Programación II, 2015-2016

Escuela Politécnica Superior, UAM

Práctica 1: Estructuras de Datos y Tipos Abstractos de Datos

OBJETIVOS

- Profundizar en el concepto de **TAD (Tipo Abstracto de Dato).**
- Aprender a elegir la **estructura de datos** apropiada para implementar un TAD.
- Codificar sus **primitivas** y utilizarlo en un programa principal.

NORMAS

Los ***programas*** que se entreguen deben:

- Estar escritos en ***ANSI C,*** siguiendo las **normas de programación** establecidas.
- Compilar sin errores ni *warnings* incluyendo las banderas "-ansi" y "-pedantic" al compilar.
- **Ejecutarse sin problema en una consola de comandos**.
- Incorporar un adecuado **control de errores.** Es justificable que un programa no admita valores inadecuados, pero no que se comporte de forma anómala con dichos valores.

PLAN DE TRABAJO

Semana 1: P1_E1 completo y funciones más importantes de P1_E2.
Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.

Semana 2: todos los ejercicios.
La entrega final se realizará a través de Moodle, **siguiendo escrupulosamente las instrucciones** indicadas en el enunciado de la práctica 0 referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido que se debe entregar debe llamarse **Px_Prog2_Gy_Pz**, siendo '**x**' el número de la práctica, '**y**' el grupo de prácticas y '**z**' el número de pareja (ejemplo de entrega de la pareja 5 del grupo 2161: P1_Prog2_G2161_P05.zip).

Las **fechas de subida a Moodle del fichero** son las siguientes:

- Los alumnos de ***Evaluación Continua, la semana del 15 de febrero*** *(cada grupo puede realizar la entrega hasta las 23:30 h. de la noche anterior a su clase de prácticas).*
- Los alumnos de ***Evaluación Final,*** según lo especificado en la normativa.

Contenido del .zip para esta Práctica 1:

- Carpeta de la práctica, siguiendo al pie de la letra las instrucciones indicadas en la P0.
- Fichero donde se respondan a las cuestiones planteadas en la PARTE 2 de este documento, con nombre y apellidos de la pareja de prácticas. En esta primera práctica no será necesario elaborar ninguna memoria adicional.

PARTE 1: CREACIÓN DEL TAD PUNTO Y LABERINTO

En esta práctica se va a implementar un laberinto de puntos. Para ello, primero se comenzará definiendo el tipo PUNTO (*Point*), y a continuación se trabajará sobre el TAD LABERINTO (*Maze*).

Observe que el contenido de algunos de los ficheros necesarios para esta práctica (*types.h*) se proporcionan al final de este enunciado.

EJERCICIO 1 (P1_E1)

1. Definición del tipo de dato PUNTO (*Point*). Implementación: selección de estructura de datos e implementación de primitivas.

En esta práctica, un punto se representará mediante un ***símbolo*** (un carácter) y sus coordenadas **x** e **y**.

Es necesario tener en cuenta las siguientes observaciones:

- Existen varios tipos de símbolos: entrada, salida, barrera, espacio. Se podrán crear otros si es necesario.
- Las coordenadas no podrán ser negativas. Se asume que el (0, 0) es el vértice superior izquierdo (ver información más detallada en ejercicio P1_E2).

Los tipos de datos Status (empleado para el control de errores) y Bool (verdadero/falso) se encuentran definidos en *types.h* (ver apéndice 1).

- Para definir la estructura de datos necesaria para representar el TAD PUNTO conforme ha sido descrita anteriormente, hay que escribir la siguiente declaración en *point.h*:

```
typedef struct _Point Point;
```

  Además, en point.c hay que incluir la implementación del tipo abstracto de datos previamente declarado:

```
struct _Point {
   char symbol;
   int x;
   int y;
};
```

- Para poder trabajar con datos de tipo Point serán necesarias, al menos, las funciones básicas o primitivas cuyos prototipos se encuentren declarados en el fichero *point.h* (ver apéndice 2). Escribid el código asociado a su definición en el fichero *point.c*.

2. Comprobación de la corrección de la definición del tipo Point y sus primitivas.

Se deberá crear un fichero **p1_e1.c** que defina un programa (de nombre **p1_e1**) con las siguientes operaciones:

- Declarar dos puntos.
- Inicializarlos de modo que el primero sea un punto con símbolo '+' y coordenadas (x,y) iguales a (1,2) y el segundo uno con símbolo 'o' y coordenadas (3,4).
- Imprimir ambos puntos e imprimir después un salto de línea.
- Comprobar si los dos puntos son iguales
- Imprimir el símbolo del primer punto junto con una frase explicativa (ver ejemplo más abajo).
- Imprimir la coordenada X del segundo punto (ver ejemplo más abajo)
- Copiar el primer punto en el segundo.
- Imprimir ambos puntos.
- Comprobar si los dos puntos son iguales
- Liberar ambos puntos.

Salida esperada:

```
[(1,2): +][(3,4): o]
Son iguales? No
El símbolo del primer punto es: +
La coordenada X del segundo punto es: 3
[(1,2): +][(1,2): +]
Son iguales? Sí
```

EJERCICIO 2 (P1_E2)

Definición del tipo abstracto de dato LABERINTO.

1. Implementación: selección de estructura de datos e implementación de primitivas.

En esta parte de la práctica se definirá el Tipo Abstracto de Datos (TAD) LABERINTO como un conjunto de elementos homogéneos (del mismo tipo) y otros atributos que nos permitirán movernos por el laberinto y saber dónde empieza y termina.

Se tiene que **definir una estructura de datos para representar el TAD LABERINTO** (en los ficheros *maze.c* y *maze.h*), suponiendo que los datos que hay que almacenar en él son de tipo **Point** y que su capacidad máxima son 4096 elementos.

Para implementar los TADs relacionados con el laberinto:

- Definid en el fichero *maze.h* el nuevo tipo de dato **Maze**,
- Definid en maze.c la estructura de datos **_Maze** que contendrá los puntos del laberinto.
- Para poder trabajar con datos de tipo **Maze** serán necesarias, al menos, las funciones básicas o primitivas indicadas en el apéndice 3. Las funciones se declaran, mediante sus prototipos, en *maze.h*. Escribid su código en *maze.c*.

2. Comprobación del funcionamiento de un laberinto a través de ficheros.

Definid un programa en un fichero de nombre **p1_e2.c** cuyo ejecutable se llame **p1_e2**, y que utilice la función que aparece a continuación, de manera que se cargue un laberinto en memoria de manera correcta. Una vez hecho eso, se deberá imprimir el número de filas y columnas del mismo, así como llamar a la función para imprimir el laberinto, de forma que la salida del programa debería ser igual al fichero de entrada.

Un ejemplo de fichero de datos de entrada es el siguiente, donde la primera línea indica el número de filas y columnas del laberinto y en las siguientes el contenido del mismo, comenzando (coordenada (0, 0)) con el símbolo que se encuentra más a la izquierda y arriba, y teniendo en cuenta que las coordenadas X crecen a la derecha y las Y hacia abajo:

```
7 5
+++++
+i  +
+++ +
+   +
+ + +
+ +o+
+++++
```

A continuación se proporciona el código de una función, en la que no se ha completado la gestión de los posibles errores, pero que, al margen de ello, tiene la funcionalidad que necesita para leer un laberinto:

```
Maze * maze_read (FILE *pf, Maze *pl) {
   char buff[MAX];
   int i, j, nrows, ncols, indice;
   Point *temp;

   if (pl==NULL || pf==NULL) return NULL;

   /*creamos punto que se utiliza como buffer*/
   temp = point_ini();
   if (temp==NULL) return NULL;

   /* asignamos dimensión al laberinto */
   fgets(buff, MAX, pf);
   sscanf(buff, "%d %d", &nrows, &ncols);
   pl = maze_setSize (pl, nrows, ncols);
   if (pl==NULL) return NULL;

   /* leemos el fichero linea a linea */
   for(i=0; i < nrows;i++) {
      fgets(buff, MAX, pf);
      for (j=0; j < ncols; j++) {
         /* ajustamos los atributos del punto leído (falta añadir control de errores) */
         point_setCoordinateX(temp, j);
         point_setCoordinateY(temp, i);
         point_setSymbol(temp, buff[j]);
         /* insertamos el punto en el laberinto (falta añadir control de errores) */
         maze_addPoint(pl, temp);
      }
   }
   /* libera recursos */
   point_free(temp);
   /* no cerramos el fichero ya que lo han abierto fuera */

   return pl;
}
```

PARTE 2: PREGUNTAS SOBRE LA PRÁCTICA

Responded a las siguientes preguntas **completando el fichero disponible en el .zip**. Renombrad el fichero para que se corresponda con su nº de grupo y pareja de prácticas y adjuntadlo al fichero .zip que entreguéis.

1. ¿Sería posible implementar la función de copia de puntos empleando el siguiente prototipo **STATUS point_copy(Point pDest, const Point pOrigin);** ? ¿Por qué?

2. ¿Es imprescindible el puntero Point* en **int point_print(FILE * pf, const Point* p);** o podría ser **int point_print(FILE * pf, const Point p); ?**
   Si la respuesta es sí: ¿Por qué?
   Si la respuesta es no: ¿Por qué se utiliza, entonces?

3. ¿Qué cambios habría que hacer en la función de copiar puntos si quisiéramos que recibiera un punto como argumento donde hubiera que copiar la información? Es decir, ¿cómo se tendría que implementar si en lugar de **Point* point_copy(const Point* pOrigin)**, se hubiera definido como **STATUS point_copy(const Point * pSource, Point * pDest)**? ¿Lo siguiente sería válido: **STATUS point_copy(const Point * pSource, Point ** pDest)**? Discute las diferencias.

4. Indica qué se tendría que cambiar en **maze.c/h** para tener laberintos que pudieran almacenar cualquier estructura de datos, es decir, que no estuviera limitado a almacenar puntos.

Apéndice 1: types.h

```
/*
 * File:   types.h
 * Author: Profesores de PROG2
 */

#ifndef TYPES_H
#define  TYPES_H

typedef enum {
   ERROR = 0, OK = 1
} Status;

typedef enum {
   FALSE = 0, TRUE = 1
} Bool;

#endif    /* TYPES_H */
```

Apéndice 2: point.h

```
#define ERRORCHAR 'E'

#define INPUT 'i'
#define OUTPUT 'o'
#define BARRIER '+'
#define SPACE ' '

/* Inicializa un punto, reservando memoria y devolviendo el punto inicializado si lo ha hecho correctamente o NULL si no */
Point * point_ini();
/* Libera la memoria dinámica reservada para un punto*/
void point_free(Point * );

/* Devuelve la coordenada X de un punto dado, o -1 si se produce algún error */
int point_getCoordinateX(const Point * );
/* Devuelve la coordenada Y de un punto dado, o -1 si se produce algún error */
int point_getCoordinateY(const Point * );
/* Devuelve el símbolo de un punto dado, o ERRORCHAR si se produce algún error */
char point_getSymbol(const Point * );

/* Modifica la coordenda X de un punto dado, devuelve NULL si se produce algún error */
Point * point_setCoordinateX(Point *, const int );
/* Modifica la coordenda Y de un punto dado, devuelve NULL si se produce algún error */
Point * point_setCoordinateY(Point *, const int );
/* Modifica el símbolo de un punto dado, devuelve NULL si se produce algún error */
Point * point_setSymbol(Point *, const char );

/* Devuelve TRUE si los dos puntos pasados como argumentos son iguales (revisando todos sus campos).
   Devuelve FALSE en otro caso. */
Bool point_equals(const Point * , const Point * );
/* Copia los datos de un punto a otro devolviendo el punto copiado (incluyendo la reserva de la memoria necesaria) si todo ha ido bien, o NULL en otro caso */
Point * point_copy(const Point * );

/* Imprime en un fichero dado los datos de un punto con el siguiente formato:  [(x,y): symbol]. Por ejemplo, un punto con símbolo "*", con coordenada X 3 e Y 7 se representará como [(3, 7): *]. Además devolverá el número de caracteres que se han escrito con éxito (mirar documentación de fprintf)  */
int point_print(FILE *, const Point *);
```

Apéndice 3: maze.h

```
/* Movimientos posibles en un laberinto */
typedef enum {
        RIGHT=0,
        UP=1,
        LEFT=2,
        DOWN=3,
        STAY=4
} Move;

/* Inicializa un laberinto, reservando memoria y devolviendo el laberinto inicializado si lo ha hecho correctamente o NULL si no */
Maze * maze_ini();
/* Libera la memoria dinámica reservada para un laberinto*/
void maze_free(Maze * );

/* Devuelve el número de filas de un laberinto dado, o -1 si se produce algún error */
int maze_getNrows(const Maze * );
/* Devuelve el número de columnas de un laberinto dado, o -1 si se produce algún error */
int maze_getNcols(const Maze * );
/* Devuelve el punto de entrada en un laberinto dado, o NULL si se produce algún error */
Point * maze_getInput(const Maze * );
/* Devuelve el punto de salida en un laberinto dado, o NULL si se produce algún error */
Point * maze_getOutput(const Maze * );
/* Devuelve el punto de un laberinto situado en unas coordenadas dadas, o NULL si se produce algún error */
Point * maze_getPoint(const Maze * , const int x, const int y);

/* Devuelve el punto resultante al realizar un movimiento en un laberinto a partir de un punto inicial, o NULL si se produce algún error */
Point * maze_getNeighborPoint(const Maze * , const Point *, const Move mov);

/* Indica el tamaño de un laberinto, devuelve NULL si se produce algún error */
Maze* maze_setSize(Maze * , int nrow, int ncol);
/* Añade un punto a un laberinto dado */
Maze * maze_addPoint(Maze * , const Point* );

/* Imprime en un fichero dado los datos de un laberinto. Además devolverá el número de caracteres que se han escrito con éxito (mirar documentación de fprintf) */
int maze_print(FILE *, const Maze * );
```