mientras no fdf(f) hacer
        obtenerLongitud(f, lup);
        si lup > l_max ⟶
            l_max:= lup
        [] lup ≤ l_max ⟶
            continuar
        fsi
    fmientras;
    cerrar(f)
falgoritmo
```

Sin embargo, el lenguaje algorítmico no dispone de esa operación por lo que es necesario definirla. La definición consiste en saltar los blancos previos a la palabra cuya longitud se va a obtener, y una vez saltados estos blancos, contar los caracteres que forman la palabra:

```
accion obtenerLongitud(e/s f:  fichero de caracter; sal long:  entero);
    var
        c:  caracter;
    fvar
    saltarBlancos(f, c);
    contarLetras(f, c, long);
faccion
```

donde `saltarBlancos` y `contarLetras` son operaciones que han surgido de forma natural al definir `obtenerLongitud`. Sus especificaciones son:

[3]La parte de requisitos exige que aún queden cosas por leer

```
Entrada:  f un fichero de caracter;
Requisitos:  pd(f) =αM con α ≠ ε
Modifica:  f, a la salida el elemento distinguido de f es el primer
        caracter no blanco de α
Salida:  c, un caracter que coincide con el elemento distinguido de f
accion saltarBlancos(e/s f:  fichero de caracter; sal c:  caracter);

Entrada:  f un fichero de caracter, f = αM y un caracter c;
Requisitos:  ninguno
Modifica:  f, a la salida el elemento distinguido de f es el primer
        caracter blanco de α
Salida:  long, letras leidas (incluyendo c)
accion contarLetras(e/s f:  fichero de caracter; ent c:  caracter;
        sal long:  entero);
```

De nuevo, si el lenguaje usado proporcionara operaciones que cumplieran tales especificaciones, al emplearlas en la definición de `obtenerLongitud` el problema quedaría resuelto. Dado que el lenguaje no dispone de tales operaciones es necesario definirlas. Aparecen así una serie de definiciones de segundo nivel:

```
accion saltarBlancos(e/s f:  fichero de caracter; sal c:  caracter);
    leer(f, c);
    mientras no fdf(f) y (c = ' ') hacer
        leer(f, c)
    fmientras
faccion;

accion contarLetras(e/s f:  fichero de caracter; ent c:  caracter;
        sal long:  entero);
    long:= 0;
    mientras no fdf(f) y (c <> ' ') hacer
        long:= long + 1;
        leer(f, c)
    fmientras;
    si c <> ' ' ⟶
        long:= long + 1
    [] c = ' ' ⟶
        continuar
    fsi
faccion
```

Finalmente, estas operaciones se han podido definir recurriendo únicamente a elementos que proporciona el lenguaje por lo que el problema queda ya resuelto completamente.

Recapitulación: La filosofía de diseño que se ha seguido consiste en lo siguiente:

- En primer lugar, abordar la solución del problema en los términos en los que éste se plantea, imaginando una *unidad lógica de tratamiento* adecuada. En el ejemplo presentado, la unidad lógica de tratamiento sería la palabra.
- En segundo lugar, aplicar alguno de los esquemas algorítmicos conocidos empleando operaciones que manejen elementos del tipo de la unidad lógica de tratamiento.

- Por último, definir estas operaciones si el lenguaje empleado no dispone de ellas. Estas definiciones pueden dar lugar a nuevos problemas y la solución de los mismos puede intentarse poniendo de nuevo en práctica esta misma filosofía en contextos que, probablemente, obliguen a realizar nuevas abstracciones aunque, es de esperar, más sencillas.

3.2 Otro ejemplo: palabra más larga

Dado un fichero de texto, el problema consiste en devolver una palabra de ese texto cuya longitud sea máxima.

```
Entrada:  f, un fichero de caracteres;
Requisitos:  f no esta vacio;
Salida:  pMax es la primera aparición en f de una palabra cuya longitud
        sea mayor o igual que la del resto de palabras de f
```

La abstracción que va a facilitar el diseño de un algoritmo para este problema consiste, nuevamente, en recorrer el fichero como si éste estuviese formado por palabras (obviamente haciendo una primera lectura para iniciar `pMax` con la primera palabra del fichero). Suponemos que nuestro lenguaje permite declarar objetos de tipo palabra y que, además, para manejar objetos de este tipo dispone al menos de las operaciones `longitudDePalabra` y `esPalabraVacia`, así como de la operación `leerPalabra`. Entonces como solución al problema se podría proponer lo siguiente:

```
algoritmo palabraMasLarga;
    var
        p:palabra;
    fvar
    abrir_lec(f);
    prepararPalabra(pMax);
    mientras no fdf(f) hacer
        leerPalabra(f, p);
        si longitudDePalabra(p) > longitudDePalabra(pMax) ⟶
            pMax:= p
        [] longitudDePalabra(p) <= longitudDePalabra(pMax) ⟶
            continuar
        fsi
    fmientras;
    cerrar(f)
falgoritmo
```

Sin embargo, el lenguaje no dispone del tipo palabra y, por tanto, tampoco de las operaciones que permiten manejarlo, por lo que es necesario definir todos estos elementos.

Comencemos por definir un nuevo tipo de dato que permita representar los objetos que aparecen en el nivel de abstracción en el que se plantea el problema: el tipo palabra. Emplearemos en su definición el constructor de tipos estructurados **tupla**:

```
tipo
    palabra = tupla
```

```
        letras:  tabla [1..M] de caracter;
        long:  entero;
    ftupla;
ftipo
```

Cada variable o parámetro que se declare de tipo `palabra` estará formado por dos campos, el primero de ellos se llamará `letras` y su tipo será el de una tabla de caracteres del tamaño indicado por la constante `M`. El segundo de los campos tendrá por nombre `long` y guardará memoria del número de posiciones ocupadas en el campo `letras`.

A continuación definimos las operaciones que, en la solución propuesta, operan con palabras:

```
Entrada:  p, una palabra;
Requisitos:
Salida:  long, la longitud de p
funcion longitudDePalabra(p:palabra) dev long:  entero ;
    long:= p.long;
    dev long
funcion;

Entrada:  p, una palabra;
Requisitos:  Salida:  el booleano b es verdadero si y solo si p no tiene
        letras
funcion esPalabraVacia(p:palabra) dev b:  booleano ;
    b:= (longitudDePalabra(p) = 0);
    dev b
ffuncion;
Entrada f:  fichero de caracter;
Requisitos:  El fichero no esta vacio y pd(f) = αM y α ≠ ε;
Modifica:  f avanzando el elemento distinguido hasta el principio de la
        siguiente palabra;
Salida:  p (entero) contiene la primera palabra de α;
accion leerPalabra(e/s f:  fichero de caracter; sal p:palabra);
    var
        c:  caracter;
    fvar
    saltarBlancos(f, c);
    copiarLetras(f, c, p)
faccion;
```

Las definiciones anteriores, tanto de nuevos tipos de datos como de acciones y funciones, corresponden al *primer nivel de refinamiento* de la solución del problema. Para completar la solución es necesario proporcionar definiciones para las operaciones `saltarBlancos` y `copiarLetras`. Estas son definiciones correspondientes al *segundo nivel de refinamiento*. Obviamente, como acción `saltarBlancos` podemos utilizar la misma que se usó en el anterior ejemplo. Para `copiarLetras` podemos usar:

```
Entrada f:  fichero de caracter pd(f) = βM y c, un caracter
Requisitos:  c coincide con el elemento distinguido de f;
Modifica:  f avanzando el elemento distinguido hasta el principio de la
```

```
        siguiente palabra;
Salida:  p (entero) contiene la primera palabra de β
accion copiarLetras(e/s f:  fichero de caracter; sal p:  palabra);
    prepararPalabra(p);
    mientras no fdf(f) y (c <> ' ') hacer
        p.long:= p.long + 1;
        p.letras[p.long]:=c;
        leer(f, c)
    fmientras;
    si c <> ' ' ⟶
        p.long:= p.long + 1;
        p.letras[p.long]:=c
    [] c = ' ' ⟶
        continuar
    fsi
faccion;
```

Que utiliza una acción de un *tercer nivel de refinamiento* para inicializar las variables de tipo `palabra`. El código de esta última función podría ser[4]:

```
Entrada:
Requisitos:
Salida:  p, una palabra inicializada a 0 letras;
accion prepararPalabra(sal p:palabra);
    p.long:= 0
faccion;
```

3.3 Tuplas en C

Para codificar en C el algoritmo anterior tan sólo necesita saber, además de lo que aparece en las prácticas anteriores, como se crean tuplas en C. La definición y el manejo de tuplas en lenguaje C es muy similar a lo visto en el lenguaje algorítmico empleado en las clases de teoría. El constructor de tipos `tupla` recibe en C el nombre de `struct`. Este constructor se utiliza poniendo la palabra clave `struct` seguida, opcionalmente, por un nombre y, obligatoriamente, por un listado de campos, separados por `;` entre llaves. Los campos se definen de igual modo que las variables de un programa, es decir un nombre de tipo seguido de un listado separado de comas de nombres de campo. El nombre del tipo es `struct` *`nombreOpcional`*.

La construcción `struct` *`nombreOpcional`*`{`*`listaDeCampos`*`}` puede aparecer en los mismos sitios de un programa que cualquier tipo básico. A pesar de ello lo más habitual es definir el tipo y crear un alias para usarlo de forma má cómoda. Es decir, la forma habitual de trabajo es escribir:

```
typedef struct nombreOpcional {
    tipo nombreCampo1;
    ...
```

[4]Nótese que no es necesario inicializar los elemento de la tabla. El campo `long` nos dice que posiciones tienen sentido y cuáles no. Esto,además, es mucho más rápido y segiro que utilizar un valor concreto para indicar que a partir de una determinada posición los valores son irrelevantes.

```
    tipo nombreCampo_n;
} Alias;
```

Al definir el tipo de esta manera, el tipo tiene dos nombres `struct` *`nombreOpcional`* o bien `Alias`. Es costumbre muy extendida usar como `nombreOpcional` el mismo término que se usa para `Alias`.

Las tuplas en C tienen un comportamiento muy similar a las variables de tipos básicos. Se pueden asignar y se pasan siempre por valor a las funciones. La única diferencia es que no se pueden comparar. Es decir, para que se puedan comparar es preciso que se defina una función que las compare.

Como operador de acceso a los campos del `struct` también se emplea el punto (.). En el acceso a un campo del `struct`, el identificador de la variable precede al punto, a continuación, va el punto, y finalmente el identificador del campo al que se quiere acceder. Ahora bien, como sabe, es muy común que trabajemos con apuntadores a variables en lugar de trabajar con variables. Esto nos llevaría a tener que escribir en múltiples ocasiones `(* variable).nombreDeCampo`. Para evitar esto en C existe otra forma de acceso a los campos de un struct del que se conoce su apuntador: el operador flecha `->`. Escribir `(* variable).nombreDeCampo` es lo mismo que poner `variable->nombreDeCampo`

El siguiente ejemplo muestra cómo definir objetos de tipo `struct`, declarar variables de esos tipos y cargar los campos de esas variables con valores. Obsérvese que el tipo *coordenada* es empleado en la definición del tipo *pixel*, por lo que su definición debe preceder a la definición de *pixel*.

```
typedef struct coordenada {
    int x;
    int y;
} Coordenada;

typedef struct pixel {
    Coordenada c;
    int color;
} Pixel;

void cambiaColorYCoordX(Pixel * q; int c; int cx){
        ⋮
    q->color = c;
    q->c.x = cx;
        ⋮
}

void main ()
    Pixel p;
        ⋮
    p.c.x = 10;
    p.c.y = 10;
    p.color = 23;
        ⋮
    cambiaColorYCoord(&p, 40, 30);
}
```

3.4 Actividades a realizar

Sígase la filosofía de diseño descendente para construir algoritmos que cumplan las especificaciones de los apartados siguientes.

1. **Apariciones de la primera palabra** (`cuenta.c`).

   Dado un fichero de caracteres cuéntese el número de veces que la primera palabra aparece a lo largo del mismo. Para implementar este algoritmo conviene utilizar acciones y funciones del segundo ejemplo de esta práctica.

2. **Decodificación del código ASCII** (`decascii.c`).

   Sea f un fichero de caracteres que contenga sólo ceros y unos, en número múltiplo de 8. Si cada bloque de ocho elementos de f se ve como la representación en binario de un número entre 0 y 255, se pide construir la secuencia de elementos delcódigo ASCII formada por los caracteres que corresponden a los códigos numéricos obtenidos al hacer dicha interpretación del contenido del fichero f.

Función para chequear si dos palabras son iguales

```
Entrada:  p1=P y p2=Q, dos palabras
Requisitos:  p1 y p2 están inicializadas
Salida:  el booleano b será verdadero si y sólo si P=Q
función sonPalabrasIguales(p1, p2:  palabra) dev b:  booleano
    var
        i:  entero;
    fvar
    si p1.long = p2.long →
        i:=1:
        mientras ((i < p1.long) && (p1.letras[i]=p2.letras[i])) hacer
            i:= i+1
        fmientras;
        b:=(p1.letras[i]=p2.letras[i])
    [] p1.long <> p2.long →
        b:= FALSO
    fsi;
    dev b
ffunción
```

Aunque puede que lo parezca, la práctica no acaba aquí. Lea el siguiente apartado

Algunas cuestiones sobre los ficheros de texto

En esta práctica le pido que trabaje con ficheros de texto. Para ello debe recordar el uso de las funciones `fopen`, `fclose`, `fgetc`, `fgets`, `fread` y `fscanf`. También es necesario que conozca algunas peculiaridades de los ficheros de caracteres.

Pruebe a crear un fichero de texto vacío haciendo en el terminal de comandos, por ejemplo, `touch ficheroTexto.txt`. Si ejecuta posteriormente `ls -l` comprobará que el

fichero contiene 0 bytes [5]. Lógico, pues no ha escrito usted nada. Ahora bien, si ahora abre ese fichero con un editor de textos, escribe un carácter y cierra el fichero (salvando, claro) el resultado de `ls -l` no es obvio. Lo lógico sería que le indicase que contiene un byte, pero dependiendo del editor empleado comprobará que pueden ser 2 o incluso más. ¿Que sucede?. Lo que le he dicho en clase alguna vez: Las aplicaciones le engañan. Muchos editores de texto meten un salto de línea, `\n` al final de cualquier fichero de texto con el que trabajan (si no lo tenía, claro). Otros utilizan representaciones más extensas para los caracteres que un único byte, y otros añaden información administrativa de la aplicación. El comportamiento puede incluso diferir para la misma aplicación en diferentes sistemas operativos.

Por lo tanto, la única manera de saber sin genero de dudas que un fichero, incluso de texto, contiene exactamente lo que usted quiere es que sólo se cree y lea con aplicaciones cuyo funcionamiento conozca exhaustivamente o que usted mismo haya creado.

Para que me entienda, si utiliza editores normales para crear los ficheros de ejemplo para comprobar los programas de esta práctica, es posible que no le funcionen bien porque el fichero creado no contenga lo que usted supone.

[5] lo indica la primera columna numérica del listado que obtiene