Tema A

Ejercicio 1

Considerar la siguiente asignación múltiple:

```
var x, y : Int;
{Pre: x = X, y = Y, X * Y ≠ 0}
x, y := y * x, x * y
{Post: x = X * Y, x = y}
```

Escribir un programa en lenguaje C equivalente usando asignaciones simples. Verificar las pre y post condiciones usando la función `assert()`. Los valores iniciales de `x` e `y` deben ser ingresados por el usuario y los valores finales se deben mostrar por pantalla.

Ejercicio 2

Supongamos que contamos la cantidad de segundos que toma hacer cierta actividad (por ejemplo la cantidad de segundos usada para rendir un parcialito). Cómo no es sencillo interpretar si por ejemplo 2000 segundos es mucho o poco tiempo, vamos a hacer una función que nos diga a cuantas horas, minutos, segundos equivale:

```
struct lapso_t calcular_lapso(int segs) {
    ...
}
```

La estructura `lapso_t` se define como:

```
struct lapso_t {
    int horas;
    int minutos;
    int segundos;
};
```

Para calcular se puede seguir las siguientes formulas:

```
cantidad de horas: segs div 3600
cantidad de minutos: (segs mod 3600) div 60
cantidad de segundos: (segs mod 3600) mod 60
```

Programar la función `calcular_lapso()` y dentro de ella asegurar mediante `assert()` que la cantidad de segundos pasada como parámetro sea mayor o igual a cero.

En la función `main` pedir al usuario una cantidad de segundos e imprimir el resultado de la función `calcular_lapso()`. Si el usuario ingresa un valor negativo, mostrar un mensaje de error y terminar sin llamar a la función que calcula el lapso.

Ejercicio 3

Programar la función:

```
struct sum_t suma_acotada(int cota, int array[], int tam) {
    ...
}
```

donde la estructura `sum_t` se define como sigue:

```
struct sum_t {
    int inferior;
    int superior;
};
```

La función toma un valor entero `cota`, un arreglo `array[]` y su tamaño `tam` devolviendo una estructura con dos enteros. En el campo `inferior` debe estar la suma de los elementos de `array[]` que son menores a `cota`. En el campo `superior` debe estar la suma de los elementos que son mayores o iguales a `cota`. La función `suma_acotada()` debe contener un solo ciclo.

En la función `main` se deben pedir los datos del arreglo al usuario asumiendo un tamaño constante. Además se debe pedir el valor de la cota. Por ultimo se debe mostrar el resultado de la función por pantalla (los dos valores de la estructura).

Ejercicio 4*

Dada la estructura:

```
struct asoc_t {
    int clave;
    int valor;
};
```

Programar la función

```
bool hay_asoc(int key, struct asoc_t a[], int tam) {
    ...
}
```

que devuelve `true` si y solo si en el arreglo `a[]` (de tamaño `tam`) hay una asociación que tiene la clave `key`.

Además programar la función de entrada de datos:

```
void pedirArreglo(struct asoc_t a[], int n_max) {
    ...
}
```

que permite al usuario ingresar los datos de un arreglo de tipo `struct asoc_t`

Tema B

Ejercicio 1

Considerar la siguiente asignación múltiple:

```
var x, y : Int;
{Pre: x = X, y = Y, X > Y}
x, y := y + x, x - y
{Pos: x = X + Y, y = x - 2 * Y}
```

Escribir un programa en lenguaje C equivalente usando asignaciones simples. Verificar las pre y post condiciones usando la función `assert()`. Los valores iniciales de `x` e `y` deben ser ingresados por el usuario y los valores finales se deben mostrar por pantalla.

Ejercicio 2

Los colores en la computadora están identificados con números que van entre 0 y (2^{24} - 1), generalmente expresados en hexadecimal. Todos los colores se descomponen en sus componentes de rojo, verde y azul cuyos valores van de *0* a *255*. Vamos a hacer una función que dado un identificador de color nos devuelva las componentes de rojo, verde y azul que posee:

```
struct color_t calcular_color(int color_id) {
    ...
}
```

la estructura `color_t` se define como:

```
struct color_t {
    int rojo;
    int verde;
    int azul;
};
```

Para calcular las componentes se puede seguir las siguientes formulas:

```
valor de azul: color_id div 65536
valor de verde: (color_id mod 65536) div 256
valor de rojo: (color_id mod 65536) mod 256
```

Programar la función `calcular_color()` y dentro de ella asegurar mediante `assert()` que `color_id` sea mayor o igual a cero y menor que *16777215* (que es 2^{24} - 1).

En la función `main` pedir al usuario un identificador de color e imprimir el resultado de la función `calcular_color()`. Si el usuario ingresa mayor a *16777215* o negativo, mostrar un mensaje de error y terminar sin llamar a la función que calcula los componentes del color.

Ejercicio 3

Programar la función:

```
struct sum_t suma_multiplo(int mul, int array[], int tam) {
    ...
}
```

donde la estructura `sum_t` se define como sigue:

```
struct sum_t {
    int total_multiplo;
    int total_no_multiplo;
};
```

La función toma un valor entero `mul`, un arreglo `array[]` y su tamaño `tam` devolviendo una estructura con dos enteros. En el campo `total_multiplo` debe estar la suma de los elementos de `array[]` que son múltiplos de `mul`. En el campo `total_no_multiplo` debe estar la suma de los elementos que no son múltiplos de `mul`. La función `suma_multiplo()` debe contener un solo ciclo.

En la función `main` se deben pedir los datos del arreglo al usuario asumiendo un tamaño constante. Además se debe pedir un valor entero para pasarle como parámetro del múltiplo a `suma_multiplo()`. Por ultimo se debe mostrar el resultado de la función por pantalla (los dos valores de la estructura).

Ejercicio 4*

Dada la estructura:

```
struct asoc_t {
    int clave;
    int valor;
};
```

Programar la función

```
bool hay_asoc(int key, struct asoc_t a[], int tam) {
    ...
}
```

que devuelve `true` si y solo si en el arreglo `a[]` (de tamaño `tam`) hay una asociación que tiene la clave `key`.

Además programar la función de entrada de datos:

```
void pedirArreglo(struct asoc_t a[], int n_max) {
    ...
}
```

que permite al usuario ingresar los datos de un arreglo de tipo `struct asoc_t`

Tema C

Ejercicio 1

Considerar la siguiente asignación múltiple:

```
var x, y : Int;
{Pre: x = X, y = Y, X > 0}
x, y := y / x, x * y
{Pos: x = Y / X, y = x * X * X}
```

Escribir un programa en lenguaje C equivalente usando asignaciones simples. Verificar las pre y post condiciones usando la función `assert()`. Los valores iniciales de `x` e `y` deben ser ingresados por el usuario y los valores finales se deben mostrar por pantalla.

Ejercicio 2

A veces es útil dividir actividades según la terminación del DNI puesto que forman grupos bien definidos. Vamos a programar entonces una función que nos diga cuales son la tres últimas cifras de un DNI dado:

```
struct dni_t calcular_ultimos(int dni) {
    ...
}
```

la estructura `dni_t` se define como:

```
struct dni_t {
    int ultimo;
    int penultimo;
    int antepenultimo;
};
```

Para calcular las cifras (o dígitos) se puede seguir las siguientes formulas:

```
Último digito       : dni mod 10
Penúltimo digito    : (dni mod 100) div 10
Antepenúltimo digito: (dni mod 1000) div 100
```

Programar la función `calcular_ultimos()` y dentro de ella asegurar mediante `assert()` que `dni` sea mayor a 13000000 y menor que *200200000* ya que es el rango de DNIs de las alumnas y alumnos de la cátedra.

En la función `main` pedir al usuario un dni e imprimir el resultado de la función `calcular_ultimos()`. Si el usuario ingresa un valor fuera del rango 13000000 a *200200000*, mostrar un mensaje de error y terminar sin llamar a la función que calcula los últimos dígitos.

Ejercicio 3

Programar la función:

```
struct sum_t suma_rango(int a, int b, int array[], int tam) {
    ...
}
```

donde la estructura `sum_t` se define como sigue:

```
struct sum_t {
    int suma_dentro;
    int suma_fuera;
};
```

La función toma dos enteros `a` y `b`, un arreglo `array[]` y su tamaño `tam` devolviendo una estructura con dos enteros. En el campo `suma_dentro` debe estar la suma de los elementos de `array[]` que están en el rango `[a, b]`, es decir, los que son mayores iguales que `a` y son menores o iguales que `b`. En el campo `suma_fuera` debe estar la suma de los elementos que no están en el rango `[a, b]`, o sea los menores que `a` o los mayores que `b`. La función `suma_rango()` debe contener un solo ciclo.

Por ejemplo si el rango definido es `a=3, b=8` y el arreglo tiene elementos [**3**, 1, 2, **4**, 0,**7**, 2], entonces en el resultado de `suma_rango()` el campo `suma_dentro` debe valer `14` y `suma_fuera` debe valer `5`.

En la función `main` se deben pedir los datos del arreglo al usuario asumiendo un tamaño constante. Además se deben pedir los valores para el rango que se le pasará luego a la función `suma_rango()`. Por ultimo se debe mostrar el resultado de la función por pantalla (los dos valores de la estructura).

Ejercicio 4*

Dada la estructura:

```
struct asoc_t {
    int clave;
    int valor;
};
```

Programar la función

```
int cuenta_asoc(int key, struct asoc_t a[], int tam) {
    ...
}
```

que devuelve la cantidad de asociaciones que tienen como clave a `key` en el arreglo `a[]` de tamaño `tam`.

Además programar la función de entrada de datos:

```
void pedirArreglo(struct asoc_t a[], int n_max) {
    ...
}
```

que permite al usuario ingresar los datos de un arreglo de tipo `struct asoc_t`

En la función `main` deben pedirle al usuario el contenido de un arreglo `a[]` de tamaño constante con elementos del tipo `struct asoc_t`. Luego se le debe pedir un entero que se usará como clave en el llamado a `cuenta_asoc()`. Finalmente se debe mostrar la cantidad de asociaciones que tienen la clave ingresada por el usuario.

Tema D

Ejercicio 1

Considerar la siguiente asignación múltiple:

```
var x, y : Int;
{Pre: x = X, y = Y, Y > 0}
x, y := x mod y, x / y
{Pos: y * Y + x = X}
```

Escribir un programa en lenguaje C equivalente usando asignaciones simples. Verificar las pre y post condiciones usando la función `assert()`. Los valores iniciales de `x` e `y` deben ser ingresados por el usuario y los valores finales se deben mostrar por pantalla.

Ejercicio 2

En computación es muy habitual tener que descomponer un número en su representación de bits (digitos del numero en su base binaria). Vamos a programar una función que permite obtener la representación en bits de un numero entre 0 y 15 (con cuatro bits será suficiente):

```
struct bits_t calcular_bits(int num) {
    ...
}
```

la estructura `bits_t` se define como:

```
struct bits_t {
    int b0;
    int b1;
    int b2;
    int b3:
};
```

Para ilustrar se pueden ver las siguiente conversiones:

	Numero en binario			
Numero decimal	b3	b2	b1	b0
1	0	0	0	1
2	0	0	1	0
5	0	1	0	1

Para calcular las cifras (o dígitos) se puede seguir las siguientes formulas:

```
Primer bit (b0): num mod 2
Segundo bit    : (num div 2) mod 2
Tercer bit     : (num div 4) mod 2
Cuarto bit     : (num div 8) mod 2
```

Programar la función `calcular_bits()` y dentro de ella asegurar mediante `assert()` que `num` sea mayor o igual a 0 y menor que 16 (ya que no nos vamos a meter con los números negativos y para los números mayores a 15 serán necesarios más de 4 *bits*).

En la función `main` pedir al usuario un número e imprimir el resultado de la función `calcular_bits()`. Si el usuario ingresa un valor fuera del rango 0 a 15, mostrar un mensaje de error y terminar sin llamar a la función que calcula los *bits*.

Ejercicio 3

Programar la función:

```
struct sum_t suma_parimpar(int array[], int tam) {
    ...
}
```

donde la estructura `sum_t` se define como sigue:

```
struct sum_t {
    int suma_pares;
    int suma_impares;
};
```

La función toma un arreglo `array[]` y su tamaño `tam` devolviendo una estructura con dos enteros. En el campo `suma_pares` debe estar la suma de los elementos de `array[]` que son múltiplo de dos. En el campo `suma_impares` debe estar la suma de los elementos que no son pares. La función `suma_parimpar()` debe contener un solo ciclo.

En la función `main` se deben pedir los datos del arreglo al usuario asumiendo un tamaño constante. Por ultimo se debe mostrar el resultado de la función por pantalla (los dos valores de la estructura).

Ejercicio 4*

Dada la estructura:

```
struct asoc_t {
    int clave;
    int valor;
};
```

Programar la función

```
int cuenta_asoc(int key, struct asoc_t a[], int tam) {
    ...
}
```

que devuelve la cantidad de asociaciones que tienen como clave a `key` en el arreglo `a[]` de tamaño `tam`.

Además programar la función de entrada de datos:

```
void pedirArreglo(struct asoc_t a[], int n_max) {
    ...
}
```

que permite al usuario ingresar los datos de un arreglo de tipo `struct asoc_t`

En la función `main` deben pedirle al usuario el contenido de un arreglo `a[]` de tamaño constante con elementos del tipo `struct asoc_t`. Luego se le debe pedir un entero que se usará como clave en el llamado a `cuenta_asoc()`. Finalmente se debe mostrar la cantidad de asociaciones que tienen la clave ingresada por el usuario.