Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 4

Programación Imperativa en C

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
```

Compilar para gdb Agregar un flag al momento de compilar .c escribir en la terminal:

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

Para ejecutar gdb:

```
$> gdb miprograma
```

Ver el teórico de gdb para aprender comandos básicos.

Ejercicios

1. (Traducción de Lisa a C) Completar los siguientes archivos

   a) `expresiones1.c`

   b) `condicional.c`

   c) `ciclo1.c`

   d) `ciclo2.c`

   e) `fibonacci.c`

   traduciendo a C los archivos de extensión `.lisa` del mismo nombre. Parte de las traducciones ya estan hechas a modo de ejemplo. Compilar y ejecutar cada uno de los programas.

2. (Intercambio de variables) Considera el siguiente programa que intercambia los valores de dos variables `x` e `y` de tipo `Int`.

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

   a) Especificar la pre y la post condición.

   b) Verificar que el programa es correcto (por ejemplo, demostrando que la precondición más débil implica la postcondición).

   c) Traducir ese programa a C en un archivo nuevo llamado `intercambio.c`. Pedir los valores iniciales de las variables al usuario, e imprimir en pantalla su valor final.

3. (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`. Pedir al usuario el valor inicial de la variable y luego imprimir en pantalla su valor absoluto.

4. (Asignaciones múltiples) Considerar las siguiente 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. Demostrar que los programas son correctos.

   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.

5. (Función suma_hasta) Completar el código del archivo `suma_hasta.c`. Escribir 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.

6. (Algoritmo de la división) Completar el archivo `division.c` que contiene 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 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.

   Ayuda: En el cuerpo de la función se puede usar un ciclo del primer ejercicio.

7. (Función sumatoria). Hacer el ejercicio 4 del práctio 4 (con derivaciones incluídas) y luego hacer un programa completando el archivo `sumatoria.c` que contiene la función

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

   que recibe un arreglo y su tamaño como argumento, y devuelve la suma de sus elementos. En la función `main` pedir los datos del arreglo al usuario asumiendo un tamaño constante.

8. (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 tipo:

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

   donde la estructura `comp_t` se define como sigue:

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

La función toma un arreglo, su tamaño 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.

9. (GDB) Usar GDB para encontrar los errores del programa del archivo `gdb-test.c`. Aprender a usar al menos los comandos *breakpoint*, *next*, *step*, *continue* y *print*.

Ejercicios Estrella

(a) (Función es_primo) En un archivo nuevo `primo.c` hacer una función

```
bool es_primo(int N)
```

que devuelve verdadero si el número `N` es primo, y falso en caso contrario.

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 `es_primo`.

b) Modificar la función `main` para que se repita el proceso anterior indefinidamente hasta que el usuario ingrese un número primo. Es decir, si ingresa un número no primo volver a pedirle otro número.

Ayuda: En el cuerpo de la función se puede usar un ciclo del primer ejercicio.

(b) (Funciones varias) Hacer el ejercicio 18 del práctico 3 (con derivaciones incluídas) y luego hacer una función por cada programa:

a) Máximo común divisor.

b) Exponencial.

c) Exponencial optimizado.

Las tres funciones (más la función `main`) deben estar en el mismo archivo nuevo `varias.c`. En la función `main` pedir al usuario dos números enteros y luego permitirle elegir qué función ejecutar a continuación. Imprimir en pantalla el resultado de la función elegida.

(c) (Función positivo). Hacer el ejercicio 5 del práctico 4 (con derivaciones incluídas) y luego hacer un programa en el archivo nuevo `positivo.c` que contenga las siguientes funciones:

```
bool existe_positivo(int a[], int tam)
bool todos_positivos(int a[], int tam)
```

Estas funciones reciben un arreglo y su tamaño como argumento, y devuelven respectivamente verdadero si existe un número positivo en el arreglo o si todos los elementos son positivos. En la función `main` pedirle al usuario los elementos del arreglo (asumiendo un tamaño constante) y luego permitirle elegir qué función ejecutar.

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

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

que recibe un arreglo, su tamaño 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 esten en el rango correcto y luego imprimir en pantalla el arreglo modificado.

(e) (Función min_max). Hacer un programa en un archivo nuevo `min_max.c`, que calcula el mínimo y el máximo de un arreglo no vacío. La función tiene el siguiente tipo:

```
struct datos_t min_max(int a[], int tam)
```

donde la estructura `datos_t` se define como sigue:

```
struct datos_t {
  int maximo;
  int minimo;
};
```

La función pedida debe implementarse con un único ciclo. En la función `main` pedir al usuario los datos del arreglo e imprimir en pantalla el máximo y el mínimo devuelto por la función.

(f) (Punto opcional super estrella). Hacer un programa en el archivo nuevo `ordenar.c` que reciba un arreglo y lo ordene de menor a mayor.

```
void ordenar(int a[], int tam)
```

Investigar en internet algún algoritmo de ordenación de su agrado (y que luego sepa explicar con sus palabras cómo funciona!).