ALGORITMOS Y TIPOS DE DATOS EN C

Miguel Toro Bonilla

Editorial Universidad de Sevilla

Algoritmos y tipos de datos en C

Editorial Universidad de Sevilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Algoritmos y tipos de datos en C // Miguel Toro Bonilla

Miguel Toro Bonilla

Algoritmos y tipos de datos en C

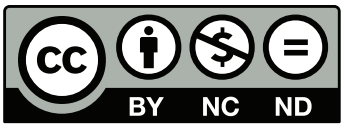
Editorial Universidad de Sevilla

Sevilla 2023

Algoritmos y tipos de datos en C // Miguel Toro Bonilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Núm.: 5



Editorial Universidad de Sevilla 2023
c/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: https://editorial.us.es

DOI: https://dx.doi.org/10.12795/9788447225088

Maquetación: Miguel Toro

Diseño de cubierta y edición electrónica:
referencias.maquetacion@gmail.com

Índice

Agradecimientos

Para aprender a programar, lo mejor es ponerlo en práctica. En este manual vamos a abordar algoritmos y estructuras de datos en el lenguaje C. Los lenguajes de programación tienden a ir compartiendo las mismas ideas básicas, dado que unos y otros van tomando prestadas recíprocramente las más novedosas. Aunque existen particularidades, podemos decir que los lenguajes de programación van convergiendo hacia elementos comunes a todos ellos. Por ello, parece sensato aprender las ideas compartidas entre los más actuales. En este sentido, vamos a abordar en C conceptos que hemos aprendido en Java y que se pueden extender a otros. El diseño de tipos ocupará un lugar central en este volumen. Al final, se incluyen varios ejemplos diseñados con estas ideas. El material que conforma el texto procede de la experiencia acumulada durante varios años en la enseñanza de la asignatura Análisis de Datos y Algoritmos en la Universidad de Sevilla. En https://github.com/migueltoro/C_v2.git se encuentra el código de los ejemplos.

Sevilla, septiembre de 2023

Introducción

En este capítulo vamos a ver los elementos de la programación en C. Este es un lenguaje no orientado a objetos, por lo que las técnicas y el estilo de programación son muy diferentes a las que hemos usado en Java. En Java hacemos programación orientada a objetos. En C seguimos una filosofía de programación denominada programación estructurada. Pero, como veremos, muchas técnicas aprendidas nos serán útiles en este nuevo estilo.

Introduciremos los elementos de C y, cuando sea necesario, veremos los elementos que se comparten con Java y los que no. En general, decimos que C es un lenguaje de más bajo nivel que Java.

Un programa en C es un conjunto de funciones. Una de ellas tiene de nombre *main* y es el comienzo del programa. Cada función tiene una cabecera (que también denominaremos *prototipo* o *signatura*) y un cuerpo.

La compilación de un programa C se hace en dos pasos bien diferenciados: preprocesamiento y compilación propiamente dicha. La primera fase es llevada a cabo por el preprocesador y se encarga, como veremos abajo, de incluir ficheros y expandir definiciones de macros. Esta fase no existe en lenguajes como Java. La segunda fase se encarga de la traducción del código fuente a código máquina.

Estructura de un programa en C

El código de un programa en C se compone de un conjunto de ficheros con extensiones *.h* y *.c*. Los ficheros *.h* contienen definiciones (de tipos y signaturas de funciones). Los ficheros *.c* contienen el código de las funciones. Debe haber un fichero que contenga la función *main* de la forma:

Importaciones

```
void main ()  {
       declaración de variables;
       instrucciones;
}
```

Declaración de importaciones

Las importaciones son un conjunto de líneas que comienzan con *#include*. Cada línea de este tipo indica al preprocesador que incluya en ese punto el fichero que le sigue. Pueden ser de los dos tipos siguientes:

```
#include <fichero1.h>
#include "fichero2.h"
```

Estas líneas indican al preprocesador que sustituya la línea correspondiente por el contenido del fichero indicado. Si es fichero está entre " " entonces se busca en la misma ubicación del código fuente. Si el nombre de fichero está entre < > entonces se busca en una ruta definida por el entorno.

Ejemplos:

```
#include <stdio.h>
#include <math.h>
#include "punto.h"
```

Estas líneas equivalen a las declaraciones *import* de JAVA

Definición de macros

Son un conjunto de líneas que comienzan por la declaración *#define.* Indican al preprocesador que en en el resto de instrucciones del código sustituya un nombre (posiblemente con parámetros) por una definición dada. Hay dos tipos según si el nombre va seguido de paréntesis o no.

```
#define id1 exp1
#define id2(p1, p2) exp2(r1, r2)
```

La primera línea indica que el preprocesador debe sustituir en el resto del código el *id1* por *exp1*. La segunda línea indica al preprocesador que sustituya las apariciones de *id2(p1, p2)* por *exp2(r1, r2)* y los parámetros *p1, p2* que había en la definición se sustituirán a su vez por a los parámetros concretos *r1, r2*.

Ejemplos:

```
#define PI 3.14159
#define MENSAJE "Introduzca su edad: "
#define area(r) PI*r*r
```

Tipos, declaraciones, operadores, expresiones

Los tipos básicos son:

- *char*, que representa un carácter
- *int / long*, que representa un número entero
- *float / double*, que representa un número real
- *bool,* que representa un boolean
- *void,* que representa a un tipo sin valores
- En los primeros compiladores de C no existía el tipo boolean. El tipo *int* ocupaba su lugar. El valor cero es equivalente a false y los demás valores a true. Los operadores relacionales y lógicos devuelven *int*. La relación entre *bool* e *int* sigue existiendo
- No existe el tipo *String*. Se trabaja con cadenas de caracteres acabadas en '\0'.

Estos tipos básicos equivalen a los tipos predefinidos de Java. En C, de forma parecida, se pueden declarar tipos formados por una serie de valores. Esto se hace, como en Java, con la declaración *enum*.

Sobre los tipos básicos se pueden aplicar algunos calificadores como *unsigned.* Esta puede aplicarse al tipo *char* o a un tipo entero para representar enteros positivos o cero.

Funciones de librería matemáticas

Las funciones matemáticas usuales vienen proporcionadas en la librería *math.h.* Algunas de ellas son:

```
double log(double x);       // logaritmo neperiano
double log10(double x);     // logaritmo decimal
double sin(double x);       // seno
double cos(double x);       // coseno
double pow(double x, double y);    // x elevado a y
double exp(double x);       // e elevado a x
double sqrt(double x);      // raíz cuadrada
double ceil(double x);      // menor número entero mayor o igual que x
double floor(double x);     // mayor número entero menor o igual que x
```

Además, se proporcionan un conjunto de funciones para convertir cadenas de caracteres con un formato dado a valores de los tipos básicos. Vienen en *stdlib.h* y algunas de ellas son:

```
int atoi(const char * s);  //convierte la parte inicial de s a un entero
long int atol(const char * s); //convierte la parte inicial de s a un long
double atof(const char * s); //convierte la parte inicial de s a un double
int abs(int num);          // valor absoluto
long int labs(long int num);  // valor absoluto
div_t div(int n, int d); //devuelve cociente y resto de n entre d
int rand(void);            // número entero aleatorio entre 0 y RAND_MAX
```

Funciones de librería sobre cadenas de caracteres

El lenguaje C no tiene el tipo *String* de Java para almacenar cadenas de caracteres. Una cadena de caracteres es una tabla de caracteres donde los

elementos útiles van desde la casilla 0 hasta que una contenga el carácter `\0`. Este carácter es el fin de cadena. Las funciones sobre cadenas de caracteres funcionan mal si usamos cadenas sin el carácter final `\0`. Las constantes de tipo cadena son secuencias de caracteres entre comillas como en Java.

El entorno de C proporciona un conjunto de funciones de librería para manipular cadenas de caracteres. Los prototipos de estas funciones están en *string.h.* Son:

*int strlen (const char * s);* Calcula la longitud de s

*char * strchr(const char * s, char c);* Busca la primera posición del carácter c y devuelve un puntero al mismo o NULL si no lo encuentra.

*char * strstr(const char * s1, const char * s2);* Devuelve un puntero a la primera posición de la secuencia de caracteres en s2 (excluyendo el carácter nulo) dentro de s1 o NULL si no la encuentra.

*char * strcpy (char * s1, const char * s2);* Copia s2 en s1 y devuelve un puntero a s1.

*int strcmp(const char * s1, const char * s2) ;* Devuelve un entero negativo, cero o positivo según que s1 sea menor, igual o mayor que s2 en el orden lexicográfico.

*char * strcat(char * s1, const char * s2);* Concatena s2 detrás de s1 y devuelve un puntero a s1.

El significado de *char** lo veremos en próximos apartados.

Declaración de variables

A diferencia de Java las declaraciones de variables deben ir al principio de un bloque. Tienen como ámbito hasta el final del bloque (más adelante veremos variables globales con ámbito en todo el programa). La declaración de las variables se hace de acuerdo con el siguiente formato:

```
tipo lista_de_identificadores;
```

Ejemplos:

```
char c;
int i, j;
long potencia;
double radio, longitud;
```

En C es posible declarar una variable fuera de todos los bloques. Son las denominadas variables globales. Estas variables tienen una vida que dura todo el programa. Son equivalentes a las variables *static* de Java.

Definición de constantes

Para definir constantes usamos el preprocesador. La forma recomendada es:

```
#define identificador_de_constante  valor_constante
```

Las constantes es usual escribirlas con letras mayúsculas.

Ejemplos:

```
#define PI 3.14159
#define MAXIMO 999
#define ULTIMALETRA 'Z'
#define MENSAJE "Introduzca su edad:"
```

Una forma alternativa de declarar que una variable no puede cambiar de valor es precediendo el tipo del calificador *const.* Por ejemplo:

```
const int i = 2;
```

Definición de tipos

Con la cláusula *typedef* se pueden declarar nuevos tipos.

Ejemplos:

```
typedef enum {R, G, B} Color;
Color c;
```

En el ejemplo anterior se ha declarado el tipo Color y la variable c de ese tipo.

Expresiones

Como en Java, las expresiones son combinaciones de constantes, variables operadores y funciones. Todas las expresiones, si tienen una sintaxis correcta, tienen un tipo.

Los operadores disponibles en C son similares a los de Java. Se clasifican en aritméticos, relacionales, lógicos y de asignación. Más adelante veremos otros tipos de operadores.

Estructuras y operadores asociados

Una estructura es un agregado de datos. Cada dato se llama un *campo.* Tienen las siguientes características:

- Cada campo puede ser de distinto tipo (*int, char, float, etc.*).
- Cada campo debe tener un identificador distinto.
- Una estructura se declara con la palabra *struct.*

En general, a partir de una declaración *struct* definiremos un tipo nuevo usando *typedef* tal como se muestra abajo. Una variable de un tipo construido mediante un *struct* puede inicializarse con un conjunto de valores para los campos separados por comas y delimitados por llaves.

```
typedef struct {
   T1 c1;
   T2 c2;
   ...
   Tn cn;
} Tr;

Tr r = {v1, v2, ..., vn};
```

Ejemplo:

```
typedef struct {
       double x;
       double y;
} punto;

punto p = {2.0,3.1};
```

Hay disponibles dos operadores para acceder a los campos de un tipo *struct*. Es el operador punto. Si tenemos la declaración vista arriba para el tipo *Punto,* tenemos:

```
punto  p = {2.0,3.1};
int x = p.x;
```


Punteros

Un puntero es una variable capaz de almacenar direcciones de memoria y se usa para señalar donde hay guardada una información. Una variable puede ser declarada de tipo *puntero a un tipo T*. Esto se consigue posponiendo * al tipo T.

Ejemplos

```
int *pint;
float *pf;
```

En el ejemplo anterior hemos declarado la variable *pint* de tipo puntero a *int* y *pf* de tipo puntero a *float*. Eso quiere decir que los valores de *pint* serán direcciones de memoria adecuadas para señalar la dirección donde se ubica un entero. Igualmente ocurre con *pf* pero respecto a un *float*.

La constante *NULL* es un valor que pueden tomar los punteros para explicitar que tienen un valor indefinido.

Como todos los tipos, junto con los valores, el tipo puntero dispone de un conjunto de operadores. Los operadores disponibles son: ** y &*. El operador & se aplica a una variable de tipo T y devuelve un valor de tipo T * (la dirección donde se encuentra ubicada en la memoria la variable). El operador * se aplica sobre una variable de tipo T * y devuelve un valor de tipo T (el valor contenido en la memoria señalada por el puntero). El

operador & se le suele denominar obtención de la dirección y al operador * obtención del contenido (señalado por un puntero).

Estos operadores se relacionan por la propiedad siguiente, que es válida para cualquier tipo T.

```
T a, b;
T * p;
p = & a;
b = * p;
```

Es decir, *p* se ha declarado T * (puntero a *T*), e inicializado con la dirección de *a*. La variable *b* se ha inicializado con el contenido señalado por *p*. Entonces a y b contienen el mismo valor de tipo T.

Ejemplo. Sean dos variables enteras i y *j* y un puntero a entero *p* y un puntero a entero *p2* inicializado con la dirección de *j*.

```
int i, j;
int * p;
int * p2 = &j;
i = 15;
p = &i;                                    // p "apunta" a i
j = *p +7;                                 // se accede  a  lo
                                           //apuntado por p
```

El valor de j es 22.

Punteros a *struct*

Los punteros a *struct* tienen un operador especial para acceder a los campos del *struct*. Es el operador ->.

```
typedef struct {
       double x;
       double y;
} punto;

punto p1 = {2.0,3.1};
punto * p = &p1;
double x = p->x;
```

El valor de x es 2.0.

Los punteros a *struct* permiten una definición de tipos recursivos como en el caso:

```
typedef struct bt {
       tree_type tree_type;
       type label_type;
       void * label;
       struct bt * left;
       struct bt * right;
} binary_tree;
```

Punteros a *void*

El tipo *void*, como en Java, no tiene valores. Un puntero de tipo *void ** representa la dirección de un objeto, pero no su tipo y pueden convertirse a punteros de cualquier tipo haciendo un *casting*.

Los tipos genéricos de Java se pueden representar en C mediante un puntero a *void,* que indica su dirección en memoria, y un entero, que indica el tamaño del dato.

Punteros a funciones

En C también existen punteros a funciones. Se declaran rodeando el nombre de la función prefijado con un * en un paréntesis. En el ejemplo se declara un puntero a función que toma dos parámetros `(double, char)` y devuelve un *double*.

```
int sum (int num1, int num2) {
    return num1+num2;
}
```

```
double (*p2f)(double, char); //Declaración del puntero a
       // función
f2p = sum; //Inicialización del puntero a función
int op1 = f2p(10, 13); //Llamada a la función
int op2 = sum(10, 13); //Llamada a la función
```

Los punteros a función tienen el papel de las funciones lambda en Java.

Arrays

El concepto de array es equivalente al visto en Java. Un array es un conjunto de datos (*llamados elementos*), con las siguientes propiedades:

- Todos son del mismo tipo (*int, char, float, etc.*)
- Se accede a cada uno de ellos mediante un índice.
- Pueden ser multidimensionales. A estos se accede con varios índices. Pueden ser reducidos a arrays con un solo índice.

Declaración e inicialización de arrays

```
tipo1 nombre[dim];
tipo2 nombre[] = {v1, v2, ..., vn};
tipo3 nombre[d1][d2];
```

Ejemplos:

```
int v [10];
char palabra [256];
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  inicializa fila 0 */
   {4, 5, 6, 7} ,   /*  inicializa fila 1 */
   {8, 9, 10, 11}   /*  inicializa fila 2 */
};
int b[3] = {7,8,9,10}
b[1] = a[1];
```

La variable v es un array de 10 enteros. La variable palabra es un array de 256 caracteres. Un array de *char* es la manera adecuada para ubicar cadenas de caracteres. El tipo array tiene asociado el operador [] de acceso a los elementos del array. Se usan números enteros como índices dentro del operador []. Pero el índice puede tomar cualquier valor y si no lo controlamos los programas que construyamos funcionarán mal y el error será difícil de detectar.

Ejemplos:

```
int v[10];         // Se declara el array pero no se
             // inicializan las casillas
int a[] = {2,-3,7}; // Se declara el array y se inicializan
las casillas
a[2] = a[1] +2;    // a ha quedado modificada a {2,-3,-1}
v[4] = v[3];      // incorrecto la casilla v[3] se está usando
             // antes inicializarse,
         // pero no es detectado por el compilador.
```

En el ejemplo anterior se declara *v* pero no se inicializan sus casillas. Las casillas no inicializadas toman valores basura por lo que no deben ser usadas.

Arrays y punteros: aritmética de punteros

Existen equivalencias claras entre el tipo puntero a T y el array de T. Veamos algunas equivalencias entre tablas y punteros. Estas equivalencias son la llamada aritmética de punteros.

Ejemplos

```
#define d
#define d1
#define d2
typedef struct {
       T1 x;
       T2 y;
} Tr;
Ta a[d1][d2];
Ta * b = (Ta *) a;
Tr v[d];
Tr * p = v;
```

Entonces existen las siguientes equivalencias (incluimos también las equivalencias cuando aparecen punteros a *struct*

v equivale a &v[0] // dirección del primer elemento del vector
&v[i] equivale a v+i // dirección del elemento i
p+i equivale a v+i
p[i] equivale a v[i]
v[i] equivale a *(p+i)
v[i] equivale a *(v+i)
p->x equivale a (*p).x
a[i][j] equivale a *(a+d2*i+j)

Como vemos el nombre de un array es esencialmente un puntero, pero constante. Es decir, no puede ser colocado a la izquierda de un operador de asignación. Por otra parte, a cualquier puntero es posible sumarle un entero *i* (también restarle). Esa operación nos da otro puntero que señala a la casilla correspondiente. Estas operaciones se denominan aritmética de punteros.

El operador *[]*, denominado operador de indexación puede colocarse detrás de un nombre de un array o de un puntero. En general si p es un puntero, lo que incluye al identificador de un array, entonces *p[i]* es equivalente a **(p+i)* para cualquier *i.* Ambas expresiones pueden ser usadas indistintamente ya a la derecha, consulta del contenido del

elemento *i*, o a la izquierda, modificación del elemento *i*, de un operador de asignación.

La aritmética de punteros no es adecuada para el tipo *void ** pero sí para *void ***.

Algunas versiones del compilador de C puede que no acepten algunas de las equivalencias anteriores. En ese caso usaremos la versión que más se adapte a nuestras necesidades.

Ahora recordamos que las cadenas de caracteres siempre son del tipo *char ** (punteros a caracteres). La cadena de caracteres tiene reservadas un array de caracteres, pero puede que no los ocupe. Acaba cuando aparece el carácter '\0'.

Punteros y memoria dinámica

Las funciones para gestionar la memoria dinámica son:

malloc: Obtiene un bloque de memoria de num bytes
calloc: Obiene num bloques de memoria de tamaño size
realloc: Reubica un bloque de memoria y lo amplia al tamaño newsize byte
memcpy: Copia un bloque de memoria de num bytes
free: Libera un bloque de memoria

Sus prototipos son:

```
void * malloc(int num);
void * calloc(int num, int size);
void * realloc(void *address, int newsize);
void * memcpy(void * destino, const void * origen, int  num);
void free(void * addres);
```

Estas funciones pueden ser encontradas en *<stdlib.h>.*

Para el control de la memoria dinámica es conveniente diseñar un tipo para ello. Es decir, ocultar el uso de las operaciones anteriores dentro de

tipo mediante el cual podamos controlar la memoria creada y liberarla cuando sea conveniente. En el API se ofrece el tipo *memory_heap*

Tipos estáticos y tipos dinámicos

Los tipos que usaremos en C podemos clasificarlos en estáticos y dinámicos. Los estáticos no necesitan obtener memoria dinámicamente. Tienen la ventaja que se declaran a principio de un bloque, así obtienen la memoria necesaria, y desaparecen a final del bloque incluida la memoria obtenida. Tienen la desventaja de que su tamaño no puede cambiar en tiempo de ejecución. Los tipos dinámicos necesitan, en C, obtener memoria dinámica en tiempo de ejecución. Tienen la ventaja de que pueden cambiar la cantidad de memoria necesaria en tiempo de ejecución, pero hay que gestionar tanto la memoria necesaria como recogida de esa memoria cuando no sea necesaria. En otros lenguajes, como Java, de esta tarea se encarga el llamado recolector de basura.

En los tipos estáticos las variables se declaran (en este momento se les adjudica automáticamente la memoria por parte del compilador), se inicializan, y se usan. El compilador libera la memoria cuando acaba el bloque.

En los tipos dinámicos haya más fases: las variables se declaran (pero sólo es un puntero lo que se declara), se busca la memoria necesaria, se inicializan, se usan y cuando se estime oportuno se libera la memoria.

Tipos genéricos

Cada tipo genérico se puede instanciar en muchos tipos diferentes con necesidades de memoria completamente distintas. Por ello, en C, son tipos dinámicos. Para simularlos en C los representamos mediante un puntero a *void*, que puede ubicar cualquier puntero, y un entero con la memoria que necesita. Para calcular la memoria necesaria en tiempo de ejecución usaremos la función de C que calcula el tamaño en memoria de los tipos conocidos:

```
int sizeof(type);
```

Si tenemos dos variables *a, b* del tipo genérico *T*, la forma de manejarlas es asignarles memoria dinámica y trabajar con un puntero a void. La idea general para declararlas y asignarlas es:

```
void * a = malloc(n);
void * b = malloc(n);
…
memcpy(a, b, n);
```

Donde *n* se sustituirá por el tamaño del tipo concreto que instancie el genérico. En tiempo de ejecución debemos mantener la información del tamaño del tipo genérico que corresponde con el tamaño de la zona de memoria donde señala el puntero a *void*. Esto lo podemos encapsular en el tipo que usemos para gestionar la memoria dinámica.

Toda la información sobre el tipo genérico la guardaremos en una variable de tipo *type*, que veremos más adelante, que hará posible la consulta de las propiedades del tipo en tiempo de ejecución.

Sentencias de control, funciones y paso de parámetros

Un bloque es una secuencia de declaraciones y sentencias delimitadas por {} como en Java. Las declaraciones tienen como ámbito hasta el final de este, pero a diferencia de Java deben ir al comienzo del bloque.

Las sentencias de control disponibles en C son un subconjunto de las de Java:

```
if() {} else if {} else {}
while() {}
for(int i= i0; i<limit; i++) {}
switch(exp_integer) { case a:  .. break; … default …}
```

No existe el for extendido.

No existen excepciones en C. Se suele usar la macro *assert* que tiene la forma:

```
void assert(int test);
```

Esta macro se expande como un bloque *if*, en el que se comprueba la condición *test* y, dependiendo de si es o no verdadera, puede abortar el programa. Si test se evalúa como cero (es decir, si es falsa) entonces se aborta el programa mediante la función *abort()* y se imprime un mensaje en *stderr* en el que se incluyen la condición *test*, el nombre del archivo fuente y el número de línea en la que se llamó a *assert()* . Si se coloca la directiva *#define NDEBUG* (no depurar) en el archivo fuente antes de incluir *assert.h*, entonces la macro no tendrá ningún efecto.

Funciones y procedimientos: paso de parámetros

Todo programa C está construido en base a funciones. Las funciones permiten estructurar la codificación de los programas reduciendo su complejidad y como consecuencia, mejorando su desarrollo.

Como las variables la funciones y procedimientos se declaran antes de ser usadas. Igual que las variables tienen un tipo las funciones tienen también un tipo asociado su prototipo, cabecera o signatura. Es conveniente estructurar el código agrupando los prototipos de las funciones en un fichero con extensión *.h*. El código asociado (cuerpo de las funciones) se incluye en el fichero del mismo nombre con extensión *.c.* Usualmente, dentro de la cultura del C, las funciones que devuelven *void* se les llama procedimientos. En los ficheros .h se incluyen, además, la definición de tipos, y los #define con los #include necesarios. Los ficheros .h cumplen el papel de los interfaces en Java y loos .c de las clases.

Paso de parámetros

El paso de parámetros en C es como en Java: se copia el valor de los parámetros reales en los parámetros formales y se ejecuta el cuerpo del método invocado. Al terminar, si existe la sentencia *return*, se devuelve el valor de la expresión correspondiente. Es lo que se denomina paso de parámetros por valor. En Java distinguíamos dos tipos de parámetros: parámetros de entrada y parámetros de entrada-salida. Los mismos conceptos existen en C pero ahora debemos identificar con claridad que parámetros son de un tipo y de otro.

De manera simple podemos decir que los parámetros cuyo tipo es *T ** (puntero a un tipo *T*) son de entrada salida, el resto parámetros de entrada. Es decir, son parámetros de entrada aquellos que son de unos de los tipos básicos (*char, int, float, ...*), de tipo *enum* y de tipo *struct*. Son parámetros de entrada salida los que son punteros a alguno de los tipos anteriores.

Las funciones también pueden pasarse como parámetros mediante punteros a funciones. Veamos como ejemplo el prototipo de la función *qsort* (está en *<stdlib.h>*) que ordena un array de un tipo genérico mediante un orden:

```
void qsort(void *base,int n, int size, int (*compar)(const
      void *, const void*));
```

Los detalles son:

- El puntero base al comienzo del array
- El tamaño (*size*) de cada elemento
- El número (*n*) de elementos
- El orden, como el *Comparator<T>* de Java, es una función que toma dos variables de un mismo tipo genérico de tamaño *size* y devuelve un entero.

Número variable de parámetros

En C es posible tener funciones con un número variable de parámetros. Abajo diseñamos la función *toListDeParametros* que crea una lista a partir de un número variable de parámetros de un tipo dado. La función toma un primer parámetro de tipo entero (*numArgs*) que indica el número de parámetros que siguen. En este ejemplo se muestra la forma de trabajar con un número variable de parámetros. Esto se consigue en C con el uso de las macros predefinidas *va_list*, *va_start*, *va_arg* y *va_end*. Estas macros están definidas en *stdarg.h.*

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

double average(int num,...) {

   va_list valist;
   double sum = 0.0;
   int i;

   /* inicializa valist con num, el número de argumentos */
   va_start(valist, num);

   /* accede a los argumentos indicando su tipo */
   for (i = 0; i < num; i++) {
      sum += va_arg(valist, double);
   }
         /* limpia la memoria*/
   va_end(valist);

   return sum/num;
}
```

```
int main() {
   printf("Average of 2., 3., 4., 5. = %f\n",
       average(4, 2.,3.,4.,5.));
   printf("Average of 5., 10., 15. = %f\n",
       average(3, 5.,10.,15.));
}
```

Con *va_list* declaramos una variable que gestionará la secuencia variable de parámetros, *va_start* inicializa la variable declarada (*args*) tomando un parámetro adicional (*numArgs*) que indica el parámetro detrás del cual comienzan los parámetros variables y que también cuántos hay. La macro *va_arg* devuelve el siguiente parámetro disponible en la lista inicializada por *va_start*. Finalmente, usamos *va_end* para limpiar la memoria.

Entrada-salida

En C se disponen de un conjunto de funciones para mostrar resultados en la consola, ubicarlos en una cadena o en un fichero. Igualmente existen otras para leer de la consola, de una cadena de caracteres o de un fichero.

Al conjunto de todas ellas las denominamos funciones de entrada salida con formato. Su objetivo es convertir cadenas de caracteres en secuencias de valores de otros tipos (*int, float, ...)* o viceversa.

Un flujo de datos se representa en un programa en *C* con el tipo *FILE **. Las funciones entradas salida más usuales son:

```
FILE * fopen(const char * file, const char * mode); // mode "r" o "w"
int fclose(FILE * stream);
int feof(FILE * stream);
printf (char * format, arg1, arg2, ..., argn);
sprintf(char * s,char * format, arg1, arg2, ..., argn);
fprintf(FILE * stream, char * formato, arg1, arg2, ..., argn);
scanf (char * format, parg1, parg2, ..., pargn);
sscanf(char * s,char * format, parg1, parg2, ..., pargn);
fscanf(FILE * stream, char * formato, parg1, parg2, ..., pargn);
int fputs(const char * s, FILE * stream);
int puts(const char *s);
char * fgets(char * s, int n, FILE * stream);
char * gets(char *s);
```

Donde *file* es el nombre de un fichero, *stream* es un flujo de datos resultado de abrir un fichero, *format* es una cadena con el formato de las variables con una estructura similar al de Java, *arg* son variables y *parg* punteros a variables.

Estructuración de programas en C: diseño de tipos

Comparado con Java C es un lenguaje de menor nivel de abstracción. Algunas diferencias concretas son:

- C no es orientado a objetos.
- C no tiene sobrecarga de métodos.
- C no tiene paquetes, ni interfaces ni clases.
- C no tiene tipos genéricos.
- C no tiene recolector de basura.

C y Java son dos lenguajes, por tanto, con objetivos y dominios de aplicación diferentes. De todas formas, es posible y recomendable estructurar los programas en C intentando imitar, en lo posible, algunos mecanismos de modularización disponibles en Java. Para ello vamos a proponer algunas ideas que nos puedan guiar en este camino. Estas ideas nos van a permitir estructurar los programas en C, por una parte, y comprender más en profundidad el funcionamiento de los programas en Java por otra.

Cada tipo tiene un conjunto de valores y un conjunto de funciones para operar con ellos. De estas funciones hay algunas que son muy usadas: la igualdad, el orden natural, si lo tiene, la forma de construir un valor a partir de una cadena de caracteres y su inversa la representación como cadena de caracteres de un valor.

Información de los tipos en tiempo de ejecución

Para estructurar los programas en C, es conveniente construir un conjunto de tipos que nos permitan una programación más cómoda. Lo primero es diseñar un conjunto de funciones que nos permitan trabajar de forma homogénea con los tipos básicos y otros que podamos ir definiendo. Todas esas funciones ya las proporcionan algunos lenguajes de alto nivel y otras hay que diseñarlas en cada caso. Nos referimos a las funciones de *parse*, *equals*, *tostring*, *orden natural*, *size*, y otras de más bajo nivel como *swap*, *copy*, etc.

```
typedef struct gtp {
       char name[15];
       bool (*equals)(const void * e1, const void * e2,
              struct gtp * t);
       char * (*tostring)(const void * e, char * mem,
              struct gtp * t);
       int (*order)(const void * e1, const void * e2,
              struct gtp * t);
       void * (*parse)(void * out, char * text,
              struct gtp * t);
       void (*free)(void * e, struct gtp * t);
       void * (*copy_new)(void * in, heap * h, struct gtp * t);
       void (*copy)(void * out, void * in, struct gtp * t);
       int size;
       int num_types;
       struct gtp * types[2];
}type;
```

Para poder usar los tipos en tiempo de ejecución, debemos reunir sus propiedades más importantes en una variable que contenga las funciones relevantes del tipo. Estas variables serán de tipo *type* anterior. Cada valor de *type* contiene las funciones específicas para trabajar con valores de un

tipo dado. Diseñamos *type* para incluir punteros a las funciones relevantes de un tipo, el tamaño que ocupan sus valores en memoria, el número de parámetros de tipo si el tipo es genérico y punteros a los tipos concretos en su caso.

Veamos cada una de estas funciones y propiedades.

El atributo *name* guarda el nombre del tipo. Dos tipos son iguales si tienen el mismo nombre.

La función *equals* toma dos valores del tipo y decide si son iguales o no. El último parámetro es el tipo al que nos estamos refiriendo.

```
bool (*equals(const void * e1,const void * e2,struct gtp * t);
```

La función *tostring* convierte un valor del tipo en una cadena de caracteres que ubica en la memoria ya reservada *mem* y devuelve un puntero a esta memoria.

```
char * (*tostring)(const void * e, char * mem,struct gtp * t);
```

La función *order* define el orden natural del tipo si lo tiene.

```
int (*order)(const void * e1, const void * e2,struct gtp * t);
```

La función *parse* toma una cadena de texto en text y construye un valor que deja ubicado en la memoria out y devuelve un puntero al valor construido.

```
void * (*parse)(void * out, char * text, struct gtp * t);
```

La función *free* libera la memoria usada por un valor del tipo ubicado en el puntero e.

```
void (*free)(void * e, struct gtp * t);
```

La función *copy_new* crea un nuevo valor del tipo como copia el ubicado en el puntero e. Crea la memoria dinámica necesaria y la guarda en el montón de memoria h.

```
void * (*copy_new)(void * in, heap * h, struct gtp * t);
```

La función *copy* copia el valor ubicado en in a out. Asume que en out hay ya disponible la memoria necesaria.

```
void (*copy)(void * out, void * in, struct gtp * t);
```

El valor devuelto por *size* da la memoria que se necesita para ubicar un valor del tipo.

El valor *num_types* es el número de parámetros de tipo que tiene el tipo si es genérico. En principio se ha previsto la posibilidad de que sean 0, 1, 2. Instanciar el tipo genérico es actualizar los punteros *types[i]* por tipos concretos.

Para instanciar los tipos genéricos, consultar sus propiedades y copiar las propiedades del tipo usaremos las funciones:

```
type * type_copy(type * t, heap * hp);
bool type_equals(const type * e1, const type* e2);
char * type_tostring(type * t, char * mem);
type generic_type_1(type * base, type * key_type);
type generic_type_2(type *base, type * key_type,
       type * value_type);
int num_types(type * t);
type * types(type * base, int i);
void type_free(type * t);
```

La función *type_copy* copia la información de un tipo contenida en *t* creando la memoria necearia que guarda en el montón hp. La función *type_equals* decide si dos tipos, genéricos o no, son iguales. La función *type_tostring* convierte a cadena de caracteres la información de un tipo.

El diseño de las tres funciones anteriores es similar. En concreto el cuerpo de *type_copy* es:

```
type* type_copy(type *t, heap *hp) {
	type * r;
	switch (t->num_types) {
	case 0: r = heap_copy(t,hp,sizeof(type));break;
	case 1:{
		type * r0 =
			heap_copy(t->types[0],hp,sizeof(type));
		r = heap_copy(t,hp,sizeof(type));
		r->types[0] = r0;
		break;
	}
	case 2:{
		type * r0 =
			heap_copy(t->types[0],hp,sizeof(type));
		type * r1 =
			heap_copy(t->types[1],hp,sizeof(type));
		r = heap_copy(t,hp,sizeof(type));
		r->types[0] = r0;
		r->types[1] = r1;
		break;
	}
	}
	return r;
}
```

El tipo *heap* y la función *heap_copy* que nos permiten obtener memoria dinámica los veremos más adelante en la sección de tipo para gestionar la memoria dinámica.

La función *generic_type_1* instancia el tipo genérico base sustituyendo el parámetro del tipo genérico base por *key_type*.

```
type generic_type_1(type * base, type * key_type);
```

La función *generic_type_2* instancia el tipo genérico base sustituyendo los parámetros del tipo genérico base por *key_type* y *value_type*.

```
type generic_type_2(type *base, type * key_type,
	type * value_type);
```

El cuerpo de esta fución es:

```
type generic_type_2(type *base, type * key_type,
              type * value_type) {
       type rb = *base;
       rb.types[0] = key_type;
       rb.types[1] = value_type;
       return rb;
}
```

Las funciones *num_types, types* y *type_free,* respectivamente, nos dan el número de parámetros del tipo, el tipo que ocupa el parámetro i del tipo genérico y la última es capaz de liberar la memoria usada en un valor de tipo *type*.

Además, diseñamos un conjunto de funciones que serán adecuadas para todos los tipos:

```
unsigned long int hash_code(void * in, type t);
void * copy(void * out, void * in, int size);
void * copy_and_mem(void * in, int size);
void * swap(void * out, void * in, int size);
```

La función *hash_code* asigna un *hashcode* al dato, del tipo especificado, proporcionado como entrada, *copy* hace una copia de una zona de memoria de tamaño size a otra, *copy_and_mem* copia un dato de entrada de tamaño size en una memoria dinámica obtenida expresamente, *swap* intercambia los contenidos de dos bloques de memoria de tamaño size.

Tipos

Para cada tipo que necesitemos diseñaremos un *struct* con las propiedades básicas de los objetos del tipo, un conjunto de funciones para operar los valores del tipo y una variable de tipo *type* para acceder a las propiedades del tipo en tiempo de ejecución. El esquema que seguiremos para diseñar tipos nuevos tomará como punto de partida los interfaces y clases disponibles en un programa Java.

Por cada *interface* diseñamos un fichero con el mismo nombre y extensión *.h*. La definición de tipos, constantes globales y prototipos de funciones se hará en este fichero.

Por cada *clase* diseñamos un fichero con el mismo nombre y extensión .c. El código irá en este fichero. Como en C no existe sobrecarga de métodos necesitamos dar nombres de una forma sistemática a las diferentes funciones del programa en C a partir de los métodos de una clase Java. Los nombres de las funciones de C deben ser únicos dentro de un programa por lo tanto seguimos el siguiente patrón: si en Java tenemos un método individual *m* del tipo *A* y de signatura *T m(T1 a, T2 b)* entonces en C diseñamos la función *T a_m(A this, T1a, T2 b).*

Si hay varios métodos sobrecargados con nombre *m* en el tipo A las correspondientes funciones en C necesitarán, además, un sufijo para poder distinguir unas de otras. Además, necesitamos un parámetro adicional a los ya disponibles en el método Java. Es un parámetro que haga las veces del *this* de Java. Este parámetro lo colocaremos en primer lugar en muchos casos.

A partir de los detalles de implementación del tipo que aparecen en la clase Java (atributos privados) definiremos un *struct* de C con un campo por cada atributo privado con el mismo nombre y tipo. Este tipo *struct* nos servirá para definir el estado del objeto. A este struct le daremos el nombre de tipo original en Java.

Implementamos las funciones equals, order, tostring y parse específicas para el tipo. Como en Java podemos implementar constructores, métodos, equals, …

Si los tipos son dinámicos diseñamos métodos de factoría para obtener la memoria, crear e inicializar las variables y métodos free para liberar la memoria

Los objetos en *Java* son equivalentes al tipo puntero a *struct* definido antes.

Para hacer la mínima cantidad de gestión de memoria dinámica dentro de las funciones que diseñemos usaremos los siguientes patrones para diferentes números de parámetros.

Si hay un parámetro de entrada e1 y otro de salida e2.

```
e2 = exp1(e1);
-----
t2 * f(t2 * e2, const t1 *e1){
       *e2 = exp(*e1);
       return e2;
}
```

Asumimos que el puntero e2 señala a una memoria ya disponible.

Un parámetro de entrada e1 y otro de entrada-salida e2.

```
e2 = exp2(e1,e2);
-----
t2 * f(t2 * e2, const t1 *e1){
       *e2 = exp(*e2,*e1);
       return e2;
}
```

Un parámetro de entrada e1, otro de entrada-salida e2 y una variable e3 que aporta información del contexto.

```
e2 = exp2(e1,e2,e3);
------
t3 e3 = …;
t2 * f(t2 * e2, const t1 *e1){
       *e2 = exp(*e1,*e2,e3);
       return e2;
}
```

En los tres casos el tipo t2 se ha supuesto inmutable. Si fuera mutable existirían correspondencias similares.

En el repositorio se pueden encontrar el diseño e implementación de las funciones anteriores y las específicas para los tipos: *int, long, float, double, int_pair, punto, pair, pair_enumerate, optional, time, pchar, string* y otros. Para algunos de estos tipos se han implementado funciones adicionales para un manejo más cómodo. Todas ellas están declaradas en los ficheros *types.h* y *dates.h.* Veamos como ejemplo el tipo entero.

Tipos numéricos

Los tipos numericos incluyen *int, long, long_long, float* y *double*. Todos ellos tienen caracteristicas similares.

```
int * int_parse(int * out, char * text, type * type);
char * int_tostring(const int * e, char * mem, type * type);
bool int_equals(const int * e1, const int * e2, type * type);
int int_naturalorder(const int * e1,const int * e2,
      type * type);
```

El tipo entero ya viene implementado en C, pero debemos diseñar una variable, *int_type*, de tipo type para acceder a las propiedades del tipo en tiempo de ejecución. Para implementar cada una de las funciones anteriores usamos las librerías proporcionadas por C.

En la función *int_parse* usamos la función *sscanf* de la librería de C.

```
int * int_parse(int * out, char * text, type * type){
      sscanf(text,"%d",out);
      return out;
}
```

Para implementar int_tostring usamos la función de librería *sprintf*.

```
char * int_tostring(const int * e, char * mem, type * type){
    sprintf(mem,"%d",*e);
    return mem;
}
```

La función equals decide si dos enteros son iguales

```
bool int_equals(const int * e1, const int * e2, type * type){
    return *e1 == *e2;
}
```

El orden natural devuelve -1, 0,1 según que el el primer entero sea menor, igual o mayor que el segundo.

```
int int_naturalorder(const int * e1,const int * e2,type *
      type){
    int a = *e1;
    int b = *e2;
    int r;
    if(a==b) r = 0;
    else if(a<b) r = -1;
    else r = +1;
    return r;
}
```

Con estas implementaciones definimos el tipo entero como:

```
type int_type = {"int",int_equals,int_tostring,
      int_naturalorder,int_parse,
      free_0,copy_new_0,copy_0,sizeof(int),0,NULL};
```

Los restantes tipo numéricos tienen diseños similares y pueden verse en el repositorio.

Tipo fecha

La gestión del calendario, incluidas horas, minutos y segundos la haremos con los tipos *date*, *time y time_t*. El primero para gestionar fechas, el segundo horas y minutos y el tercero fechas y horas. Para su diseño partimos del tipo *time_t* y *struct tm* de las librerias de C.

El tipo *struct tm* de C incorpora la siguiente información:

```
struct tm {
      int tm_sec;   /* segundos,  rango 0 to 59  */
      int tm_min;   /* minutos, rango 0 to 59    */
      int tm_hour;  /* horas, rango 0 to 23   */
      int tm_mday;  /* dia del mes, rango 1 to 31  */
      int tm_mon;   /* mes, rango 0 to 11   */
      int tm_year;  /* el número del año desde 1900   */
      int tm_wday;  /* dia de la semana, rango 0 to 6  */
      int tm_yday;  /* día en el año, rango 0 to 365  */
      int tm_isdst; /* si es horario de verano o  no */
};
```

El tipo *time_t* es una representación compacta de la hora y fecha. En la mayoría de las implementaciones *time_t* es un entero (de 32 o 64 bits), que representa el número de segundos desde el inicio del tiempo: la medianoche del 1 de enero de 1970. Para pasar de *time_t* a *struct tm* y viceversa disponemos de las siguientes funciones en C:

```
struct tm *localtime(const time_t *timer);
time_t mktime(struct tm *timeptr);
```

Con los datos anteriores definimos dos tipos: *date* y *time*. El tipo *date* incluye fechas. El tipo *time* incluye fechas, horas, minutos y segundos. Ambos se representan internamente por el tipo *time_t*.

Las funciones para llevar a cabo el parse de valores de estos tipos o decidir la igualdad de dos valores de esos tipos son:

```
time_t * date_parse(time_t * out, char * text, type * t);
char * date_tostring(const void * p, char * mem, type * t);
bool time_equals(const void * p1, const void * p2, type * t);
int time_naturalorder(const void * t1,const void *t2,type *t);
```

Igualmente disponemos de funciones que definen su orden natural y la conversión de valores del tipo en cadenas de caracteres.

Los variables *date_type* y *time_type* que definen los tipos son:

```
type date_type = {"date",time_equals,date_tostring,
       time_naturalorder,date_parse,
       free_0,copy_new_0,copy_0,sizeof(time_t),0,NULL};
type time_type = {"time",time_equals,time_tostring,
       time_naturalorder,time_parse,
       free_0,copy_new_0,copy_0,sizeof(time_t),0,NULL};
```


Veamos la implementación de algunas de ellas. La implementación del orden natural usa la función *difftime* de C que da la diferencia en segundos de dos fechas.

```
int time_naturalorder(const void * p1,const  void * p2,
              type * t){
       time_t t1 = *(time_t *) p1;
       time_t t2 = *(time_t *) p2;
       double d = difftime(t2,t1);
       int r;
       if(d>0) r = -1;
       else if(d<0) r= +1;
       else r= 0;
       return r;
}
```

La implementación de *date_tostring* es de la forma:

```
char * date_tostring(const void * p, char * mem){
       time_t * t = (time_t *) p;
       struct tm * r = localtime(t);
       sprintf(mem,"%02d-%02d-%4d",r->tm_mday,r->tm_mon+1,
              r->tm_year+1900);
       return mem;
}
```

Adicionalmente diseñamos las siguientes funciones de factoría:

```
time_t time_now();
time_t time_of_date(int day, int month, int year);
time_t time_of(int day, int month, int year,
       int hour, int minute, int second);
```

Otras funciones para añadir o restar días, meses o años.

```
time_t time_add_days(time_t date, int days);
time_t time_add_months(time_t date, int months);
time_t time_add_years(time_t date, int years);
time_t time_minus_days(time_t date, int days);
time_t time_minus_months(time_t date, int months);
time_t time_minus_years(time_t date, int years);
```

Y por último la diferencia en horas minutos o segundos entre dos fechas.

```
double time_diff_seconds(time_t start, time_t end);
double time_diff_minutes(time_t start, time_t end);
double time_diff_hours(time_t start, time_t end);
```

Las implementaciones se pueden encontrar en el repositorio.

Cadenas de caracteres

Las cadenas de caracteres tienen un gran interés en todos los lenguajes de programación. Por ello se han diseñado dos tipos específicos: *string_fix* (cadenas de caracteres de tamaño fijo en memoria) y *string_var* (cadenas de caracteres de tamaño variable en memoria).

El tipo *string_fix* es, en definitiva, un bloque de memoria de un tamaño especificado y un puntero a la misma, *char **, con la restricción adicional que la cadena de caracteres se extiende desde el comienzo de la memoria hasta el primer carácter *'\0'*.

La declaración de variables de este tipo puede hacerse con *string_fix* o mediante un array de caracteres para reservar la memoria necesaria.

```
#define Tam_String 256
typedef char string_fix[Tam_String];
```

Las referencias a variables de este tipo son punteros a char (char *).

Definimos las funciones siguientes para obtener un valor del tipo desde una cadena, convertir un valor en una cadena, etc.

```
char * string_fix_parse(char * out, char * text, type * type);
char * string_fix_tostring(const char * e, char * mem,
       type * type);
bool string_fix_equals(const char * e1, const char * e2,
       type * type);
int string_fix_naturalorder(const char * e1,const char * e2,
       type * type);
```

Y la declaración del tipo:

```
type string_fix_type = {"string_fix",string_fix_equals,
       string_fix_tostring,string_fix_naturalorder,
       string_fix_parse,free_0,copy_new_0,copy_0,
       Tam_String,0,NULL};
```

Podemos definir tipos *string_fix* con otra cantidad de memoria reservada.

```
type string_fix_type_of_tam(int numchars){
       type r = string_fix_type;
       r.size = numchars;
       return r;
}
```

El tipo dispone de otras funciones de utilidad:

```
char char_get(const char * in, int i);
char char_set(char * in_out, int i, char c);
int string_fix_size(const char * string);
char * string_fix_substring(char * out, char * in, int from,
       int to);
char * string_fix_concat(char * out,const char * in,type *t);
char * string_fix_copy(char * out,const char * in,type *t);
char * string_fix_remove_eol(char * out, char * in);
```

Junto con el tipo anterior disponemos una cadena de caracteres de tamaño de memoria variable. Es el tipo *string_var.* Las variables de este tipo tienen asignada al principio una determinada cantidad de memoria dinámica pero al añadir nuevos caracteres se amplia esta memoria si es necesario.

Este tipo dispone de las funciones:

```
string_var string_var_empty();
bool string_var_is_empty(string_var * in);
int string_var_size(string_var * in);
char * string_var_data(string_var * in);
```

Y las funciones asociadas al tipo

```
string_var * string_var_parse(string_var * out, char * text,
       type * type);
char * string_var_tostring(const string_var * e, char * mem,
       type * type);
bool string_var_equals(const string_var * e1,
       const string_var * e2, type * type);
int string_var_naturalorder(const string_var * e1,
       const  string_var * e2, type * type);
 (string_var),0,NULL};
```

La implementación de este tipo es de la forma:

```
typedef struct {
       int tam;
       int size;
       char * data;
}string_var;
```

La información asociada al tipo se encuentra en la variable *string_var_type.*

```
type string_var_type = {"string_var",string_var_equals,
       string_var_tostring,string_var_naturalorder,
       string_var_parse,string_var_free,string_var_copy_new,
       string_var_copy,sizeof
```

La utilidad de esta implementación puede verse al implementar las funciones siguientes. A diferencia de las funciones similares para el tipo *string_fix,* ahora hay que buscar memoria adicional cuando sea necesario.

```
void * string_var_add_string_fix(string_var * out,
       const char * in_char);
void * string_var_add_string_var(string_var * out,
       const string_var * in);
```

La implementación de la primera función es

```
void * string_var_add_string_fix(string_var * out,
              const char * s) {
       check_not_null(s,__FILE__,__LINE__,"Cadena null");
       int n = strlen(s);
       if(out->size+n>out->tam){
              int tm = MAX(out->size+n,out->tam+INC_TAM);
              out->data = realloc(out->data,tm);
              check_not_null(out->data,__FILE__,__LINE__,
                  "No ha sido posible aumentar la memoria");
              out->tam = tm;
       }
       if(out->size == 0) strcpy(out->data,s);
       else strcat(out->data,s);
       out->size = out->size +n;
       return out->data;
}
```

Pares de valores

Muy frecuentemente usamos pares de valores. Aquí hemos diseñado pares de valores numéricos: pair_int, pair_long, pair_long. Todos tienen un diseño similar, veamos solamente algunos detalles de pair_long. El resto de los detalles se pueden ver en el repositorio.

La implementación del tipo es:

```
typedef struct{
       long a;
       long b;
}pair_long;
```

Y la variable que contiene las propiedades del tipo es:

```
type pair_long_type = {"pair_long",pair_long_equals,
       pair_long_tostring,pair_long_naturalorder,
       pair_long_parse,free_0,
       copy_new_0,copy_0,
       sizeof(pair_long),0,NULL};
```

Veamos como ejemplo la implementación de su orden natural considerando que este viene definido por el orden de la primera componente y si las primeras componentes de los dos valores son iguales entonces por la segunda.

```
int pair_long_naturalorder(const pair_long * p1,
              const pair_long * p2){
       pair_long np1 = *p1;
       pair_long np2 = *p2;
       int r  = long_naturalorder(&np1.a,&np2.a);
       if(r==0) r  = long_naturalorder(&np1.b,&np2.b);
       return r;
}
```

El tipo punto en el plano 2D podemos diseñarlo como sinónimo del *pair_double* con algunas funciones auxiliares.

Memoria dinámica

Para la gestión de la memoria dinámica, se ha diseñado el tipo *heap* que se encuentra declarado de *heap.h.*

Este tipo nos permite reserva memoria dinámica de forma que podamos liberarla toda a la vez.

```
heap heap_empty();
void * heap_copy(void * source, heap * heap, int size);
void * heap_get_memory(heap * heap,int size);
int heap_size(heap * heap);
void heap_free(heap * heap);
void heap_clear(heap * heap);
```

Es conveniente usar este tipo para tratar la gestión de la memoria dinámica de forma ordenada. Es decir, nos permite obtener bloques de memoria dinámica con la posibilidad de liberarar todos juntos. Incluye las funciones *heap_copy* y *heap_get_memory*. La primera copia el contenido de tamaño *size* en una nueva ubicación de memoria obtenida dinámicamente y mantiene la referencia en el montón para después poder ser liberado. La segunda obtiene un bloque de memoria de tamaño *size* y mantiene la referencia en el montón.

La implementación es de la forma:

```
typedef struct {
       int size;
       int tam;
       void ** elements;
       int size_memory;
} heap;
```

Como vemos, dispone de un puntero a un array de punteros a void que señalarán los diferentes bloques de memoria obtenidos, la memoria total obtenida y el número de bloques.

La implementación puede verse en el repositorio indicado a principio del libro.

Precondiciones

Para trabajar con las excepciones se ha diseñado el conjunto de funciones que pueden encontrarse en el fichero *preconditions.h* e incluye las funciones:

```
void check_argument(bool condition, char * file, int line,
       char * message);
void * check_not_null(void * reference, char * file, int line,
       char * message);
int check_element_index(int index, int size, char * file,
       int line);
int check_position_index(int index, int size, char * file,
       int line);
```

Con las funciones anteriores podemos generar errores con mensajes que incluyen el nombre del fichero y el número de línea.

Tipos genéricos

Los tipos genéricos son tipos con parámetros que son otros tipos. Un tipo genérico se instancia cuando sus parámetros son sustituidos por tipos concretos.

Incluimos aquí los tipos genéricos pair, enumerate y optional.

Veamos el diseño del tipo genérico *Pair<A,B>* que en Java implementariamos como:

```
record Pair<A, B>(A first,B second) {
	public static <A,B>  Pair<A,B> of(A a, B b){
		return new Pair<>(a,b);
	}
	@Override
	public String toString() {
		return String.format("(%s,%s)",first,second);
	}
}
```

La implementación de este tipo en C sería:

```
typedef struct {
	void * key;
	void * value;
} pair;

pair pair_of(void * key, void * value);
```

Y las funciones asociadas al tipo:

```
pair * pair_parse(pair * out, char * text, type * t);
char * pair_tostring(const pair * e, char * mem, type * t);
bool pair_equals(const pair * e1, const pair * e2, type * t);
int pair_naturalorder(const pair * e1, const pair * e2,
	type * t);
```

Y la variable que contiene la información del tipo en tiempo de ejecución.

```
type pair_type = {"pair",pair_equals,pair_tostring,
	pair_naturalorder,pair_parse,pair_free,
	pair_copy_new,pair_copy,sizeof(pair),2,NULL};
```

Como vemos, indicamos que el tipo tiene dos parámetros. Las implementaciones de las funciones *pair_naturalorder* y *pair_equals* son:

```
bool pair_equals(const pair * e1, const pair * e2, type * t){
	return equals(e1->key,e2->key,t->types[0]) &&
		equals(e1->value,e2->value, t->types[1]);
}
```

```
int pair_naturalorder(const pair * e1, const pair * e2,
       type * t){
       int r = order(e1->key,e2->key,t->types[0]);
       if(r==0) r = order(e1->value,e2->value,t->types[1]);
       return r;
}
```

Más interesante es la implementación de la fución *pair_parse*.

```
pair * pair_parse(pair * out, char * text, type * t){
       char * tokens[20];
       string_fix tt;
       string_fix_copy(tt,text,&string_fix_type);
       int n = split_text(tt,pair_delimiters, tokens);
       check_argument(n==2,__FILE__,__LINE__,
              "el numero de tokens no es 2");
       if(heap_isnull(&pair_heap)) pair_heap = heap_empty();
       out->key = heap_get_memory(&pair_heap,
              t->types[0]->size);
       out->value = heap_get_memory(&pair_heap,
              t->types[1]->size);
       parse(out->key,tokens[0],t->types[0]);
       parse(out->value,tokens[1],t->types[1]);
       return out;
}
```

O la implementación de *pair_copy_new*

```
pair * pair_copy_new(pair * p, heap * hp, type * t) {
       type * t0 = t->types[0];
       type * t1 = t->types[1];
       void * key = p!=NULL?p->key:NULL;
       void * value = p!=NULL?p->value:NULL;
       void * k = t0->copy_new(key,hp,t0);
       void * v = t1->copy_new(value,hp,t1);
       pair pc = pair_of(k,v);
       return (pair *) heap_copy(&pc,hp,sizeof(pair));
}
```

En el caso de *pair_parse* para cada componente del par se obtiene la memoria adecuada según el tamaño del tipo concreto y se hace el parsing usando la función correspondiente de ese tipo. La instanciación de un tipo

genérico se hace con las funciones *generic_type_1* y *generic_type_2* que construyen una variable con la información del tipo concreto.

En el caso de *pair_copy_new* se hace una nueva copia de cada componente del par usando la función *copy_new* y luego del propio par.

Veamos la forma de instanciar el tipo *pair* con la primera componente un doble y la segunda un *long*.

```
string_fix text = "(0,1)";
type t = generic_type_2(&pair_type, &double_type, &long_type);
parse(&p, "(34,56)", &t);
printf("2: %s\n", tostring(&p, mem, &t));
```

La implementación de la función *generic_type_2* es:

```
type generic_type_2(type *base, type * key_type,
              type * value_type) {
       type rb = *base;
       rb.types[0] = key_type;
       rb.types[1] = value_type;
       return rb;
}
```

Tuplas: coordenadas2d y coordenadas3d

Las tuplas se usan en todos los lenguajes de programación. Veamos cómo diseñar el tipo *coordenadas_2d* con propiedades latitud y longitud en grados, y la posibilidad de calcular la distancia entre dos coordenadas según la fórmula de *Haversine*.

La implementación del tipo, funciones de factoría y transformación a radianes.

```
typedef struct {
       double latitud;
       double longitud;
}coordenadas_2d;
coordenadas_2d coordenadas_2d_of(double latitud,
       double longitud);
coordenadas_2d coordenadas_2d_to_radians(coordenadas_2d c);
```

Otras funciones del tipo podrían ser:

```
double coordenadas_2d_distance(coordenadas_2d c1,
       coordenadas_2d c2);
bool coordenadas_2d_cercanas(coordenadas_2d c1,
       coordenadas_2d c2, double d);
char * coordenadas_2d_tostring(coordenadas_2d * c1,
       const char * mem);
```

Algunas implementaciones. El paso de las coordenadas en grados a radianes.

```
coordenadas_2d coordenadas_2d_to_radians(coordenadas_2d c){
       double lat = c.latitud * M_PI / 180.0;
       double lon = c.longitud * M_PI / 180.0;
       coordenadas_2d r = {lat, lon};
       return r;
}
```

La conversión a cadena de caracteres teniendo en cuenta que tanto la latitud como la longitud son de tipo double.

```
char * coordenadas_2d_tostring(coordenadas_2d * c,
              const char * mem) {
       sprintf(mem,"(%lf,%lf)",c->latitud,c->longitud);
       return mem;
}
```

La distancia a lo largo de la superficie de la tierra según la fórmula de Harvesine:

$$a = sin^2(\Delta\varphi/2) + cos\ \varphi_1 * cos\ \varphi_2 * sin^2(\Delta\lambda/2)$$

$$d = 2\ R\ atan2(a, \sqrt{1-a})$$

Donde φ_1, φ_2 son las latitudes de los puntos, , $\Delta\varphi$ la diferencia de latitudes y $\Delta\lambda$ la diferencia de longitudes.

```
double coordenadas_2d_distance(coordenadas_2d c1,
       coordenadas_2d c2){
       double radio_tierra = 6373.0;
       coordenadas_2d c1R = coordenadas_2d_to_radians(c1);
       coordenadas_2d c2R = coordenadas_2d_to_radians(c2);
       double incLat  = c2R.latitud-c1R.latitud;
       double incLong = c2R.longitud-c1R.longitud;
       double a = pow(sin(incLat/2),2)+
              cos(c1R.latitud)*cos(c2R.latitud)*
              pow(sin(incLong/2),2);
       double c = 2 * atan2(sqrt(a),sqrt(1 - a));
       return radio_tierra*c;
}
```

El tipo *coordenadas_3d*

El tipo *coordenadas_3d* con propiedades latitud y longitud en grados y altura en kilómetros y la posibilidad de calcular la distancia entre dos coordenadas.

```
typedef struct {
       double latitude;
       double longitude;
       double altitude;
}coordenadas_3d;

coordenadas_3d coordenadas_3d_of(double latitude, double
longitude, double altitude);
double coordenadas_3d_distance(coordenadas_3d c1,
coordenadas_3d c2);
char * coordenadas_3d_tostring(coordenadas_3d * c1,
       const char * mem);
```

```
coordenadas_3d coordenadas_3d_of(double latitude,
       double longitude, double altitud){
       coordenadas_3d c = {latitude, longitude, altitud};
       return c;
}
```

```
double coordenadas_3d_distance(coordenadas_3d c1,
       coordenadas_3d c2){
       coordenadas_2d c12D =
              coordenadas_2d_of(c1.latitud,c1.longitud);
       coordenadas_2d c22D =
              coordenadas_2d_of(c2.latitud,    c2.longitud);
       return sqrt(pow(coordenadas_2d_distance(c12D,c22D),2)+
              pow(c1.altitud-c1.altitud,2));
}
```

```
char * coordenadas_3d_tostring(coordenadas_3d * c,
       const char * mem){
       sprintf(mem,"(%lf,%lf,%lf)",c->latitud,c->longitud,
              c->altitud);
       return mem;
}
```

Se deja como ejercicio diseñar el *orden natural.*

Listas

El lenguaje C no proporciona listas ya definidas. Lo usual es trabajar con arrays y punteros. Pero esto es una fuente continua de problemas. En primer lugar, porque los arrays son de tamaño fijo y no tienen asociado un tamaño que no pueda informar cuanta memoria ocupa y cuantos elementos tienen.

Diseñamos por esta razón un tipo lista (*list*) que nos permita tratar de forma homogénea un array de cualquier tipo y que, además, sea de tamaño variable. Dotamos a la lista de su propia gestión de la memoria dinámica que necesite.

El tipo tiene la implementación:

```
typedef struct {
       bool is_view;
       type * type;
       int size;
       int tam;
       void ** elements;
} list;
```

El cometido de las distintas propiedades es el siguiente:

- *is_view*: Indica si la lista es una vista de otra. Una sublista será una vista de otra lista.
- *type*: Un puntero al tipo de los elementos de las casillas
- *size*: El número de elementos de la lista. La lista vacía tiene *size = 0*.
- *tam*: El número máximo de elementos que puede contener la lista. Si *size = tam* e intentamos añadir un elemento aumentará automáticamente *tam*.
- *elements*: Un array de elementos de tipo *void ** que serán punteros a la memoria reservada para los elementos añadidos a la lista. Tiene que ser de tipo *void *** al ser un array de elementos de tipo *void **.

Junto con la implementación dotamos al tipo de las siguientes operaciones con las que incluimos una pequeña explicación de su funcionalidad (hay más operaciones disponibles en el repositorio):

- *list list_empty(type * t*): Crea una lista vacía con elementos de tipo t.
- *list list_of(void * data, int size, type t):* Crea una lista a partir de una array (data) de size elementos de tipo t.
- *list * list_parse(list * out, char * text):* Parsing de una lista a partir de text y con unos delimitadores y separadores fijados.
- *list list_sublist(list * ls, int a, int b):* Crea una vista de la lista *ls* en el intervalo a, b.
- *void * list_get(list * ls, const int index):* Obtiene el elemento index de la lista.
- *int list_size(list * ls):* Número de elementos de la lista.
- *void list_add(list * ls, void * e):* Añade e al final de la lista.
- *void list_add_left(list * ls, void * e):* Añade e al principio de la lista.
- *char * list_get_string(list * ls, const int index, char * mem):* Obtiene el elemento *index* de la lista en su representación como cadena de caracteres.

Otras funciones necesarias son:

- *list list_copy(list * in):* Hace un copia de la lista.

- *list list_of_string_of_file(char * file):* Obtiene una lista formada con las líneas del fichero.
- *list list_filter(list * ls, bool (*predicate)(void * e)):* Obtiene una lista filtrada con el predicado.
- *list list_map(list * ls, type * t, void * (*f)(void * out, void * in)):* Obtiene una lista tansformada con la función.
- *char * list_tostring(list * ls, char * mem):* La representación textual de la lista.
- *void write_list_to_file(char * file, list * list, char * tostring(const void * source, char * mem)):* Escribe la lista en el fichero transformando cada elemento según tostring.
- *list * list_add_list(list * ls, list * add_list):* Añade a la lista los elementos de la lista.
- *void list_clear(list * ls):* Elimina los elementos de la lista dejándola vacía.
- *void list_free(list * list):* Elimina la memoria dinámica usada por la lista.

La implementación de estas funciones se puede encontrar en el repositorio. Solo destacaremos la función *list_add*. Comencemos con *list_add_pointer*.

```
void list_add_pointer(list * list, void * element) {
	check_argument(!list->is_view,__FILE__,__LINE__,
		"no se puede modificar una vista");
	list_grow(list);
	list->elements[list->size] = element;
	list->size = list->size + 1;
}
```

Podemos ver que se comprueba si la lista es una vista para impedir su modificación. Posteriormente se llama al método list_grow para aumentar el tamaño de la lista si fuera necesario y por último se añade el puntero a al array de punteros de la lista y se aumenta el tamaño en 1.

La función *list_grow* intenta reubicar la lista de punteros en un tamaño doble del anterior. Esto se hace con la función *realloc* de C. Hay que

comprobar si lo hemos conseguido verificando que el puntero obtenido no es NULL.

```
#define tam_default 20

void list_grow(list * list) {
	if (list->size == list->tam) {
		list->tam = 2 * list->tam;
		list->elements = realloc(list->elements,
			list->tam *sizeof(void *));
		check_not_null(list->elements,__FILE__,__LINE__,
			"Problemas de memoria en el
			creciminento de la lista");
	}
}
```

Ahora ya podemos implementar la función *list_add* usando las anteriores. Primero copiamos el dato que queremos añadir a la lista en el montón de memoria de la propia lista. El puntero obtenido lo añadimos a la lista.

```
void list_add(list * ls, void * element){
	check_argument(!ls->is_view,__FILE__,__LINE__,
		"no se puede modificar una vista");
	void * e = copy_new(element,NULL,ls->type);
	list_add_pointer(ls,e);
}
```

Algoritmos sobre listas

Las listas son tipos de datos muy adecuados para implementar algoritmos de diversos tipos, con objeto de conseguir una alta eficiencia cuando los datos se pueden cargar en memoria.

Algoritmos de ordenación

Se proporcionan las siguientes operaciones de utilidad sobre listas:

- *int_pair bh(list * ls, void * p, int i, int j, int (*order)(const void * e1, const void * e2)):* Algoritmo de la bandera holandesa aplicado al segmento i, j de la lista ls con pivote p y criterio de ordenación *order*.

- *void list_sort(list * ls, int (* order)(const void * e1, const void * e2)):* Algoritmo de quicksort aplicado a la lista.
- *void basic_sort(list * ls, int (*order)(const void * e1, const void * e2)):* Algoritmo de la burbuja para ordenar la lista.
- *list merge_list(list * ls1, list * ls2, int (* order)(const void * e1, const void * e2)):* Fusión de las dos lista ordenadas para producir otra ordenada.
- *int bs(list * ls, void * e, int (* order)(const void * e1, const void * e2)):* Búsqueda binaria de la posición de e en la lista ordenada ls o -1 si no está.
- *void list_merge_sort(list * ls, int (*order)(const void * e1, const void * e2)):* Algoritmo de mergesort aplicado a la lista.
- *void * k_esimo(list * ls,int k, int (*order)(const void * e1, const void * e2)):* Algoritmo del k-esimo sobre una lista.

En el repositorio se ofrecen versiones de las funciones que usan el orden natural de los elementos de la lista.

Algoritmo de la *Bandera Holandesa*

Veamos la implementación de algunas de estas funciones. En primer lugar, el algoritmo de la *Bandera Holandesa*. Este algoritmo reordena la lista tomando un elemento *p* que lo llamamos el pivote. La ordenación se produce en tres partes: la primera, los elementos que son menores que el pivote, la segunda los que son iguales al pivote y la tercera los que son mayores. El algoritmo devuelve un par de enteros (*a*,*b*) que delimitan las tres partes.

La explicación con más detalle del algoritmo puede verse en el libro *Análisis y Diseño de Algoritmos y Tipos de Datos*, que aparecerá publicado en esta misma colección.

```
pair_int bh(list * ls, void * pivot, int i, int j,
              int (*order)(const void * e1, const void * e2)) {
       int a =i, b=i, c=j;
       while (c - b > 0) {
              void * elem = list_get(ls,b);
              int r = order(elem, pivot);
              if (r < 0) {
                     swap_in_list(ls,a,b);
                     a++;
                     b++;
              } else if (r > 0) {
                     swap_in_list(ls,b,c-1);
                     c--;
              } else {
                     b++;
              }
       }
       pair_int r = pair_int_of(a, b);
       return r;
}
```

Se usa la función *swap_in_list* para intercambiar los valores de dos casillas.

La ordenación por el método de la burbuja de un segmento de lista es:

```
void basic_sort_g(list * ls, int inf, int sup,
              int (*order)(const void * e1, const void * e2)) {
       for (int i = inf; i < sup; i++) {
          for (int j = i + 1; j < sup; j++) {
              if (order(list_get(ls, i),list_get(ls, j)) > 0) {
                           swap_in_list(ls,i,j);
              }
          }
       }
}
```

Y la función para la lista completa:

```
void basic_sort(list * ls,
              int (*order)(const void * e1, const void * e2)){
       return basic_sort_g(ls,0,ls->size,order);
}
```

El algoritmo de quicksort usa el de la bandera holandesa y la ordenación por burbuja. La versión generalizada ordena un segmento de lista.

```
void list_quick_sort_g(list * ls, int i, int j,
              int (*order)(const void * e1, const void * e2)) {
       assert(j >= i);
       if (j - i <= 4) {
              basic_sort_g(ls, i, j, order);
       } else {
              void * pivote = piv(ls,i,j);
              pair_int p = bh(ls, pivote, i, j, order);
              list_quick_sort_g(ls, i, p.a, order);
              list_quick_sort_g(ls, p.b, j, order);
       }
}
```

El algoritmo quicksort usa el algoritmo anterior generalizado.

```
void list_quick_sort(list * ls,
              int (*order)(const void * e1, const void * e2)){
       list_quick_sort_g(ls,0,ls->size,order);
}
```

Ejemplos de listas

1. *Implementar una lista de n números reales aleatorios en el intervalo (a,b)*

Lo primero es diseñar una función para obtener un número real aleatorio en un intervalo *(a,b).*

```
double double_aleatorio(double a, double b) {
       double r = ((double) rand() / (RAND_MAX));
       r = a + (b - a) * r;
       return r;
}
```

```
list random_list(int n, int a, int b) {
	list ls = list_empty(&double_type);
	for (int i = 0; i < n; i++) {
		double r = double_aleatorio(a, b);
		list_add(&ls, &r);
	}
	return ls;
}
```

2. *Dada una lista, ordenar con respecto al orden natural el subintervalo (a,b) de la lista*

```
void sort_interv(list * ls, int a, int b){
	list ls2 = list_sublist(ls,a,b);
	list_quick_sort_naturalorder(&ls2);
}
```

3. *Implementar una función para decidir si dos listas son iguales*

```
bool list_equals(const list * ls1, const list * ls2) {
	bool res = ls1->size==ls2->size;
	if(!res) return false;
	int i = 0;
	while(i<ls1->size && res) {
			void* e1 = list_get(ls1, i);
			void* e2 = list_get(ls2, i);
			res = equals(e1, e2,ls1->type);
			i++;
	}
	return res;
}
```

La función comprueba que el tamaño de ambas listas es igual y el contenido de cada casilla también.

4. *Implementar una función para obtener una lista a partir de números reales*

```
list list_of_double(int n, ...) {
	list r = list_empty(&double_type);
	va_list valist;
	va_start(valist, n);
	for (int i = 0; i < n; i++) {
		double e = va_arg(valist, double);
		list_add(&r, &e);
	}
	va_end(valist);
	return r;
}
```

La función se usaría de la forma:

```
list ls3 = list_of_double(5, 6., 7., 8., 9., 10.);
```

Para diseñar este tipo de funciones con número variable de parámetros usamos las macros de C *va_start* y *va_list*.

De forma similar, se pueden diseñar funciones para obtener listas a partir de un número variable de *int, long, string* o punteros a un tipo dado.

Tablas hash

Las tablas hash son necesarias para implementar tipos como diccionarios, conjuntos y multiconjuntos. Una tabla hash es un conjunto de pares clave valor tal que las claves no se repiten.

Las tablas hash son tipos genéricos con dos parámetros de tipo: uno para las claves y otro para los valores.

Implementación

La implementación parte del tipo entry que es una tupla *(clave, valor, entero).*

```
typedef struct {
	void * key;
	void * value;
	int next;
} entry;
```

El entero nos va a servir para enlazar tuplas de este tipo para formar listas. Una lista acaba cuado next es -1.

La tabla hash dispondrá de su propio montón de memoria para ubicar los pares que se vayan añadiendo. Además, tendrá otras propiedades: el número de claves, los tipos de las claves y los valores, etc. Una tabla hash puede ser implementada mediante el tipo *map* siguiente:

```
typedef struct {
       type * key_type;
       type * value_type;
       int * blocks;
       entry * data;
       int size;
       int capacity_blocks;
       int capacity_data;
       int first_free_data;
       float load_factor_limit;
} map;
```

Las propiedades tienen el significado siguiente:

- *key_type:* Tipo de las claves
- *value_type:* Tipo de los valores
- *blocks:* Un array de números enteros. Cada entero nos indicará el comienzo de una lista enlazada de *entrys* en el bloque data.
- *data:* Un array de *entry*. Este array está organizado como un conjunto de listas enlazadas que llamaremos bloques. El bloque *i* comienza en la casilla con índice *blocks[i]*. Si blocks[i] es -1 el bloque correspondiente está vacío.
- *size:* Número de claves. La tabla hash vacía tiene *size = 0.*
- *capacity_blocks:* El número máximo de bloques. Por razones asociadas a las funciones de hashing este número será un primo mayor o igual a 13.
- *capacity_data:* El número máximo de claves.
- *first_free_data:* Índice de la primera casilla del bloque de casillas libres en *data*.
- *load_factor_limit:* La relación *capacity_data/capacity_blocks.* Solemos establecerlo en 0.75. Hacemos que *capacity_blocks* sea un número primo. Establecido este calculamos *capacity_data* dado *load_factor_limit.* Estos valores quedan fijos hasta que *size =*

capacity_data. Cuando eso ocurre hay que ampliar *capacity_blocks* al siguiente primo y recolocar todos los elementos en los nuevos bloques (la operación *rehash*).

Se proporcionan las siguientes operaciones de utilidad sobre tablas hash:

- *map map_empty(type * key_type, type * value_type):* Tabla hash vacía con claves de tipo *key_type* y valoes de tipo *value_type*.
- *int map_size(map * table):* Número de claves
- *void * map_put(map * table, void * key, void * value):* Añadir un par clave valor.
- *void * map_remove(map * table, void * key):* Eliminar una clave.
- *void * map_get(map * table, void * key):* Valor asociado a una clave o NULL si la clave no existe.
- *bool hash_table_contains(hash_table * table, void * key):* Si la tabla contiene esa clave.

Para comprender el funcionamiento interno de la tabla hash hay que tener en cuenta que el array data está compuesto por un conjunto de listas enlazadas de *entrys*. Hay una lista que contiene las casillas libres. Esta lista comienza en el entero señalado por la variable *first_free_data.* El resto de listas comienza en los enteros señalados por los distintos bloques *blocks[i]*. Si *blocks[i]* es -1 el bloque i está vacío.

Veamos, en primer lugar, dos funciones para gestionar las listas asociadas a los bloques.

*entry * ocupa_primera_libre(map * table, int bq):* Devuelve la primera casilla libre después de ocuparla y unirla al bloque *bq*.

*void libera(map * table, int bq, int index_data):* Libera la casilla index_data, que estaba en el bloque *bq* y la une a las casillas libres.

El código de la primera función es:

```
entry * ocupa_primera_libre(map * table, int bq) {
	int first_free = table->first_free_data;
	table->first_free_data = table->data[first_free].next;
	if (table->blocks[bq] < 0) {
		table->data[first_free].next = -1;
	} else {
		table->data[first_free].next = table->blocks[bq];
	}
	table->blocks[bq] = first_free;
	return table->data+first_free;
}
```

Y el código de la segunda

```
void libera(map * table, int bq, int index_data) {
	if (table->blocks[bq] == index_data) {
		table->blocks[bq] = table->data[index_data].next;
	} else {
		int i = table->blocks[bq];
		while (table->data[i].next != index_data) {
			i = table->data[i].next;
		}
		table->data[i].next =
			table->data[index_data].next;
	}
	table->data[index_data].next = table->first_free_data;
	table->first_free_data = index_data;
}
```

Con estas funciones para gestionar las listas podemos diseñar otras que encuentren el bloque donde debe ubicarse una nueva clave y la entrada concreta dentro de un bloque si existe. Son las funciones:

*int get_index_block(map * table, void * key):* Calcula el bloque donde se encuentra la clave. Este bloque es el resto del *hash_code* de la clave con respecto al número de bloques (*capacity_blocks).*

*int find_data_entry(map * table, int index_block, void * key):* Encuentra el índice de la casilla dónde está la clave dentro del bloque *index_block* (recordemos que cada bloque es una lista enlazada) o -1 si no la encuentra. Para la búsqueda se usa la igualdad asociada al tipo de la clave.

El código de la primera función es:

```
int get_index_block(map * table, void * key) {
       unsigned long int hash_index = hash_code(key,
              table->key_type);
       int index = (int) (hash_index %
              (table->capacity_blocks));
       return index;
}
```

Y el de la segunda:

```
int find_data_entry(map * table, int bq, void * key) {
  int r = -1;
  int next = table->blocks[bq];
  while (next >= 0) {
    if (equals(key,table->data[next].key,table->key_type)){
       r = next;
       break;
    }
       next = table->data[next].next;
  }
  return r;
}
```

Como se puede observar se busca secuencialmente la clave *key* en el bloque *bq*. La primera entrada del bloque es *table->blocks[bq].* Se busca hasta que la encuentra según la igualdad definida por el tipo de la clave (*equals(key,table->data[next].key,table->key_type))*, en ese caso se hace break, o hasta que acaba la lista (cuando no se cumpla *next >=0*).

El *hash_code* de una clave se calcula a partir de su representación como cadena de caracteres:

```
unsigned long int hash_code(void * in, type * t){
       char mem[256];
       char * r = tostring(in,mem,t);
       unsigned long int hash_index = hash(r);
       return hash_index;
}
```

Ahora podemos diseñar las funciones para obtener el valor asociado a una clave. Hemos de tener en cuenta que el valor asociado a una clave está en la misma entrada.

```
void * map_get(map * table, void * key){
	int index = get_index_block(table,key);
	int pos = find_data_entry(table,index,key);
	void * r = NULL;
	if(pos >=0){
		r = table->data[pos].value;
	}
	return r;
}
```

A continuación, se muestra la función para decidir si la tabla hash contiene o no una clave.

```
bool map_contains_key(map * table, void * key) {
	int index = get_index_block(table, key);
	int pos = find_data_entry(table, index, key);
	bool r = false;
	if (pos >= 0) {
		r = true;
	}
	return r;
}
```

La función para añadir un nuevo par clave valor comprueba, en primer lugar, si se ha sobrepasado o no el factor de carga con la función *rehash*.

```
void * map_put_pointer(map * table, void * key,void * value){
	rehash(table);
	return map_put_private(table,key,value);
}
```

Cuando el factor de carga aumenta, hay que aumentar el número de bloques y recolocar los pares clave valor en los nuevos bloques. Este trabajo es llevado a cabo por la función *int rehash(hash_table * table).* El código se puede encontrar en el respositorio.

Para añadir un par clave valor primero hay que obtener memoria para la clave y el valor. Luego, se deben añadir los punteros correspondientes con la función anterior *map_put_pointer* que usa *map_put_private*.

```
void * map_put(map * table, void * key, void * value){
	void * k = copy_new(key,NULL,table->key_type);
	void * v = copy_new(value,NULL,table->value_type);
	return map_put_pointer(table,k,v);
}
```

```
void * map_put_private(map * table, void * key,
		void * value) {
	int index = get_index_block(table, key);
	int pos = find_data_entry(table, index, key);
	if (pos >= 0) {
		table->data[pos].value = value;
	} else {
		table->size = table->size + 1;
		entry * e = ocupa_primera_libre(table,index);
		e->key = key;
		e->value = value;
	}
	return key;
}
```

La función *map_put_private* es la función auxiliar que añade el par clave valor tras haberse obtenido memoria y hacer el rehash. Como puede verse, encuentra el bloque donde debería estar la clave, después busca una entrada con la clave. Si la encuentra actualiza el valor. Si no la encuentra, busca una casilla libre la ubica en el bloque correspondiente y actualiza la clave y el valor.

Conjuntos y multiconjuntos

Vemos en este capìtulo la implementación de conjuntos, multiconjuntos, sus usos y algunos tipos relacionados como ListMultimap y SetMultimap. Un ListMultimaps es un diccionario (Map) cuyos valores son listas y un SetMultimap es también un diccionario cuyos valores con conjuntos.

Implementación de conjuntos

Los conjuntos (*set*) se pueden implementar mediante tablas hash. Las claves en una tabla hash forman un conjunto. La implementación delega, por lo tanto, en una tabla hash. Declaramos entonces set como un sinónimo de map con funciones añadidas.

```
typedef map set;
```

El conjunto de operaciones disponibles es:

```
set set_empty(type type_element);
set set_of(list * ls);
void set_add(set * st, void * element);
void set_add_all(set * st, iterator * it;
void set_remove(set * st, void * element);
int set_size(set * st);
bool set_contains(set * st, void * element);
```

Donde el significado de las funciones es:

set set_empty(type t): Se crea un conjunto vacío con elementos de tipo *t*
*set set_of(list * ls)*: Se crea un conjunto a partir de los elementos de la lista
*void set_add(set * st, void * e)*: Se añade el elemento al conjunto
*void set_add_all(set * st, iterator * it):* Añade al conjunto todos los elementos del iterador.
*void set_add_remove(set * st, void * e):* Elimina del conjunto el elemento e
*int set_size(set * st):* El tamaño del conjunto
*bool set_contains(set * st, void * element*): Indica si el conjunto contiene al elemento

La implementación de *set_add* y *set_remove* es:

```
void set_add(set * st, void * element){
	map_put(st,element,NULL);
}

void set_remove(set * st, void * element) {
	map_remove(st,element);
}
```

Otras funciones disponibles para los conjuntos son:

```
char * set_tostring(set * st, char * mem);
iterator set_iterable(set * st);
set set_map(set * st, type * type_out,
	void * (*f)(void * out, void * in));
set set_filter(set * st, bool (*p)(void * in));
list set_tolist(const set * s);
set * set_parse(set * out, char * text);
```

Cuyo significado es:

*char * set_tostring(set * st, char * mem):* Convierte eel conjunto en una cadena de caracteres.

*iterator set_iterable(set * st*): Obtiene un iterador del conjunto

*set set_map(set * st, type * t, void * (*f)(void * out, void * in)):* Construye un nuevo conjunto aplicando a cada elemento la función f.

*set set_filter(set * st, bool (*p)(void * in)):* Construye un nuevo conjunto con los elementos que cumplen el predicado.

*list set_tolist(const set * s):* Obtiene una lista a partir de los elementos del conjunto

*set * set_parse(set * out, char * text):* Construye un conjunto a partir una cadena de texto. Asume como separadores y delimitadores los caracteres "{ ,}".

La transformación de un conjunto mediante una función:

```
set set_map(set * st, type * t,
              void * (*f)(void * out, void * in)){
       iterator it = set_iterable(st);
       set r = set_empty(type_out);
       char mem[type_out->size];
       while(iterable_has_next(&it)){
              void * e = iterable_next(&it);
              f(mem,e);
              set_add(&r,mem);
       }
       return r;
}
```

El filtro de un conjunto mediante un predicado usa un iterador que más adelante veremos.

```
set set_filter(set * st, bool (*p)(void * in)) {
	iterator it = set_iterable(st);
	set r = set_empty(st->hash_table.key_type);
	while (iterable_has_next(&it)) {
		void * e = iterable_next(&it);
		if(p(e)) set_add(&r,e);
	}
	return r;
}
```

Implementación de multiconjuntos

A su vez un *multiconjunto* (*multiset*) puede ser implementado también como una tabla hash con valores enteros que representan en número de veces que se repite un elemento. La implementación, similar a la del cojunto:

```
typedef map multiset;
```

Con las operaciones disponibles:

```
multiset multiset_empty(type type_element);
multiset multiset_of(list * ls);
void multiset_add(multiset * st, void * element);
void multiset_add_n(multiset * st, void * element, int n);
void multiset_remove(multiset * st, void * element, int n);
int multiset_size(multiset * st);
bool multiset_contains(multiset * st, void * element);
int multiset_count(multiset * st, void * element);
char * multiset_tostring(multiset * st, char * mem);
iterator multiset_items_iterable(multiset * st);
```

Donde el significado de las funciones es:

multiset multiset_empty(type t): Crea un multiset vacío con elementos de tipo t.

*multiset multiset_of(list * ls):* Crea un multiset con los elementos de la lista

*void multiset_add_n(multiset * st, void * e, int n*): Añade n copias de e al multiset

*void multiset_add(multiset * st, void * e*): Añade 1 copia de e al multiset

*void multiset_remove_n(multiset * st, void * e, int n):* Elimina n copias del elemento e del multiconjuto

*int multiset_size(multiset * st):* El cardinal del multiconjunto

*bool multiset_contains(multiset * st, void * e):* Si el multiconjunto contiene a *e*.

*int multiset_count(multiset * st, void * element):* El numero de veces que el multiconjunto contiene a *e*.

*char * multiset_tostring(multiset * st, char * mem):* Obtiene una cadena de caracteres con los elementos del multiconjunto.

*iterator multiset_items_iterable(multiset * st):* Obtiene un iterable de con los elementos del multiconjunto.

La implementación de *multiset_add_n* se basa en las correspondientes funciones de la tabla hash.

```
void multiset_add_n(multiset * st, void * element, int n) {
	if (map_contains_key(st, element)) {
		int count = *(int *)
			hash_table_get(st, element);
		count = count + n;
		map_put(st, element,&count);
	} else {
		int nn = n;
		map_put(st, element,&nn);
	}
}
```

La implementación de *multiset_remove_n* también se basa en las correspondientes funciones de la tabla hash.

```
void multiset_remove(multiset * st, void * e, int nu) {
       if (map_contains_key(st, e)) {
              void * n = map_get(st,e);
              int m = MAX(*(int*) n -nu, 0);
              if(m>0) map_put(st,e,&m);
              else map_remove(st, e);
       }
}
```

ListMultimap y SetMultimap

Los tipos ListMultimap y SetMultimap son casos específicos de diccionarios donde los valores son listas o conjuntos de elementos. Su implementación se basa en una tabla hash.

```
typedef map list_multimap;
typedef map set_multimap;
```

La forma de crear un list_multimap vacío

```
list_multimap list_multimap_empty(type * key_type,
              type * value_type){
       type  t = generic_type_1(&list_type, value_type);
       return map_empty(key_type,type_copy(&t));
}
```

La forma de crear un set_multimap vacío

```
set_multimap set_multimap_empty(type * key_type,
              type * value_type){
       type  t = generic_type_1(&set_type, value_type);
       return map_empty(key_type,type_copy(&t));
}
```

Como podemos observar ambos tipos se implementan sobre tablas hash. Los valores son listas y conjuntos. Ambos tipos son genéricos y por lo tanto debemos instanciarlos a tipos concretos sustituyendo los parámetros de tipo por tipos específicos.

Iteradores

El iterador es un tipo que no permite representar un flujo secuencial de datos. Es similar, en sus aspectos secuenciales, al tipo *Stream* de *Java*. Un iterator tiene un estado que guarda el valor actual del flujo de datos (también un estado auxiliar para posibles operaciones) y funciones específicas para recorrer el flujo de datos:

- bool *iterable_has_next*(iterator * st);
- void * *iterable_next*(iterator * st);

En el repositorio se ofrece una factoría de iteradores que emulan a algunos de la factoría de Java:

- iterator *iterable_empty*();
- iterator *iterable_range_long*(long a, long b, long c);
- iterator *iterable_range_double*(double a, double b, double c);
- iterator *iterable_iterate*(int size_state,void * initial_value,
  bool (*hash_next)(void * element),
  void * (*next)(void * out, void * in));
- iterator *file_iterable_string_fix_tam*(char * file, int n);
- iterator *split_iterable*(char * text, const char * delimiters);

También un conjunto de operaciones para producir nuevos iteradores a partir de otros:

- iterator *iterable_map*(iterator * st,int size_state,
  void * (*map_function)(void * out, const void * in));
- iterator *iterable_filter*(iterator * st, int size_state,
  bool (*map_filter)(void * in));
- iterator *iterable_flatmap*(iterator * st, int size_state,
  void * (*map_function)(void * out, void * in));
- iterator *iterable_consecutive_pairs*(iterator * st, int size_element);
- iterator *iterable_enumerate*(iterator * st);

El código puede verse en el repositorio. Veamos las implementciones de algunos de ellos

Implementación de iteradores

Los ficheros son de uso muy general. Es conveniente disponer de un iterador que recorra sus líneas y oculte los detalles del acceso al fichero. Veamos los detalles de implementación de un iterador junto a los particulares de un iterador sobre las líneas de un fichero.

El tipo iterador lo definimos como:

```
typedef struct st {
	type * type;
	void * state;
	void * a_state;
	void * dp;
	int dp_size;
	bool (*has_next)(struct st * iterator);
	void * (*next)(struct st * iterator);
	void (*free_dp)(void * in);
} iterator;
```

Con el significado siguiente para cada campo.

*type * type*: El tipo de los elementos del iterador

*void * state, void * a_state*: El estado que guarda el valor siguiente a devolver por el iterador y un estado auxiliar del mismo tipo para hacer operaciones. Ambos son de tipo type.

*void * dp*: Información específica de cada tipo de iterador que denominaremos dependencias.

int dp_size: Tamaño de la memoria necesaria para dp.

*bool (*has_next)(struct st * iterator):* Función que indica si hay un elemento siguiente.

*void * (*next)(struct st * iterator):* Función que proporciona el siguiente elemento y actualiza el estado.

*void (*free_dp)(void * in)*: Función para liberar las dependencias.

Iterador sobre las líneas de un fichero

Para el caso del iterador de un fichero los detalles específicos son: *file*, de tipo *FILE **proporcionado por las librerías de C, y *has_next* de tipo *bool* que indica si hay un siguiente elemento o no.

```
typedef struct{
	FILE * file;
	bool has_next;
}dp_file;
```

La función *iterable_file_has_next* devuelve la propiedad específica *has_next* del iterador.

```
bool iterable_file_has_next(iterator * c_it) {
	dp_file * dp = (dp_file *) c_it->dp;
	return dp->has_next;
}
```

La función *iterable_file_next* copia el estado en el estado auxiliar e intenta buscar el siguiente elemento. El tamaño del estado es el tamaño que indica el tipo del iterable.

Para conseguir el siguiente elemento invoca la función *fgets* de la librería de C pasando como parámetros el estado, donde quedará el dato leído, el tamaño de estado que indicará el máximo de caracteres a leer y el puntero al fichero.

Si el resultado de la llamada es NULL entonces sabemos que no hay más elementos y actualizamos la propiedad específica *has_next*.

Eliminamos el salto de fin de línea antes de devolver la línea leída.

```
void * iterable_file_next(iterator * c_it){
	dp_file * dp = (dp_file *) c_it->dp;
	type * t = c_it->type;
	copy(c_it->a_state,c_it->state,c_it->type);
	char * r = fgets(c_it->state, t->size, dp->file);
	dp->has_next = r!=NULL;
	remove_eol(c_it->a_state);
	return c_it->a_state;
}
```

Con las funciones anteriores ya implementadas y definidas las propiedades específicas del iterador, podemos implementarlo.

Como podemos observar, usamos la función open de la librería de C que nos devuelve un puntero de tipo FILE *. Si este puntero es NULL no se ha encontrado el fichero y generamos el mensaje de error correspondiente.

Definimos un tipo *string_fix* con el número de caracteres máximo que estimemos habrá por línea. Es el tipo t. Construimos el iterador con las funciones definidas antes y el tipo decidido.

Con la función *fgets* de la librería de C intentamos leer una línea dejando el resultado en el estado y actualizando la propiedad específica *has_next*.

El iterador está implementado de tal manera que el siguiente valor válido se almacenará en el estado si *has_next* es verdadero. Al pedir un nuevo valor el estado se copiará en el estado auxiliar que es el que se devolverá y se volverá a actualizar el estado junto con la propiedad *has_next*.

Todo ello se implementa en la función en la función *iterable_file_string_fix_tam* convierte ficheros en iterables. Cada vez que la llamamos nos devuelve un iterador.

```
iterator iterable_file_string_fix_tam(char * file,
              int line_tam) {
       FILE * st = fopen(file,"r");
       char  ms[Tam_String];
       if(st==NULL) sprintf(ms,
              "no se encuentra el fichero %s",file);
       check_not_null(st,__FILE__,__LINE__,ms);
       dp_file dp = {st,false};
       int size_dp = sizeof(dp_file);
       void * dp_p = malloc(size_dp);
       memcpy(dp_p,&dp,size_dp);
       type t = string_fix_type_of_tam(line_tam);
       iterator it_file = iterable_create(type_copy(&t,NULL),
              iterable_file_has_next,iterable_file_next,
              dp_p,size_dp,free_dependencies_file);
       dp_file * dp2 = (dp_file *)it_file.dp;
       char * r = fgets(it_file.state,line_tam,dp2->file);
       dp2->has_next = r!=NULL;
       return it_file;
}
```

La implementación de los restantes iteradores puede verse en el repositorio. Algunos los veremos como ejercicios.

Iterador sobre las partes de una cadena de texto

Es un iterador de uso habitual cuando queremos dividir una cadena de texto en partes, tokens, separados por caracteres dados. Para conseguirlo usaremos la función *strtok_r* que es una versión reeentrante de *strtok*. La función *strtok* puede dividir una cadena en partes dados unos delimitadores. La versión reentrante puede ser interrumpida en medio de su ejecución y volver a llamarse de forma segura. Aquí usaremos *strtok_r2* que hace el mismo trabajo que *strtok_r*.

Este iterador tiene la siguiente información específica (dependencias):

```
typedef struct{
       char * text;
       char * token;
       char * delimiters;
       char * saveptr[1];
}dp_split;
```

Donde los campos indican, respectivamente, el texto a separar en partes, el último token encontrado, los delimitadores usados y un puntero a puntero que será usado por la función *strtok_r2* para guardar la última posición. No entramos aquí en los detalles de la función *strtok_r2* que pueden verse en el repositorio.

El iterador no tendrá más elementos cuando strtok_r2 devuelva NULL

```
bool iterable_split_has_next(iterator * c_it) {
	dp_split * dp = (dp_split *) c_it->dp;
	return dp->token != NULL;
}
```

El siguiente token será el guardado en las dependencias, pero antes de devolverlo lo guardamos y encontramos el siguiente.

```
void * iterable_split_next(iterator * c_it){
	dp_split * dp = (dp_split *) c_it->dp;
	char * old = dp->token;
	dp->token = strtok_r2(NULL,dp->delimiters,dp->saveptr);
	return old;
}
```

Ambos punteros a char, old y dp_token, señalan a posiciones en la cadena de caracteres que estamos partiendo.

Con estos elementos podemos implementar el iterador en la función *iterable_split_text_tam* que toma como parámetros la cadena, los delimitadores y el tamaño máximo de los tokens.

```
iterator iterable_split_text_tam(char * text,
              const char * delimiters, int tam) {
       type t = string_fix_type_of_tam(tam);
       if(string_fix_all_space(text))
              return iterable_empty(type_copy(&t,NULL));
       dp_split dp;
       int size_dp = sizeof(dp_split);
       dp.text = malloc(strlen(text)+2);
       dp.text = strcpy(dp.text,text);
       dp.delimiters = delimiters;
       dp.token = strtok_r2(dp.text,delimiters,dp.saveptr);
       void * dp_p = malloc(size_dp);
       memcpy(dp_p,&dp,size_dp);
       iterator r = iterable_create(type_copy(&t,NULL),
              iterable_split_has_next,
              iterable_split_next,dp_p,size_dp,
              dependencies_split_free);
       return r;
}
```

Como vemos, si la cadena solo tiene caracteres blancos devuelve un iterador vacío.

El tipo de los elementos del iterador es *string_fix* es decir cadenas de texto de tamaño *tam*.

Iterador sobre las palabras de un fichero

Este es un iterador que podemos obtener combinando el iterador sobre las líneas de un fichero con funciones sobre iterables como *iterable_flat_map* y otros iterables anteriores como *iterable_split_text.*

La idea es convertir el fichero en un iterable de líneas cuyo tamaño máximo será *line_tam* y posteriormente expandir cada línea en las partes separadas por los separadores indicados mediante *iterable_flat_map* y *iterable_split_text_tam.* La función *iterable_split_text_function* es equivalente a *iterable_split_text_tam* con los parámetros adecuados.

```
iterator iterable_words_in_file(char *file, int line_tam,
              int word_tam, char * sep) {
       type t = string_fix_type_of_tam(word_tam);
       strcpy(text_to_iterable_delimiters,sep);
       iterator r1 = iterable_file_string_fix_tam(
              file, line_tam);
       string_fix_function_tam = word_tam;
       iterator r2 = iterable_flatmap(iterable_copy_new(&r1),
              type_copy(&t,NULL),iterable_split_text_function);
       return r2;
}
```

Iterable sobre palabras y líneas

Es un iterable que podemos obtener de los anteriores para devolvernos pares de palabras y las líneas en las que se encuentran.

```
iterator iterable_words_and_line_in_file(char *file, int
init,int line_tam, int word_tam, char *sep) {
       type t = string_fix_type_of_tam(word_tam);
       type t2 = generic_type_1(&enumerate_type,&t);
       type_expand = type_copy(&t2,NULL);
       strcpy(text_to_iterable_delimiters, sep);
       word_tam_expand = word_tam;
       iterator r1 = iterable_file_string_fix_tam(
              file, line_tam);
       iterator r2 = iterable_enumerate(
              iterable_copy_new(&r1),init);
       iterator r3 = iterable_flatmap(iterable_copy_new(&r2),
              type_expand,enumerate_expand_f);
       return r3;
}
```

Como podemos observar obtenemos primero un iterador sobre las líneas del fichero. En segundo lugar, usamos *iterable_enumerate* para obtener pares de líneas y el número de línea correspondiente. Por último, usamos *iterable_flatmap* combinado con la función *enumerate_expand* para expandir cada par anterior en otros formados por el mismo número de línea y cada una de las palabras de la línea.

Iteradores sobre agregados de datos

Veamos el diseño de un *iterador para una lista*. Todo iterador debe tener un estado. En este caso el estado será el par formado por un puntero a la lista y un entero.

```
typedef struct{
	list * ls;
	int i;
}dp_list;
```

La función *has_next* devolverá true siempre que la *i* del estado sea menor que el tamaño de la lista

```
bool iterable_list_has_next(iterator * current_iterable) {
	dp_list * dp = (dp_list *) current_iterable->dps;
	return dp->i < list_size(dp->ls);
}
```

La función *next* devolverá la casilla de índice *i* e incrementará el índice

```
void * iterable_list_next(iterator * current_iterable){
	dp_list * dp = (dp_list *) current_iterable->dps;
	int old_i = dp->i;
	dp->i = dp->i +1;
	return list_get(dp->ls,old_i);
}
```

Con las funciones anteriores y el estado diseñado podemos implementar el iterador.

```
iterator list_iterable(list * ls){
	dp_list dp = {ls,0};
	int size_dp = sizeof(dp_list);
	void * dp_p = malloc(size_dp);
	memcpy(dp_p,&dp,size_dp);
	iterator s_list = iterable_create(
		type_copy(ls->type,NULL),iterable_list_has_next,
		iterable_list_next,dp_p,size_dp,free);
	return s_list;
}
```

El iterador sobre los pares de una tabla hash recorre los bloques de la tabla y dentro de cada bloque los elementos en la lista asociada. Tiene la información asociada:

```
typedef struct{
	map * ht;
	int nb;
	int i;
	int j;
}dp_hash_table;
```

Donde *nb* es el número de bloques de la tabla, *i* el número de bloque y *j* la entrada concreta en ese bloque.

Con esa información podemos diseñar las funciones *has_next* y *next* específicas del iterador. La primera es:

```
bool iterable_hash_table_has_next(iterator * c_iterable) {
	dp_hash_table * dp = (dp_hash_table *) c_iterable->dp;
	return dp->i < dp->nb;
}
```

La segunda requiere usar una función que encuentre el siguiente par (i,j) entre las entradas disponibles.

```
void next_state(dp_hash_table * dp){
	map * table = dp->ht;
	int i = dp->i;
	int j = dp->j;
	while(i<dp->nb){
		if(j<0) j = table->blocks[i];
		else j = table->data[j].next;
		if(j>=0) break;
		i++;
	}
	dp->i = i;
	dp->j = j;
}
```

La función next es entonces

```
void * iterable_hash_table_next(iterator * c_iterable){
	dp_hash_table * dp = (dp_hash_table *) c_iterable->dp;
	map * table = dp->ht;
	pair * state = (pair *)c_iterable->state;
	state->key = table->data[dp->j].key;
	state->value = table->data[dp->j].value;
	next_state(dp);
	return c_iterable->state;
}
```

A partir de esas funciones podemos diseñar el iterador.

```
iterator map_items_iterable(map * ht){
	dp_hash_table dp = {ht,ht->capacity_blocks,0,-1};
	int size_dp = sizeof(dp_hash_table);
	void * dp_p = malloc(size_dp);
	memcpy(dp_p,&dp,size_dp);
	type t = generic_type_2(&pair_type,ht->key_type,
		ht->value_type);
	iterator r_hash_table = iterable_create(
		type_copy(&t,NULL),
		iterable_hash_table_has_next,
		iterable_hash_table_next,dp_p,size_dp,free);
	next_state(r_hash_table.dp);
	return r_hash_table;
}
```

El *iterador sobre un conjunto* se obtiene quedándonos solo con las claves del iterador de la tabla hash en la que se delega.

```
iterator set_iterable(set * st){
	iterator r = map_items_iterable(st);
	iterator im = iterable_map(iterable_copy_new(&r),
		type_copy(st->key_type,NULL),pair_key);
	return im;
}
```

El *iterador sobre un multiconjunto* se obtiene de forma similar a partir del iterador de la tabla hash que lo implementa.

```
iterator multiset_iterable(multiset * st){
	return map_items_iterable(st);
}
```

Los iteradores sobre *list_multimap* y *set_multimap* usan los iteradores proporcionados por las tablas hash que los implementan.

Factorías y funciones sobre iteradores

En el repositorio se encuentran varias factorías y funciones sobre iteradores. Las primeras nos permiten obtener iteradores nuevos, las segundas obtener nuevos iteradores a partir de otros.

En la factoria de iteradores incluimos además de los anteriores:

iterator iterable_empty(): Un iterador vacío.

iterator iterable_range_long(long a, long b, long c): Un iterador que proporciona los valores enteros de una progresión aritmética de primer valor a, diferencia c hasta el valor b excluido.

iterator iterable_range_double(double a, double b, double c): Un iterador que proporciona los valores relaes de una progresión aritmética de primer valor a, diferencia c hasta el valor b excluido.

*iterator iterable_iterate(type * type, void * initial_value,*
*bool (*has_next)(void * element),*
*void * (*next)(void * out, void * in)):* Un iterador que proporciona los valores de tipo type de una secuencia que comienza en initial y para cada valor obtiene el siguiente aplicando la función *next*. La secuencia continua mientras se cumpla el predicado *has_next*.

iterator iterable_random_long(long n, long a, long b): Un iterador que proporciona n números enteros entre a y b.

iterator iterable_primos(int a, int b): Un iterador que proporciona los números primos mayores o iguales que a y menores que b.

Junto a la factoría disponemos de funciones que permiten obtener iteradores a partir de otros.

*iterator iterable_map(iterator * st, type * type,*
*void * (*function)(void * out, const void * in)):* Un nuevo iterador con los valores del antiguo transformados por la función function.

*iterator iterable_filter(iterator * st, bool (*filter)(void * in)):* Un nuevo iterador con los valores del antiguo filtrados por el predicado filter.

*iterator iterable_flatmap(iterator * st, type * type,*
*iterator * (*map_function)(void * in)):* Un nuevo iterador con los valores del antiguo expandidos mediante la función function.

*iterator iterable_consecutive_pairs(iterator * st):* Un nuevo iterador con pares formados por dos elementos consecutivos.

*iterator iterable_enumerate(iterator * st, int n):* Un nuevo iterador con pares formados por enteros consecutivos y los elementos del antiguo iterador.

*iterator iterable_zip(iterator * it1, iterator * it2):* Un nuevo iterador con pares formados por un elemento de cada iterador.

Ejemplos iteradores y de su uso

1. *Dado un fichero de texto, calcular el número de líneas*

```
int num_lineas(char * file) {
	iterator g1 = file_iterable_string_fix_tam(file,100);
	int n = 0;
	for(;iterable_has_next(&g1);iterable_next(&g1)) n++;
	return n:
}
```

Escogemos un iterable sobre las líneas de un fichero. El tipo de los elementos de este iterable es *string_fix* de tamaño *string_fix_tam* que

hemos fijado en 100. Es decir, asumimos que las líneas del fichero no tendrán más de 100 caracteres.

Este iterador podemos recorrerlo con una sentencia for y contar el número de líneas.

---

2. Dado un fichero de texto, calcular el número de líneas vacías (que están vacías o solo tienen espacios en blanco)

Ahora necesitamos filtrar los elementos del iterable quedándonos con aquellas líneas que tienen todos los caracteres blancos o no tienen ninguno.

Podemos diseñar un iterable a partir de otro definiendo un filtro, es decir un predicado sobre los elementos del iterable. Esta operación con iterables se le llama *filter*. Escribamos una función que nos indique si todos los elementos del iterable son blancos o no. Es una iteración de tipo para todo. La función la llamamos *string_fix_all_space.*

```
bool string_fix_all_space(char * in){
       bool r = true; int i =-1;
       while(in[++i] != '\0' && r) r = isspace(in[i]);
       return r;
}
```

El código anterior hemos de tener en cuenta que las cadenas de caracteres en C se terminan cuando aparece el carácter *'\0'*. El código va comprobando que el carácter en la posición i es un espacio y continua mientras *r* sea verdadero, es decir mientras los caracteres sean espacios, lo que es comprobado por la función *isspace* de C. El operador unario prefijo *++* incrementa el valor de *i* antes de ser usada como índice.

Con esta función podemos diseñar un iterable que filtre las líneas vacías y contarlas.

```
int num_lineas_vacias(char * file) {
	string_fix_tam = 100;
	iterator g1 = file_iterable_string_fix(file);
	iterator g2 = iterable_filter(&g1,string_fix_all_space);
	int n = 0;
	for(;iterable_has_next(&g2);iterable_next(&g2)) n++;
	return n;
}
```

3. *Contar el número de palabras de un fichero*

Podemos usar el iterador sobre las palabras de un fichero y contarlas.

```
int num_palabras(char * file) {
	iterator r = iterable_words_in_file(file,100,20,
		" ,;.()");
	int n = 0;
	for(;iterable_has_next(&g3);iterable_next(&g3)) n++;
	return n;
}
```

4. *Dado un fichero con una fecha por línea, contar cuántas son posteriores a una fecha dada*

Asumimos que el fichero tiene las fechas en el formato:

```
2-3-1991
1-5-1995
2-9-1989
2-7-2001
…
```

Las líneas del fichero debemos transformarlas a fechas. Es la operación entre iterables que hemos llamado *map*.

Diseñamos un filtro y una función de transformación.

```
time_t ref;

bool posterior(time_t * f){
	return date_type.order(f,&ref,&date_type) > 0;
}

time_t * p_date(time_t * out, char * in){
	return date_type.parse(out,in,&date_type);
}
```

Con estas funciones podemos encontrar un iterable transformado (map) y otro filtrado. A partir de este último podemos contar sus elementos.

```
int num_posteriores(char *file) {
	string_fix_tam = 100;
	ref = time_of_date(1,1,2000);
	iterator g1 = file_iterable_string_fix(file);
	iterator g2 = iterable_map(&g1,&date_type,p_date);
	iterator g3 = iterable_filter(&g2,posterior);
	int n = 0;
	for (; iterable_has_next(&g3); iterable_next(&g3)) n++;
	return n;
}
```

5. *Obtener una cadena de caracteres a partir de los elementos de un iterador, un separador y unos delimitadores*

Dado un iterador, es interesante en muchos casos obtener una cadena de caracteres con los elementos del iterador convertidos en cadenas de caracteres separadas por el separador indicado. Podemos, además, fijar un prefijo y un sufijo par la cadena obrenida. El parámetro *mem* será la memoria para ubicar la cadena resultante, por lo que deberá ser dimensionada adecuadamente.

```
char * iterable_tostring_sep(iterator * st,char * sep,
              char * prefix,char * suffix,char * mem){
       char m[Tam_String];
       bool first = true;
       strcpy(mem,prefix);
       while(iterable_has_next(st)){
              void *  next = iterable_next(st);
              char * ns = tostring(next,m,st->type);
              if(first) first = false;
              else strcat(mem,sep);
              strcat(mem,ns);
       }
       strcat(mem,suffix);
       return mem;
}
```

6. *Obtener un iterador de enteros aleatorios en un rango*

La idea es partir de un iterador que genere los enteros de 0 a n-1. Posteriormente transformar el iterador para convertir cada entero en un entero aleatorio.

No hacen falta varias funciones: incrementar y obtener un entero aleatorio en un rango. Todas las funciones que usamos en los iteradores deben tener la cabecera *void* f(void * out, void * in).* Es decir, sólo están permitidos la entrada (*in*) y la salida *out.* La salida debe señalar a una memoria del tamaño adecuado al tipo del resultado. Para poder usar funciones con más parámetros que podamos reutilizar pasamos los parámetros adicionales mediante variables globales. Una variable asociada a cada función.

A continuación, podemos ver la función entero aleatorio

```
long entero_aleatorio_long_a;
long entero_aleatorio_long_b;

long * entero_aleatorio_long_f(long * out, void * in){
       *out = entero_aleatorio(entero_aleatorio_long_a,
              entero_aleatorio_long_b);
       return out;
}
```

Como vemos, cuando llamamos a la función *entero_aleatorio_long_f* los parámetros adicionales, como los límites en los que debemos obtener el entero, se le pasan dando valores a las variables globales long_*entero_aleatorio_long_a, long entero_aleatorio_long_b*.

```
long inc_long_ref;
long * inc_long_f(long * out, long * in){
       *out = *(in)+inc_long_ref;
       return out;
}

long menor_que_long_ref;
bool menor_que_long(long * a){
       return *a < menor_que_long_ref;
}
```

De forma similar, pasamos los parámetros a la función incrementar y a los predicados. Con estos elementos el iterador de enteros aleatorios en un rango es de la forma siguiente.

```
iterator iterable_random_int(int n, int a, int b) {
	new_rand();
	entero_aleatorio_long_a = a;
	entero_aleatorio_long_b = b;
	menor_que_long_ref = n;
	inc_long_ref = 1;
	int e = 0;
	iterator r = iterable_iterate(&long_type, &e,
		menor_que_long, inc_long_f);
	iterator r2 = iterable_map(iterable_copy(&r),
		&long_type, entero_aleatorio_long_f);
	return r2;
}
```

Es conveniente destacar cómo se ha hecho una copia del iterador *r* . Esto es necesario porque al devolver r2 los punteros que señalen a variables locales quedarán rotos. Al hacer la copia del iterador se consigue memoria dinámica, por lo que al devolver r2 los punteros a variables locales señalarán a la memoria dinámica obtenida.

7. *Obtener un iterable con los números primos posteriores o iguales a un número y menores que otro*

Para el diseño de este iterador, usamos *iterable_iterate* con los parámetros: el primer primo mayor a igual que a, un predicado que nos indica si el primo es menor que b y una función que nos da el siguiente primo de uno dado.

```
iterator iterable_primos(int a, int b) {
	if (!es_primo(a)) a = siguiente_primo(a);
	menor_que_long_ref = b;
	iterator r = iterable_iterate(&long_type,&a,
		menor_que_long, siguiente_primo_f);
	return r;
}
long * siguiente_primo_f(long * out, long * in);
```

La funcion *siguiente_primo_f* debe tener la cabecera indicada.

Tratamientos secuenciales: iterables, iteradores y acumuladores

Combinando los tipos y los iteradores podemos estructurar y sistematizar los tratamientos secuenciales. Un tratamiento secuencial se puede entender como un flujo de datos, que modelaremos como un iterador, más un acumulador que nos permitirá obtener el resultado deseado del flujo de datos. Un iterable es una factoría de iteradores.

En muchos casos hablaremos de *acumulación*: la combinación de un iterable con un acumulador concreto.

Iteradores y acumuladores

Un acumulador es un estado, la base del acumulador, y un conjunto de funciones que nos permite obtener el resultado deseado (*acumular*) a partir de un iterator. Llamamos acumular a la combinación de un iterador y un acumulador.

Un acumulador tiene las siguientes propiedades:

- *Estado* (state): De tipo *tb*. Es donde se mantiene el valor de lo ya acumulado. También lo denominaremos base del acumulador.
- *Valor Inicial:* De tipo *tb*. Es el valor de la base cuando instanciamos el acumulador.
- *Función de acumulación* (add): Acumula el valor recibido del iterator (de tipo *te*) en la base.

```
bool acum(tb * b, const te * e);
```

La funcion acumula *e* en la base *b* ydevuelve *true* si la base del acumulador ha alcanzado un valor que nos permite ignorar el resto de los valores del iterador. A ese aspecto de la función de acumulación se le llama función de cortocircuito y algunas veces se presenta como una función separada.

- *Función de retorno:* Calcula el valor del resultado a partir de la base. Si el tipo del resultado es *tr*, es de la forma:

```
tr * result(tr * r, const tb * b);
```

En muchos casos, si *tb* y *tr* son los mismos tipos, el resultado devuelto es la misma base. Consideraremos este caso en los acumuladores planteados.

Combinando distintos tipos de acumuladores con un iterable podemos obtener determinados resultados. El cálculo del mínimo y el máximo de los elementos de un iterable:

```
void * iterable_min_naturalorder(iterator * st);
void * iterable_min(iterator * st,
       int (*order)(const void * e1, const void * e2));
void * iterable_max_naturalorder(iterator * st);
void * iterable_max(iterator * st,
       int (*order)(const void * e1, const void * e2));
```

Otros iterables conocidos son:

```
bool iterable_all(iterator * st, bool (*p)(const void * in));
bool iterable_any(iterator * st, bool (*p)(const void * in));
void * iterable_first(iterator * st,
      bool (*p)(const void * in));
```

Acumulación de valores numéricos.

```
double iterable_sum(iterator * st);
int iterable_size(iterator * st);
double iterable_average(iterator * st);
```

Veamos cada uno de ellos:

*void * iterable_min_naturalorder(iterator * st)*: Obtiene el mínimo de los elementos del iterable según el orden natural incorporado al tipo.

*void * iterable_min(iterator * st,*
*int (*order)(const void * e1, const void * e2)):* Obtiene el mínimo de los elementos del iterable según el orden definido en el parámetro.

*void * iterable_max_naturalorder(iterator * st)*: Obtiene el máximo de los elementos del iterable según el orden natural incorporado al tipo.

*void * iterable_max(iterator * st,*
*int (*order)(const void * e1, const void * e2)):* Obtiene el máximo de los elementos del iterable según el orden definido en el parámetro.

*bool iterable_all(iterator * st, bool (*p)(const void * in)):* Devuelve true si todos los elementos del iterable cumplen el predicado.

*bool iterable_any(iterator * st, bool (*p)(const void * in)):* Devuelve true si alguno de los elementos del iterable cumple el predicado.

*void * iterable_first(iterator * st, bool (*p)(const void * in)):* Devuelve el primer elemento del iterable que cumple el predicado.

*double iterable_sum(iterator * st):* Devuelve la suma de los elementos del iterable.

*int iterable_size(iterator * st):* Devuelve el número de elementos del iterable.

*double iterable_average(iterator * st):* Devuelve la media de elementos del iterable.

La implementación es similar para todos ellos. Veamos dos ejemplos.

El cálculo del mínimo puede resultar en un valor o en NULL si el iterable está vacío. Declaramos una variable del tipo de los elementos del iterador para que guarde el mínimo valor y una variable *first* que indica si el siguiente es el primer elemento. Si terminado el bucle *first* es true entonces el resultado es NULL.

```
void * iterable_min(iterator * st,
       int (*comparator)(const void * out, const void * in)){
       type * t = st->type;
       void * minvalue = malloc(t->size);
       bool first = true;
       while(iterable_has_next(st)){
              void * next = iterable_next(st);
              if(first) {
                     copy(minvalue, next, t->size);
                     first = false;
              } else if(comparator(next,minvalue) < 0) {
                     copy(minvalue, next, t->size);
              }
       }
       if(first) {
              free(minvalue);
              minvalue = NULL;
       }
       return minvalue;
}
```

Los acumuladores *all, any, none* acumulan en su base un valor booleano y lo van combinando con el valor obtenido por un predicado aplicado a cada elemento del iterador. Si la combinación de la base con el resultado del predicado es con el operador *and* resulta *all*, si es con *or* resulta *any*. *None* es el contrario de *any*. En el caso de *all* devuelve true si todos los elementos del iterable cumplen el predicado. Por las propiedades del operador *and* una implementación de la acumulación *all* es:

```
bool iterable_all(iterator * st, bool (*p)(const void * in)){
	bool r = true;
	while(iterable_has_next(st) && r) {
		void * e = iterable_next(st);
		r = p(e);
	}
	return r;
}
```

Podemos observar que la base del acumulador es la variable *r* de tipo *bool* inicializada a *true*. Teniendo en cuenta que *false && e = false* escogemos la función de tal forma que cuando *r* sea false podemos dar por acabada la acumulación.

Otros acumuladores permiten obtener listas, conjuntos, multiconjuntos, tablas hash, etc.

```
list iterable_to_list(iterator * st);
set iterable_to_set(iterator * st);
multiset iterable_to_multiset(iterator * st,
multiset iterable_to_multiset_groups(iterator * st,
	type * key_type,
	void * (*f_key)(void * out, void * in));
```

La función *iterable_to_multiset_groups* obtiene las frecuencias de los grupos definidos por *f_key*.

Acumulación por la izquierda y por la derecha

En general dado un *iterator* y un *acumulador* (definido por el valor inicial de la base, su tipo y la función de acumulación) podemos acumularlo por la izquierda o por la derecha. A esta combinación se le llama *acumular* y se puede implementar de forma genérica. Con la misma función de acumulación podemos acumular por la izquierda o por la derecha según empecemos a acumular desde el primer elemento del iterador o desde el último.

Ambas formas de acumular dan el mismo resultado si el operador asociado a la función de acumulación es asociativo y conmutativo.

```
void * accumulate_left(iterator * st, void * base,
       bool (*add)(void * out, const void * e));
void * accumulate_right(iterator * st, void * base,
       bool (*add)(void * out, const void * e));
```

La implementación de *accumulate_left* la haremos iterativa.

```
void * accumulate_left(iterator * st, void * base,
              bool (*add)(void * out, const void * e)) {
       bool f = false;
       while (iterable_has_next(st) && !f) {
              void * e = iterable_next(st);
              f = add(base, e);
       }
       return base;
}
```

La implementación de *accumulate_right* recursiva.

```
bool accumulate_right_private(iterator * st, void * base,
              bool (*add)(void * out, const void * e)) {
       bool f;
       if (iterable_has_next(st)) {
              char se[st->type->size];
              void * e = iterable_next(st);
              copy(se, e, st->type->size);
              f = accumulate_right_private(st, base, add);
              if(!f) f = add(base, se);
       }
       return f;
}
```

```
void * accumulate_right(iterator * st, void * base, bool
(*add)(void * out, const void * e)) {
       accumulate_right_private(st,base,add);
       return base;
}
```

Sean *tb* el tipo de la base y *te* de los elementos del iterator. Si ambos tipos son iguales podemos prescindir en algunos casos del valor inicial del

acumulador tomándolo igual al primer elemento. Esta forma de acumulación se llama *reducción* y también puede ser por la izquierda y por la derecha.

La acumulación se suele ampliar con una función adicional: la *función de retorno* que calcula el valor a devolver a partir del valor de la base. Este detalle se deja como ejercicio.

En este caso el resultado de la acumulación puede ser NULL si el iterador está vacío.

```
void * reduce_left(iterator * st, void * base,
       bool (*add)(void * out, const void * e));
void * reduce_right(iterator * st, void * base,
       bool (*add)(void * out, const void * e));
```

La implementación de *reduce_left* la hacemos iterativa

```
void * reduce_left(iterator * st, void * base,
              bool (*add)(void * out, const void * e)) {
       bool first = true; bool f = false;
       while (iterable_has_next(st) && !f) {
              void * e = iterable_next(st);
              if (first) {
                     copy(base, e, st->type->size);
                     first = false;
              } else if(!f) f = add(base, e);
       }
       if(first) return NULL; else return base;
}
```


Y la implementación de *reduce_right* recursiva

```
bool reduce_right_private(iterator * st, void * base,
       bool (*add)(void * out, const void * e), bool * first) {
       bool f = false;
       if (iterable_has_next(st)) {
              char se[st->type->size];
              void * e = iterable_next(st);
              copy(se, e,st->type->size);
              f = reduce_right_private(st, base, add, first);
              if (*first) {
                     copy(base, e,st->type->size);
                     * first = false;
              } else if(!f){
                     f = add(base, se);
              }
       }
       return f;
}
```

```
void * reduce_right(iterator * st, void * base,
              bool (*add)(void * out, const void * e)) {
       bool first = true;
       reduce_right_private(st, base, add, &first);
       if(first) return NULL;
       else return base;
}
```

Ejemplos de acumuladores

1. *Diseñar una función que convierta una lista en una cadena de caracteres*

La implementación la podemos conseguir obteniendo un iterador de la lista y posteriormente convirtiendo el iterador a cadena de caracteres con un separador, prefijo y sufijo determinados.

```
char * list_tostring(list * ls, char * mem){
	iterator st = list_iterable(ls);
	iterable_tostring_sep(&st,",","[","]",mem);
	iterable_free(&st);
	return mem;
}
```

2. *Obtener una lista con las líneas de un fichero*

Para obtener la lista pedida partimos de un iterador sobre un fichero ya visto previamente. Especificamos que el número de caracteres por línea es n. Diseñamos un tipo string_fix de tamaño n para los elementos de las casillas de la lista.

A media que leemos una línea del fichero eliminamos sus caracteres de fin de línea y la añadimos a la lista

```
list list_of_file(char * file, int n){
	type t = string_fix_type_of_tam(n);
	list r = list_empty(type_copy(&t));
	iterator f = file_iterable_string_fix_tam(file,n);
	while(iterable_has_next(&f)){
		char * s = iterable_next(&f);
		remove_eol(s);
		list_add(&r,s);
	}
	return r;
}
```

3. *Obtener una lista con las líneas de un fichero transformadas en objetos de un tipo que se da como parámetro*

Procedemos como en el ejemplo anterior pero ahora cada vez que leemos una línea la transformamos en uno de los objetos del tipo especificado y lo añadimos a la lista. Para poder hacer esa trasformación necesitamos una memoria temporal de tamaño *type->size*.

```
list list_of_file_type(char * file, type * type, int n) {
	iterator it = file_iterable_string_fix_tam(file,n);
	list ls = list_empty(type);
	char e[type->size];
	while (iterable_has_next(&it)) {
		char * line = (char *) iterable_next(&it);
		parse(e, line, type);
		list_add(&ls,e);
	}
	return ls;
}
```

4. *Obtener una lista de listas formadas con los fragmentos de las líneas de un fichero transformados en objetos de un tipo que se da como parámetro*

En este caso tras leer cada línea del fichero debemos obtener los fragmentos de la misma. Esto los conseguimos con el iterador *text_to_iterable_string_fix_tam*. Indicamos que las líneas tendrán un máximo de *n* caracteres, los fragmentos un máximo de *m* y los separadores los caracteres incluidos en la cadena *sep*.

La función tiene dos bucles que recorren los dos iteradores: el primero las líneas del fichero y el segundo los fragmentos. Cada uno de los fragmentos obtenidos en el bucle interior lo transformamos en uno de los objetos del tipo especificado y lo añadimos a la lista. Para poder hacer esa trasformación necesitamos como anteriormente una memoria temporal de tamaño *type->size*. Cada lista obtenida en el bucle interior la añadimos a la lista principal.

```
list list_of_list_of_file_type(char * file, type * type,
             char * sep, int n, int m){
       list res = list_empty(&list_type);
       iterator it1 = file_iterable_string_fix_tam(file,n);
       char e[type->size];
       while(iterable_has_next(&it1)) {
             char* linea = (char*)iterable_next(&it1);
             list ls = list_empty(type);
             iterator it2 = text_to_iterable_string_fix_tam(
                    linea,sep,m);
             while(iterable_has_next(&it2)) {
                    char * tx = iterable_next(&it2);
                   parse(e,tx,type);
                list_add(&ls,e);
             }
             list_add(&res, &ls);
       }
       return res;
}
```

5. *Obtener una lista a partir de una cadena de texto especificados los separadores*

Abajo vemos el código para obtener una lista de objetos de un tipo dado. Usamos como delimitadores y separadores los caracteres en el variable global *list_delimiters* y el tamaño de los fragmentos indicado en *string_fix_tam.* La lista la construimos a partir de los fragmentos obtenidos por el iterador y transformados por la función *parse*.

```
list * list_parse(list * out, char * text) {
       iterator it = text_to_iterable_string_fix_tam(text,
             list_delimiters,string_fix_tam);
       char tmp[out->type->size];
       while(iterable_has_next(&it)){
             void * txt = iterable_next(&it);
             parse(tmp,txt,out->type);
             list_add(out,tmp);
       }
       iterable_free(&it);
       return out;
}
```

6. *Calcular la suma de los primos menores que 2000*

Se presenta este ejemplo como ilustración de los iterables y los acumuladores.

Como el operador asociado al acumulador suma es asociativo y conmutativo podemos comprobar como la reducción por la izquierda resulta igual a la reducción por la derecha.

```
void suma_primos(){
	iterator it1 = iterable_primos(2,2000);
	iterator it2 = iterable_map(&it1,&long_type,
		square_long_f);
	long s = *(long*) reduce_left(&it2, &suma,long_sum);
	printf("%d\n",s);
	it1 = iterable_iterate(&long_type,&p,
		menor_que_long,siguiente_primo_f);
	it2 = iterable_map(&it1,&long_type,square_long_f);
	s = *(long*) reduce_right(&it2, &suma,long_sum);
	printf("%d\n",s);
}
```

7. *Combinar en una cadena de texto todas las palabras de una cadena dada*

Ahora el operador asociado (concatenar cadenas) no es conmutativo. El resultado de acumular por la derecha o por la izquierda es diferente.

```
void combina_palabras(char * text){
	iterator p3 = text_to_iterable_string_fix_tam(text,
		" ;.", 20);
	char b[10000];
	void *r2 = accumulate_left(&p3, b, str_cat);
	printf("1: %s\n", b);
	p3 = text_to_iterable_string_fix_tam(text, " ;.", 20);
	b[0] = '\0';
	r2 = accumulate_right(&p3, b, str_cat);
	printf("2: %s\n", b);
}
```

Hemos usado la función *str_cat*

```
bool str_cat(char * out, char * in){
	strcat(out,in);
	return false;
}
```

Y *text_to_iterable_string_fix_tam* que construye un iterador que nos la devolviendo las palabras que hemos estimado tienen 20 caracteres como mucho.

8. Obtener la palabra más larga de un trozo de texto

La obtención del *máximo* es un caso particular de la reducción con la función de acumulación adecuada. Lo mismo ocurre con el mínimo, suma, producto, all, any, none, contador, etc. En todos ellos la reducción por la izquierda resulta en el mismo valor que la reducción por la derecha porque los operadores involucrados son asociativos y conmutativos.

```
void palabra_mas_larga(char * text){
	char p[20];
	iterator p3 = text_to_iterable_string_fix_tam(text,
		" ;.", 20);
	char * s = (char *) reduce_left(&p3,p,max_len);
	printf("%s\n",s);
}
```

Hemos usado la función:

```
bool max_len(char * p1, char * p2){
	int r1 = strlen(p1);
	int r2 = strlen(p2);
	int r = order(&r1,&r2,&int_type);
	if(r < 0) strcpy(p1,p2);
	return false;
}
```

9. *Obtener la primera palabra en una cadena de caracteres que comienza por un carácter dado*

Podemos abordar el problema con una acumulación con la función de acumulación

```
char c;
bool p_palabra(char * out, char * in){
       bool r = in[0] == c;
       if(r) strcpy(out,in);
       return r;
}
```

Inicializamos la base del acumulador para que nos indique un mensaje por defecto.

```
void primera_palabra() {
       char p[20] = "No existe";
       c = 'x';
       iterator p3 = text_to_iterable_string_fix_tam(text,
              " ;.", 20);
       void * s = accumulate_left(&p3,p,p_palabra);
       printf("%s\n",s);
       p3 = text_to_iterable_string_fix_tam(text, " ;.", 20);
       s = accumulate_right(&p3,p,p_palabra);
       printf("%s\n",s);
}
```

Podemos observar cómo, en este caso, la acumulación por la izquierda produce un resultado distinto a la acumulación por la derecha.

Agrupación de iterables

Hay un conjunto de tareas de uso frecuente que son las agrupaciones de iterables; es decir, formar grupos con los elementos de un iterable. Los grupos podemos definirlos mediante el valor devuelto por una función aplicada a cada elemento del iterable. Estas operaciones son casos particulares de acumulación.

Tipos de agrupación de iterables

Vamos a considerar varias de estas operaciones: agrupación con reducción, agrupación en listas y agrupación en conjuntos.

La agrupación en listas consiste en formar grupos en forma de listas a partir de los valores devueltos por una función. La función será de la forma *void* (*f_key)(void *out, void *in).* Hay que especificar el tipo de las claves. El resultado será una tabla hash con claves del tipo especificado y como valores listas de elementos del iterable.

La agrupación en conjuntos forma grupos en forma de conjuntos a partir de los valores devueltos por una función similar al caso anterior. En este caso los grupos, al ser conjuntos, no tendrán elementos repetidos. Hay que especificar el tipo de las claves. El resultado será una tabla hash con

claves del tipo especificado y como valores conjuntos de elementos del iterable.

Por último, la agrupación con reducción forma grupos a partir de los valores devueltos por una función similar a los casos anteriores y posteriormente acumula todos los elementos del grupo mediante una fución de acumulación. Esta función de acumulación es de la forma *bool (*add)(void *out, const void *e).* Hay que especificar el tipo de las claves. El resultado será una tabla hash con claves del tipo especificado y valores del tipo de los elementos del iterable.

La implementación de *iterable_grouping_reduce* es como sigue:

```
map iterable_grouping_reduce(iterator *st,
                    type * key_type,
              void* (*f_key)(void *out, void *in),
              bool (*add)(void *out, const void *e)) {
       type *value_type = st->type;
       char mem_key[key_type->size];
       char mem_value[value_type->size];
       map ht = map_empty(key_type, value_type);
       while (iterable_has_next(st)) {
              void *next = iterable_next(st);
              void *key = f_key(mem_key, next);
              void * e = map_get(&ht, key);
              if(e == NULL){
                     map_put(&ht, key, next);
              } else {
                     copy(mem_value,next,value_type->size);
                     add(mem_value,next);
                     map_put(&ht, key, mem_value);
              }
       }
       return ht;
}
```

Algunas cuestiones a tener en cuenta. Los tipos de las claves y los valores son *key_type* y *value_type*. Hay que reservar memoria para los cálculos que harán las fuciones *f_key* y *add*. La correspondiente memoria es *mem_key* y *mem_value*. Tras tener en cuenta lo anterior se trata de comprobar si la clave asociada al elemento procedente del iterable está en la tabla o no (valor devuelto NULL). Si no lo está se añade el par clave

elemento. Si está se calcula el valor acumulado del nuevo elemento junto con el valor previamente asociado a la clave y se actualiza la tabla.

Este tipo de agrupación se puede generalizar añadiendo una función de transformación a los elementos del iterable. Se trata ahora de formar grupos, transformar los elementos de cada grupo mediante la función de transformación y acumular los resultados.

La función cuya implementación queda como ejercicio tendrá la cabecera siguiente:

```
hash_table iterable_grouping_reduce_map(iterator * st,
       type * key_type, type * value_type,
       void * (*f_key)(void * out, void * in),
       bool (*add)(void * out, const void * e),
       void * (*f_map)(void * out, void * in));
```

La implementación de *iterable_grouping_list* forma grupos en forma de listas usando una función que calcula las claves de los grupos. Para ello va recorriendo los elementos del iterable y va poniendo los pares clave valor en un *list_multimap* que hemos visto en el capìtulo anterior.

```
list_multimap iterable_grouping_list(iterator * st,
              void * (*f_key)(void * out, void * in),
              type * key_type){
       char mem[100];
       type * value_type = st->type;
       list_multimap lm =
              list_multimap_empty(key_type,value_type);
       while(iterable_has_next(st)){
              void * e = iterable_next(st);
              void * key = f_key(mem,e);
              list_multimap_put(&lm,key,e);
       }
       return lm;
}
```

La implementación de *iterable_grouping_set* forma grupos en forma de listas usando una función que calcula las claves de los grupos. Para ello va recorriendo los elementos del iterable y va poniendo los pares clave valor en un *set_multimap* que hemos visto en el capítulo anterior.

```
set_multimap iterable_grouping_set(iterator * st,
             void * (*f_key)(void * out, void * in),
             type * key_type){
       char mem[100];
       type * value_type = st->type;
       set_multimap sm =
             set_multimap_empty(key_type,value_type);
       while(iterable_has_next(st)){
             void * e = iterable_next(st);
             void * key = f_key(mem,e);
             set_multimap_put(&sm,key,e);
       }
       return sm;
}
```

Se dejan como ejercicio funciones similares que transformen y posiblemente filtren los elementos del iterable antes de agregarlos a la lista o conjunto.

Ejemplos de agrupación de iterables

1. *Dado un iterable de enteros aleatorios, obtener la suma de los cuadrados de los grupos de restos al ser divididos por n*

```
void suma_cuadrados_restos(int n) {
       remainder_base = n;
       iterator r = iterable_random_int(100,0,20);
       map ms = iterable_grouping_reduce_map(&r,
             &long_type, &long_type,
             remainder_f, sum, square_long_f);
       iterator r3 = map_items_iterable(&ms);
       iterable_to_console_sep(&r3, ",", "{", "}");
}
```

Hemos usado la función *remainder_f* para calcular el resto de la división entera.

```
long remainder_base;
long * remainder_f(long * out, long * in){
	*out = (*in)%remainder_base;
	return out;
}
```

2. *Acumular un iterable en un set*

Esta es una operación que se repite a menudo para obtener los elementos distintos de un iterable

```
set iterable_to_set(iterator * st) {
	set r = set_empty(st->type);
	while (iterable_has_next(st)) {
		void * e = iterable_next(st);
		set_add(&r, e);
	}
	return r;
}
```

3. *Acumular un iterable en un multiset*

Esta es una operación que podemos usar para encontrar las frecuencias con que aparecen los elementos en un iterable. Debemos tener en cuenta que un *multiset* es un conjunto donde se pueden repetir los elementos y por lo tanto podemos usarlo para guardar las veces que aparece un elemento dado.

```
multiset iterable_to_multiset(iterator * st) {
	multiset r = multiset_empty(st->type);
	while (iterable_has_next(st)) {
		void * e = iterable_next(st);
		multiset_add(&r, e);
	}
	return r;
}
```

4. *Obtener la frecuencia de las palabras de un texto contenido en un fichero*

Para abordar esta tarea se trata de obtener un iterable con las palabras de un texto y posteriormente acumularlo en un multiset.

El iterable con las palabras lo obtenemos a partir del iterable de las líneas del fichero, filtrando solo las líneas no vacías, y posteriormente dividiendo cada línea en palabras con la operación flatmap.

```
multiset frecuencias_de_palabras(char * file){
	iterator r = iterable_words_in_file(
		file,100,20," ,;.()");
	multiset ms = iterable_to_multiset(&r);
	return ms;
}
```

5. *Obtener las frecuencias de los grupos de elementos de un iterable definidos por una clave*

```
multiset iterable_to_multiset_groups(iterator * st,
              type * key_type,
              void * (*f_key)(void * out, void * in)){
       char mem[key_type->size];
       multiset ms = multiset_empty(key_type);
       while(iterable_has_next(st)){
              void * e = iterable_next(st);
              void * key = f_key(mem,e);
              multiset_add(&ms,key);
       }
       return ms;
}
```

Ahora buscamos obtener un multiset con las frecuencias de los grupos definidos por una función que calcula una clave para cada elemento.

Árboles

Un tipo recursivo es aquel que tiene alguna propiedad con valores del mismo tipo. Veremos aquí ejemplos de estos y técnicas para su diseño e implementación.

Los tipos de datos recursivos dan lugar a algoritmos recursivos específicos. Estos algoritmos están guiados por la estructura del tipo de datos recursivo.

Veamos el tipo árbol. Un árbol es un tipo de datos jerárquico, con una raíz y subárboles hijos. Vamos a definir un árbol que puede ser vacío, puede tener un elemento, que llamaremos *etiqueta*, o puede ser un árbol *n-ario* que tiene n hijos. Estos tres tipos de árboles son los que vamos a permitir. Podemos decir que cada tipo es un subtipo de árbol. Sea *e* una etiqueta de tipo *E* y *t0, t1,..., tn−1* árboles que los llamamos hijos y al árbol resultante padre de los mismos. Un árbol que no tiene padre lo llamamos raíz.

En segundo lugar, definimos un conjunto propiedades del tipo y de cada uno de los subtipos.

Definimos los predicados *is_empty(), is_leaf(), is_nary()* que serán *true* si el tipo del árbol es el adecuado. El valor devuelto por *get_subtype()* indicará el subtipo del árbol dependiendo del constructor usado.

Algunas nociones sobre árboles:

- Un árbol binario se dice *ordenado* si es vacío, hoja o es binario, sus hijos están ordenados y su etiqueta es mayor que todas las de su hijo izquierdo y menor que las de su hijo derecho.
- Un árbol binario es *equilibrado* si es vacío, es hoja o si es binario, sus hijos están equilibrados y sus alturas no difieren en más de una unidad.
- *El tamaño de un árbol* (*size*) es el número de etiquetas que tiene.
- Un *camino en un árbol* es cualquier secuencia de árboles*, t0, t1, …, tr−1,* tal que cada uno es padre del siguiente. Entre cada padre y uno de sus hijos existe una arista. La longitud del camino se define como el número de sus aristas.
- La *altura de un árbol* (*height*) se define como la longitud del camino más largo que comienza en la raíz y termina en una hoja. La altura de una hoja será de cero. La altura de un árbol se define como la altura de su raíz.
- La *profundidad de un árbol* dentro de otro se define como la longitud del camino que comienza en la raíz y termina en él. La profundidad de la raíz es cero. A la profundidad de un árbol también se la denomina *nivel* del árbol en el árbol que lo contiene.
- Un árbol es *raíz* si no tiene padre.

A partir de la definición del tipo podemos diseñar métodos para implementar sus propiedades. Este código estará estructurado alrededor de un *switch* sobre los valores de la propiedad *tree_type*. Los algoritmos sobre árboles, y en general sobre todos los tipos recursivos, son algoritmos recursivos. Para un árbol, los casos base son los árboles *empty, leaf*. El caso recursivo el árbol *n-ary (*o *binary* en el caso de árboles binarios). Como en todos los algoritmos recursivos la solución de los casos base debe ser inmediata, la solución del caso recursivo se obtendrá componiendo las soluciones asociadas a cada uno de los hijos.

Vamos a diseñar dos tipos de árboles: binarios y n-arios. Los constructores del tipo nos permiten contruir un árbol binario vacío, un árbol binario con una etiqueta y sin hijos o un árbol binario con dos hijos.

Las propiedades nos permiten acceder a la etiqueta, si la tiene, y a los hijos izquierdo y derecho si existen.

Un ejemplo de árbol nario es:

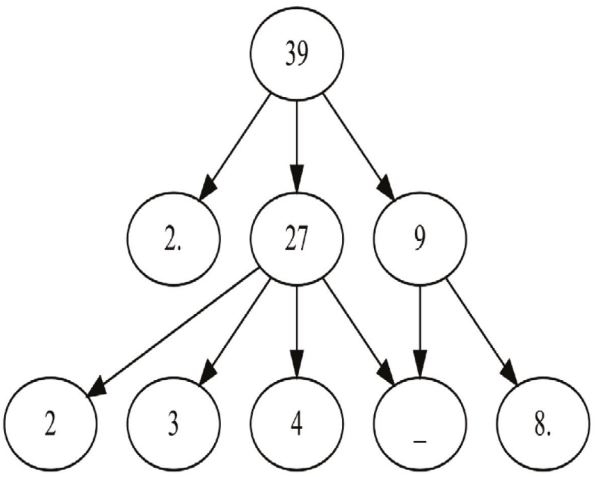


Ilustración 1: Un *árbol nario*

Implementación de los árboles

La implementación de los árboles binarios es:

```
typedef enum{Empty_Binary_Tree, Leaf_Binary_Tree, Binary_Tree}
      binary_tree_subtype;
```

```
typedef struct bt {
      binary_tree_subtype tree_type;
      type * label_type;
      void * label;
      struct bt * left;
      struct bt * right;
}binary_tree;
```

Los constructores para estos árboles binarios son:

```
binary_tree * binary_tree_empty();
binary_tree * binary_tree_leaf(void * label, type * type);
binary_tree * binary_tree_new(void * label, type * type,
      binary_tree * left, binary_tree * right);
```

Algunas propiedades:

```
binary_tree_subtype binary_tree_subtype(
       const binary_tree * tree);
void * binary_tree_label(const binary_tree * tree);
binary_tree * binary_tree_left(const binary_tree * tree);
binary_tree * binary_tree_right(const binary_tree * tree);
```

Disponemos de algunos predicados para decidir el tipo de árbol binario.

```
bool binary_tree_is_empty(const binary_tree * tree);
bool binary_tree_is_leaf(const binary_tree * tree);
bool binary_tree_is_binary(const binary_tree * tree);
```

Los árboles n-arios tienen un diseño similar:

```
typedef enum{Empty_Tree, Leaf_Tree, Nary_Tree} tree_subtype;
```

```
typedef struct tr {
       tree_subtype tree_type;
       type * label_type;
       void * label;
       int num_children;
       struct tr ** children;
}tree;
```

Los constructores del tipo nos permiten construir un árbol vacío, un árbol con una etiqueta y sin hijos o un árbol binario con varios hijos.

```
tree * tree_empty();
tree * tree_leaf(void * label, type * type);
tree * tree_new(void * label, type * type,
              int num_children, tree ** children);
```

Las propiedades nos permiten acceder a la etiqueta, si la tiene, al número de hijos y a uno de ellos en particular.

```
tree_subtype tree_get_subtype(const tree * tree);
void * tree_label(const tree * tree);
void * tree_label_f(void * out, const tree * tree);
tree * tree_get_child(const tree * tree, int i);
int tree_child_number(const tree * tree);
```

Los árboles contarán con un montón de memoria dinámica: uno para los árboles binarios y otro para los n-arios. Son las variables:

```
extern heap heap_binary_tree;
extern heap heap_tree;
```

Veamos la implementación de algunas de las funciones anteriores. El resto pueden verse en el repositorio. Veamos la función *tree_new.* Esta función usa *tree_new_me* que gestiona la memoria dinámica necesaria.

```
tree * tree_new_me(void * label, type * label_type,
       int num_children, tree ** children, heap * hp){
       void * lb = heap_copy_and_mem(hp,label,
              label_type->size);
       tree ** ch = heap_copy_and_mem(hp,children,
              num_children*sizeof(tree *));
       tree t = {Nary_Tree,label_type,lb,num_children,ch};
       return (tree *) heap_copy_and_mem(hp,&t,sizeof(tree));
}
```

Con la función anterior el constructor resulta ser:

```
tree * tree_new(void * label, type * type_label,
       int num_children, tree ** children) {
       return tree_new_me(label,type_label,
              num_children,children,&heap_tree);
}
```

Los árboles que hemos diseñado son inmutables. En particular un árbol concreto tiene un número fijo de hijos.

Algunas propiedades tienen precondiciones. La propiedad *tree_get_child* existe solo para los árboles n-arios. Las precondiciones las establecemos con la función *check_argument*. La precondición también incluye que el hijo pedido sea menor que el número de hijos.

```
tree * tree_get_child(const tree * tree, int child){
       check_argument(tree->tree_type == Nary_Tree,
              __FILE__,__LINE__,"el árbol no es nario");
       check_argument(child < tree->num_children,
              __FILE__,__LINE__,"el árbol no tiene ese hijo");
       return tree->children[child];
}
```

Ejemplos de árboles

1. *Calcular el tamaño de un árbol binario*

```
int binary_tree_size(const binary_tree * tree){
       int r;
       switch(tree->tree_type){
       case Empty_Binary_Tree: 0; break;
       case Leaf_Binary_Tree: r = 1; break;
       case Binary_Tree: r = 1 +
              binary_tree_size(binary_tree_left(tree))+
              binary_tree_size(binary_tree_right(tree));
       }
       return r;
}
```

Como vemos, los casos base son que el árbol sea vacío o una hoja, en cuyo caso el número de etiquetas es cero o uno respectivamente. En el caso recursivo el número de etiquetas de un árbol es la suma del número de etiquetas de sus hijos más uno.

2. *Calcular la altura de un árbol n-ario.*

Para el cálculo de la altura los casos base son, como antes, el árbol vacío o una hoja, en cuyo caso la altura es cero. En el caso recursivo la altura de un árbol es el máximo de las alturas de sus hijos más uno.

```
int binary_tree_height(const binary_tree * tree){
	int r;
	switch(tree->tree_type){
	case Empty_Binary_Tree: r = 0; break;
	case Leaf_Binary_Tree: r = 0; break;
	case Binary_Tree: r = 1 + MAX(
			binary_tree_height(binary_tree_left(tree)),

	binary_tree_height(binary_tree_right(tree)));
	}
	return r;
}
```

3. *Transformar un árbol binario en una lista en preorden*

```
list * tree_to_list_private(const tree * tree, list * ls) {
	switch (tree->tree_type) {
	case Empty_Tree: break;
	case Leaf_Tree: list_add_pointer(ls,
			binary_tree_label(tree)); break;
	case Nary_Tree:
		list_add_pointer(ls,tree_label(tree));
		for(int i=0; i < tree_child_number(tree); i++) {
			tree_to_list_private(
				tree_get_child(tree,i),ls);
		}
	}
	return ls;
}
```

Como podemos ver, la función anterior, privada, tiene un parámetro acumulador. Una lista que hay que inicializar a lista vacía en la llamada principal que vemos a continuación. Asumimos que los datos ya tienen memoria en el árbol por lo que es suficiente añadir un puntero a la lista sin obtener memoria adicional.

```
list tree_to_list_preorder(const tree * tree){
	list ls = list_empty(tree->label_type);
	tree_to_list_private(tree,&ls);
	return ls;
}
```

4. Transformar un árbol aplicando una función a cada una de sus etiquetas

```
tree* tree_map(tree* tree_in, type * tipo_out,
		void* fmap(void* out, const void* in)) {
	tree *res = NULL;
	char mem[tipo_out->size];
	switch (tree_get_subtype(tree_in)) {
	case Empty_Tree: res = tree_empty();break;
	case Leaf_Tree:
		etq_in = tree_label(tree_in);
		etq_out = fmap(mem, etq_in);
		res = tree_leaf(etq_out, tipo_out);
		break;
	case Nary_Tree:
		etq_in = tree_label(tree_in);
		etq_out = fmap(mem, etq_in);
		tree* children[tree_in->num_children];
		for (int i = 0; i < tree_in->num_children; i++) {
			children[i] = tree_map(tree_get_child(
				tree_in, i), tipo_out, fmap);
		}
		res = tree_new(etq_out, tipo_out,
			tree_in->num_children,children);
	}
	return res;
}
```

Matrices

Veamos un ejemplo: obtener la multiplicación de dos matrices:

- De forma iterativa
- De forma recursiva dividiéndolas en cuatro submatrices

Para enfocar el problema proponemos un sistema de coordenadas para nombrar las casillas de la matriz. Nombremos las filas de arriba abajo por i y las columnas por j de tal forma que (0,0) es la casilla superior izquierda. Una matriz vendrá definida por su número de filas y su número de columnas: : $nf \times nc$.

Una submatriz es una vista de una matriz definida por un vértice superior izquierdo y un tamaño de filas y columnas. Es una idea similar a *sublist*. Una matriz la podemos dividir en cuatro vistas que denominaremos a*, b, c, d*. La vista *a* vendrá definida por $(0, nf/2, 0, nc/2)$ y de forma similar el resto. Cada submatriz tiene asociadas unas coordenadas locales tales que el vértice superior izquierdo es (0,0).

Llamemos *matrix* al tipo que representa una matriz. Las funciones para operar con este tipo son:

- *matrix matrix_of_array(void * a, int nf, int nc, type * type_element):* Una nueva matriz definida por un array.
- *matrix matrix_of(int nf, int nc, type * type_element):* Una nueva matriz.
- *matrix matrix_of_file(char * file, type * type, int nf, int nc):* Una nueva matriz con los datos leídos de un fichero.
- *void * matrix_get(matrix * s, int f, int c):* El valor de la casilla i, j.
- *void matrix_set(matrix * s, int f, int c, void * value):* Actualización de la casilla i, j con value.
- *matrix_views views_of_matrix(matrix * s*): Las cuatro submatrices de una matriz.
- *matrix matrix_compose(matrix * m0, matrix * m1, matrix * m2, matrix * m3):* Construye una matriz a partir de cuatro submatrices.

El código de la suma y la multiplicación iterativa de matrices es sencilllo de abordar. La suma iterativa asumiendo que la matriz es de valores enteros es de la forma que vemos abajo.

Como los datos de las casillas son de tipo *void ** debemos convertirlos a int para hacer las operaciones aritméticas necesarias. Esto lo conseguimos con la funcion *int to_int(void * in).*

```
matrix sum_iterativa(matrix * in1, matrix * in2) {
    matrix out = matrix_of(in1->nf,in1->nc,in1->type);
    int i, j;
    for (i = 0; i < in1->nf; i++) {
        for (j = 0; j < in1->nc; j++) {
             int val = to_int(matrix_get(in1,i,j))+
                   to_int(matrix_get(in2,i,j));
            matrix_set(&out,i,j,&val);
        }
    }
    return out;
}
```

Y la multiplicación iterativa asumiendo que la matriz es de valores enteros:

```
matrix multiply_iterativa(matrix * in1, matrix * in2) {
    matrix out = matrix_of(in1->nf,in2->nc,in1->type);
    int i, j, k;
    for (i = 0; i < in1->nf; i++) {
        for (j = 0; j < in2->nc; j++) {
             int v = 0;
            matrix_set(&out,i,j,&v);
            for (k = 0; k < in1->nc; k++){
             int val = (to_int(matrix_get(in1,i,k))*
                    to_int(matrix_get(in2,k,j)))+
                           to_int(matrix_get(&out,i,j));
             matrix_set(&out,i,j,&val);
            }
        }
    }
    return out;
}
```

Con estos elementos podemos diseñar la suma y la multiplicación recursiva de matrices.

Para conseguirlo la suma obtenemos las cuatro submatrices y operamos recursivamente de la forma:

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} + \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} a1+a2 & b1+b2 \\ c1+c2 & d1+d2 \end{pmatrix}$$

Para la multiplicación recursiva obtenemos las cuatro submatrices y operamos recursivamente de la forma:

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} a1*a2+b1*c2 & a1*b2+b1*d2 \\ c1*a2+d1*d2 & c1*b2+d1*d2 \end{pmatrix}$$

Cálculos sobre un libro

El objetivo es leer un libro de un fichero y hacer cálculos sobre las palabras que contiene, en que líneas aparecen, ordenarlas por frecuencias, etc. Se usará el fichero *quijote.txt* para hacer los cálculos, que está disponible en el repositorio.

Se supone que las palabras están separadas una de otras por separadores que podemos definir. Los separadores que usaremos son *" ,;.()"* aunque podemos añadir o eliminar separadores.

Queremos implementar las funciones siguientes. Indicamos los tipos en la forma Java pero lo implementaremos en C.

- *numeroDeLineas(file:String):int*
- *numeroDePalabrasDistintas(file:String):int*
- *palabrasNoHuecas(file:String):Set<String>*
- *longitudMediaDeLineas(file:str):float*
- *numeroDeLineasVacias(file:str):int*
- *lineaMasLarga(file:str):String*
- *primeraLineaConPalabra(file:str,palabra:str):int*
- *lineaNumero(file:str,n_int):String*
- *frecuenciasDePalabras(file:String)*:Map<String,*Integer*>, Frecuencia de cada palabra. Ordenadas por palabras

- *palabrasPorFrecuencias(file:String:Map<Integer,Set<String>>,* palabras agrupadas por sus frecuencias de aparición. Ordenadas por frecuencias
- *lineasDePalabra(file:String):Map<String,Set<Integer>>,* grupos de líneas donde aparece cada palabra.

Ahora vamos a resolver el problema como un conjunto de funciones. Algunos detalles se incluyen aquí y el resto se puede ver en el repositorio. Diseñaremos previamente varios iterables sobre ficheros.

Ya hemos diseñado en el capítulo de iteradores un iterador que nos proporciona las líneas de un fichero y otro que nos da las palabras del mismo.

Diseñaremos otros para abordar diversas cuestiones sobre un libro.

1. *Encontrar el número de líneas de un fichero*

Usaremos el iterador que nos proporciona las líneas de un fichero

```
int numero_de_lineas(char * file) {
	iterator r = iterable_file_string_fix_tam(file,100);
	int n = iterable_size(&r);
	iterable_free(&r);
	return n;
}
```

2. *Encontrar el número de líneas vacías de un fichero*

Filtramos el iterable sobre líneas de un fichero para quedarnos solo con las líneas vacías

```
int numero_de_lineas_vacias(char * file) {
	iterator r = iterable_file_string_fix_tam(file,100);
	iterator r2 = iterable_filter(&r,string_fix_all_space);
	int n = iterable_size(&r2);
	iterable_free(&r);
	return n;
}
```

El predicado *string_fix_all_space* comprueba que todos los caracteres de una línea son espacios. Es lo que llamamos una línea vacía.

```
bool string_fix_all_space(char * in){
	bool r = true; int i =-1;
	while(in[++i] != '\0' && r)
		r = isspace(in[i]);
	return r;
}
```

3. *Encontrar el número de palabras de un fichero*

Usaremos el iterador que nos va proporcionando las palabras de un fichero.

```
int numero_de_palabras(char * file) {
	iterator r = iterable_words_in_file(file,
			100,20," ,;.()");
	int n = iterable_size(&r);
	iterable_free(&r);
	return n;
}
```

4. Encontrar el número de palabras no huecas de un fichero

Definimos como palabras huecas las incluidas en un fichero dado. Son palabras que por alguna razón no nos interesan.

Debemos leer el conjunto de palabras huecas de un fichero y posteriormente filtrar las palabras que no pertencen a ese conjunto. Hemos de tener en cuenta pasar los parámetros adicionales al predicado mediante variables globales.

En primer lugar, leemos el fichero de palabras huecas con el correspondiente iterador y construimos un conjunto a partir de él.

```
set palabras_huecas(char * file){
	iterator r1 = iterable_file_string_fix_tam(file,30);
	set s = iterable_to_set(&r1);
	return s;
}
```

```
set sg;
bool no_hueca(char * in){
	return set_contains(&sg,in);
}
int numero_de_palabras_no_huecas(char * file) {
	sg = palabras_huecas("ficheros/palabras_huecas.txt");
	iterator r = iterable_words_in_file(
		file,100,20," ,;.()");
	iterator r2 = iterable_filter(
		iterable_copy(&r),no_hueca);
	return iterable_size(&r);
}
```

5. Encontrar la primera línea donde aparece una palabra

Usamos un iterador que nos proporciona pares de palabras y líneas visto anteriormente para encontrar el primer par en que aparece una palabra. El campo *counter* del par nos proporciona el resultado. Encontrar el

primer par lo hacemos con iterable_first. Y asumimos que si no encuentra la palabra el resultado será -1.

```
int primera_linea_con_palabra(char * file, char * word){
	strcpy(word_g,word);
	iterator r = iterable_words_and_line_in_file(f
		ile,100,20," ,;.()");
	enumerate * p =
		(enumerate *) iterable_first(&r,es_palabra);
	if(p != NULL ) return p->counter;
	else return -1;
}
```

Usando *iterable_last* podemos encontrar la última línea en la que aparece la palabra. El predicado *es_palabra* será como se indica:

```
string_fix word_g;
bool es_palabra(enumerate * in){
	return strcmp(in->value,word_g) == 0;
}
```

6. *Encontrar la línea n*

```
int n_g;
bool es_numero(enumerate * in){
	return n_g == in->counter;
}
```

```
char * line_n(char * file, int n){
	n_g = n;
	iterator r = iterable_file_string_fix_tam(file,100);
	iterator r2 = iterable_enumerate(&r);
	enumerate * p = (enumerate *)
		iterable_first(&r2,es_numero);
	if(p != NULL ) return p->value;
	else return NULL;
}
```

Hemos usado *iterable_enumerate* para formar pares número-línea.

Posteriormente se trata de encontrar el primer par con el componente *counter* igual al número de línea especificado.

7. Encontrar la longitud media de las líneas del fichero

La primera tarea es diseñar un acumulador que calcule la media de un iterable de elementos numéricos.

```
double iterable_average(iterator * st) {
	double r = 0;
	int n = 0;
	Number nm = numeric_type(st->type);
	while (iterable_has_next(st)) {
		void * e = iterable_next(st);
		switch(nm) {
		case INTEGER: r +=  *(int *) e; break;
		case DOUBLE: r +=  *(double *) e; break;
		case LONG: r +=  *(long *) e; break;
		case FLOAT: r +=  *(float *) e; break;
		}
		n = n+1;
	}
	check_argument(n != 0, __FILE__, __LINE__,"el número de
elementos es cero y es %d",n);
	return r/n;
}
```

Para poder hacer las operaciones numéricas necesarias debemos hacer la conversión adecuada desde el puntero generico. Para ello diseñamos una función que obtenga el tipo numérico de los elementos del iterable. El tipo Number es un tipo que nos indica cual de los tipos numéricos es.

```
typedef enum {INTEGER,LONG,FLOAT,DOUBLE} Number;
```

La función siguiente obtiene el tipo a partir del tamaño en memoria de los elementos del mismo

```
Number numeric_type(type *t) {
	Number r;
	int sz = t->size;
	if(sz == int_type.size) r = INTEGER;
	else if(sz == double_type.size) r = DOUBLE;
	else if(sz == long_type.size) r = LONG;
	else if(sz == float_type.size) r = FLOAT;
	else check_argument(false,__FILE__,__LINE__,
		"No es un tipo numerico");
	return r;
}
```

Con estos elementos la longitud media de las líneas se obtiene combinando iteradores y acumuladores.

```
int * to_len(int * out, int * in){
	*out = strlen(in);
	return out;
}
```

```
double longitud_media(char * file){
	iterator r = iterable_file_string_fix_tam(file,100);
	iterator r2 = iterable_map(&r,&int_type,to_len);
	double r3 = iterable_average(&r2);
	return r3;
}
```


8. Obtener la línea más corta que no esté vacía

La idea es recorrer las líneas, filtrar las que no son vacías y encontrar la más corta.

```
char * linea_mas_corta(char * file){
       iterator r = iterable_file_string_fix_tam(file,100);
       iterator r2 = iterable_filter(&r,
              string_fix_not_all_space);
       char * ln = (char *) iterable_min(&r2, cmp_len);
       return ln;
}
```

La función que proporciona el orden entre líneas es:

```
int cmp_len(char * e1, char * e2){
        int n1 = strlen(e1);
        int n2 = strlen(e2);
        return int_type.order(&n1,&n2,&int_type);
}
```

Coordenadas

Las coordenadas 2D y 3D son tipos muy importantes para resolver problemas donde aparecen mapas y aspectos geográficos

La distancia a lo largo de la superficie de la tierra según la fórmula de Harvesine:

$$a = sin^2(\Delta\varphi/2) + cos\ \varphi_1 * cos\ \varphi_2 * sin^2(\Delta\lambda/2)$$

$$d = 2\ R\ atan2(a, \sqrt{1-a})$$

Donde φ_1, φ_2 son las latitudes de los puntos, $\Delta\varphi$ la diferencia de latitudes y $\Delta\lambda$ la diferencia de longitudes

Coordenadas2D

Propiedades

- latitud: Double, básica, en grados
- longitud: Double, básica, en grados

Funciones

- to_radianes(c1: Coordenada2D): Coordenada2D
- distancia(c1: Coordenada2D, c2: Coordenada2D): Double, *formula de harvesine*
- cercanas(c1: Coordenada2D, c2: Coordenada2D, d:Double): Boolean
- center(coordenadas: List[Coordenada2D]): Coordenada2D

Factoría

- of(latitude:float,longitude:float): Coordenadas2D

Este tipo y el siguiente son muy reutilizables. Para su implentación seguimos el mismo esquema que para los tipos vistos anteriormente. En primer lugar definimos el tipo y las funciones para su uso:

```
typedef struct {
	double latitud;
	double longitud;
}coordenadas_2d;
```

Veamos el código de algunas de estas funciones. El resto se puede ver en el repositorio. En primer lugar, la distancia usando la fórmula de Harvesine y la comprobación sobre si están cerca.

```
double coordenadas_2d_distance(coordenadas_2d c1,
		coordenadas_2d c2){
	double radio_tierra = 6373.0;
	coordenadas_2d c1R = coordenadas_2d_to_radians(c1);
	coordenadas_2d c2R = coordenadas_2d_to_radians(c2);
	double incLat  = c2R.latitud-c1R.latitud;
	double incLong = c2R.longitud-c1R.longitud;
	double a = pow(sin(incLat/2),2)+
		cos(c1R.latitud)*
		cos(c2R.latitud)*pow(sin(incLong/2),2);
	double c = 2 * atan2(sqrt(a),sqrt(1 - a));
	return radio_tierra*c;
}
```

```
bool coordenadas_2d_cercanas(coordenadas_2d c1,
              coordenadas_2d c2, double d){
       return coordenadas_2d_distance(c1,c2) <=d;
}
```

La conversión a cadena de caracteres.

```
char * coordenadas_2d_tostring(coordenadas_2d * c,
              const char * mem) {
       sprintf(mem,"(%lf,%lf)",c->latitud,c->longitud);
       return mem;
}
```

El cálculo de la coordenada media:

```
double * lat(double * out, const coordenadas_2d * c){
       return &c->latitud;
}
double * lng(double * out, const coordenadas_2d * c){
       return &c->longitud;
}
coordenadas_2d  coordenadas_2d_center(list lc){
       iterator it = list_iterable(&lc);
       iterator ilat = iterable_map(&it,&double_type,lat);
       double average_lat = iterable_average(&ilat);
       it = list_iterable(&lc);
       iterator ilng = iterable_map(&it,&double_type,lng);
       double average_lng = iterable_average(&ilng);
       coordenadas_2d c =
              coordenadas_2d_of(average_lat,average_lng);
       iterable_free(&it);
       iterable_free(&ilat);
       iterable_free(&ilng);
       return c;
}
```

Coordenadas3D

Propiedades

- latitud: Double, básica, en grados
- longitud: Double, básica, en grados

- altitud: Double, básica, en kms

Funciones

- coordenada2d: Coordenada2D
- distance(c1: Coordenada3D, c2: CoordenadaDD): float

Factoría

- of(latitud:float,longitud:float,altitud:Double) Coordenadas3D, en grados y km

Este tipo se implementa siguiendo la misma pauta.

```
typedef struct {
	double latitud;
	double longitud;
	double altitud;
}coordenadas_3d;
```

Y algunas funciones.

```
coordenadas_3d coordenadas_3d_of(double latitude,
		double longitude, double altitud){
	coordenadas_3d c = {latitude, longitude, altitud};
	return c;
}

double coordenadas_3d_distance(coordenadas_3d c1,
		coordenadas_3d c2){
	coordenadas_2d c12D = coordenadas_2d_of(c1.latitud,
		c1.longitud);
	coordenadas_2d c22D = coordenadas_2d_of(c2.latitud,
		c2.longitud);
	return sqrt(pow(coordenadas_2d_distance(c12D,c22D),2)+
		pow(c1.altitud-c1.altitud,2));
}
```

Ruta

Veamos varios ejemplos de problemas más completos. En cada problema de una determinada complejidad la primera tarea es abordar el diseño de tipos que necesitaremos. Esta tarea es básica para una programación de calidad. También lo es el diseño de funciones reutilizables. Un primer ejemplo es el de cálculos sobre rutas de las que conocemos la secuencia de sus coordenadas.

Las rutas GPS (también llamadas tracks) contienen información sobre los puntos de un determinado trayecto. Casi cualquier dispositivo que tenga GPS (móviles, relojes, pulseras fitbit, ...) permite registrar esta información. Los datos que acompañan el ejercicio se corresponden con una ruta real.

Queremos hacer una serie de cálculos sobre los datos de entrada. Todos esos cálculos los acumularemos en tipo Ruta descrito más abajo.

El formato de entrada con el que trabajaremos contempla una línea por cada punto del trayecto que incluye cuatro informaciones:

- *tiempo* en el que fue tomada la medición en horas, minutos y segundos
- *Latitud* del punto en el que fue tomada la medición en grados

- *longitud* del punto en el que fue tomada la medición en grados
- altitud del punto en el que fue tomada la medición en metros

He aquí un fragmento de dicho fichero con las cinco primeras líneas:

```
00:00:00,36.74991256557405,-5.147951105609536,712.2000122070312
00:00:30,36.75008556805551,-5.148005923256278,712.7999877929688
00:01:30,36.75017642788589,-5.148165263235569,714.0
00:02:04,36.750248931348324,-5.148243047297001,714.5999755859375
00:02:19,36.750430315732956,-5.148255117237568,715.0
```

Marca

Propiedades

- tiempo: Time, básica
- latitud: Double, básica, en grados
- longitud: Double, básica, en grados
- altitud: Double, básica, en Km

Representación:

- (00:00:30,2.3,0.5,7.9)

Factoría

- parse(linea:String)->Marca
- of(tiempo:Time,latitud: Double,longitud: Double,altitud:Double))->Marca

Intervalo

Propiedades

- principio: Marca, básica
- fin: Marca, básica
- desnivel: Double, derivada, km
- velocidad: Double, derivada, km/hora

- longitud: Double, derivada, km
- tiempo: Double, derivada, km

Representación:

- ((00:00:30,2.3,0.5,7.9), (00:00:35,2.4,0.6,8.1))

Factoría

- of(p: Marca, f: Marca)->Marca,

Ruta

Propiedades

- marcas: List<Marca>, básica
- tiempo: , derivada
- longitud: Double, derivada
- velocidad: Double,
- velocidad_en_intervalo(i:Integer): Double
- desnivel_en_intervalo(i:int):Double
- desnivel_en_intervalo(i:int):Double
- desnivel_acumulado: Pair<Double,Double>

Representación:

- (00:00:30,2.3,0.5,7.9) ..., (00:00:35,2.4,0.6,8.1)}

Factoría

- leer_de_fichero(fichero:String)->Ruta

Veamos el código de cada uno de ellos. En primer lugar el tipo Marca.

```
typedef struct {
	time_t time;
	coordenadas_3d coordenadas;
}marca;
```

En este tipo usamos coordenadas_3d y time_t ya diseñados previamente. Otras funciones asociadas al tipos son:

```
marca marca_of(time_t time, coordenadas_3d coordenadas);
marca * marca_parse(marca * out, const char * in);
char * marca_tostring(marca * m, char * mem);
```

La función de factoría marca_of puede verse en el repositorio. La función marca_parse convierte una línea del fichero comentado arriba en una marca. Como en la línea del fichero se encuentran cuatro campos separados por comas usamos *iterable_split_text* para obtener una lista con los cuatro campos.

Hacemos la comprobación de que los campos son cuatro. Posteriormente realizamos el parsing para cada campo y construimos la marca.

```
marca * marca_parse(marca * out, const char * in) {
	iterator it = iterable_split_text(in,",");
	list ls = iterable_to_list(&it);
	char mens[30];
	sprintf(mens,"numero de campos %d",list_size(&ls));
	check_argument(list_size(&ls)==4,__FILE__,
		__LINE__,mens);
	marca m;
	m.time = hour_parse(list_get(&ls,0));
	coordenadas_3d coordenadas = coordenadas_3d_of(
			double_parse_s(list_get(&ls,1)),
			double_parse_s(list_get(&ls,2)),
			double_parse_s(list_get(&ls,3)) / 1000);
	m.coordenadas = coordenadas;
	*out = m;
	iterable_free(&it);
	list_free(&ls);
	return out;
}
```

Para implementar la conversión de una marca a cadena de caracteres usamos memorias temporales, mr1, mr2, para almacenar la conversión a cadenas del tiempo y las coordenadas 3d. Usamos la función *sprintf* para conseguir el formato deseado.

```
char * marca_tostring(marca * m, char * mem){
	char mr1[Tam_String]; char mr2[Tam_String];
	sprintf(mem,"(%s,%s)",
		hours_tostring(&m->time,mr1),
		coordenadas_3d_tostring(&m->coordenadas,mr2));
	return mem;
}
```

En segundo lugar, veamos el tipo intervalo que se compone de dos marcas: principio y fin.

```
typedef struct {
	marca * p;
	marca * f;
}intervalo;

intervalo intervalo_of(marca * p, marca * f);
double intervalo_tiempo(intervalo * in);
double intervalo_longitud(intervalo * in);
double intervalo_velocidad(intervalo * in);
double intervalo_desnivel(intervalo * in);
```

Veamos el código de algunas de estas funciones.

```
double intervalo_tiempo(intervalo * in){
	time_t initial_time = in->p->time;
	time_t last_time = in->f->time;
	return time_diff_hours(initial_time,last_time);
}
```

```
double intervalo_longitud(intervalo * in) {
	coordenadas_3d c0 = in->p->coordenadas;
	coordenadas_3d c1 = in->f->coordenadas;
	double d = coordenadas_3d_distance(c0,c1);
	return d;
}
```

```
double intervalo_velocidad(intervalo * in){
	return intervalo_longitud(in)/intervalo_tiempo(in);
}
```

```
double intervalo_desnivel(intervalo * in) {
	coordenadas_3d c0 = in->p->coordenadas;
	coordenadas_3d c1 = in->f->coordenadas;
	double d = c1.altitud-c0.altitud;
	return d;
}
```

Por último, el tipo ruta y algunas de sus funciones asociadas.

```
typedef struct{
	list marcas;
}ruta;

ruta * ruta_of_file(ruta * r, char * file);
double ruta_longitud(ruta * r);
double ruta_duracion(ruta * r);
char * ruta_tostring(ruta * r, char * mem);
double ruta_velocidad_intervalo(ruta * r, int i);
double ruta_velocidad_media(ruta * r);
pair_double ruta_desnivel(ruta *r);
void ruta_printf(ruta * r);
```

Veamos la implementación de algunas de las funciones anteriores. Obtenemos una ruta a partir de un fichero. Para conseguirlo usamos un iterable que nos proporciona las líneas de un fichero. Este iterable lo tansformamos en otro de marcas con *iterable_map* y la función *marca_parse* peviamente diseñada. Por último, construimos una lista de marcas a partir del iterable anterior.

```
ruta * ruta_of_file(ruta * r, char * file){
	iterator i1 = iterable_file_string_fix(file);
	iterator i2 = iterable_map(&i1,&marca_type,marca_parse);
	list ls = iterable_to_list(&i2);
	r->marcas = ls;
	iterable_free(&i1);
	iterable_free(&i2);
	return r;
}
```

Definimos una función que nos proporciona el intervalo que ocupa la posición *i*.

```
intervalo ruta_intervalo(ruta * r, int i){
	return intervalo_of(list_get(&r->marcas,i),
		list_get(&r->marcas,i+1));
}
```

A partir de un intervalo y las funciones asociadas al intervalo es fácil encontrar el tiempo total de la ruta, su longitud y su velocidad media.

```
double ruta_longitud(ruta * r){
	double s = 0;
	for(int i =0; i<r->marcas.size-1;i++){
		intervalo in = ruta_intervalo(r,i);
		s = s + intervalo_longitud(&in);
	}
	return s;
}
```

```
double ruta_velocidad_media(ruta * r) {
	return ruta_longitud(r)/ruta_tiempo(r);
}
```

```
double ruta_tiempo(ruta *r) {
	double s = 0;
	for (int i = 0; i < r->marcas.size - 1; i++) {
		intervalo in = ruta_intervalo(r,i);
		s = s + intervalo_tiempo(&in);
	}
	return s;
}
```

El cálculo del desnivel lo hacemos siguiendo el esquema anterior y filtrando cuando el desnivel es positivo o negativo.

```
pair_double ruta_desnivel(ruta *r) {
	double dc = 0;
	double dd = 0;
	for (int i = 0; i < r->marcas.size - 1; i++) {
		intervalo in = ruta_intervalo(r, i);
		if(intervalo_desnivel(&in) > 0) dc =
			dc + intervalo_longitud(&in);
		if(intervalo_desnivel(&in) < 0) dd =
			dd + intervalo_longitud(&in);
	}
	return pair_double_of(dc,dd);
}
```

La presentación de la ruta en pantalla consiste en presentar cada marca separada por un fin de línea.

```
void ruta_printf(ruta * r){
	char mem[Tam_String];
	printf("\n");
	for(int i = 0; i < r->marcas.size; i++){
		printf("%s\n",list_get_string(&r->marcas,i,mem));
	}
}
```


Bibliografía

- El estudio de C está basado en el libro de obligada lectura *El lenguaje de programación C*, segunda edición, autores Brian W. Kernighan, Dennis M. Ritchie, Ed. Pearson Educación
- Un buen libro de consulta del lenguaje C es Practical Programming In C
- Una librería puede encontrarse en C standard library
- Otra librería de funciones puede encontrarse en Wikipedia

Miguel Toro Bonilla es doctor en Ingeniería Industrial por la Universidad de Sevilla y catedrático del Departamento de Lenguajes y Sistemas Informáticos de la misma institución. Ha ocupado cargos de responsabilidad como director de la Oficina de Transferencia de Resultados de la Investigación (OTRI) y director general de Investigación, Tecnología y Empresa de la Junta de Andalucía. Asimismo, ha tenido un papel activo en varias agencias nacionales de evaluación universitaria, entre las que se encuentran la Agencia Andaluza de Evaluación de la Calidad y Acreditación Universitaria (AGAE) y la Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA). Ha sido presidente de la Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software (Sistedes) y de la Sociedad Científica Informática de España (SCIE), que engloba a los informáticos de las universidades españolas. Su labor académica ha sido reconocida con varios premios, entre los que cabe señalar el Premio Fama de la Universidad de Sevilla, el Premio Sistedes en reconocimiento a su labor de promoción y consolidación de la informática en España y el Premio Nacional de Informática José García Santesmases a la trayectoria profesional otorgado por la Sociedad Científica Informática de España. Actualmente es director del Instituto de Ingeniería Informática de la Universidad de Sevilla

Este libro está orientado a la enseñanza del diseño de datos y algoritmos en el lenguaje de programación C; constituye la continuación natural del volumen *Análisis y diseño de algoritmos y tipos de datos*, publicado también en esta colección de manuales. Los conceptos sobre análisis y diseño de datos y algoritmos se concretarán en el lenguaje de programación C. Para abordar el diseño de estos últimos, es necesario tener asimilados los elementos de la programación en algún lenguaje previo, como Java o Python. Junto al diseño de algoritmos, el de tipos y el uso de tipos diseñados por otros conforman un lugar importante en este material. Al final del volumen, se incluyen ejemplos, algunos de ellos con las correspondientes soluciones. Este texto procede de la experiencia acumulada durante varios años de enseñanza de la asignatura Análisis y Diseño de Algoritmos en la Universidad de Sevilla.

Editorial Universidad de Sevilla