Proyecto 4

Algoritmos y Estructuras de Datos I - Laboratorio
TADs, Arreglos y otros elementos del lenguaje C

2 de junio de 2010

El proyecto consta de una serie de ejercicios para programar algoritmos en el lenguaje C.

La implementación en lenguaje C debe hacerse traduciendo los programas escritos en el formalismo del teórico producto de su derivación, tal como se vio en el teórico del laboratorio, a menos que en el enunciado se diga lo contrario. Los programas deben tomar los datos de entrada del usuario y mostrar los resultados en pantalla.

Para hacer los programas tener en cuenta:

- Escribir los programas de manera legible, con la indentación (espaciado y alineación) correcta.
- En todos los programas utilizar la técnica de archivos separados según se indique.
- Recordar que la entrada debe chequear el cumplimiento de la precondición en la especificación del programa.
- Cuando se creen "headers" (archivos `.h`) hacer la cerradura `ifndef` como se explicó en el teórico.
- Compilar todos los archivos y linkear el programa con las opciones `-ansi`, `-pedantic` y `-Wall`. Ver que no haya mensajes de error o advertencias.
- No usar ninguna variable global.
- Hacer primero todos los ejercicios y después los puntos estrella.

1. Derivar y programar en C el programa que calcula la suma del segmento de suma máxima (ejemplo 19.9 del libro).
   - Escribir el algoritmo en una función

     ```
     int suma_maxima(int arr[], int n);
     ```

     donde `arr` es el arreglo y n es su longitud.
   - Escribir esta función en los archivos separados `operaciones.h` y `operaciones.c`. En el primero debe esta la signatura (o prototipo) de la función y en el segundo su implementación.
   - Agregar las funciones de entrada/salida a los archivos separados `in_out.h` e `in_out.c`. En el primero deben esta las signaturas (o prototipos) de la funciones y en el segundo sus implementaciones.
2. Derivar y programar en C el programa que dice si todos los elementos en un arreglo son positivos, y el programa que dice si algún elemento en un arreglo es positivo (ejercicio 19.3 del libro). Para ello:

- Escribir en un archivo separado `bool.h` el tipo `bool`.
- Escribir el primer algoritmo en una función

  ```
  bool todos_positivos(int arr[], int n);
  ```

  y el segundo en una función

  ```
  bool algun_positivo(int arr[], int n);
  ```

  a donde `arr` es el arreglo y n es su longitud.
- Agregar estas funciones a los archivos separados `operaciones.h` y `operaciones.c`. En el primero debe estar la signatura (o prototipo) de las funciones y en el segundo su implementación.
- Escribir las funciones de entrada/salida en los archivos separados `in_out.h` e `in_out.c`. En el primero deben estar las signaturas (o prototipos) de la funciones y en el segundo sus implementaciones.

Punto $\star$: Programar la función

```
int hay_negativo(int arr[], int n)
```

que devuelva la primera posición en el arreglo a donde hay un número negativo, o -1 si no hay ningún elemento negativo. No hace falta demostrar este agregado.

3. Derivar y programar en C el programa que dice si los paréntesis en un string están balanceados (ejemplo 20.3 del libro). Para ello:

   - Completar la derivación del ejemplo.
   - Escribir el algoritmo en una función

     ```
     bool parentesis_balanceados(char *str);
     ```

     donde `str` es la cadena de caracteres.
   - Agregar esta función a los archivos `operaciones.h` y `operaciones.c`. En el primero debe esta la signatura (o prototipo) de la función y en el segundo su implementación.
   - Agregar las funciones de entrada/salida a los archivos separados `in_out.h` e `in_out.c`. En el primero deben esta las signaturas (o prototipos) de la funciones y en el segundo sus implementaciones.

   Punto $\star$: Al momento de programar los algoritmos anteriores se puede definir el tamaño del arreglo como una constante `N` (definida mediante `#define`). ¿Se puede hacer que este tamaño sea decidido por el usuario? Lea la info page de libc donde se habla de arreglos, responda a la pregunta y lea si la posible solución tiene algún inconveniente. Programar todos los ejercicios anteriores según la respuesta.

4. Hacer dos implementaciones del TAD Número Complejo (no hace falta derivar los algoritmos):

   a) Escribir en archivos separados la definición del tipo complejo como un par parte real y parte imaginaria, su constructor y demás funciones (TAD) siguiendo el esquema de los archivos:

   - En el archivo complejo.h definir el tipo como una estructura y escribir los prototipos. El archivo debe quedar como el que figura en el apéndice A.

- En el archivo complejo.c, incluir el archivo complejo.h e implementar todas sus funciones.

b) Escribir en otro archivo el bloque *main* que implemente una interface con el usuario donde el mismo pueda cargar 2 números complejos `x` e `y` y muestre por pantalla el valor:

```
a * x + b * y + c
```

con

```
a = 2 + i * 3
b = 8 + i
c = 5 + i * 7
```

Este programa debe operar con los complejos únicamente a través de las funciones del TAD, y nunca accediendo a su estructura interna.

c) Después de terminar este programa, hacer otra implementación donde el número complejo se represente por su módulo y su ángulo. Para hacer esto, cambiar únicamente el archivo complejo.c y la definición de la estructura en el archivo complejo.h.

Apéndice A

```
#ifndef COMPLEJO_H
#define COMPLEJO_H

struct scompl {
  double rl, img;
};

typedef struct scompl compl;

compl  compl_create(double rl, double img);
/* DESC: Constructor del tipo */

real compl_preal(compl c);
/* DESC: Devuelve la parte real. */

real compl_pimag(compl c);
/* DESC: Devuelve la parte imaginaria. */

compl compl_suma(compl c1, compl c2);
/* DESC: Suma c1 y c2. */

compl compl_resta(compl c1, compl c2);
/* DESC: Resta c2 a c1. */

compl compl_prod(compl c1, compl c2);
/* DESC: Multiplica c1 y c2. */

char* compl_print(compl c);
/* DESC: Devuelve c en forma de string con formato a + i b. */

compl compl_leer(void);
/* DESC: Lee por teclado y construye. */

#endif /* COMPLEJO_H */
```