# Programación II
# 1º Grado Ingeniería Informática

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

## Objetivos:

- Aprender a utilizar un TAD ya implementado, usando como documentación su fichero de interfaz (.h)
- Aprender a construir bibliotecas de estructuras de datos, definiendo los tipos de datos como TIPOS DE DATOS OPACOS.
- Aprender a manejar la memoria dinámica.
- Ser capaz de pasar por línea de comandos argumentos al `main`.
- Ser capaz de automatizar la compilación del proyecto a través del `makefile`

## Descripción:

En esta práctica veremos cómo manipular y crear una clave que almacenará contraseñas cifradas utilizando un TAD. Sabemos que en C es posible tratar las cadenas de caracteres de forma muy directa, como agrupaciones estáticas o dinámicas de datos como

```
char v[10];
ó
char *v;
(y luego generando con malloc tantas posiciones como se requiera)
```

Sin embargo, el uso de TAD es en general una buena práctica de programación que resulta especialmente adecuada para la reutilización de código. El único inconveniente es que la implementación es más costosa, puesto que exige **definir, diseñar y construir** todos los procedimientos y funciones para construir el TAD (en nuestro caso la "clave") y manejar la información que contenga. A cambio, el diseño con TAD conlleva innumerables ventajas puesto que, una vez implementado, su manipulación es muy intuitiva y simple y, sobre todo, independiente de los tipos de datos que encapsule el TAD.

Lo crucial de definir un TAD correctamente está en tres aspectos:

- Hacerlo totalmente independiente de los detalles de implementación interna: esto se consigue usando **tipos opacos de datos** (`void *` en el módulo de definición de la biblioteca del TAD `fichero.h`)
- Permitir de manera natural **cambiar el tipo de datos concreto** que almacena el TAD: esto se consigue utilizando una **definición abstracta** (en nuestro caso el nuevo tipo de datos `TELEMENTO`) en el módulo de definición, mediante una sentencia `typedef` (para cambiar el tipo de datos concreto basta modificar el `TELEMENTO` en `fichero.h` y en `fichero.c`). Esto permitiría por ejemplo alternar entre claves que sean meramente numéricas (p.e. el típico PIN de un teléfono móvil) o claves que tengan cualquier carácter. Para esto habría simplemente que alternar entre un TELEMENTO short o un TELEMENTO char, por ejemplo.
- Construir un **repertorio de procedimientos y funciones** suficientemente amplio para manipular el TAD, pero en los que no se incluya ningún aspecto concreto (dependencia) con determinadas tareas o tipos de datos. Así garantizamos su completa reutilización.

# Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

En esta práctica crearemos un TAD para familiarizarnos inicialmente con esta nueva forma de diseñar programas. Seguidamente construiremos los procedimientos/funciones básicas de manipulación para dicho TAD. En prácticas posteriores haremos uso de este para tareas de más alto nivel, y ahí es donde comenzaremos a apreciar las ventajas de reutilización de código.

## Cómo trabajar con tipos de datos opacos:

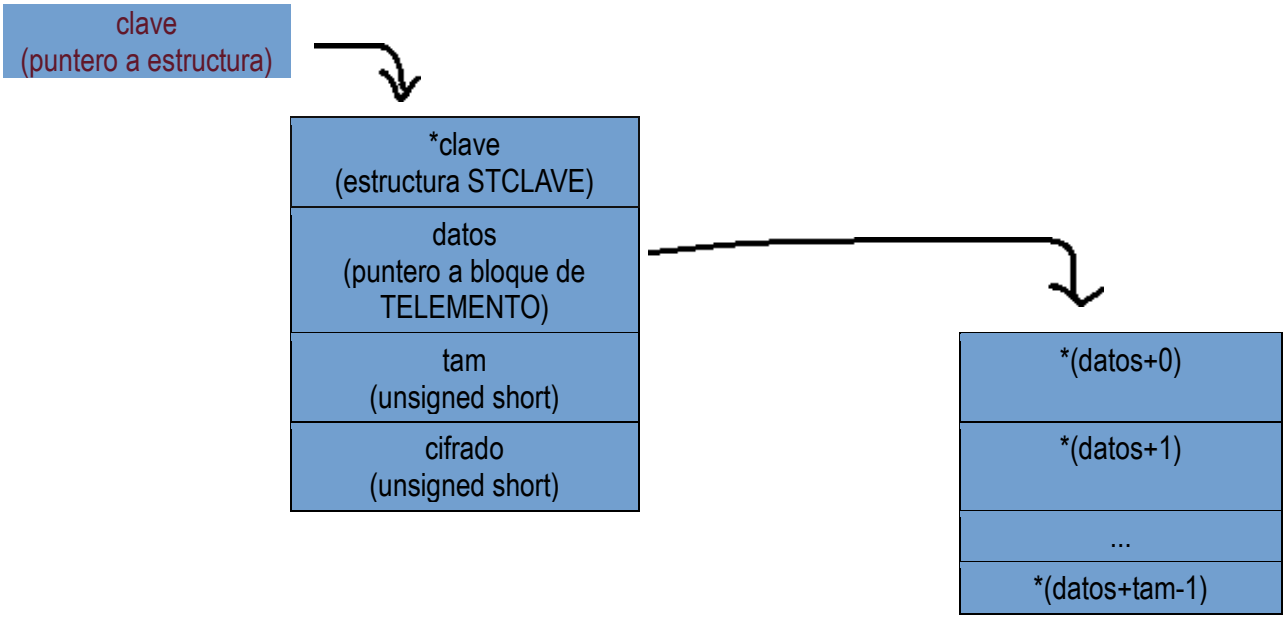
Abordaremos esta tarea directamente a través del ejemplo que consideramos en esta práctica, y que va a ser una "clave" que en principio almacenará datos de tipo carácter (**char**).

Podemos pensar inicialmente en dos formas distintas para definir dicho TAD, al que por simplicidad vamos a denominar `clave`:

a) Como un puntero a un bloque de memoria de N componentes, donde N es el número de caracteres de la contraseña. En este caso, como el tamaño de la contraseña se conocerá en tiempo de ejecución (porque el usuario la escribirá por teclado), todos los procedimientos de manipulación del TAD deberían recibir N como parámetro. **Esto no es una buena práctica**, pues nada evita que desde `main()` se llame a cualquiera de estos procedimientos con un tamaño de N erróneo.

a) Por lo dicho en el apartado anterior, el tamaño N de la clave debe estar ENCAPSULADO con el TAD `clave` pues, una vez creada una clave, esta no cambia durante todo su uso. No se debe permitir por tanto que el usuario realice operaciones erróneas utilizando los procedimientos de manipulación de la clave.

Por tanto, utilizaremos la opción b), que define el TAD `clave` como un puntero a una estructura con tres campos: el puntero al bloque de memoria de caracteres (`char`), el tamaño de la clave (número de caracteres, unsigned short), y el número que se ha utilizado para cifrar la clave (unsigned short).

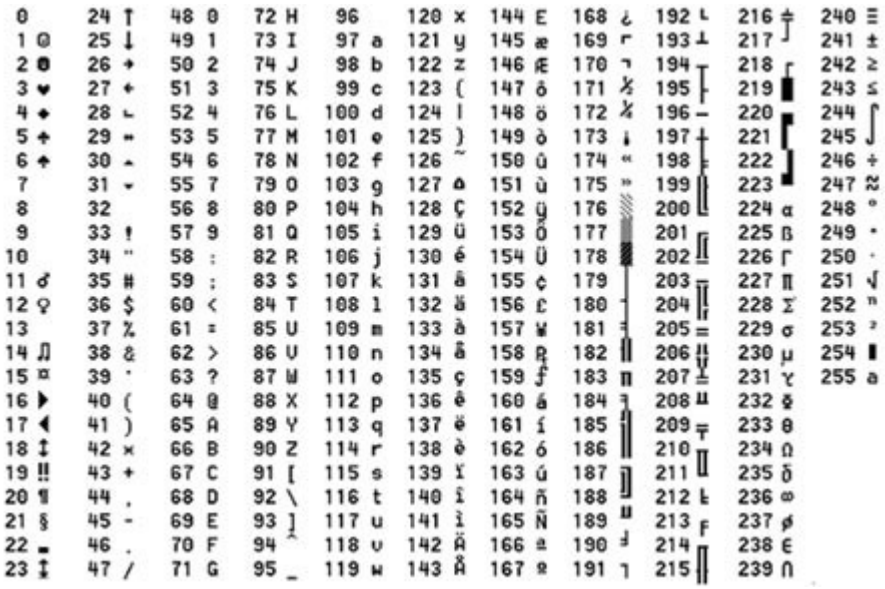


# Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

Por seguridad, no guardaremos los caracteres de la contraseña tal como nos los proporciona el usuario sino que almacenaremos una versión cifrada de los mismos. Seguiremos una aproximación muy simple para hacer el cifrado: cuando el usuario proporcione una clave (p.e. "peca") se le pedirá a continuación un número entero positivo (que deberá memorizar) para cifrar su clave (p.e. 5) y el programa automáticamente guardará en el bloque de caracteres la siguiente transformación de los caracteres (cada carácter se representa por el carácter ASCII original sumándole la clave multiplicada por la posición que ocupa el carácter original en la cadena de caracteres):

| | | | | |
|---|---|---|---|---|
| cadena original | 'p' | 'e' | 'c' | 'a' |
| cifrado | 5 | | | |
| offset a aplicar | 5*0 | 5*1 | 5*2 | 5*3 |
| cadena modificada | 'p' | 'j' | 'm' | 'p' |

Téngase en cuenta para esto la codificación ASCII:



Empezaremos definiendo los tipos de datos que requiere el TAD:

Vamos a asumir que la clave almacenará valores **char.** Dicho tipo de datos se representará definiendo un **nuevo tipo de datos** (utilizando `typedef`) que deberá incluirse tanto en clave`.h` como en clave`.c`:

| clave.h | clave.c |
|---|---|

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

| | |
|---|---|
| `typedef char TELEMENTO;` | `typedef char TELEMENTO;` |

Esto hace que el TAD sea lo más general posible y que se vea claramente el tipo de datos que CONTIENE. Cambiar este tipo de datos es simplemente cambiar esta línea de código. Esto permitiría, por ejemplo, usar el TAD para almacenar PINs (simplemente cambiando el TELEMENTO a un unsigned short).

Todos los TAD, al igual que las bibliotecas, tienen un fichero de interfaz (`.h`) y un fichero de implementación (`.c` o `.a`). El primero de ellos contiene las definiciones de tipos de datos y los prototipos de las funciones del TAD. El segundo la implementación (código fuente o compilado) de las mismas.

Todas las operaciones que manipulen o traten las claves deben estar incluidas en un módulo de biblioteca independiente donde el tipo `clave` debe estar definido como **opaco** (deben ocultarse los detalles de implementación a los usuarios de la biblioteca). Esto quiere decir que:

a) debe ser declarado como *tipo de datos puntero en el módulo de implementación* (`clave.c`),

b) quedando como *puntero indefinido* (`void *`) en el *módulo de definición* (`clave.h`) en el que únicamente se establece el *nombre* del tipo opaco

clave.h

```
typedef char TELEMENTO;
typedef void * clave;
```

clave.c

```
typedef char TELEMENTO;
typedef struct {
 TELEMENTO *datos;    /*caracteres de la contraseña*/
 unsigned short tam;           /*tamaño de la clave*/
 unsigned short cifrado;           /*numero para el cifrado*/
}STCLAVE;         /*definición del tipo de datos estructura*/

typedef STCLAVE *clave;  /*puntero a estructura*/
```

**Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD**

## Práctica 1.0: implementación del TAD (versión 0):

Crea un nuevo proyecto al que le pondrás como nombre **P1_v0.** Bajo la instalación de Windows Subsystem Linux (WSL) explicada en la práctica anterior, abriremos el Visual Studio Code (VSC), conectado con el WSL:Ubuntu y crearemos una nueva carpeta vacía para este proyecto dentro del sistema de archivos de WSL, y abrimos el proyecto vacío desde VSC con el botón Open Folder del Explorer. Pedirá confirmación a lo que debemos seleccionar, *Yes, I trust the authors*.

Este proyecto tendrá un programa principal `main.c` y los archivos del TAD. A continuación, se detalla la implementación de los ficheros de interfaz e implementación del TAD `clave`:

Interfaz de usuario: `clave.h`

En la interfaz de usuario se indican los tipos de datos que es posible utilizar y los prototipos de las funciones que los manejan.

Para crear un fichero de interfaz en un proyecto VSC debemos acudir al explorador de VSC y bajo la carpeta del nuevo proyecto seleccionar la opción "New File.." y a continuación, indicamos el nombre "**`clave.h`**". El contenido del fichero de interfaz **`clave.h`** será el siguiente:

```
typedef char TELEMENTO;
/* tipo de datos de los elementos de la contraseña */

typedef void *clave;
/* tipo clave opaco, el usuario del TAD sólo ve el .h, no tiene que
saber como está implementada la clave por dentro */

void crear(clave *c,  unsigned short longitud, unsigned short cifrado);
/* para crear una clave se proporciona un puntero a la dirección de
memoria donde el TAD almacenará la clave, la función crear del TAD
asignará memoria, y guardará encapsuladamente la longitud y número
de cifrado de la clave */

void asignar(clave *c, unsigned short posicion, TELEMENTO valor);
/* para introducir individualmente cada posición de la clave */
```

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

Módulo de implementación: `clave.c`

Los detalles de la implementación o construcción del tipo de datos `clave` estarán en un fichero denominado `clave.c`. Estos detalles de implementación siempre se ocultarán al usuario del TAD, que únicamente necesita la interfaz `clave.h` para poder escribir sus programas.

Para crear un fichero de implementación en un proyecto VSC debemos acudir al explorador de VSC y bajo la carpeta del nuevo proyecto seleccionar la opción "New File.." y a continuación, indicamos el nombre "**`clave.c`**". El contenido del fichero **`clave.c`** será el siguiente:

```
typedef char TELEMENTO;
/* tipo de datos de los elementos de la contraseña */

/* implementación, ahora ya no es opaca pues estamos en el .c de
la librería, del tipo clave */

typedef struct {
 TELEMENTO *datos;    /*caracteres de la contraseña*/
 unsigned short tam;           /*tamaño de la clave*/
 unsigned short cifrado;           /*numero para el cifrado*/
}STCLAVE;         /*definición del tipo de datos estructura*/
typedef STCLAVE *clave;  /*puntero a estructura*/


/* para crear una clave se proporciona un puntero a la dirección de
memoria donde el TAD almacenará la clave, la función crear del TAD
asignará memoria, y guardará encapsuladamente la longitud y número
de cifrado de la clave */

void crear(clave *c,  unsigned short longitud, unsigned short cifrado)
{
unsigned short i;

/* crea espacio para el struct */
*c = (clave) malloc(sizeof(STCLAVE));

/* dentro del struct creamos el bloque dinámico para almacenar los
caracteres de la clave */
(*c)→datos = (TELEMENTO *) malloc(longitud*sizeof(TELEMENTO));
```

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

```
/* guardamos el tamaño y número de cifrado como datos encapsulados en el
struct del TAD */
(*c)→tam = longitud;
(*c)→cifrado = cifrado;

/* inicializamos la contraseña a letras a */

for (i=0; i<longitud; i++)
      *((*c)→datos + i) = 'a';


}

/* para introducir individualmente cada posición de la clave */
void asignar(clave *c, unsigned short posicion, TELEMENTO valor)
{

/* en lugar de asignar el valor que nos dan representamos el carácter
ASCII correspondiente a ese carácter ASCII desplazado el offset que
corresponda según su posición en la clave y el número de cifrado*/

      *((*c)→datos + posicion) = valor + posicion*(*c)→cifrado;
}
```

Con la construcción de los ficheros de interfaz (`clave.h`) e implementación (`clave.c`) queda concluido el diseño e implementación del TAD.

### Programa principal: `main.c`

En primer lugar, recuerda que para utilizar la biblioteca que acabas de escribir, debes incluirla en main como:
`#include "clave.h"`

A continuación, declara las variables que vas a utilizar y usa los procedimientos que manipulan el vector especificados en `clave.h`, teniendo en cuenta los tipos de los argumentos. Lo importante es pensar que el tipo de datos que estás usando (clave) es como cualquier otro de los tipos de datos estándar (int, float, char, etc.) y que por tanto lo usarás de la misma forma. Cada tipo de datos tiene definidas las operaciones que lo manipulan y, en el caso de las claves, por ahora sólo tiene definidas las operaciones `crear()` y `asignar()`. A medida que vayas avanzando irás añadiéndole más funcionalidades.

ES MUY IMPORTANTE DARSE CUENTA DE QUE, DESDE EL PROGRAMA PRINCIPAL, COMO NO SABEMOS CÓMO ESTÁ CONSTRUIDO EL TIPO DE DATOS `clave`, NO PODEMOS ACCEDER A SUS COMPONENTES (por ejemplo, tam o cifrado). ES DECIR, LA MANIPULACIÓN Y EL ACCESO AL TAD O A

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

SUS ELEMENTOS SE REALIZA **ÚNICA Y EXCLUSIVAMENTE** A TRAVÉS DE LOS PROCEDIMIENTOS DEFINIDOS EN EL TAD.

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

int main() {

clave clave1;
unsigned short longitud, cifrado, posicion;
TELEMENTO valor;
char opcion;

do{
      printf("\n--------------------\n");
      printf("\na) Crea clave");
      printf("\ns) Salir");
      printf("\n--------------------\n");

      printf("\nOpcion: ");
      scanf(" %c",&opcion);

      switch(opcion){
            case 'a':

                  printf("Introduce el tamanho de la clave: ");
                  scanf("%hu",&longitud);

                  printf("Introduce el número para cifrar la clave: ");
                  scanf("%hu",&cifrado);

                  crear(&clave1,longitud,cifrado);

                  /*Asigno valores a la clave a partir de lo que teclea el
usuario*/
                  for (posicion=0;posicion<longitud;posicion++)
                  {
                      printf("Elemento %d de la clave: ",posicion);
                      scanf(" %c",&valor);
                      asignar(&clave1,posicion,valor);
                  }
```

```
                break;

            case 's':
                printf("Salimos del programa\n");

                break;
            default:
                printf("Opcion incorrecta\n");
        }
}while (opcion!='s');


return (EXIT_SUCCESS);
}
```

A continuación, desde VSC, compila todo, depura y ejecuta el programa, comprobando que funcione correctamente.

# Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

## Práctica 1.1: utilización del TAD a través de una librería compilada

Ahora vamos a trabajar en la consola de WSL y vamos a ver cómo se crearía una librería compilada a partir de nuestro archivo de implementación (`clave.c`) para ocultar al usuario los detalles de implementación.

Para ello, en nuestra carpeta de trabajo, crearemos la carpeta **`P1_v1`** y copiaremos en ella los archivos del proyecto **`P1_v0`** con los que acabamos de trabajar: `main.c, clave.h y clave.c`. Nos cambiamos de carpeta de trabajo, situándonos en la carpeta `P1_v1`.

### Creación de una librería compilada para ocultar los detalles de implementación

En primer lugar, vamos a ver cómo se crea una librería que se puede proporcionar al usuario para ocultar todos los detalles de implementación, es decir, para ocultar totalmente el archivo `clave.c`.

- Como el funcionamiento de la librería compilada depende del sistema operativo y su versión, la vamos a crear a partir del fichero `clave.c` que habéis creado. Para crear la librería estática necesitáis crear el fichero .o mediante el `comando:`

  ```
  gcc –static –c –o clave.o clave.c
  ```

  A continuación, con el comando `ar` creamos la librería (con extensión .a):

  ```
  ar –rcs –o libclave.a clave.o
  ```

  Podéis consultar el contenido de una librería compilada del siguiente modo:

  ```
  ar –t libclave.a
  ```

La práctica consistirá en la compilación y ejecución del programa que has creado, pero SIN UTILIZAR clave.c en la compilación (de hecho, una vez compilado en la librería, puedes borrar de esta carpeta clave.c y clave.o), sino utilizando únicamente la librería compilada `libclave.a`.

**Descarga el `makefile` que tienes junto con este guión**, necesario para compilar este proyecto a partir de una librería propia. Para ello, fíjate que se han incluido en la fase de enlazado las opciones `-L` (para indicar la carpeta donde está la librería) y `-l` para indicar el nombre de la librería modificando el contenido del makefile base como se indica a continuación:

```
 …
#carpeta de las librerías estáticas propias (si están en la actual, ponemos .)
LIB_FILES_DIR = .
#opciones de compilación, indica dónde están nuestra librería estática (si es una
#librería estándar, no es necesario)
LIBRARIES= -L $(LIB_FILES_DIR)
#si incluye una librería, en este caso la de la clave
LIBS=-lclave
…
#REGLA 1: genera el ejecutable, dependencia de los .o
$(OUTPUT): $(OBJS)
	$(CC) -o $(OUTPUT) $(OBJS) $(LIBRARIES) $(LIBS)
…
```

## Práctica 1.2: utilización de los argumentos de main.

Ahora vamos a modificar la función main.c para poder pasar argumentos por la línea de comandos. **Cread una carpeta llamada P1_v2 y copiad los archivos main.c**, **clave.c** y **clave.h de la carpeta P1_v0 y el fichero Makefile de la carpeta P1_v1**. En este makefile, debéis eliminar todas las referencias a la librería compilada (lo que está en rojo en la página anterior) y añadir el fichero clave.c a los fuentes: `SRCS = main.c clave.c`
Los archivos los podéis editar en VSC o simplemente arrastrándolos encima de un editor, si os es más cómodo para trabajar. Pero la compilación debéis hacerla desde el terminal de Ubuntu de WSL.

La función main() tendrá ahora el siguiente formato:

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

donde `argc` indica el **número de argumentos** que recibe el programa al ejecutarse y `argv` es un **vector de cadenas de texto** que contiene los argumentos que ha incorporado el usuario al lanzar el programa.

Vamos a configurar un programa que permita al usuario ejecutarlo incorporando ya los caracteres a almacenar en la clave (separados por espacio) y, por último, el número de cifrado que quiere aplicar. El uso desde consola de este programa (que llamaremos ejecutable) sería:

```
./ejecutable p e d r o 9
```

Los argumentos de `main` tomarán los siguientes valores:

- `argc`, que es el número de cadenas que se escriben en la línea de comandos, tomará el valor de 7, y esas cadenas se almacenan en el vector `argv[]` de tamaño 7:

| | | |
|---|---|---|
| `argv[0]` | `"./ejecutable"` | Nombre del programa |
| `argv[1]` | `"p"` | Componentes de la clave (5 componentes) |
| `argv[2]` | `"e"` | |
| `argv[3]` | `"d"` | |
| `argv[4]` | `"r"` | |
| `argv[5]` | `"o"` | |
| `argv[6]` | `"9"` | Número para el cifrado |

El sistema operativo siempre pasa al programa los argumentos de entrada como cadenas de caracteres. Luego internamente el programa las transforma al tipo de datos interno que necesite para trabajar con esos datos. En nuestro caso, para las letras de la contraseña, tendremos que extraer el primer carácter de cada una de esas cadenas (que tienen en todo caso un sólo carácter cada una) para obtener el TELEMENTO que necesitamos para incorporar a la clave. Por otra parte, el último dato hay que transformarlo a dato numérico mediante la función strtoul.

Recuerda que es necesario validar el número de argumentos. El número de argumentos que se pasen al ejecutable debe ser mayor o igual que 3 (que corresponden al nombre del ejecutable, y al menos a una componente de la clave, junto con el número para el cifrado).

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

Como todavía no tenemos la función imprimir, para comprobar que estás rellenando la clave correctamente, puedes ir imprimiendo por pantalla los valores a medida que los vas asignando a cada componente.

## Práctica 1.3: ampliación de la funcionalidad del TAD:

Crea una nueva carpeta para un nuevo proyecto, **P1_v3,** y copia en ella los archivos .c y .h de **P1_v2**. En esta parte de la práctica os facilitaremos las especificaciones informales para **ampliar la funcionalidad** del TAD, de modo que se incremente el repertorio de procedimientos que lo manipulen y realicen tareas algo más complejas que las incluidas hasta el momento.

De nuevo, puedes usar VSC para editar los ficheros, pero debes utilizar el makefile anterior para compilar tu proyecto desde el terminal.

### Función `liberar()`

En este momento podemos plantear una nueva pregunta. Imaginemos que ejecutamos nuestro programa introduciendo una clave `c` con 10 caracteres y que a continuación introducimos una nueva clave con 8 caracteres.

¿Qué sucede con la memoria que habíamos reservado para la primera clave? No la hemos liberado y, al mismo tiempo, hemos perdido la referencia a dicha memoria, pues con la segunda creación de un bloque dinámico apuntamos a **otra** zona de memoria con 8 posiciones consecutivas.

La solución a este problema es liberar la memoria ocupada por la clave previa **ANTES** de realizar una nueva asignación de memoria.

Además, hemos de tener en cuenta que esta liberación sólo se puede realizar si antes se ha hecho la creación de la clave (esto es, es incorrecto liberar una clave si antes no la habíamos creado). Por tanto, desde el punto de vista del uso del TAD, el programa debe ser cauto en sólo liberar si antes había sido creada la clave (de otro modo, tendríamos un error de ejecución pues el programa intentaría liberar unas posiciones de memoria no reservadas). Una posibilidad para controlar si una clave ha sido creada o no es introducir en el programa principal una variable bandera que se ponga a 0 o 1 en función de si el usuario ha introducido ya una clave (y se vuelva a poner a 0 cuando desde el menú se libera la clave anterior).

Para escribir la función liberar puedes inspirarte en la función `destruirMatriz.c` que has corregido para la práctica P0_E5, pero ten en cuenta que en esa práctica no se trabajaba con TADs opacos y por tanto puede cambiar la forma de hacer referencia al TAD (en ese caso la variable era una estructura y ahora es un puntero a estructura).

Añade la llamada a esta función como una opción de tu menú: **b) Liberar clave.**

### Función `recuperar()`

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

La función `recuperar()` recibirá como argumentos una determinada `clave` y la posición y DEVOLVERÁ al programa principal el carácter (`TELEMENTO`) que se encuentra en dicha posición. Será el programa principal el que se encargue de imprimirlo en pantalla. Puedes tomar como ejemplo la función `obtenerElemento.c` corregida en la práctica P0_E5. **Pero observa que para recuperar el elemento de la posición correspondiente no basta con tomarlo del vector de caracteres sino que hay que "descifrarlo" a su carácter original según el proceso descrito anteriormente.**

Observa que todas estas funciones tienen algunos argumentos en los cuales deben volcar resultados y otros argumentos que actúan únicamente como valor de entrada (sólo lectura). Esto afecta a cómo debes pasar los argumentos (con puntero al correspondiente tipo de datos o sin puntero).

### Función `longitud()`

Es necesario introducir una función en el TAD que nos permitan obtener el tamaño actual de la clave, y que así pueda ser utilizado por procedimientos auxiliares que operen con las claves, como por ejemplo la función `imprimir()`.

### Función `imprimir()`

Es necesario introducir una función en el TAD de `imprimir()`, que imprime la clave. Para acceder a las componentes de la clave usaremos la función del TAD `recuperar()` y para acceder a su tamaño usaremos la función `longitud()` escrita anteriormente.

La función de impresión tendrá tres posibles modos de operación en función de un parámetro de entrada **modo**. Cuando modo es 0 el programa escribe los caracteres normalmente, cuando el modo es 1 el programa escribe la clave pero sustituyendo cada carácter original por un asterisco (con tantos asteriscos como caracteres tiene la clave), en el modo 2 el programa ocultará sólo parte de la clave (aleatoriamente escogiendo las posiciones que se ocultan – "pe***" , "*e**o", etc. ). Para ello, consultad el uso de las funciones srand y rand para generación de números aleatorios.

### Función `cadena2clave()`

Dado que la entrada de datos para un usuario es engorrosa en la opción de crear clave (porque se le pide iterativamente cada carácter), crearemos aquí otra opción en la que el usuario podrá introducir de una sola vez todo el texto de la clave y el programa usará esa cadena para crear internamente una clave. El programa principal leerá de teclado una cadena de caracteres e invocará a una nueva función del TAD llamada **cadena2clave** que se ocupará de generar la clave a partir de la cadena de caracteres. El programa también deberá pedir independientemente el valor de cifrado para pasárselo a la función cadena2clave (lo necesita para generar internamente la clave).

## Práctica 1: creación y manipulación de la Estructura de Datos "clave" usando TAD

**Función** `compruebaclave()`

Esta función del TAD recibirá la clave y un **modo** de operación. Si el modo de operación es 0 entonces la función le pedirá al usuario todos los caracteres de la clave (leyéndolos en una única operación de lectura) y comprobará que la clave proporcionada por el usuario es correcta (si lo es devolverá un 1, en caso contrario un 0). Si el modo de operación es 1, la función mostrará primero la clave ocultando algunos caracteres (poniéndolos como asteriscos, por ejemplo, p*d**) y le pedirá al usuario los caracteres que faltan (leyéndolos en una única operación de lectura). Nuevamente, la función devolverá un 1 si el usuario ha acertado los caracteres que faltan y un 0 en caso contrario.

## Entregables

Deberás entregar por el Campus Virtual el ejercicio correspondiente a la práctica 1, versión 3 (P1_v3). Las fechas de entrega se especifican en el Campus Virtual, y las instrucciones para generar el fichero son las siguientes:

- Deberás subir un único fichero comprimido con el nombre **Apellido1Apellido2_1.zip**.
- Incluye ÚNICAMENTE los ficheros fuente (.c) y de cabecera (.h), así como su correspondiente makefile.

Para ser válida, la práctica debe compilar directamente con el makefile (sin opciones) en Linux/WSL, en otro caso será evaluada con la calificación de 0. Este criterio se mantendrá en el resto de las prácticas de la asignatura.

## Ejemplo de menú en el programa principal:

```
a) Crea clave
b) Liberar clave
c) Recuperar posición de una clave
d) Longitud de la clave
e) Imprime clave sin ocultar caracteres
f) Imprime clave ocultando todos los caracteres
g) Imprime clave ocultando algunos caracteres
h) Crea clave a partir de cadena de texto
i) Comprueba contraseña (completa)
j) Comprueba contraseña (partes)
s) Salir
```

El programa en todo momento llevará control de si hay o no una clave creada (por ejemplo con una variable bandera) y además de la posibilidad de crear claves con las opciones de menú a) o h) podrá recibir la clave como entrada en la línea de comandos, tal y como se explicó anteriormente.

## Anotaciones sobre reserva dinámica de memoria en C

### Conceptos previos:

- Función **`sizeof`**: calcula en tiempo de ejecución el tamaño en bytes de cualquier tipo de dato. **`sizeof(tipo_de_dato)`**
- Puntero Nulo: la constante **`NULL`** contiene por definición el valor 0, que corresponde a una posición de memoria donde no puede existir ningún dato válido. Suele utilizarse para indicar que un puntero no está inicializado, o que no dispone de ningún dato al que hacer referencia. Es una buena técnica de programación inicializar todos los punteros a **`NULL`**.

### Algunas funciones para la gestión dinámica de memoria:

- **`malloc(int tamanho)`**: devuelve un puntero que contiene la dirección de memoria a partir de la cual se ha reservado un bloque de memoria del tamaño requerido (**`tamanho`** se expresa en bytes). Este puntero a **`void`** hay que moldearlo para que apunte al tipo de dato para el que se hace la reserva, para asegurarnos así que funcionará correctamente la aritmética de punteros.
- Si no es posible realizar la reserva, **`malloc()`** devuelve **`NULL`**.
- Ejemplo: Reserva dinámica de una matriz de n componentes enteras, mostrando el puntero void moldeado a int *: **`p=(int *) malloc(n*sizeof(int));`**
- **`free(void *pMemoria)`**: pasándole el puntero devuelto por **`malloc()`**, libera la memoria cuando ya no la necesitemos, para su uso posterior en el programa.

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

1. SIEMPRE inicializa los punteros a **`NULL`**. De este modo te aseguras de que, si se utilizan posteriormente sin darles otro valor, se generará un error de ejecución, que es más fácil de detectar que el error lógico que tendríamos en caso de no inicializarlos.

2. SIEMPRE comprueba si la reserva dinámica tiene éxito. Una comprobación un tanto drástica, pero muy efectiva, es:

```
m=…malloc(…); /*Intento de reserva dinámica*/
if (!m)       /*Si no se ha podido reservar, m vale NULL*/
   exit(EXIT_FAILURE);   /*Salir del programa*/
```

3. Cuando vayas a recorrer una agrupación (array) con un puntero **`p`**, debes mantener otro puntero **`pInicio`** apuntando al inicio de dicha agrupación, para poder volver a reapuntar **`p`** en cualquier momento a un lugar conocido.