LENGUAJES, TECNOLOGÍAS Y PARADIGMAS DE LA PROGRAMACIÓN

Práctica 2.
Programación Imperativa con C

Contenido

1. Introducción

El objetivo de esta práctica consiste en profundizar en los conceptos más importantes que subyacen bajo el lenguaje de programación C. Para ello, plantearemos diversos ejercicios, que demuestren las capacidades de este lenguaje. Por último, se planteará un reto a resolver, que deberá ser entregado en tiempo y forma.

En las siguientes secciones, trataremos los aspectos más interesantes del lenguaje de programación C en cuanto a la programación estructurada, sus tipos de datos, la gestión de la memoria y finalmente la programación procedural.

Proporcionamos en Poliforma-T una tarjeta de referencia rápida (refcard) que en 2 hojas resume la sintaxis, tipos de datos y funciones más importantes del lenguaje de programación C. Recomendamos encarecidamente su consulta en el transcurso de esta práctica.

2. Programación estructurada

Utilizar comandos no estructurados (saltos arbitrarios) nos puede llevar a generar código spaghetti, es decir, código muy difícil de comprender e imposible de mantener. Afortunadamente, los lenguajes de alto nivel introducen comandos estructurados, que generalmente poseen un único punto de entrada y un único punto de salida, evitando saltos arbitrarios, manteniendo el código controlado y determinable. La mayoría de lenguajes de programación introduce un subconjunto básico de constructores que nos permite definir el flujo de control de nuestro programa de una manera muy sencilla.

En el caso de C, listamos los principales brevemente:

Secuencia

```
statement;
statement;
```

Bloque

```
{
     statement;
     statement;
}
```

En las expresiones anteriores simplemente tener en cuenta que *statement* puede ser reemplazado en cualquier momento por un bloque de instrucciones { … }. En este caso no sería necesario añadir un ';' al finalizar el bloque.

No parece necesario tener que explicar el significado de estos constructores, ya que son los que nos encontramos en todos los lenguajes de programación. Los usaremos en las próximas secciones de la práctica. Para mayor información sobre su sintaxis es posible consultar la tarjeta de referencia rápida de C en Poliforma-T.

Bifurcación

```
if (expr) statement;
else if (expr) statement;
else statement;

switch (expr) {
     case const 1: statement; break;
     case const 2: statement; break;
     default: statement
}
```

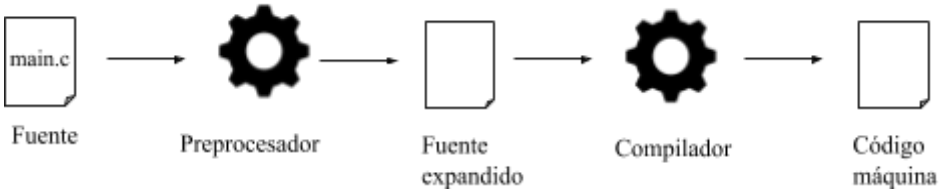
Iteración

```
while (expr) {
     statement;
}

for (init_expr; cond_expr; end_expr) {
     statement;
}

do {
     statement;
} while (expr);
```

3. Tipos de datos

Como hemos visto en clase, el cometido fundamental de un programa imperativo consiste en manipular datos para modificar el estado de la máquina y alcanzar la solución a nuestro problema. En esta sección exploraremos los tipos de datos existentes en C, cómo podemos manipularlos.

Para empezar, todo lenguaje de programación proporciona una manera de definir constantes. En C, las constantes se definen utilizando la "directiva" **`#define`**. Una directiva es un comando que comienza por el símbolo "**#**" y que es tratada por el ***pre****procesador* de C, un elemento que analiza el código y lo expande antes de ser pasado al compilador. El proceso sería como sigue:



Podemos definir una constante MAX del siguiente modo:

```
#define MAX 100
```

Tras efectuar dicha declaración es posible utilizar MAX en el resto de nuestro código fuente:

```
int vector[MAX];
```

Cuando el preprocesador de C encuentra el identificador MAX lo reemplaza por el valor que aparece a su derecha en la directiva **`#define`**, sea lo que sea.

Una vez ya sabemos definir constantes, analicemos los tipos de datos existentes en C. Como hemos visto en clase, existen primitivos y compuestos. Los primitivos son: **`int`**, **`char`**, **`float`**, **`double`**, algunos modificables con los modificadores **`long`**, **`short`**, **`unsigned`**. Los complejos son: **`struct`**, **`union`** y **`arrays`** (donde un string es un tipo particular de array).

Una primitiva extremadamente útil en nuestros programas, especialmente cuando manipulamos memoria, es **`sizeof()`**, que acepta o bien un tipo (int, float, etc.) o bien una variable, y devuelve el número de bytes que ocupa en memoria:

```
int i = 10;
char c = 'a';
float f = 4.5;
double d = 4.5;
long l = 100;

printf("sizeof variables: %lu %lu %lu %lu %lu\n", sizeof(i), sizeof(c),
sizeof(f), sizeof(d), sizeof(l));
printf("sizeof types: %lu %lu %lu %lu %lu\n", sizeof(int), sizeof(char),
sizeof(float), sizeof(double), sizeof(long));
```

Los tipos de datos compuestos son más interesantes. Empezaremos por el **`struct`**. Este tipo de datos nos permite “empaquetar” varias variables juntas (similar a una clase de OOP). Para usarlo, generalmente primero definimos un nuevo tipo de estructura, y después creamos todas las variables de ese tipo que queramos.

```
struct MyStruct {
  int i;
  double d;
  char c;
};
```

Acabamos de crear un nuevo tipo de estructura, pero todavía no hemos definido ninguna variable de ese tipo. Lo hacemos a continuación, y accedemos a sus campos:

```
struct MyStruct myVar;
myVar.i = 100;
myVar.d = 4.5;
myVar.c = 'a';
```

Cuando creamos una nueva estructura generalmente definimos un nuevo tipo de datos, con un nombre único. Para simplificar su uso, podemos utilizar la palabra **`typedef`**, de este modo:

```
typedef definición_tipo nombre_tipo;
```

En el ejemplo que sigue definimos un nuevo tipo de datos denominado “MyStruct”.

```
typedef struct {
  int i;
  double d;
  char c;
} MyStruct;
```

A continuación podemos declarar todas las variables de este tipo de datos que deseemos:

```
MyStruct myVar; // No prefijamos la palabra struct
myVar.i = 100;
myVar.d = 4.5;
myVar.c = 'a';
```

Respecto a los arrays, su declaración es muy simple, usando la sintaxis básica:

```
tipo variable[DIMENSION];
```

Por ejemplo, en el siguiente fragmento de código declararíamos un array de 100 componentes, las inicializaríamos y las imprimiríamos por pantalla.

```
#define MAX 100
int v[MAX];
int i;
for (i = 0; i < MAX; i++) v[i] = i;
for (i = 0; i < MAX; i++) printf("%d\n", v[i]);
```

El tipo del array puede ser cualquier cosa, incluso un tipo compuesto:

```
MyStruct v[10];
int i;
for (i = 0; i < 10; i++) {
  v[i].i = i;
  v[i].d = 4.5;
  v[i].c = 'a';
}
for (i = 0; i < 10; i++) printf("%d %f %c\n", v[i].i, v[i].d, v[i].c);
```

Podemos inicializar un vector con la sintaxis:

```
tipo variable[tamaño] = { elem1, elem2, ...};
```

Por ejemplo:

```
int v[5] = { 1, 2, 3, 4, 5}, v2[] = { 1, 2, 3, 4, 5 };
```

Esto nos obliga a que conozcamos el número de elementos de antemano.

Podemos definir vectores multidimensionales, por ejemplo, para definir una matriz de enteros de 10x10:

```
int m[10][10];
```

Un string es un vector de caracteres, donde el último carácter es el carácter nulo '\0'. Podemos declararlo de la manera habitual:

```
char str[10];
char str2[] = {'h', 'o', 'l', 'a', '\0'};
```

Aunque C nos ofrece una manera más cómoda de hacerlo (syntactic sugar):

```
char str[] = "hola"; // \0 implícito
```

Hay que tener en cuenta que un string en C es un vector normal y corriente, y lo único que podemos hacer con él de manera nativa es indexarlo con expresiones del tipo `v[i]` y obtener su longitud con el comando `sizeof()`. Recordar que `sizeof()` devuelve el número de bytes de su argumento. En nuestro caso es un vector de chars, donde cada char ocupa un byte, y por lo tanto el número de bytes del vector coincide con la longitud de la cadena (incluyendo el carácter nulo). Probar el código:

```
char str[] = "hola";
printf("%s -> %lu bytes", str, sizeof(str));
```

Si queremos manipular el string de una manera más cómoda, debemos de definir nosotros las primitivas, o bien utilizar las funciones incluidas en la librería estándar y declaradas en <string.h>: `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, etc. Consultar la tarjeta de referencia rápida de C para mayor información.

```
#include <string.h>
...
char str1[] = "hola";
char str2[] = " mundo";
char res[20];

strcpy(res, str1);
printf("%s\n", res);
strcat(res, str2);
printf("%s\n", res);
printf("%lu vs %lu\n", sizeof(res), strlen(res));
```

Por último, C soporta un tipo de datos denominado **puntero** (😟)**,** que nos permite apuntar a las direcciones de memoria donde se almacenan otras variables, y se declara utilizando la sintaxis:

```
tipo *ptr;
```

Que nos permite apuntar a la dirección de memoria de cualquier variable del tipo de datos especificado. Para obtener la dirección de memoria de una variable podemos utilizar el operador **address-of &**. El valor devuelto por este operador puede ser almacenado en un puntero del mismo tipo de la variable, por ejemplo:

```
int i = 100;
int *ptr = &i; // * forma parte del tipo, & para extraer la dir. memoria
printf("La variable i en direccion: %p, %p\n", ptr, &i);
```

Si poseemos un puntero a una posición de memoria, siempre podemos modificar el contenido de dicha posición de memoria utilizando el operador **dereference ***, por ejemplo:

```
int i = 100;
int *ptr = &i;
*ptr = 200;
printf("El valor de i es %d\n", i);
```

Podemos utilizar esta sintaxis para cualquier tipos de datos, por ejemplo, estructuras.

```
typedef struct {
  int i;
  double d;
} MyStruct;
MyStruct st;
MyStruct *ptr = &st;  // recuperamos la dirección de la estructura
```

Cuando tenemos un puntero a estructura, podemos usar el operador -> para acceder a sus campos:

```
ptr->i = 100;
```

Pregunta 1: ¿Cómo accederíamos a ese campo si no existiese el operador -> ?

Existe una relación especial entre punteros y arrays. Cuando declaramos una variable de tipo array, el nombre de la variable es un puntero al primer elemento del array. Así pues, un array se puede tratar como un puntero. La relación contraria también es cierta, un puntero se puede tratar como un vector, pudiendo usar índices:

```
int v[] = {1, 2};
int *ptr = v;
int i = *v;
ptr[1] = 200;
```

Esta relación puntero – array resulta especialmente útil cuando trabajamos con strings.

```
char v[] = "hi";
char *ptr = v;
char *str = "hola";
```

Por último, podemos incrementar/decrementar un puntero para iterar entre los elementos de un array, lo que denominamos aritmética de punteros:

```
int v[] = {1, 2};
int *ptr = v;
*ptr = 4;
++ptr;
*ptr = 5;
///
int v[] = {1, 2};
int *ptr = v;
ptr[0] = 4;
ptr[1] = 5;
```

Los punteros resultarán de gran utilidad cuando trabajemos con funciones más adelante.

4. Gestión de memoria

En C podemos distinguir dos tipos de memoria en cuanto a su gestión:

- **Memoria estática**: es la memoria que se utiliza cuando definimos variables en el cuerpo de una función. La mayoría de estas variables se almacena en el stack.
- **Memoria dinámica**: es la memoria que se utiliza cuando trabajamos con punteros, y la reservamos y liberamos a voluntad. Esta memoria se encuentra en el heap.

La gestión de la **memoria estática** es automática. Esto significa que no nos tenemos que preocupar por ella (🥳). Cuando declaramos una variable en el cuerpo de una función el compilador automáticamente reserva memoria en la pila para dicha variable. Cuando salimos de la función la variable se destruye automáticamente.

```
int main(int argc, char *argv[]) {
  int i;      // la variable 'i' se almacena en la pila
  struct {    // definimos una estructura anónima y declaramos una variable
    int i;
    float f;
  } myVar;    // la variable 'myVar' reside en la pila

  i = 10;
  myVar.i = 10;
}           // Memoria liberada, duración automática
```

Cuando se ejecuta la función `main()` la memoria para las variables `i` y `myVar` se asigna automáticamente en la pila. Cuando se sale de la función `main()` la memoria asignada a dichas variables se destruye automáticamente.

Por contra, la gestión de la **memoria dinámica** es manual. Para reservar memoria dinámica utilizamos la función **`malloc(bytes)`**. Esta función acepta como parámetro el número de bytes que queremos reservar y devuelve un puntero a dicha zona de memoria. El puntero devuelto es un puntero **`(void *)`**, totalmente agnóstico sobre aquello que va a almacenar, únicamente marca una zona de memoria como usable. Esto significa que a priori no posee ningún tipo, por lo que es necesario hacer un cast:

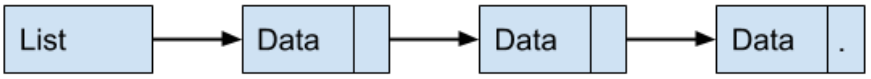
```
void *ptr1 = malloc(100); // espacio para 100 bytes (de lo que sea)
char *ptr2 = (char *)malloc(10*sizeof(char)); // espacio para 10 caracteres
```

Posteriormente, debemos liberar dicha memoria utilizando la función **`free(ptr)`**, que acepta como parámetro el puntero devuelto anteriormente.

Podemos usar esta estrategia para cualquier tipo de datos. Por ejemplo, con estructuras:

```
typedef struct {
  int i;
  double d;
} MyStruct;

MyStruct *st = (MyStruct *)malloc(sizeof(MyStruct));
st->i = 100;
free(st);
```

Con punteros y la gestión de memoria dinámica (`malloc()`, `free()`), somos capaces de crear estructuras de datos dinámicas, como por ejemplo, listas enlazadas. Una lista enlazada puede contemplarse como una secuencia de nodos entrelazados. Cada nodo puede contener en su interior cualquier cosa.



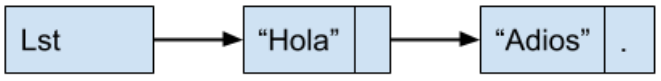
En C, el nodo se puede modelar utilizando un struct. Por lo tanto, una lista enlazada sería una secuencia de estructuras donde la primera apunta a la segunda, la segunda a la tercera, etc. A la hora de definir el nodo, nos encontramos con una relación recursiva: la estructura *node* posee un puntero que apunta a la misma estructura, que está siendo definida. Para resolver este tipo de definiciones es necesario asignar un nombre temporal a la estructura, como se muestra a continuación:

```
typedef struct st_node {
  char *data;
  struct st_node *next;
} node;
```

Con esta definición ya hemos creado la base para nuestra lista enlazada. A continuación creamos una lista enlazada sencilla:

```
node *lst = (node *)malloc(sizeof(node));
lst->data = "hola";
lst->next = (node *)malloc(sizeof(node));
lst->next->data = "adios";
lst->next->next = NULL;
```

Con lo que conseguiríamos algo parecido a esto:



Posteriormente, deberíamos de liberar toda la memoria reservado por `malloc()`. Así, siempre debemos recordar que cualquier `malloc()` debe ir seguido de un `free()`, una vez hayamos acabado de usar la memoria reservada.

Pregunta 2: ¿Cómo liberarías la memoria asignada a tras la llamada a esos 2 malloc()? Escribe el código en C y justifica tu respuesta.

5. Programación procedural

En ocasiones nos interesa incluir un fragmento de código en el interior de una caja negra e invocar dicha caja negra de manera repetida. Esta caja negra es lo que denominamos función, procedimiento, rutina o módulo. En C, recibe el nombre de función, y acepta parámetros de entrada y puede devolver un valor. Utilizamos la siguiente sintaxis:

```
<tipo> nombre(<tipo1> param1, <tipo2> param2, ...)
{
  /* código */
  return <valor>;
}
```

Si la función no devuelve nada, es necesario especificar como tipo de retorno *void*. A continuación presentamos un ejemplo:

```
void hello(int s) {
  printf("%d", s);
}
```

Por defecto, todos los parámetros se **pasan por valor**. Esto significa que su valor se copia en el cuerpo de la función invocada, y por tanto se convierte en una variable local a la función. Si se modifica, los cambios no son visibles desde fuera. Esto sucede con cualquier tipo, ya sea primitivo, compuesto o puntero. Probar el código:

```
typedef struct {
  int i;
} MyStruct;

void test(int i, MyStruct st) {
  i = 200;
  st.i = 200;
  printf("dentro: %d - %d\n", i, st.i);
}

int main(int argc, char *argv[]) {
  int i = 100;
  MyStruct st;
  st.i = 100;
  printf("antes: %d - %d\n", i, st.i);
  test(i, st);
  printf("despues: %d - %d\n", i, st.i);
}
```

Para permitir que una función modifique un parámetro, y este cambio sea visible desde fuera, debemos de trabajar con punteros. Cuando se pasa un puntero (una dirección de memoria) como parámetro de una función, el parámetro se pasa por valor, como todos. Sin embargo, en esta ocasión el valor del parámetro apunta a una dirección de memoria cuyo contenido es posible manipular desde el interior de la función, como en el siguiente ejemplo:

```
typedef struct {
  int i;
} MyStruct;

void test(int *i, MyStruct *st) {
  *i = 200;
  st->i = 200;
  printf("dentro: %d - %d\n", *i, st->i);
}

int main(int argc, char *argv[]) {
  int i = 100;
  MyStruct st;
  st.i = 100;
  printf("antes: %d - %d\n", i, st.i);
  test(&i, &st);
  printf("despues: %d - %d\n", i, st.i);
}
```

Este paso de parámetros (con punteros), es lo que denominamos en C como **paso por referencia**. En el caso de los arrays, por defecto el paso es por referencia, ya que cuando especificamos el nombre de un array, realmente estamos referenciando la dirección de memoria de su primer elemento. A continuación presentamos una función que permite concatenar dos cadenas de entrada y las deposita en una cadena de salida.

```
void concat(char *dst, char *src1, char *src2) {
  int i, j = 0;
  for (i = 0; src1[i] != '\0'; i++) {
    dst[j++] = src1[i];
  }
  for (i = 0; src2[i] != '\0'; i++) {
    dst[j++] = src2[i];
  }
  dst[j] = '\0';
}

int main(int argc, char *argv[]) {
  char src1[] = "hola";
  char src2[] = "mundo";
  char dst[20];

  concat(dst, src1, src2);
}
```

Si bien una función puede tomar un puntero como parámetro, también puede devolver un puntero como valor de retorno, teniendo en cuenta la duración de la memoria de tal variable (cuándo se libera, automática/pila vs. dinámica/heap):

```
int *func(){...}
```

Pregunta 3: ¿Qué error, respecto a la gestión de la memoria, encuentras en este código?

```
int *f(int a, int b) {
  int res = a + b;
  return &res;
}
```

En C, cuando utilizamos una función (o cualquier tipo de datos), es necesario que dicha función (o tipo de datos) haya sido previamente declarada. De otro modo el compilador nunca sabría si estamos invocando correctamente una función (o usando correctamente un tipo de datos). En los ejemplos anteriores siempre *hemos definido la función antes de invocarla* desde `main()`.

Realmente no es necesario **definir** la función antes de usarla; basta con **declararla**. **Declarar** una función consiste en listar el nombre de la función junto con sus parámetros de entrada y su tipo de retorno, es lo que denominamos el "prototipo" de una función. **Definirla** sería escribir su implementación. En el ejemplo anterior, el prototipo de la función `concat()` sería:

```
void concat(char *dst, char *src1, char *src2);       // Declaración
void concat(char *dst, char *src1, char *src2){...}   // Definición
```

Declarar una función consiste en escribir la signatura de la función, sin su cuerpo. Si hacemos esto para todas las funciones que usamos en nuestro programa, la compilación no debería de dar problemas. Estas declaraciones de funciones (y tipos de datos) suelen incluirse en ficheros especiales **.h**, que denominamos **ficheros de cabecera**, y que incluimos en nuestro programa con la directiva **`#include`**, antes de usar todas sus funciones. La directiva **`#include`** simplemente inserta el contenido del fichero **.h** especificado en el archivo fuente.

A modo de ejemplo, a continuación vamos a desarrollar una librería de funciones que nos permitan implementar una sencilla calculadora. Cuando diseñamos una librería de funciones que usaremos en nuestro programa, generalmente la creamos en un fichero **.c** independiente. En nuestro caso, denominaremos a este fichero **calculator.c**, y **definirá** las siguientes rutinas:

- `int add(int a, int b){...} // suma dos números enteros`
- `int sub(int a, int b){...} // resta dos números enteros`
- `int mult(int a, int b){...}// multiplica dos números enteros`

Definiremos un fichero de cabecera **calculator.h**, que **declare** (el prototipo, sin cuerpo) las rutinas anteriores:

- `int add(int a, int b);`
- `int sub(int a, int b);`
- `int mult(int a, int b);`

La librería será utilizada por nuestra función principal `main()` incluida en el fichero main.c, y por lo tanto antes de utilizar sus funciones, deberemos declararlas, importando para ello **calculator.h**. La función principal main() debería de funcionar con un código similar al siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include "calculator.h"
int main(int argc, char *argv[]) {
  int a, b;
  printf("Introduce a: ");
  scanf("%d", &a);
  printf("Introduce b: ");
  scanf("%d", &b);
  printf("a+b=%d, a-b=%d, a*b=%d", add(a, b), sub(a, b), mult(a, b));
  return 0;
}
```

Nótese que incluimos nuestros ficheros de cabecera entre "", mientras que las librerías de C se incluyen entre <>

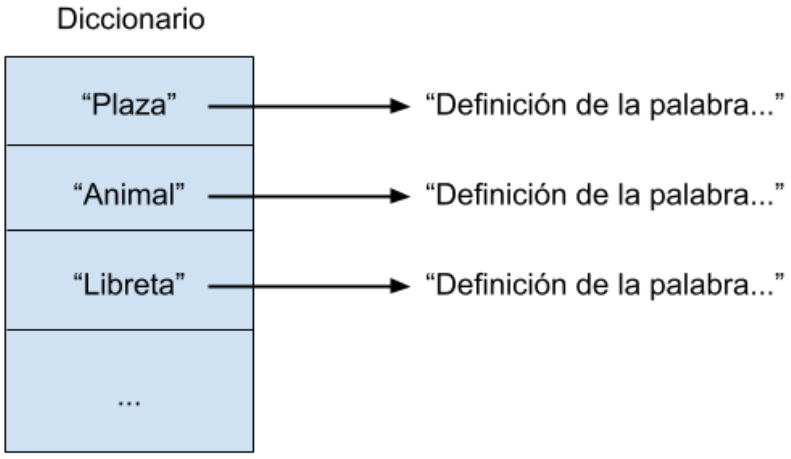
Por último, a continuación se presenta una posible implementación de la librería **calculator.c**:

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

int add(int a, int b) {
  return a+b;
}
int sub(int a, int b) {
  return a-b;
}
int mult(int a, int b) {
  return a*b;
}
```

6. Entregable

Una vez efectuados todos los ejercicios planteados en la práctica, proponemos efectuar una implementación dinámica del tipo de datos diccionario en C, algo que encontramos en otros lenguajes de programación como Python.

Podríamos pensar que un diccionario funciona como un 'diccionario en la vida real'



Un diccionario es una colección de pares (clave, valor). Podría ser considerado como una especie de array dinámico, donde los índices (las claves) no tienen por qué ser necesariamente números enteros. El diccionario que implementaremos poseerá las siguientes características:

- Será **dinámico**, en el sentido de que no se creará el par (clave, valor) hasta que sea necesario. Deberá trabajar por tanto internamente con *listas enlazadas de nodos*.
- Tanto claves como valores pertenecerán a los tipos de datos **int**, **float** o **char***, es decir, números y strings.
- Como C es fuertemente tipado, necesitaremos *rutinas de conversión* de tipos de datos para almacenar/recuperar los elementos del diccionario.
- Un diccionario, por definición, **no permite claves duplicadas**. Si insertamos un valor con una clave que ya existe, el valor asociado a la clave ha de ser *modificado* (cambiar el valor para dicha clave)

Para ello planteamos implementar una nueva librería. Como hemos visto en el punto 5, crearemos dos ficheros:

- **dict.c**: este fichero contendrá la **definición/implementación** de todas las operaciones de la librería.
- **dict.h**: este fichero contendrá la **declaración** de los tipos de datos y las operaciones que se implementarán en dict.c. Cualquier programa que desee trabajar con nuestra librería deberá primero incluir (`#include`) este fichero.

Paso 1

Como hemos comentado, el diccionario debe ser capaz de trabajar con claves/valores que pertenezcan a los tipos de datos int, float o char*. Para simplificar el almacenamiento de estos elementos crearemos un nuevo tipo *object* donde almacenaremos información sobre cada elemento, básicamente su tipo y su valor.

```
// type=0,1,2-int/float/string
typedef struct {int type; char data[MAX_DATA];} object;
```

Utilizaremos estas “cápsulas” para almacenar las claves y los valores del diccionario. El campo *data* contiene un vector de *char*, pero nosotros podremos almacenar en su interior cualquier cosa, haciendo castings entre punteros y tipos de datos. Para ello implementaremos rutinas que nos permitan construir estos valores a partir de los básicos:

```
void int_to_obj(int src, object *dst);
void float_to_obj(float src, object *dst);
void str_to_obj(char *src, object *dst);
```

Necesitaremos también rutinas que nos permitan efectuar las operaciones inversas, es decir, a partir de una “cápsula” de tipo *object* obtener los tipos básicos:

```
void obj_to_int(object src, int *dst);
void obj_to_float(object src, float *dst);
void obj_to_str(object src, char *dst);
```

Como se puede observar, todas estas rutinas depositan los resultados de la conversión en una dirección de memoria, de manera que deberemos haber reservado memoria previamente para ello. A continuación se muestra un fragmento de código usando estas rutinas de conversión, que se recomienda probar antes de continuar con el desarrollo de la práctica:

```
// reserva estática de variables
object obj;
int i;

// conversiones
int_to_obj(10, &obj);  // int → object
obj_to_int(obj, &i);   // object → int

// impresión
printf("%d", i);
```

Para implementar estas rutinas de conversión podemos trabajar con las siguientes herramientas:

- `int atoi(char *str)`, `float atof(char *str)`: declaradas en <stdlib.h>, permiten transformar strings a números.
- `sprintf(char *str_dest, char *format, ...)`: declarada en <stdio.h>, funciona como `printf`, pero en lugar de imprimir por consola, imprime en un string; puede usarse para transformar números a cadena.
- castings regulares (int) (float) para transformar entre números.

Se recomienda, una vez implementadas estas funciones, probar el siguiente código antes de continuar con la práctica:

```
// Demo int
object obj;
int i;

int_to_obj(10, &obj);
obj_to_int(obj, &i);

printf("%d\n", i);

// Demo float
float f;

float_to_obj(76.43f, &obj);
obj_to_float(obj, &f);

printf("%f\n", f);

// Demo string
char s[20];

str_to_obj("hola mundo", &obj);
obj_to_str(obj, s);

printf("%s\n", s);
```

Paso 2

Un diccionario es una lista enlazada que contiene una colección de pares (clave, valor) donde tanto la clave como el valor son del nuevo tipo *object*. Definiremos por tanto el tipo de nodo como:

```
typedef struct st_dict_node {
  object key;
  object value;
  struct st_dict_node *next;
} dict_node;
```

Un diccionario sería un puntero al primer elemento de la lista. Podemos declarar un nuevo tipo para ello:

```
typedef struct {
  dict_node *first;
  int len;
} dict;
```

Paso 3

Deberás definir las siguientes funciones:

```
// crear, eliminar
dict *dict_new();                  // Retorna un puntero
void dict_destroy(dict *dic);      // Libera la memoria, de forma correcta!!

// añadir, buscar, eliminar por clave
int dict_add(dict *dic, object key, object value);    // 0 si éxito, -1 si fallo
int dict_search(dict *dic, object key, object *dst);  // 0 si éxito, -1 si fallo
int dict_remove(dict *dic, object key);               // 0 si éxito, -1 si fallo

// utilidades
int dict_len(dict *dic);                  // Retorna longitud del diccionario
int dict_equals(dict *dic1, dict *dic2);  // 0 si iguales, -1 si no

// iterar
dict_node *dict_first(dict *dic);           // primera entrada; NULL si no hay
dict_node *dict_next(dict *dic, dict_node *current); // sig ent; NULL si no hay

// obtener clave/valor desde una entrada del diccionario
void dict_key(dict_node *pair, object *dst);
void dict_value(dict_node *pair, object *dst);
```

* Queda a juicio del alumno definir qué significa que 2 diccionarios sean iguales, así como los posibles códigos de error que puedan devolver las funciones.

* También queda a juicio del alumno decidir cuándo la llamada a una función ha fallado o no

A continuación mostramos cómo manipular un diccionario en un hipotético fichero main.c, por lo que recomendamos probar este código para comprobar la correcta implementación del diccionario:

```
#include <stdio.h>
#include <stdlib.h>
#include "dict.h"

void main() {

  dict *dic;            // Diccionario
  object key, value;    // Almacenes de datos

  // creamos diccionario
  dic = dict_new();

  // insertamos dos elementos
  int_to_obj(1, &key); str_to_obj("Pepe", &value);
  dict_add(dic, key, value);

  int_to_obj(2, &key); str_to_obj("Juan", &value);
  dict_add(dic, key, value);

  // buscamos elemento
  char str[50];
  int_to_obj(1, &key);
  dict_search(dic, key, &value);
  obj_to_str(value, str);
  printf("%s", str);

  // iteramos
  int i;
  dict_node *node;

  node = dict_first(dic);
  while (node != NULL) {
    dict_key(node, &key); dict_value(node, &value);
    obj_to_int(key, &i); obj_to_str(value, str);
    printf("%d->%s", i, str);

    // Accedemos al siguiente nodo
    ??¿¿
  }

  // destruimos el diccionario (libera TODO)
  dict_destroy(dic);

}
```