Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 4

Programación Imperativa en C

El objetivo de este proyecto es:

- Extender los conceptos relacionados con el desarrollo de programas en lenguaje C en base al formalismo visto en el teórico/práctico de la materia.
- Introducir el uso de la librería `assert.h` para garantizar el cumplimiento de estados.
- Familiarizarse con el uso de Arreglos en el lenguaje "C";
- Definir tipos abstractos nuevos básicos, utilizando el comando `struct`.

1. Lenguaje "C"

A lo largo de todo el proyecto, se utilizará el lenguaje C, y algunas herramientas como el `GDB: The GNU Project Debugger`, para ayudar a la comprensión del concepto de estado y del paradigma imperativo.

En el caso del lenguaje "C", para poder ejecutar un programa, lo vamos a tener que "compilar", y de esa manera generamos un archivo binario que podrá ser ejecutado en la computadora.

Cómo compilar en C:

Para compilar un archivo `.c` escribir en la terminal:

```
$> gcc -Wall -Wextra -std=c99 miarchivo.c -o miprograma
```

Para ejecutar escribir:

```
$> ./miprograma
```

Para compilar para gdb se debe agregar el flag -g al momento de compilar .c escribir en la terminal:

```
$> gcc -Wall -Wextra -std=c99 -g miarchivo.c -o miprograma
```

Ejercicios

1. (Funciones en C, Assert) Escribí los programas siguientes:

   a) `ejercicio01.c` que lee una variable `n` de tipo `int` e imprime por pantalla la palabra "hola" repetida `n` veces. En esta ocasión el programa debe utilizar dos funciones a definir (además de la función `main`). Programá en un archivo `ejercicio01.c` la función de prototipo (también conocido como signatura o firma):

   ```
   void hola_hasta(int n)
   ```

que dado un `int n`, imprime `n` veces "Hola". (Usar una bucle `while`). La función `main` pide un número en la entrada antes de llamar `hola_hasta` (¿qué función podes usar ya implementada?). Usá la función `assert` (ver teórico) para chequear que $x > 0$.

b) Los ejemplos a continuación han sido vistos en el teórico práctico. Para cada uno de ellos, se debe obtener la pre y post condición y la derivación. Luego, se debe traducir a Lenguaje C el programa y las pre y post condiciones utilizando el comando `assert`.

- (Mínimo) Cálculo del mínimo entre dos variables enteras `x` e `y`. El programa en C se debe llamar `minimo.c`.
- (Valor Absoluto) Especificar un programa que calcule el valor absoluto de un número entero. Verificar que el programa es correcto, y luego traducir el programa a C en un archivo nuevo llamado `absoluto.c`.
- (Intercambio de variables) Traducir en `intercambio.c` el siguiente programa que intercambia los valores de dos variables `x` e `y` de tipo `Int`.

```
z := x;
x := y;
y := z;
```

Nota: En todos los casos el programa en C, debe solicitar los valores de las variables de entrada, e imprimir el resultado para que lo pueda ver el usuario.

2. (Asignaciones múltiples) Considerar las siguientes asignaciones múltiples

```
{Pre: x = X, y = Y}
x, y := x + 1, x + y
{Post: x = X + 1, y = X + Y}
```

```
{Pre: x = X, y = Y, z = Z}
x, y, z := y, y + x + z, y + x
{Post: x = Y, y = Y + X + Z, z = Y + X}
```

a) Escribir un programa equivalente que sólo use secuencias de asignaciones simples.

b) Traducir los programas resultantes a C en archivos nuevos llamados `multiple1.c` y `multiple2.c` respectivamente.

Recordar: Como C no tiene asignaciones múltiples, siempre será necesario traducirlas primero a secuencias de asignaciones simples.

3. (Función suma_hasta) Crear un archivo llamado `suma_hasta.c`, que contenga la función

```
int suma_hasta(int n)
```

que toma un número entero `n` como argumento, y devuelve la suma de los primeros `n` naturales. En la función `main` pedir al usuario que ingrese el entero `n`, si es negativo imprimir un mensaje de error, y si es no negativo imprimir el resultado devuelto por `suma_hasta`.

Ayuda: La función puede hacer un ciclo o directamente usar la fórmula de Gauss.

4. (vocales) Crear un archivo `vocales.c` que contenga la función:

```
bool es_vocal(char letra)
```

que dado el caracter `letra` devuelve `true` si es una vocal y `false` en caso contrario. En la función `main` se le debe solicitar al usuario que ingrese un caracter y luego se debe mostrar un mensaje que indique si dicho caracter es una vocal o no según el resultado de la función `es_vocal()`. Tener en cuenta vocales mayúsculas y minúsculas.

NOTA: Definir una función que pida un caracter análoga a `pedir_entero()` pero para el tipo `char`.

NOTA: Recordar usar `%c` en vez de `%d` en el uso de `scanf()` y `printf()` para obtener / mostrar caracteres al usuario.

5. (Algoritmo de la división) Crear un archivo llamado `division.c` que contenga la siguiente función:

```
struct div_t division(int x, int y){
...
}
```

donde la estructura `div_t` se define como

```
struct div_t {
  int cociente;
  int resto;
};
```

Esta función recibe dos enteros no negativos (divisor no nulo) y devuelve el cociente junto con el resto de la división entera. En la función `main` pedir al usuario los dos números enteros, imprimir un mensaje de error si el divisor es cero, o imprimir tanto el cociente como el resto en otro caso.

6. (Arreglos, entrada-salida) Escribir un programa que solicite el ingreso de un arreglo de enteros `int a[]` y luego imprime por pantalla. El programa debe utilizar dos nuevas funciones además de la función `main`:

   - una que dado un tamaño máximo de arreglo y el arreglo, solicita los valores para el arreglo y los devuelve, guardándolos en el mismo arreglo `int a[]`; función con prototipo (también conocido como signatura o firma):

     ```
     void pedir_arreglo(int n_max, int a[])
     ```

   - otra que imprime cada uno de los valores del arreglo `int a[]`, de prototipo:

     ```
     void imprimir_arreglo(int n_max, int a[])
     ```

7. (Arreglos, Función sumatoria). Hacer un programa en un archivo con nombre `sumatoria.c` que contenga la función

   ```
   int sumatoria(int tam, int a[])
   ```

   que recibe un tamaño máximo de arreglo y un arreglo como argumento, y devuelve la suma de sus elementos (del arreglo). En la función `main` pedir los datos del arreglo al usuario asumiendo un tamaño constante previamente establecido (en tiempo de compilacilación).

8. (Múltiplos) . Hacer un programa en un archivo `multiplos.c` que contenga las siguientes funciones:

   ```
   bool todos_pares(int tam, int a[])
   bool existe_multiplo(int m, int tam, int a[])
   ```

   La primera recibe un tamaño máximo de arreglo y un arreglo como argumento devolviendo `true` cuando todos los elementos del arreglo `a[]` son numeros pares y `false` en caso contrario. La segunda determina si hay en el arreglo `a[]` algún elemento que es múltiplo de `m`. En la función `main` se debe pedir al usuario los elementos del arreglo (asumiendo un tamaño constante) y luego permitirle elegir qué función ejecutar. En caso que se elija la función `existe_multiplo()` se le debe pedir al usuario un valor para `m`.

9. (Procedimiento intercambio). Hacer un programa en el archivo nuevo `intercambio_arreglos.c` que contenga la siguiente función:

   ```
   void intercambiar(int tam, int a[], int i, int j)
   ```

que recibe un tamaño máximo de arreglo, un arreglo y dos posiciones como argumento, e intercambia los elementos del arreglo en dichas posiciones. En la función `main` pedirle al usuario que ingrese los elementos del arreglo y las posiciones, chequear que las posiciones estén en el rango correcto y luego imprimir en pantalla el arreglo modificado.

10. (Mínimos). Hacer un programa en un archivo con nombre `minimos.c` que contenga las siguientes funciones:

    ```
    int minimo_pares(int tam, int a[])
    int minimo_impares(int tam, int a[])
    ```

    Estas funciones reciben un tamaño máximo de arreglo y un arreglo como argumentos, devolviendo el elemento par más pequeño del arreglo y el elemento impar más pequeño del arreglo respectivamente.

    a) En la función `main` se debe pedir al usuario los elementos del arreglo (asumiendo un tamaño constante) y luego mostrar por pantalla:

    - El resultado de `minimo_pares()`, para el arreglo ingresado
    - El resultado de `minimo_impares()`, de nuevo, para el arreglo ingresado
    - El elemento mínimo del arreglo ingresado (utilizando el resultado de ambas funciones para calcularlo).

    Pueden definir alguna función auxiliar si les resulta necesario.

    NOTA: Investigar las constantes definidas en la librería `<limits.h>` para definir el neutro de la operación *mínimo*

    b) (Punto estrella) Hacer una segunda versión del programa en el archivo `minimos_estrella.c` y usar las funciones del ejercicio 8 en la fucnión `main` para que en caso de no haber elementos pares no se muestre el resultado de `minimo_pares()` y en caso de no haber impares no se muestre el resultado de `minimo_impares()`

11. (Función prim_iguales) Programar en el archivo `prim_iguales.c` la función

    ```
    int prim_iguales(int tam, int a[])
    ```

    que siendo `tam` el tamaño del arreglo `a[]` devuelve **la longitud** del tramo inicial más largo cuyos elementos son todos iguales (parecida a la función `primIguales` del Proyecto 1).

    a) En la función `main` se le debe pedir al usuario los elementos del arreglo asumiendo un tamaño constante previamente establecido (en tiempo de compilación) y luego mostrar el resultado de la función `prim_iguales` por pantalla

    b) (Punto Estrella) Mostrar por pantalla el mayor tramo inicial del arreglo `a[]` que tiene a todos sus elementos iguales.

12. (Función cuantos). Hacer un programa en un archivo nuevo `cuantos.c` que calcula cuántos elementos menores, iguales y mayores a un número hay en un arreglo. La función tiene el siguiente prototipo:

    ```
    struct comp_t cuantos(int tam, int a[], int elem)
    ```

    donde la estructura `comp_t` se define como sigue:

    ```
    struct comp_t {
        int menores;
        int iguales;
        int mayores;
    };
    ```

La función toma un tamaño máximo de arreglo, el arreglo y un entero, y devuelve una estructura con tres enteros que respectivamente indican cuántos elementos menores, iguales o mayores al argumento hay en el arreglo. La función `cuantos` debe contener un único ciclo.

13. (Función stats). Hacer un programa en un archivo nuevo `stats.c`, que calcula el mínimo, el máximo, y el promedio de un arreglo no vacío de números flotantes (tipo `float`). La función tiene el siguiente prototipo:

    ```
    struct datos_t stats(int tam, float a[])
    ```

    donde la estructura `datos_t` se define como sigue:

    ```
    struct datos_t {
        float maximo;
        float minimo;
        float promedio;
    };
    ```

    La función pedida debe implementarse con un único ciclo. En la función `main` pedir al usuario los datos del arreglo e imprimir en pantalla los tres valores devueltos por la función.

14. (Arreglo de asociaciones) En `asoc.c` programar la función

    ```
    bool asoc_existe(int tam, struct asoc a[], clave_t c)
    ```

    Donde la estructura `struct asoc` y los tipos `clave_t`, `valor_t` se definen como:

    ```
    typedef char clave_t;
    typedef int valor_t;

    struct asoc {
        clave_t clave
        valor_t valor
    };
    ```

    El llamado a `asoc_existe(tam, a, c)` debe indicar si la clave `c` se encuentra en el arreglo de asociaciones `a[]`. En la función `main` pedir al usuario los datos del arreglo (asumiendo un tamaño constante) y luego pedir una clave. Finalmente usar la función `asoc_existe` para verificar la existencia de la clave ingresada y mostrar por pantalla un mensaje indicando si la clave existe o no en el arreglo de asociaciones.

15. (Función nesimo_primo) En un archivo nuevo `primo.c` hacer una función

    ```
    int nesimo_primo(int N)
    ```

    que devuelve el n-ésimo primo.

    a) En la función `main` pedir al usuario que ingrese el entero `n`, si es negativo imprimir un mensaje de error, y si es no negativo imprimir el resultado devuelto por `nesimo_primo`.

    b) Modificar la función `main`, para que al ingresar un valor negativo, solicite un nuevo valor hasta que se ingrese un `n` no negativo.

16. (Punto estrella). Se define el tipo `persona_t` como sigue:

    ```
    typedef struct _persona {
        char *nombre;
        int edad;
        float altura;
        float peso;
    } persona_t;
    ```

    Definir las siguientes funciones:

```c
float peso_promedio(unsigned int longitud, persona_t arr[]);
persona_t persona_de_mayor_edad(unsigned int longitud, persona_t arr[]);
persona_t persona_de_menor_altura(unsigned int longitud, persona_t arr[]);
```

Las tres funciones toman como argumento una longitud máxima de arreglo y un arreglo de personas. Devuelven respectivamente el promedio de peso, la persona de mayor edad y la persona de menor altura que se encuentra en el arreglo. Ayuda: Para probar las funciones, hacer una función `main` como la siguiente:

```c
int main(void) {
    persona_t p1 = {.nombre="Paola", .edad=21, .altura=1.85, .peso=75};
    persona_t p2 = {.nombre="Luis", .edad=54, .altura=1.75, .peso=69};
    persona_t p3 = {.nombre="Julio", .edad=40, .altura=1.70, .peso=80};
    unsigned int longitud = 3;
    persona_t arr[] = {p1, p2, p3};
    printf("El peso promedio es %f\n", peso_promedio(longitud, arr));
    persona_t p = persona_de_mayor_edad(longitud, arr);
    printf("El nombre de la persona con mayor edad es %s\n", p.nombre);
    p = persona_de_menor_altura(longitud, arr);
    printf("El nombre de la persona con menor altura es %s\n", p.nombre);
    return 0;
}
```