Introducción a la Programación

Tema 9. Introducción al Lenguaje C

Contenido

1. 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.

2. Elementos de C

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.

2.1 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 (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.* Indica al preprocesador que en lo que sigue sustituya un nombre (posiblemente con parámetros) por una definición dada. Hay dos tipos según que el nombre vaya seguido de paréntesis o no.

```
#define identificador1  expresion1
#define indentificador2(p1,p2)  expresion2(p1,p2)
```

La primera línea indica que el preprocesador sustituya en el resto del código el identificador1 por la expresion1. L a segunda línea indica al preprocesador que sustituya las apariciones de identificador(r1,r2) por expresion2(r1,r2) y los parámetros p1, p2 que había en la definición se sustituirán a su vez por a los parámetros concretos r1, r2 que aparezcan detrás del identificador2.

Ejemplos:

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

2.2 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
- *void,* que representa a un tipo sin valores

Tipos básicos equivale a los tipos predefinidos de Java. Pero no existe el tipo *boolean*. En C, de forma parecida a Java, 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.* Este puede aplicarse al tipo *char* o a un tipo entero para representar enteros positivos o cero.

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;
```

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;
typedef enum {falso, verdadero} boolean;
Color c;
boolean a;
```

En el ejemplo anterior se ha declarado el tipo Color y el tipo *boolean*.

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.

Operadores Aritméticos: Operan sobre datos de tipo numérico. Forman expresiones cuyo tipo es real (float/doublé) o entero (int/long) y son los siguientes:

- resta (menos unario) * producto % módulo
+ suma / división ++ incremento -- decremento

Ejemplos de expresiones aritméticas:

```
(votos/electores)*100
a*x*x + b*x + c
horas*3600+minutos*60+segundos
(alto-bajo)/2
```

Operadores relacionales. Operan sobre elementos de diferentes tipos construyendo expresiones de tipo lógico. El tipo *boolean* no existe en C. En su lugar se suele usar un entero que si es cero indica falso y en otro caso verdadero. Nosotros usaremos el tipo *boolean* definido arriba pero teniendo en cuenta que en C se usa normalmente un entero en su lugar.

>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que
==	igual que
!=	distinto de

Ejemplos de expresiones relacionales:

x >= (y+z)
contador < 10o
numero%2 == 0
ordenado != Cierto

Operadores Lógicos. Estos operadores operan sobre valores lógicos construyendo expresiones de tipo lógico (*boolean*):

&&	Y
||	O
!	NO

Ejemplos de expresiones lógicas:

(opcion < 5) || (opcion > 20)
(opcion >= 5) && (opcion <= 20)
! encontrado

Operador de Asignación. Son los operadores *=, +=, -=, etc.* con una semántica similar a la de Java. En C todos estos operadores son operadores binarios básicos con la misma prioridad. Son legales, aunque no recomendables, expresiones del tipo:

```
a = b = c = 18;
```

Una expresión construida con un operador de asignación tiene el tipo del operando izquierdo y devuelve el valor asignado a ese operando izquierdo.

Conversiones explícitas e implícitas de tipo:

Existen conversiones automáticas (implícitas) de de cada uno de los tipos al siguiente en la sucesión: *char, int, long, float, double*. Como en Java una operación que involucre dos tipos de los anteriores diferentes se hace en dos pasos: en primer lugar se convierten ambos tipos al mayor (en el orden anterior) y posteriormente se hace la operación correspondiente sobre los valores de los operandos.

Si se quiere hacer una conversión de tipo de uno mayor a otro menor hace falta usar el operador de casting que tiene la misma forma que en Java y las mismas consecuencias.

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

```
double log(double x);        // logaritmo nenperiano
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 absolute
long int labs(long int num);  // valor absoluto
div_t div(int num, int denom); //devuelve cociente y resto de num entre denom
int rand(void);            // número entero aleatorio entre 0 y RAND_MAX
```

El tipo *div_t* es un *struct* (que veremos más adelante) con dos campos: *int quot* (cociente) y *int rem* (resto).

3. Punteros, tablas y estructuras

El concepto de puntero no existe directamente en Java aunque todos los objetos (de Java) están implementados usándolos. 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. En tipo puntero a T se declara como 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*. En general si tenemos un tipo T el tipo T * será el tipo puntero a T.

Como todos los tipos junto con los valores se dispone de un conjunto de operadores y funciones. 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 como *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 variable 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.

3.1 Tablas (arrays)

El concepto de tabla es equivalente al visto en Java. Una tabla 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.

Declaración e inicialización de tablas

```
tipo nombre [dim];
tipo nombre [dim1][dim2];
tipo nombre[] = {v1, v2, ..., vn};
```

Ejemplos:

```
int v [10];
char palabra [256];
int a[] = {2,-3,7};
```

La variable v es una tabla de 10 enteros. La variable palabra es una tabla de 256 caracteres. La variable a es una tabla de 4 enteros.

Asociado a una tabla vamos a definir dos conceptos que no existen en C pero que es conveniente usarlos para hacer una programación más segura.

- Tamaño de una tabla: Es el número de elementos que tiene. El tamaño de una tabla no se puede modificar. Cada vez que declaremos una tabla declararemos una variable de tipo *int* que guardará el tamaño de la tabla. La variable tendrá de identificador *nombreTam*. Donde el identificador de la tabla es *nombre*. La variable que guarda el tamaño de una tabla debe ser un valor constante de tipo *int*. Por los tanto la declararemos *const int nombreTam = valor.*
- Longitud: Es el número de elementos útiles de la tabla. Esta longitud si puede variar. En muchos programas en C usaremos tablas que no están completamente llenas. En este caso declararemos una variable con identificador *nombreLength* que guarde el número de elementos útiles. Debe cumplirse que *nombreLength <= nombreTam* y ambas son mayores o iguales a cero.

Declaración e inicialización de los elementos de una tabla: cuando declaramos una tabla se reserva memoria para ubicar cada uno de sus elementos. Pero estos elementos no están inicializados y por lo tanto si los consultamos obtenemos, en general, un valor no definido. Igual que para las demás variables los elementos de una tabla deben de ser inicializados antes de ser consultados. Mediante la sentencia *T b[] = {v1,v2,…,vn}* es posible declarar la tabla b, de tipo T y tamaño n, e inicializar sus elementos con los valores indicados entre {}. El primer valor, v1, quedará ubicado en la casilla de índice 0 y así sucesivamente.

Acceso a los elementos de una tabla: Se realiza usando números enteros como índices dentro del operador []. Si *i* es un índice deben cumplir *0 <= i < nombreTam* y si la tabla no está

completamente llena entonces *0 <= i < nombreLength*. Si no se cumplen esas restricciones los programas que construyamos funcionarán mal y el error es difícil de detectar.

Modificación de los elementos de una tabla: Se realiza usando números enteros como índices dentro del operador [], como para la consulta, pero ubicando la expresión en la parte izquierda de un operador de asignación.

Ejemplos:

```
#define vTam 10
int v[vTam];
int a[] = {2,-3,7};
int aTam = 3;
a[2] = a[1] +2;   // la tabla a ha quedado modificada a {2,-3,-1}
v[4] = v[3];   //incorrecto la casilla v[3] se está usando antes inicializarse
v[14] = 12;       // incorrecto pero no es detectado por el compilador
```

En el ejemplo anterior se declara la tabla v pero no se inicializan sus elementos. La tabla a es declarada e inicializada.

Tablas y Punteros: aritmética de punteros

Existen equivalencias claras entre el tipo puntero a T y tabla de T. El nombre de una tabla, tomado aisladamente es la dirección donde comienza la tabla. Es una constante que no se puede modificar equivalente a un puntero al elemento que está en la posición cero. Veamos algunas equivalencias entre tablas y punteros. Sean las variables v y p declaradas de la forma siguiente para cualquier T:

```
#define vTam 20
T v[vTam];
T * p = v;
```

Entonces existen las siguientes equivalencias:

- *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)*

Como vemos el nombre de una tabla 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 *i* de una tabla. Estas operaciones se denominan aritmética de punteros.
El operador *[]*, denominado operador de indexación, puede colocarse detrás de un nombre de tabla o de un puntero. En general si p es un puntero, lo que incluye al identificado de una tabla, 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.

3.2 Cadenas de caracteres

Es un caso especial de tablas con algunas características específicas

El lenguaje C no tiene 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 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. En el fichero *string.h* se definen las funciones sobre cadenas proporcionadas en la librería de C.

Definimos el tipo *String* y el tipo *StringD* tal como vemos abajo. El tipo *String* es equivalente a *char ** y por lo tanto al ser declarado no reserva espacio para los caracteres. El tipo *StringD*, por el contrario es una tabla de caracteres de tamaño *MAX*. El tipo *StringD* lo usaremos para reservar memoria y posiblemente inicializarla con una cadena constante. El tipo *String* para recorrer, consultar y modificar los caracteres de una cadena. Las variables de este tipo las podemos inicializar con la dirección de otra cadena de caracteres.

Estas declaraciones las incluimos en el fichero *tipos.h* junto con otros tipos que puedan ser de interés como el tipo *boolean* visto más arriba. Iremos aclarando cuando usar un tipo.

```
#include <string.h>

#define MAX 256
typedef char StringD[MAX];
typedef char * String;
```

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.* Los prototipos de estas funciones son:

```
int strlen (const char * s);
char * strchr(const char * s, char c);
char * strstr(const char * s1, const char * s2);
char * strcpy (char * s1, const char * s2);
int strcmp(const char * s1, const char * s2) ;
char * strcat(char * s1, const char  * s2);
```

- *strlen:* Calcula la longitud de s
- *strchr:* Busca la primera posición del carácter c y devuelve un puntero al mismo o NULL si no lo encuentra.
- *strstr:* 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.
- *strcpy:* Copia s2 en s1 y devuelve un puntero a s1.

- *strcmp*: Devuelve un entero negativo, cero o positivo según que s1 sea menor, igual o mayor que s2 en el orden lexicográfico.
- *strcat:* Concatena s2 detrás de s1 y devuelve un puntero a s1.

Junto a las anteriores es conveniente disponer de una función que sea capaz de fragmentar una cadena de entrada en una secuencia de tokens delimitados por un conjunto de delimitadores. Es la función *split* con cabecera:

```
String * split(const String s, const String delim, int * nt, String * bs, String bd);
```

- split: Toma como entrada la cadena *s*, y los delimitadores *delim* y devuelve un puntero a un array de *String* cada uno de los cuales es un token. Devuelve ,además el número de tokens (*nt*), la cadena modificada (*bd*) al sustituirse los delimitadores por caracteres nulos. Hay que proporcionar como entrada los parámetros *bd*, bs que señalan respectivamente a la memoria reservada para almacenar la cadena modificada y los punteros a cada uno de los tokens.

Hay que hacer algunas consideraciones. En las funciones anteriores los parámetros formales con la etiqueta *const* son parámetros de entrada y no van a ser modificados. Los que no tienen la etiqueta *const* si serán modificados. Así ocurre con s1 en *strcpy* y *strcat*. Como s1 va a ser modificado hay que tener en cuenta que debe apuntar a una zona de memoria, previamente reservada, con tamaño suficiente. En el caso de *strcpy* para ubicar la cadena s2 (junto con el carácter final '\0') y en el caso de *strcat* para ubicar a s1 seguida de s2.

Hay que tener en cuenta algunos detalles importantes de implementación:

- Las cadenas en C acaban siempre con el carácter '\0'.
- El número de caracteres útiles de la cadena es el devuelto por *strlen* pero la cadena necesita al menos un carácter más de memoria para ubicar el '\0' final. Por lo tanto para una cadena dada *s* el tamaño de la tabla de caracteres donde se ubique la cadena debe cumplir *sTam > strlen(s).*

Ejemplos:

```
int i1, i2;
StringD s1 = "Hola";
StringD s2 = " Mundo";
String s3;
i1 = strlen(s1);
s3 = strcat(s1,s2);
i2 = strlen(s3);
```

El resultado para *i1, i2* es 4, 10. Hemos tenido en cuenta que los tamaños de las tablas de caracteres (256) donde se ubican s1, s2 (antes y después de la concatenación) cumplen la restricción anterior.

Por otra parte hemos de tener en cuenta que el tipo *StringD* define una tabla de caracteres cuyo identificador, como en todas las tablas, es un puntero constante y por lo tanto no puede estar en la parte izquierda de una asignación. Sería incorrecta la sentencia

```
s1 = strcat(s1,s2);  //incorrecto: s1 es un puntero constante y no puede estar
                     // en la izquierda de una asignación
```

También hemos de tener en cuenta que una variable de tipo *String* puede ser inicializada con la dirección de una cadena constante pero no puede modificarla.

```
String nn1 = "Nuevo Mundo";
StringD s1 = "Hola";   // La cadena constante se copia en la memoria de s1
nn1[3] = '5'; // produce un error porque intenta modificar la cadena constante
s1[3] = '5';  // correcto
```

3.3 Estructuras y punteros a estructuras

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.*

Declaración e inicialización de estructuras

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;

typedef Tr * TrP;

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

Arriba definimos el tipo nuevo *Tr* a partir de un *struct* y *TrP* como un puntero a *struct.* También se muestra la forma de inicializar variables de esos tipos.

Ejemplo:

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

```
typedef PuntoD * Punto;

PuntoD p = {2.0,3.1};
Punto pp = &p;
```

Operadores de acceso a campos

Hay disponibles dos operadores para acceder a los campos de un tipo *struct* o puntero a *struct*. Son los operadores . y ->. El primero se usa para acceder a los campos de una variable de tipo *struct.* El segundo para acceder a los campos de una variable de tipo puntero a *struct*.

Si tenemos la declaración vista arriba para los tipos *PuntoD* y *Punto* y las declaraciones de las variables p, pp siguientes:

```
PuntoD  p;
Punto pp;
```

El acceso a los campos se hará con los operadores . y -> tal como sigue:

- Acceso a través de variable de tipo *struct*: *p.x*
- Acceso a través de puntero a *struct*: *pp->x* o también *(*pp).x*

Es decir si tenemos una variable de tipo puntero a *struct* y queremos acceder a un campo usamos el operador ->. Si la variable es de tipo *struct* usamos el operador punto (.).

Las expresiones construidas con estos operadores (. ->) tienen el tipo del campo correspondiente y pueden ser colocadas a la derecha o a la izquierda de un operador de asignación. Si se colocan a la derecha estamos consultando el valor del campo. Si se colocan a la izquierda estamos modificando su valor.

Ejemplos:

```
pp->x = pp->y+18.;              o equivalentemente
(*pp).x = (*pp).y+18.;
```

4. Bloques y sentencias de control

Un bloque es una secuencia de declaraciones y sentencias delimitadas por { }. Las declaraciones deben ir al comienzo del bloque y tienen como ámbito hasta el final del mismo.

Una sentencia puede ser una sentencia básica, una sentencia de control o un bloque tal como se ha definido antes. La sentencias básicas son expresiones terminadas en ;.

Las sentencias de control, permiten modificar y controlar el flujo secuencial de ejecución de los programas y se clasifican en:

- Selectivas: *Para bifurcar el flujo de ejecución*
- Repetitivas: *Para establecer bucles iterativos de ejecución*

Las sentencias de control selectivas permiten evaluar una condición y según el resultado de la misma se ejecutará un conjunto determinado de sentencias

```
if(condición_Lógica){
	sentencias_1
}else{
	sentencias_2
}
```

Las sentencias de control repetitivas permiten ejecutar reiteradamente un conjunto determinado de sentencias, mientras cierta condición evalúe a cierto (distinto de cero).

Veremos dos sentencias: while y for cuyo funcionamiento es similar al de Java.

```
while(condicion){
	sentencias
}
```

```
for(ini;cond;incremento){
	sentencias
}
```

C no incluye el *for-extendido* pero es sencillo de simularlo mediante un *for* clásico tal como se muestra abajo.

```
int t[] = {1,3,5,7,11};
int s = 0;
for(int e : t){
	s = s + e;
}

int tTam = 5;
int t[] = {1,3,5,7,11};
int s = 0;
int i;
for(i= 0; i < tTam; i++){
	s = s + t[i];
}
```

A su vez cualquier sentencia *for* podemos convertirla en una sentencia *while*. La transformación es de la forma:

```
for(ini;cond;incremento){
	sentencias;
}
===
ini;
while(condicion){
```

```
        sentencias;
        incremento;
}
```

5. 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.

Declaración de prototipos de funciones.

Igual que las variables tienen un tipo las funciones tienen un tipo asociado su prototipo, cabecera o signatura. Es conveniente estructurar el código agrupando los prototipos de las un 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.

Las funciones permiten la reutilización de código. Las funciones junto con su agrupación en ficheros son los mecanismos de modularización del código en C. Son mecanismos primitivos si los comparamos con los disponibles en Java: interfaces, clases y paquetes.

Ejemplos de prototipos de funciones.

```
double getXPunto(const Punto p);
void setXPunto(Punto p, double x);
double getYPunto(const Punto p);
void setYPunto(Punto p, double y);
```

La estructura de una función es como sigue:

```
Tr nombre (T1 e1, ..., Tn en) {
        declaración de variables locales
        sentencias
        return r;    // no hay return si la function devuelve void.
}
```

Estructura igual que en Java, pero las declaraciones de variables siempre al inicio de la función.

Un programa en C se compone de un conjunto de ficheros con extensión .h y otro conjunto con extensión .c. Los primeros son ficheros de declaraciones e incluyen líneas de *#include* (para incluir otros ficheros *.h*), líneas *#define* (que definen macros en general y constantes en particular), declaraciones de tipos (mediante *typedef*) y prototipos de funciones. Los ficheros *.c* contienen líneas *#include* (para inclusión de ficheros .h) y código de funciones (cabecera más código).

Los ficheros con extensión *.h* suelen tener una estructura particular para facilitar su uso. Así el fichero *tipos.h* tiene la estructura:

```
#ifndef TIPOS_H_
#define TIPOS_H_

typedef enum{falso,verdadero} boolean;

#endif /* TIPOS_H_ */
```

El objetivo es definir una variable similar al nombre del fichero, en este caso TIPOS_H_, que nos sirve para conseguir que el fichero se incluya una sola vez. Esto se consigue por el funcionamiento de las cláusulas del preprocesador *#ifndef*, *#define*, *#endif*. La primera decide si se ha definido antes la variable. Si no se ha definido expande el fichero hasta *#endif.* La primera línea define la variable que no estaba definida. Si ya está definida, porque el fichero se había incluido previamente en otro lugar el preprocesador se salta todo el contenido del fichero hasta *#endif*.

Un programa en C es, por lo tanto, un conjunto de funciones. Como no existe sobrecarga todas deben tener nombres diferentes. De todas ellas hay una única función distinguida cuyo nombre es *main.* La llamada a esta función es el comienzo del programa.

La llamada a una función es como en Java salvo que en C no se invoca sobre ningún objeto. Igualmente la invocación de función puede formar una sentencia básica (si la función devuelve *void*) o formar parte de una expresión si devuelve un valor. El concepto de parámetros formales y parámetros reales e igual que en Java.

Ejemplos:

```
double x;
PuntoD pd = {3.0,4.5};
Punto p = &pd;
…
x = getXPunto(p);
setYPunto(p,3.0);
```

Paso de parámetros: C vs. Java

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*). El resto parámetros son 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.

Veamos con algunos ejemplos por qué los parámetros de tipo puntero son de entrada salida.

```
double mediaAritmética (int a, int b, int c) {
     a = a + b + c;
     return (a / 3.0);
}
```

Los parámetros anteriores son de entrada. Sus tipos son básicos. Sin embargo los parámetros de la función intercambio son de entrada salida.

```
void intercambio (int *p1, int *p2) {
     int tmp;
     tmp = *p1;
     *p1  = *p2;
     *p2  = tmp;
}
```

```
#include <stdio.h>
void intercambio (int *, int *);
double mediaAritmetica(int a, int b, int c);


void main (void) {
     int x = 5, y = 8, z = 10;
     double r;
     intercambio (&x, &y);
     printf("%d %d\n",x,y);
     r = mediaAritmetica(x,y,z);
     printf("%d %f\n",x,r);
}
```

Podemos comprobar el que valor de x e y ha cambiado después de la llamada a la función *intercambio* en el ejemplo anterior. Vemos que como parámetros reales pasamos punteros a las correspondientes variables. Esta técnica (declarar el parámetro formal de tipo *T **, pasar como parámetro real un puntero a una variable dada y posiblemente modificar en el cuerpo de la función el contenido apuntado por el puntero) se le denomina **paso de parámetros por referencia**. Es la técnica adecuada en C para disponer de parámetros entrada-salida.

Sin embargo cuando llamamos a la función *mediaAritmetica* la variable *x* no cambia (aunque en el cuerpo de la función el correspondiente parámetro formal si es modificado) porque es un parámetro de entrada.

Los parámetros formales pueden llevar el calificador *const* lo que indica que el parámetro formal correspondiente es constante y no puede ser modificado dentro del cuerpo de la función. Es decir no puede aparecer a la izquierda de un operador de asignación. Esto es útil para declarar que una tabla no puede ser modificada (y de esa forma convertirla en un parámetro de entrada). En general es recomendable etiquetar los parámetros de entrada con el calificador *const.*

6. 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 función es convertir cadenas de caracteres es secuencias de valores de otros tipos (*int, float, ..)* o viceversa secuencias de valores en una cadena de caracteres.

Un fichero se representa en un programa en C por el tipo FILE *. Abrir un fichero se hace con la función *fopen* cuyo prototipo es:

```
FILE * fopen( const char * nombreFichero, const char * modo);
```

El parámetro *nombreFichero* es el nombre de fichero y el parámetro modo es una cadena de caracteres que indica si queremos leer o escribir en el fichero. Algunas posibilidades para modo son:

- "r" se abre el fichero en modo lectura
- "w" crea un fichero y lo abre en modo escritura

Si el fichero *nombreFichero* no existe en el sistema de fichero la función anterior devuelve NULL. Una vez terminado de leer o escribir debemos cerrar el mimos con la función

```
int fclose(FILE *fp);
```

Para saber si hemos alcanzado el fin de fichero usamos la función:

```
int feof(FILE *fp);
```

Salida con formato: printf, sprintf, fprintf

La función *printf* permite escribir en la consola de salida una lista de valores con un formato preestablecido. Acepta diferentes tipos de argumentos: carácter, valor numérico entero o real o cadena de caracteres, y un formato y a partir de ellos construye una cadena de salida que

escribe. El formato se especifica como un primer parámetro de tipo *char **. La forma de la función *printf* es:

```
printf ("formato", arg1, arg2, ..., argn);
```

Los parámetros *argi* pueden ser constantes, variables o expresiones, y *"formato"* es una cadena de caracteres que especifica el formato. La cadena de formato se compone de caracteres que formarán parte de la cadena resultado y los símbolos especificadores del formato. Los símbolos especificadores de formato comienzan con *%* seguido de uno o varios símbolos que indican el tipo de dato esperado y los detalles de su conversión a cadena.

Algunos de ellos especificadores de formato son:

%c	carácter (char)		
%d	número entero (int)	%ld	número entero (long)
%f	número real (float)	%lf	número real (double)
%s	cadena de caracteres		

Por cada tipo argumento a imprimir existe un especificador de formato, que indica para mostrar el dato. Por ejemplo para el caso *float %.3f* indica que se muestre el número real con tres decimales. Dentro de la cadena de símbolos del formato se pueden incluir tabuladores (\t), saltos de línea (\n), etc.

Ejemplos de uso de *printf*:

```
printf ("El cuadrado de %f vale %f\n", x, x*x);
printf ("La edad de %s es %d años\n", "Juan", edad);
printf ("Seno (%lf) = %lf\n", x, sin(x));
```

La función *printf* puede emplearse sólo con un argumento, el formato, en cuyo caso sólo se imprimen los caracteres presentes en la cadena de formato.

De manera completamente similar podemos ubicar el resultado en una cadena que se da como parámetro en vez de en la consola. Este es el objetivo de la función *sprintf* cuya cabecera de la forma siguiente (donde s es una cadena de caracteres):

```
sprintf(s,"formato", arg1, arg2, …, argn);
```

La cadena construida puede también mandarse a un fichero. Esto lo conseguimos con la función f*printf* cuya cabecera de la forma siguiente:

```
fprintf(f,"formato", arg1, arg2, …, argn);
```

Ahora el parámetro f es del tipo *FILE **. Es el puntero a una variable que obtiene al abrir un fichero.

En el caso de escritura sobre un fichero o sobre la consola existen algunas funciones que pueden resultar útiles para escribir líneas enteras. Son las funciones: *puts*, *fputs*, cuyas signaturas son:

```
int fputs(const char * s, FILE * f);
int puts(const char *s);
```

Ambas escriben una línea (en el fichero *f* o en la consola) y retornan un valor positivo si no ha habido error.

Entrada con formato: scanf, sscanf, fscanf

Las funciones de entrada hacen el trabajo inverso a las de salida: convertir una cadena, procedente de la consola, de una cadena dada o de un fichero, en una secuencia de valores cuyos tipos y estructura está especificada por un formato. La función *scanf* permite leer valores desde la entrada estándar y almacenarlos en las variables que se especifican como argumentos. Las funciones *sscanf* y *fscanf* hacen los mismo desde una cadena dada o desde un fichero. Sus cabeceras son de la forma (donde s es una cadena y f una variable de tipo FILE * como antes):

```
scanf ("formato", arg1, arg2, ..., argn);
sscanf(s, "formato", arg1, arg2, ..., argn);
fscanf(f, "formato", arg1, arg2, ..., argn);
```

En las funciones de salida, como en las de entrada, debe haber tantos especificadores de formato en la cadena de *"formato"* como variables en la lista de argumentos. Debemos tener en cuenta que los parámetros de las funciones de entrada deben ser parámetros de entrada-salida y por lo tanto de tipo puntero a T.

Por ejemplo:

```
int n;
double x;
scanf("%d %lf", &n, &x);
```

En el ejemplo anterior se lee de la consola un valor de tipo *int* y otro de tipo *double*. Los valores se introducirán desde la entrada estándar separándolos por espacios o retornos de carro. En la cadena de formato aparecen especificadores de formato (los mismos que ya se han visto).

En el caso de lectura desde un fichero o desde la consola existen algunas funciones más que pueden resultar útiles para leer líneas enteras. Son las funciones: *gets*, *fgets*, cuyas signaturas son:

```
char * fgets(char * s, int n, FILE * f);
char * gets(char *s);
```

La función *fgets* lee un máximo de *n-1* caracteres o hasta que encuentra un fin de línea del fichero *f* poniendo el resultado en la cadena *s*. El puntero s debe señalar a una zona cons suficiente memoria reservada para ubicar la línea leída. Regresa un puntero a la cadena leída o *NULL* si ha encontrado el fin de fichero. Similarmente funciona *gets* pero leyendo una línea de la consola.

7. Otros elementos

En C hay dos formas de obtener memoria para ubicar los valores de una variable. La primera forma es simplemente declarando la variable de un tipo dado. En ese momento se reserva memoria en un tamaño adecuado para almacenar valores del tipo. Las declaraciones, como hemos visto más arriba, se hacen al principio de un bloque y en ese momento se reserva la memoria para la variable. Esa memoria se libera automáticamente cuando acaba el bloque. Las variables que declaran al principio de un bloque las denominamos variables locales. 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. Igual que para las variables locales la memoria para esta variables globales se les asigna en el momento de la declaración y no se libera porque la vida de la variable dura a lo largo de todo el programa.

La memoria obtenida de la forma anterior se llama **memoria estática**: se reserva memoria, en un tamaño adecuado para almacenar valores del tipo, cuando se declara al variable y se libera al final del bloque. En C hay una segunda forma de obtener memoria. Es la que se denomina **memoria dinámica**. Es una memoria que se obtiene y se libera a demanda del programador. No tiene un equivalente en Java. Esta forma de memoria es muy importante para guardar los valores apuntados por una variable de tipo puntero. En lenguaje C proporciona funciones para obtener bloques de memoria, para liberarlos y para copiar bloques de memoria de una ubicación a otra. C proporciona, además, un operador muy importante para calcular la cantidad de memoria necesaria. Es el operador *sizeof* que aplicado a un tipo devuelve un entero que indica el número de bytes necesarios para ubicar un valor de ese tipo. En el ejemplo siguiente la variable *a* guarda el número de bytes necesarios para almacenar un valor de tipo *PuntoD.*

```
a = sizeof(PuntoD)
```

Las Funciones para gestionar la memoria dinámica son:

- *malloc:* Obtiene un bloque de memoria
- *memcpy:* Copia un bloque de memoria
- *free:* Libera un bloque de memoria

Sus prototipos son:

```
void * malloc(size_t numeroDeBytes);
void * memcpy(void * destino, const void * origen, size_t  n);
void  free(void * ptr);
```

El tipo *size_t* es un entero sin signo. Los tipos *void ** los veremos en seguida.

Punteros a void y Punteros a Funciones.

Junto con los punteros a un tipo *T* dado se dispone en C del tipo **puntero a *void*** que se declara *void **. Es un tipo que se puede convertir sin pérdida de información en puntero a otro tipo T cualquiera. Este tipo es muy adecuado para diseñar funciones que queremos que tomen parámetros de diversos tipos. C no dispone de tipos genéricos pero es posible simular mucha de la funcionalidad de los métodos genéricos en Java mediante funciones que toman parámetros de tipo *void **. Los punteros a void no admiten el operador * (contenido) y la aritmética de punteros.

En C hay también punteros a Funciones. Estos son punteros al código propio de la función. Para declarar un **puntero a función** incluimos el nombre de la variable entre paréntesis y precedida de *. Veamos ejemplos de funciones y punteros a funciones:

```
int * f();
int (* pf) ();
void (*ptrFun) (int a,int b);
```

En la primera línea se declara una función que devuelve un puntero a entero. En la segunda línea se declara un puntero a función que no toma ningún parámetro y devuelve un entero. En la tercera se declara un puntero a función que toma dos enteros y devuelve un entero.

Emulación de tipos y métodos genéricos en C

En C es posible diseñar funciones reutilizables sobre una tabla de un tipo cualquiera. Para ello debemos proporcionar tres parámetros: base (un puntero de, tipo void *, a la primera casilla de la tabla), n (el número de casillas útiles) y *size* (el tamaño de los valores que ocupan las casillas). El puntero a la primera casilla de la tabla tiene que ser de tipo *void ** porque este es el único tipo que puede ser convertido sin pérdida de información en un puntero a cualquier otro tipo.

El tipo puntero a función puede ser el tipo de un parámetro formal. Veamos como ejemplo el prototipo de la función de biblioteca *qsort* que ordena una tabla:

```
void qsort(void * base,  size_t n,  size_t size,  int (*cmp)(void *, void *));
```

Esta función ordena una tabla. El primer parámetro es un puntero a la primera casilla de la tabla, el segundo el número de casillas (el tamaño de la tabla), el tercero el número de bytes que se necesitan para almacenar un valor de una de las casillas, y el último un puntero a una función que representa el orden con el que se ordenará la tabla. Este último es un puntero a una función que toma dos datos de cualquier tipo *(void *)* y devuelve un entero.

Veamos un ejemplo concreto de su uso:

```
int ordenPuntoX(const  Punto * p1, const  Punto * p2);
int ordenPuntoY(const  Punto * p1, const  Punto * p2);
#define aTam 5
Punto a[aTam];
…
qsort(a,aTam,sizeof(Punto),ordenPuntoX);
...
```

En el ejemplo anterior tenemos los prototipos de dos funciones que implementan posibles formas de ordenar puntos: una los ordena según su coordenada *X* y otra según la *Y*. La llamada a *qsort* ordena la tabla de elementos de tipo *Punto* según la coordenada *X* de los respectivos puntos. La implementación de las mismas la veremos abajo. También debemos tener en cuenta que cuando declaramos una función el nombre de la misma usado aisladamente es un puntero a función del tipo adecuado. Así *ordenPuntoPX* y *ordenPuntoPY* son, por una parte, los identificadores de dos funciones pero por otra, usados aisladamente, son punteros a funciones. Por lo tanto la llamada a *qsort* puede recibir uno de estos identificadores como parámetro. Ocurre algo similar que a una tabla. El nombre de una tabla, usado aisladamente, es una constante de tipo puntero al tipo de las casillas de la tabla y el nombre de una función es constante de tipo puntero a función.

Predicados, expresiones y órdenes genéricos en C

Como en Java es importante para poder reutilizar código disponer de tipos adecuados para representar predicados, expresiones y órdenes. Cada uno de ellos los representaremos por un puntero a función con parámetros de tipo *void **. Así tenemos las definiciones reutilizables:

```
#define Predicate boolean (* predicate)(const void * element)
#define Function void * (* function)(const void * element, void * memResult)
#define Comparator int (* order)(const void * element1, const void * element2)
```

Más adelante aprenderemos a implementar predicados, expresiones y órdenes. Un *Predicate* lo definimos como un puntero a una función que a partir de un valor devuelve un *boolean*. Para que pueda ser genérico el parámetro formal de entrada debe ser de tipo *void ** y será un puntero a la variable sobre la que queremos calcular el predicado. Igual ocurre con el tipo *Function* y *Comparator*. El primero, además del puntero al valor de entrada a la función, tiene un segundo parámetro que señale a una zona de memoria adecuada para ubicar el resultado. Con esta definición la cabecera de la función *qsort* podría escribirse como:

```
void qsort(const void * base,  size_t n,  size_t size,  Comparator);
```

Junto a la función anterior se nos proporciona otra función de la librería de C capaz de buscar en una ordenada un elemento (clave) y nos devuelve un puntero al elemento encontrado o NULL si no lo encuentra. La cabecera es:

```
void* bsearch(const void *clave, const void *base, size_t n, size_t size, Comparator);
```

Ambas funciones están en *stdlib.h*

8. Estructuración de programas en C a partir de un diseño en Java: diseño de tipos

Tal como hemos ido viendo Java es un lenguaje con un nivel de abstracción mayor que C. Por lo tanto las principales diferencias vienen de aquí. 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 una metodología que nos puede guiar en este camino. Esta metodología nos va a permitir estructurar los programas en C, por una parte, y comprender más en profundidad el funcionamiento de los programas en Java por otra.

Tomaremos 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*. Por cada *clase* diseñamos un fichero con el mismo nombre y extensión .c. La *definición de tipos, constantes globales y prototipos de funciones* se hará en el fichero *.h*, el código correspondiente en un fichero *.c*. Los tipos tomarán el nombre del correspondiente interface Java.

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 métodos: si en Java tenemos un método *m* del tipo *A* y de signatura *T m(T1 a, T2 b)* entonces en C diseñamos la función *T mA(A this, T1a, T2 b).* La signatura de esta función estará en el fichero *A.h.* El código en el *.c* correspondiente. Hemos dado a la función el nombre compuesto formado por el antiguo método en Java concatenado con el nombre del tipo correspondiente. 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. Los sufijos más simple son de la forma mA1, mA2, etc. Pero en cada caso los elegiremos para que sea fácil de recordar el cometido de la función. 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 siempre lo colocaremos en primer lugar. En C, además, deberemos tener en cuenta la gestión de la memoria por lo que en muchas funciones aparecerán, por este motivo, algunos parámetros adicionales.

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. Los tipos

correspondientes los designaremos añadiendo una D final al nombre del tipo. Así el estado del tipo *Punto* lo denominaremos *PuntoD.*

Por lo tanto por cada tipo A de Java definimos dos en C:

- Uno mediante un *struct* definido a partir de los atributos privados de la clase correspondiente y que contenga el estado del objeto. A este le daremos el nombre que tenía el interface Java acabado en D.
- Otro mediante un puntero al *struct* anterior. El nombre será el mismo que tenía en Java.

Las variables del primer tipo (*struct*) se inicializarán como hemos visto para los *struct* y tendrán una vida que se extiende desde la declaración hasta el final del bloque donde se declararon. Cuando se usan como parámetros siempre son parámetros de entrada. Las variables de segundo tipo (puntero a *struct*) se inicializarán a partir de las primeras mediante el operador *&* u obteniendo bloques de memoria dinámica como veremos ahora. Las variables de este segundo tipo cuando se pasan como parámetros son parámetros de entrada-salida. Si se han inicializado obteniendo bloques de memoria dinámica su vida se extiende hasta que la memoria sea liberada. Esto hace necesaria una política adecuada de la liberación de memoria para este tipo de variables.

Como en Java podemos implementar constructores, métodos clone, órdenes, equals, …

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

Por lo tanto en C tenemos dos posibilidades:

- Definir variables de tipo *struct*. Estos se pueden inicializar como se ha visto para los *struct*. Su vida se extiende desde el momento de la declaración hasta el final del bloque donde están declarados. Si se usan como parámetros son siempre parámetros de entrada. En este caso no tenemos que preocuparnos por liberación de la memoria que usan. Esta se libera cuando termina el ámbito de la variable.
- Definir variables de tipo puntero equivalentes a los objetos de Java. Para inicializar estas variables se puede hacer obteniendo memoria dinámica o aplicando el operador & a un variable ya inicializada de tipo *struct*. En este caso la variable cuando pasa como parámetro actúa como parámetro de entrada-salida y debemos preocuparnos por liberar la memoria ocupada cuando deje de estar en uso. Este es uno de los grandes desafíos cuando se programa en C.

Vamos a desarrollar con más detalle la piemra idea. Se trata, por tanto, de definir variables de tipo *struct* para cada tipo definido. Estas se pueden inicializar como se ha visto para los *struct*. Su vida se extiende desde el momento de la declaración hasta el final del bloque donde están declaradas. Este diseño, aunque menos abstracto, tiene la ventaja de que no tenemos que preocuparnos de los problemas de gestión de la memoria dinámica. Esta se libera cuando termina el ámbito de la variable.

Las funciones que se derivan de los métodos del tipo llevarán parámetros de entrada-salida adecuados para actualizar las propiedades relevantes. También podemos diseñar funciones similares a los constructores que tomar punteros a variables del tipo struct adecuado y las actualizan.

El código sería el siguiente. No tendríamos constructores e inicializamos los objetos al inicializar el *struct* correspondiente. Los métodos tomarán los parámetros necesarios para devolver los resultados mediante parámetros de entrada-salida que ya proporcionan la memoria necesaria.

Veamos el caso concreto de implementar en C el tipo *Punto*. Obtenemos dos ficheros el fichero *Punto.h* y el *Punto.c*. El primero corresponde aproximadamente al interface *Punto* de Java y el segundo a la clase *PuntoImpl*. El tipo *Punto* es equivalente a un objeto Java de tipo *Punto* y por lo tanto hemos incluido las funciones que tenía en Java y los constructores. Se incluyen, también, dos funciones que representan órdenes posibles sobre variables de tipo *Punto:* los órdenes según la coordenada X y la Y.

```
#ifndef PUNTO_H_
#define PUNTO_H_

#include <stdio.h>
#include <stdlib.h>
#include "Tipos.h"

typedef struct {
	double x;
	double y;
} Punto;

void createPunto0(Punto * r);
void createPunto2(Punto * r, const double x, const double y);
void createPuntoDeString(Punto * r, const String s);
void setPunto0(Punto * r);
void setPunto2(Punto * r, const double x, const double y);
void setPuntoDeString(Punto * r, const String s);
double getXPunto(const Punto p);
void setXPunto(Punto * r, const double x);
double getYPunto(const Punto p);
void setYPunto(Punto * r, const double y);
String toStringPunto(const Punto p, String mem);
void clonePunto(Punto * r, const Punto p);
int ordenPuntoX(const void * o1, const void * o2);
int ordenPuntoY(const void * o1, const void * o2);

...

#endif /* PUNTO_H_ */
```

En el fichero *Punto.c* incluimos el código para las funciones anteriores:

```
#include "Punto.h"
#include <stdio.h>
#include <memory.h>
#include "Tipos.h"


void createPunto0(Punto * r) {
	if(r!=NULL){
		r->x = 0;
		r->y = 0;
	}
}

void createPunto2(Punto * r, const double x, const double y) {
	if(r!=NULL){
		r->x = x;
		r->y = y;
	}
}

void createPuntoDeString(Punto * r, const String s){
	char * as[2];
	String * asr;
	StringD sd;
	int nt;
	double x;
	double y;
	if(r!=NULL){
	 asr = split(s, ", ",&nt,as,sd);
		if(nt==2){
			x = atof(asr[0]);
			y = atof(asr[1]);
			r->x = x;
		  r->y = y;
		} else {
		 r = NULL;
		}
	}
}

void setPunto0(Punto * r){
	if(r!=NULL) {
		r->x = 0.;
		r->y = 0.;
	}
}

void setPunto2(Punto * r, const double x, const double y){
	if(r!=NULL) {
		r->x = x;
		r->y = y;
	}
}

void setPuntoDeString(Punto * r, const String s){
	String as[2];
	String * asr;
```

```
	StringD sd;
	int nt;
	double x;
	double y;
	if(r!=NULL) {
		asr = split(s, ", ",&nt,as,sd);
		if(nt==2){
			x = atof(asr[0]);
			y = atof(asr[1]);
			r->x = x;
		    r->y = y;
		} else {
			r = NULL;
		}
	}
}


double getXPunto(const Punto p){
	return p.x;
}

double getYPunto(const Punto p){
	return p.y;
}

void setXPunto(Punto * r, const double x){
	r->x=x;
}

void setYPunto(Punto * r, const double y){
	r->y=y;
}

String  toStringPunto(const Punto p, String mem){
	sprintf(mem,"(%.2f,%.2f)",p.x,p.y);
	return mem;
}


void clonePunto(Punto * r, const Punto p){
	if(r!=NULL) {
		r->x = p.x;
		r->y = p.y;
	}
}

int ordenPuntoY(const void * p1, const void * p2){
	double r;
	Punto * o1 = (Punto *)p1;
	Punto * o2 = (Punto *)p2;
	r = getYPunto(*o1)-getYPunto(*o2);
	return sgnDouble(r);
}

int ordenPuntoX(const void * p1, const void * p2){
	double r;
	Punto * o1 = (Punto *)p1;
	Punto * o2 = (Punto *)p2;
```

```
        r = getXPunto(*o1)-getXPunto(*o2);
        return sgnDouble(r);
}
```

Algunos comentarios al código anterior:

- Como podemos ver se implementan métodos de factoría para inicializar variables que deben haber sido declaradas previamente.
- En el caso del constructor que toma una cadena de caracteres como parámetro usamos la función split para partir la cadena de entrada en tokens separados por los delimitadores contenidos en la cadena " ,". La conversión de cada token a *double* lo hacemos con la función *atof.*
- La implementación de los órdenes tiene en cuenta que debe devolverse un entero. Para ello se usa la función *sgnDouble* implementada en otro lugar que convierte números reales positivos en 1 (entero), los negativos en -1 y cero en 0 entero.
- Cada función asociada a un método de un tipo dado la dotamos de un primer parámetro, de tipo Punto * o Punto según que se modifique el estado del objeto o no.
- Para trabajar con cadenas de caracteres usamos el tipo String (char *) vistos anteriormente.

A partir de lo anterior diseñamos la función principal en el fichero *main1.c* :

```
#include "Punto.h"

void main(){
        int i;
        char mem[256];
        char * s1 = mem;
        int aTam = 5;
        Punto a[5];

        createPunto2(a,2.0,3.1);
        createPuntoDeString(a+1,"7.9,-2.0");
        createPunto2(a+2,-2.0,7.1);
        createPunto2(a+3,14.0,-3.1);
        clonePunto(a+4,a[0]);
        printf("-------\n");
        qsort(a,aTam,sizeof(Punto),ordenPuntoY);
        for(i=0;i < aTam;i++){
                s1 = toStringPunto(*(a+i),s1);
                printf("%s\n",s1);
        }
        printf("-------\n");
}
```

La salida esperada es:

```
-------
-2.00,7.10
2.00,3.10
2.00,3.10
7.90,-2.00
14.00,-3.10
-------
```

Y si cambiamos por *ordenPuntoY* el parámetro de la llamada a *qsort* el resultado es:

```
-------
14.00,-3.10
7.90,-2.00
2.00,3.10
2.00,3.10
-2.00,7.10
-------
```

Como vemos se declaran la variables que necesitemos y se inicializan mediante una función que emula los métodos de factoría o con el mecanismos para inicializar un struct.

Otro ejemplo es el diseño de una lista de elementos. En el código siguiente se muestran los ficheros correspondientes y otro de prueba. Podemos ver la forma de implementar un orden y la forma de ordenar una lista.

La implementación que se ofrece es una lista de tamaño máximo dado. Si que quiere hacer una implementación sin esa limitación hay que hacerlo con memoria dinámica.

```
// ListaDouble.h

#define Comparator int (* order)(const void * e1, const void * e2)

typedef struct {
       int tam;
       int size;
       double * datos;
} ListaDouble;

void createListaDouble(ListaDouble * lis, double * dt, const int tam);
int addListaDouble(ListaDouble * lis, const double e);
int get(const ListaDouble lis, const int index, double * e);
int size (const ListaDouble lis);
int isEmpty(const ListaDouble lis);
void sortListaDouble(ListaDouble * lis, Comparator);
void printLista(const ListaDouble lis);

//ListaDouble.c

int sgnDouble(double n){
    int r;
    if(n < 0){
        r = 1;
    } else if(n>0) {
        r = +1;
    } else {
        r = 0;
    }
    return r;
}

void createListaDouble(ListaDouble * lis, double *dt, const int tam){
    lis->datos = dt;
    lis->tam = tam;
    lis->size=0;
}
```

```
int addListaDouble(ListaDouble * lis, const double e){
    int r = +1;
    if(lis->size<lis->tam){
        lis->datos[lis->size]=e;
        lis->size++;
    } else {
        r = -1;
    }
    return r;
}

int get(const ListaDouble lis, const int index, double * e){
    int r = +1;
    if(index >=0 && index < lis.size){
        *e = lis.datos[index];
    } else {
        r = -1;
    }
    return r;
}

int size (const ListaDouble lis){
    return lis.size;
}

int isEmpty(const ListaDouble lis){
     return lis.size == 0;
}

void printLista(const ListaDouble lis){
     int i = 0;
     for(; i< lis.size; i++){
        printf("%lf\n",lis.datos[i]);
     }
}

void sortListaDouble(ListaDouble * lis, Comparator){
    qsort(lis->datos,lis->tam,sizeof(double),order);
}

int ordenRealInverso(const void * p1, const void * p2){
       double r;
       double * o1 = (double *)p1;
       double * o2 = (double *)p2;
       r = -(*o1 - *o2);
       return sgnDouble(r);
}

//Main.c

int main(){
    double dt[6];
    ListaDouble lis;
    createListaDouble(&lis,dt,6);
    int i = 0;
    for(; i<6; i++){
        addListaDouble(&lis, 2.*i);
    }
    sortListaDouble(&lis, ordenRealInverso);
    printLista(lis);
}
```

Para conseguir una lista de enteros se trata de sustituir Double por Integer y double por int en el código anterior. Esto se puede conseguir definido algunas macros para el preprocesador.

9. Esquemas secuenciales en C

Los esquemas secuenciales, como vimos en temas anteriores para el caso del lenguaje Java, se compone de varios elementos: agregados de datos (fuentes de datos y mecanismos para recorrerlos), filtros, funciones de transformación y acumuladores. Componiendo los elementos anteriores resultan los diferentes esquemas. Veamos esos elementos en C.

Agregados de Datos y mecanismos para recorrerlos

Las fuentes de datos que veremos son:

- Secuencias numéricas (de enteros o reales). Por ejemplo la secuencia de 0 a n-1 de 1 en 1. O la secuencia aritmética de *a* hasta *b* con razón de *c* (siendo *b > a* y *c > 0*). O la secuencia aritmética de *a* hasta *b* con razón de *c* (siendo *a > b* y *c < 0*). O la secuencia geométrica de *a* hasta *b* con razón de *c* (siendo *b > a* y *c > 1*).
- Tablas de una o varias dimensiones
- Listas. Las listas, por simplicidad, las implementaremos como un array medio lleno. Exactamente una lista será representada por la tupla *a,m,n*. Dónde a es el puntero a la base del *array*, m es el tamaño del mismo y n el número de elementos en la lista. Con esa sencilla implementación podemos, si lo vemos conveniente, modelar la lista como un tipo de datos con las funciones correspondientes. Por lo tanto la gneración de los elementos de una lista se reduce a la una tabla o array.
- Ficheros de texto

Para cada fuente de datos vamos a ver la forma de generar sus valores para poder usar posteriormente los esquemas secuenciales. Vamos a usar la estructura de control *for* en todos los casos (esta se puede transformar en otra *while* tal como hemos visto).

1. Generación de secuencias numéricas:

```
int i;
…
for(i = a; i < b; i = i+c ){    // secuencia aritmética con a <= b, c > 0
       //     usar i
}
for(i = a; i > b; i = i+c ){    // secuencia aritmética con a >= b, c < 0
       //     usar i
}
for(i = a; i < b; i = i*c ){    // secuencia geométrica con a <= b, c > 1
       //     usar i
}
```

En las secuencias numéricas seguimos la convención Java. Las secuencias de *a* hasta *b* incluyen *a* pero no *b*.

2. Generación secuencias de pares de enteros *i, j* con *i* desde *a1* hasta *b1* y pasos de *c1* (*a1 <= b1, c1 > 0*) y *j* desde *a2* hasta *b2* y pasos de *c2 (a2 <= b2, c2 > 0)*. De la misma forma podríamos generar tripletas *i, j, k.*

```
int i, j;
…
for(i = a1; i < b1; i = i+c1){
       for(j =a2; j < b2; j= j+c2){
              // usar i,j
       }
}
```

3. Generación de elementos de un array *d* cuyo número de elementos es *dSize.* Es un caso particular de los esquemas en 1 usando los enteros generados como índices del *array.*

```
#define dTam …
T d[dTam];
int dSize;
int i;
…
for(i = 0; i < dSize; i++ ){
       //     usar d[i]
}
```

La generación de los caracteres de una cadena (donde *T* es *char*) es un caso particular de caso anterior con *dSize* dado por *strlen(d).*

4. Generación de elementos de un *array d* de dimensión 2 cuyo tamaño es *dFilas x dColumnas.*

```
#define dFilas …
#define dColumnas  …
T d[dFilas][dColumnas];
int i, j;
…
for(i = 0; i < dFilas ; i++){
       for(j =0; j < dColumnas; j++){
              // usar d[i][j]
       }
}
```

5. Generación de los elementos de un fichero.

```
FILE * f;
String s;
StringD sm;

…

f = fopen("Nombre del Fichero","r");

for(s = fgets(sm,100,f); s!=NULL; s = fgets(sm,100,f)) {
       // usar s
}
```

Acumuladores

En los temas anteriores vimos diferentes acumuladores: máximo, mínimo, suma, contador, existe, para todo, buscar, seleccionar, ejecuta para todo, etc.

Un acumulador de elementos de tipo S, en general, los podemos definir como:

- Una variable de tipo T
- Una función de inicialización
- Una función de acumulación de elementos de tipo S en la variable de tipo T.
- Una función para obtener los resultados acumulados en el acumulador
- Una condición sobre el acumulador para salir del bucle.

Algunos ejemplos de acumulador son:

Suma

- acum de tipo Integer o Double
- acum = 0;
- acum = acum +e.
- el resultado es la propia acum
- no hay

Existe

- acum de tipo boolean
- acum = false;
- acum = condición a cumplir por el elemento buscado.
- el resultado es la propia acum
- acum es verdadero

Agrega en Lista

- acum de tipo lista que en C sería de la forma T*, int m, int n.
- int = 0;
- añadir elemento a lista
- el resultado es la propia lista
- no hay

Y de forma similar el resto de los acumuladores.

Filtros y Transformaciones

Con las ideas anteriores podemos abordar la solución de problemas mediante esquemas secuenciales. Para usar los esquemas secuenciales vistos en temas anteriores tenemos que hacer algunos ajustes para partir de la generación de los datos tal como la hemos explicado. El esquema general será de la forma:

```
Declaración de variables;

Declaración de la variable del acumulador;
inicializa acumulador;

for(Generación de valores) {
              transformación de valores si es necesario;
              if(filtro de valores transformados) {
                     acumula;
              }
              transformación de valores si es necesario;
              if(condición sobre acumulador){
                     salir del bucle;
              }
}
obtener resultados del acumulador;
```

En el esquema anterior con el *for* (en alguna de sus versiones anteriores) generamos los valores. En algunos casos estos tienen que ser transformados. El ejemplo más sencillo es cuando leemos un fichero cuyas líneas representan números reales. La etapa de transformación consiste en obtener un número real a partir de la cadena de caracteres leída. En algunos casos habrá filtro y en otros no. Para completar el esquema tenemos que elegir un acumulador. Los acumuladores básicos que vimos en temas anteriores eran: existe, para todo, contador, suma, agrega en lista, agrega en conjunto, etc.

Ejemplos

Ejemplo1: Dado un fichero de número reales *(Numeros.txt),* que contiene en cada línea la representación de un número real, seleccionar en una lista los cuadrados de aquellos comprendidos entre dos valores dados *a* y *b*. Diseñar una función que tome *a*, *b* y el nombre del fichero como parámetros de entrada y devuelva una lista.

Las excepciones las devolvemos como un resultado entero: positivo si todo es correcto, -1 si no se encuentra el fichero, -2 si hay más datos de los que cabe en el array proporcionado, -3 si hay error de formato en los números leídos, -4 si los parámetros de entrada no son válidos.

El filtro es *(r > a && r < b).* La fuente de datos son las líneas de un fichero de texto. Debemos convertir las líneas de texto a números reales (con *atof*) para poder filtrarlos y añadirlos a la lista de salida.

Para representar la lista añadimos tres parámetros más: un puntero a *double* (que señalará a la base del *array* que contiene los datos de salida, un entero que indicará el tamaño disponible del *array* y un puntero a entero (número de datos seleccionados).

```
int getLista(const double a, const double b,
              const char * fichero, double * dt, int m, int * n){
```

```
	FILE * f;
	char sm[256];
	char * s;
	double num;
	int r = 1;
	int i = 0;
	char * cero = "0.";

	if(a >= b)
		return -4;
	f = fopen(fichero,"r");
	if(f==NULL)
		return -1;
	for(s = fgets(sm,100,f); s!=NULL; s = fgets(sm,100,f)) {
		num = atof(s);
		if(num==0 && strcmp(s,cero)!=0){
			r=-3;
			break;
		}
		if(num > a && num < b) {
			if(i==m){
				r = -2;
				break;
			}
			dt[i] = num;
			i++;
		}
	}
	*n = i;
	fclose(f);
	return r;
}
```

Repetir el problema usando el tipo *ListaDouble* implementado anteriormente. Repetir el problema anterior pero devolver un conjunto.

Ejemplo 2: Escribir una función que lea de la consola los elementos de una tabla de dos dimensiones. Los parámetros de entrada son la tabla, y el número de filas y columnas. Por comodidad hemos definido el tipo Matriz como:

```
#define FIL 5
#define COL 5
typedef double Matriz[FIL][COL];
```

La función pedida tiene la forma:

```
void getTabla(Matriz d, int fil, int col){
	int i, j;
	double r;
	for(i = 0; i < fil ; i++){
		for(j =0; j < col; j++){
			printf("Escriba el dato de la posicion %d %d\n", i,j);
			scanf("%f",&r);
			d[i][j] = r;
		}
	}
}
```

Ejemplo 3: Diseñar una función que imprima un dato de tipo Matriz en la consola por filas

```
void imprimeTablaEnConsola(Matriz d, int fil, int col){
	int i, j;
	double r;
	for(i = 0; i < fil ; i++){
		for(j =0; j < col; j++){
			printf("%7.3f  ",d[i][j]);
		}
		printf("\n");
	 }
}
```

Ejemplo 4: Diseñar una función que actualice una fila de un dato de tipo Matriz a partir de una tabla de números reales

```
void getFila(Matriz m, int nFil, int nCol, int fila, double * d){
	int j;
	assert(fila >= 0 && fila < nFil);
	for(j = 0; j < nCol; j++){
			m[fila][j] = d[j];
	}
}
```

Ejemplo 5: Diseñar una función que dada una tabla de cadenas de caracteres con el formato adecuado la convierta en otra tabla de números reales

```
void convierteADouble(String * ss, int n, double * d){
	int i;
	for(i = 0; i < n; i++){
			d[i] = atof(ss[i]);
	}
}
```

Ejemplo 6: Diseñar una función para leer los datos de una Matriz de un fichero. El fichero tiene en cada línea escritos números reales separados por comas. Hay tantos números reales en cada línea como elementos tiene la Matriz en cada fila.

```
void getTablaDeFichero(Matriz m, const int nFil, const int nCol,
                               const String fichero){
	FILE * f;
	StringD sm;
	String s;
	double r;
	StringD buf;
	String fString[COL];
	double fNum[COL];
	int nc;
	int nf = -1;

	assert(nCol <= COL);
	assert(nFil <= FIL);

	f = fopen(fichero,"r");

	for(s = fgets(sm,100,f); s!=NULL; s = fgets(sm,100,f)) {
		split(s," ,",&nc,fString, buf);
		assert(nc==nCol);
		convierteADouble(fString,nc,fNum);
```

```
                nf++;
                getFila(m,nFil,nCol,nf,fNum);
        }
        assert(nf == nFil-1);
        fclose(f);
}
```

Modificar la implementación para devolver valores enteros que informen de situaciones excepcionales, o del funcionamiento en modo normal, en sustitución de las sentencias assert usadas.

Ejemplo 7: Diseñar una función que devuelva el punto más cercano al origen de entre los que están escritos en las líneas de un fichero y pertenecen al primer cuadrante. En cada línea del fichero hay dos números reales separados por comas que representan la *x* y la *y* del punto. Diseñar previamente una función que dado un punto devuelva su distancia al origen.

```
void getPuntoMasCercano(const String fichero, Punto * p){
        FILE * f;
        char sm[256];
        String s;
        Punto p;
        Punto masCercano;
        double menorDistancia;
        double distancia;
        int esPrimero = 1;

        f = fopen(fichero,"r");

        for(s = fgets(sm,100,f); s!=NULL; s = fgets(sm,100,f)) {
                createPuntoDeString(&p,s);
                if(getXPunto(p) >= 0. && getYPunto(p)){
                        distancia = getDistanciaAlOrigen(p);
                        if(esPrimero || distancia < menorDistancia) {
                                clonePunto(&masCercano,p);
                                menorDistancia = distancia;
                        }
                        if(esPrimero) esPrimero = 0;
                }

        }
        fclose(f);
        *p = masCercano;
}
```

Ejemplo 8: Diseñar una función lea puntos de un fichero y escriba en un fichero de salida los puntos que están en el primer cuadrante. En cada línea del fichero de entrada hay un punto.

```
void escribePuntos(const String fichero1, const String fichero2){
        FILE * fe;
        FILE * fs;
        char sm[256];
        String s;
        Punto p;

        fe = fopen(fichero1,"r");
        fs = fopen(fichero2,"w");
```

```
	for(s = fgets(sm,100,fe); s!=NULL; s = fgets(sm,100,fe)) {
		createPuntoDeString(&p,s);
		if(getXPunto(p) >= 0. && getYPunto(p)>= 0){
			fprintf(fs,"%s\n",toStringPunto(p,sm));
		}
	}

	fclose(fe);
	fclose(fs);
}
```

Ejemplo 9: Diseñar una función que cuente el número de veces que aparece un carácter en una cadena.

```
int nv(const String s, const char c){
	char * p;
	int n =0;
	for(p=s;*p != '\0';p++){
		if(*p == c){
			n++;
		}
	}
	return n;
}
```

Para generar los caracteres de una cadena usamos las posibilidades de la aritmética de punteros y tenemos en cuenta que la cadena está acabada por un carácter nulo. Este esquema de recorrer los elementos de una tabla (inicializar un puntero a la base de la tabla, suma uno al puntero en cada iteración y comprobar el fin de la tabla cuando se cumpla alguna condición) puede ser utilizado en tablas de cualquier tipo siempre que sepamos la propiedad que cumplen el elemento final de la tabla.

Ejemplo 10: Diseñar una función que cuente el número de palabras que aparece en un fichero. Suponemos que son palabras las sub-cadenas de caracteres separadas por delimitadores que hay dentro de otra cadena o fichero. Suponemos que los delimitadores son " ;,".

```
int cuentaPalabras(const String fichero){
	FILE * fe;

	StringD sm;
	String s;
	String sp[50];
	int np = 0;
	int npl;
	fe = fopen(fichero,"r");

	for(s = fgets(sm,100,fe); s!=NULL; s = fgets(sm,100,fe)) {
		split(s,";, ",&npl,sp,sm);
		assert(npl < 50);
		np = np +npl;
	}

	fclose(fe);
	return np;
}
```

Modificar la implementación para devolver valores enteros que informen de situaciones excepcionales, o del funcionamiento en modo normal, en sustitución de las sentencias *assert* usadas.

10. 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*, *start*, *va_arg* y *va_end*. Estas macros están definidas en *stdarg.h.*

```
Lista  toListDeParametros(const int numArgs, ... ){
       TIPODEDATO p;
       int i;
       int sizeDato = sizeof(TIPODEDATO);
       Lista lis = _listas.create(sizeDato);
       va_list args;

       va_start(args, numArgs);
       for(i = 0; i < numArgs; i++){
              p =  va_arg(args, TIPODEDATO);
              _listas.add(lis,&p);
       }
       va_end(args);
       return lis;
}
```

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 y en la lista inicializada por *va_start*. Finalmente usamos *va_end*. En *TIPODEDATO* deberemos concretar el tipo de los parámetros variables.

11. Gestión de la memoria dinámica

En C es posible obtener Antes memoria dinámicamente con la funciones *malloc o calloc*. Para su manejo adecuado es muy importante incluir el archivo *stdlib.h* (donde están definidas *malloc, calloc y free*) en cualquier programa que use dichas funciones. En otro caso el comportamiento puede quedar indefinido.

La gestión de mormoria dinámica nos permite implementar timo cuyo tamaño pueda varair en tiempo de ejecución. Por ejemplo una lista de tamaño variable.

La implementación que se dio de la lista se cambiaría ahora sutiyendo el array de tamaño fijo por bloques de memoria que se pueden aumentar cuando convenga. La implementación de los métodos de factoría de la lista sería ahora (sólo incluimos algunos fragmentos del código):

```
// ListaDouble.h

#define Comparator int (* order)(const void * e1, const void * e2)

typedef struct {
       int tam;
       int size;
       double * datos;
} ListaDouble;

void createListaDouble(ListaDouble * lis, double * dt, const int tam);
int addListaDouble(ListaDouble * lis, const double e);

...

// ListaDouble.c



void createListaDouble(ListaDouble * lis, double *dt, const int tam){
    lis->datos = (double *) malloc(tam*sizeof(double));
    lis->tam = tam;
    lis->size=0;
}

int addListaDouble(ListaDouble * lis, const double e){
    int r = +1;
    if(lis->size == lis->tam){
       double * old = ls->datos;

       lis->datos = (double *) malloc(2*tam*sizeof(double));
       memcpy(lis->datos,old, tam*sizeof(double));
       tam = 2*tam;
       free(old);
    }
    lis->datos[lis->size]=e;
    lis->size++;
    return r;
}

void freeListaDouble(const ListaDouble lis){
     free(lis.datos());
}

...
```

Como vemos si la memoria está llena se busca un nuevo bloque de memoria, se copian los datos a ese bloque de memoria y se libera el antiguo. Ahora necesitamos un anueva función: *freeListaDouble*. Esta función es necesaria para liberar la memoria ocupada por los datos internos de las lista. Esta función debe ser llamada al final del ámbito donde la lista ha sido declarada.

La gestión de la memoria dinámica no es una tarea sencilla. La primera idea a tener en cuenta es que para liberar memoria hay que tener un puntero a ella porque en otro caso esto es

imposible. Por lo tanto si uno se demora demasiado en liberar un bloque de memoria, puede que se pierdan todos los punteros a ese bloque, y no queden oportunidades de liberarlo.

El momento justo para liberar un bloque de memoria, es aquel donde todos los punteros que quedan a ese bloque no van a ser usados en el futuro. Eso es más fácil de decir que de hacer. Cuando tenemos estructuras anidadas (por ejemplo, listas de cadenas, etc.), hay que elegir el orden de liberación de forma adecuada para no cortarnos la ruta hasta los objetos que queramos liberar. En general, conviene liberar desde *adentro* hacia *afuera*. Por ejemplo, si tenemos una lista de cadenas deberíamos liberar primero todas las cadenas y por último la memoria específica de la lista.

Para variables que tengan un ámbito bien delimitado (un bloque de código por ejemplo) podemos usar una aproximación sencilla: declarar las variables, obtener memoria dinámica y liberar la memoria al final del bloque. Pero en general la situación suele ser más complicada.

Hay algunos problemas que debemos tener en cuenta para una buena gestión de la memoria dinámica. El primero es el de los punteros colgados (*Dangling pointers*). Es decir punteros que señalan a zonas de memoria que ya han sido liberadas ya sea mediante free, si era memoria dinámica, o por haber terminado el correspondiente bloque si era memoria estática. Es los que ocurre en el siguiente ejemplo:

```
{
   char *dp = NULL;
       {
              char c;
             dp = &c;
              ...
       }
       /* c ha desaparecido */
       /* dp es ahora un puntero colgado */
}
```

La solución es dar el valor *NULL* al puntero cuando se libera la memoria mediante *free* o cuando se termina el ámbito de la variable local a la que señala.

```
{
   char *dp = NULL;
       /* ... */
       {
             char c;
              dp = &c;
              ...
              dp = NULL;
       }
}
```

Un caso particular del problema anterior es la devolución por una función de un puntero a una variable local de la misma. Es el problema que se presenta en:

```
int * nuevoEntero() {
       int a = 27;
       int * r = &a;
        return r;
```

```
}

….
int * e = nuevoEntero();
```

Cuando termina la llamada a la función la variable local *a* ha desaparecido. El puntero *e* señala a una zona de memoria ya liberada. Para resolver este problema en particular lo más sensato es añadir a la función diseñada un parámetro adicional que señalará a una variable que queremos actualizar:

```
int * nuevoEntero(int * r) {
       int a = 27;
       *r = a;
        return r;
}
…
int b
int * e = nuevoEntero(&b);
```

Otro problema que hay que evitar es la doble liberación de la memoria. Es decir llamar a *free(p)* varias veces sobre una misma zona de memoria. Este problema ocurre al liberar los elementos compartidos por una lista, conjunto, etc. La solución a este problema es llevar una gestión de la memoria que se ha ido pidiendo y la que se ido liberando. No entramos aquí en los detalles dejando este tema como ejercicio.

12. Ejercicios

1. Implementar las funciones sobre cadenas

```
String quitaSaltoDeLinea(String s);
String sustituyeChar(const String s, const char c1, const char c2, String sd);
String * split(const String s, const String delim, int * nt, String * as, String sd);
boolean estaEn(const char c, const String s);
String cloneString(const String s);
```

Las funciones anteriores tienen la siguiente funcionalidad:

- *quitaSaltoDeLinea:* Sustituye el carácter de fin de línea por un carácter '\0'.
- *sustituyeChar:* Sustituye en la cadena s el carácter c1 por el c2 y devuelve la cadena transformada. La función tiene un parámetro adicional (sd) que señala a una zona de memoria donde se ubicará la cadena transformada.
- *split:* Parte la cadena s en tokens delimitados por lo caracteres contenidos en la cadena *delim*. Devuelve un puntero a la base de un array de cadenas (los tokens). El parámetro *nt* queda actualizado al número de tokens. El parámetro as señala a la base de un array que quedará actualizado con los tokens. El parámetro *sd* señala a una zona de memoria donde se ubicará la cadena transformada (sustitución de los caracteres delimitadores por el carácter '\0'.
- *estaEn:* Verdadero si el carácter *c* está en *s*.
- *cloneString:* Crea una copia de la cadena *s*.

2. Implementar las funciones siguientes:

```
int sqrtInt(int a);
boolean esMultiplo(int a, int b);
boolean esDivisor(int a, int b);
boolean esPrimo(int a);
int mcd(int a, int b);
int mcm(int a, int b);
boolean esPrimo(int n);
Lista primosMenoresQue(int n);
int *  primosMenoresQueV(int n, int * np, int * p);
int sgnDouble(double d);
```

3. Diseñar e implementar el tipo *Racional* (racional.h, .c) tal como lo hemos visto en capítulos anteriores.
4. Dado um fichero de texto, cuyas líneas contienen el numerador y el denominador de um racional separados por una coma, implementar las siguientes funciones
   - Leer el fichero y devolver uma lista de racionales.
   - Encontrar si todos los racionales escritos en el fichero son mayores que uno dado.
   - Encontrar el mayor racional.
   - Construir un array ordenado de racionales con aquellos racionales del fichero que sean positivos.