Programación

Unidad 6: Funciones y Procedimientos

“La primera regla de las funciones es que deben ser pequeñas. La segunda regla de las funciones es que deberían ser más pequeñas que eso ".

~ Robert C. Martin

Funciones y Procedimientos

En esta unidad explicaremos el concepto de división por módulos, sus principales características y las razones para utilizarlo. También veremos cómo dividir en módulos un programa utilizando funciones y procedimientos.

En las unidades anteriores vimos las estructuras de control para la toma de decisiones. También los ciclos iterativos con los cuales se pueden realizar un conjunto de instrucciones más de una vez.

A lo largo de las actividades, los algoritmos se van haciendo cada vez más complejos y con una mayor cantidad de acciones. Es por eso que necesitamos una estrategia para poder construir programas más complejos.

Utilizaremos la división por módulos para dividir el problema en subproblemas y, de esa manera, simplificar el desarrollo.

Supongamos un problema de la vida real.

Imagínate que decides mudarte, no importa el motivo, piensa solamente que te vas a mudar. Ahora bien, para resolver este "problema" necesitas hacer varias tareas, por ejemplo, buscar un nuevo hogar, adquirir la propiedad y, finalmente, realizar la mudanza.

Esto que acabamos de realizar fue dividir un problema muy grande en tres problemas más pequeños. Al resolver estos tres problemas, estarás resolviendo el problema "grande" que los abarca.

A su vez el problema de **buscar un nuevo hogar**, se puede dividir, en otros subproblemas como pueden ser:

- visitar inmobiliarias,
- navegar sitios web,
- comparar propiedades,
- visitar las propiedades,
- decidir cuál adquirir.

Como habrás notado, estamos dividiendo cada problema en problemas menores.

Ahora, ¿por qué hacemos esto? Principalmente porque es mucho más fácil resolver un problema pequeño que uno mayor.

Desarrollo Top Down

La metodología de división por módulos se conoce habitualmente como “divide y vencerás” y en programación se llama **Desarrollo Top Down**.

Pensar en el problema general e ir descomponiéndolo en subproblemas. A su vez, estos subproblemas se podrán seguir dividiendo hasta llegar a un subproblema lo bastante simple como para poder resolverse de forma sencilla.

Así es como realizaremos un algoritmo principal y varios subalgoritmos para ir resolviendo cada uno de los subproblemas.

Estos subalgoritmos, en programación, tienen distintos nombres, pero representan el mismo concepto. Es por eso que podemos encontrarnos con los nombres de: módulos, subprogramas, rutinas, subrutinas. Pero son todos sinónimos para definir el mismo concepto de “subproblema”.

Razones para Modularizar

Ya vimos que una de las razones para definir módulos es que resolver un problema más simple es mucho más fácil que resolver un problema complejo. Ahora bien, hay otros beneficios al modularizar un algoritmo.

Simplifica el programa: Un módulo es más simple de programar.

Permite la reutilización de código: La reutilización de código es una característica fundamental de la programación. Una vez que se programa un módulo, si ese mismo módulo hace falta para resolver otro problema, se puede utilizar sin necesidad de pensar nuevamente cómo resolverlo. Por ejemplo, si realizaste un módulo para calcular una raíz cuadrada, ese módulo lo podrás

reutilizar en cualquier algoritmo que tenga que calcular una raíz cuadrada. Es un subproblema que ya resolviste en otro momento y no tenés que pensar nuevamente en él, porque ya sabes que está resuelto. De esta manera evitamos lo que llamamos comúnmente “reinventar la rueda”.

El algoritmo resulta más simple y más claro: En lugar de tener una lista muy larga de instrucciones que puede llegar a ser ilegible, el algoritmo principal será una pequeña lista de módulos. De esta manera, al leer el algoritmo principal, se entenderá la idea de lo que realiza, sin necesidad de acceder a los detalles de cómo lo resuelve.

El programa es más simple de verificar: Las pruebas se pueden realizar en cada módulo de forma independiente, en lugar de tener que probar todo el programa a la vez.

Reduce los tiempos de mantenimiento: Realizar modificaciones en un programa es más simple y rápido porque dividir en módulos aísla los cambios. Una vez que se detecta en dónde realizar el cambio se deberá modificar ese módulo solamente. Sabiendo que el resto del Algoritmo debería funcionar como lo hacía anteriormente.

Abstracción

Veremos ahora un concepto fundamental en el desarrollo de programas que se deriva de la técnica de división por módulos. Este concepto es la Abstracción.

Concretamente, la abstracción se produce cuando creamos módulos.

Por ejemplo; si tenemos un algoritmo principal que se encuentra dividido en tres módulos de la siguiente manera:

```
Algoritmo Principal
    Obtener Notas de Un alumno
    Calcular Promedio De Notas
    Mostrar Promedio de Notas
Fin AlgoritmoPrincipal
```

Acá estamos aprovechando una de las ventajas de la división en módulos. Como comentamos previamente, el algoritmo es más claro ya que, con los nombres que hemos dado a los módulos, queda bien especificado qué es lo que hace cada uno de ellos.

Analicemos qué sucede con el Módulo “Calcular Promedio De Notas” ...

Seguramente estarás pensando que este módulo calcula el promedio de unas notas, pero no se ocupará ni de pedir las notas ni de mostrar los resultados. Este módulo está **abstraído del resto**, solo se ocupará de calcular el promedio. No sabemos tampoco cómo lo resolverá, ya que de eso nos ocuparemos cuando pasemos a desarrollar este módulo.

Es la idea de una **“caja negra”**, sabemos qué necesita y qué hace el módulo, pero no sabemos cómo lo realiza.

Los módulos son independientes entre sí, aunque algunos pueden necesitar colaborar con otros, o trabajar de forma conjunta. Veremos, en esta misma unidad, cómo lograr que los módulos se puedan “comunicar” o transmitir datos entre ellos a través de los parámetros.

Lo importante en este momento, para entender el concepto de abstracción, es comprender que **cada módulo es independiente de los demás módulos y que es ideal que realice una sola tarea**.

Podemos definir la abstracción como el **aislamiento de un elemento de su contexto o del resto de los elementos que lo acompañan.**

Ocultamiento de la información

Un concepto que se deriva de la abstracción es el de Ocultamiento de la información.

Continuamos con el ejemplo "Calcular Promedio de Notas"....

Para calcular el promedio, seguramente se realizarán operaciones intermedias, por ejemplo, sumar todos los números, contar la cantidad de números, etc. Esa información forma parte de la manera en que el módulo va a resolver su problema, no es de interés para el resto de los módulos, es por eso que esa información queda “oculta” para el resto del sistema.

Comunicación entre Módulos

Si analizamos el ejemplo anterior, entendemos que cada módulo tiene una única tarea, y que nos oculta la forma en que la va a realizar, pero...¿Puede hacer lo que necesita sin interactuar con nadie más?

Pensémoslo un poco más ¿Es posible que el módulo *Calcular Promedio de Notas* realice el promedio sin recibir ninguna información? Y yendo un poco más allá. ¿De qué forma el Algoritmo principal se entera del resultado del cálculo?

Se tiene que definir alguna forma en la cual cada módulo pueda recibir la información necesaria para realizar su tarea, y al mismo tiempo otro mecanismo para que informe el resultado obtenido a quién lo necesite. Para esto es que existen los Parámetros y Resultado (o retorno) del Módulo.

Por ejemplo, el módulo *Calcular Promedio de Notas* necesitará recibir las notas a promediar, para lo cual definirá los parámetros de entrada, y también definirá que al finalizar su ejecución retornará un valor real que será el promedio calculado.

Si se analiza pensando los pasos del análisis del problema de las primeras unidades, vemos que cada módulo puede pensarse como un algoritmo en sí mismo, esto es fruto de la abstracción y es lo que permite achicar los problemas hasta que sean más fáciles de resolver.

Tipos de Módulos

Existen dos tipos de módulos, según su declaración y utilización. Las funciones y los procedimientos.

Función

El concepto "función" no es nuevo ya que lo hemos usado en matemática. Una función matemática tiene parámetros que pertenecen al Dominio de esa función y las funciones devuelven un resultado de acuerdo al valor de dichos parámetros.

En programación los **módulos** que llamamos Funciones tienen el mismo comportamiento, de allí su nombre. En general son módulos que realizan alguna operación sobre sus parámetros o, como lo llamaremos, datos de entrada, y **devuelven un único resultado**.

Funciones en C/C++

La forma genérica de una función en C/C++ es la siguiente

```
tipo_de_retorno nombre_funcion( lista_parametros ) {
  cuerpo de la funcion
}
```

Consta de una definición o encabezado (también llamado header) y un conjunto de instrucciones (también llamado body o cuerpo). Veamos la definición de de cada parte:

- **Tipo de Retorno:** Las funciones retornan un resultado. El tipo de retorno es el tipo de dato del valor que la función retorna.
- **Nombre de Función:** Es el nombre de la función. El nombre y la lista de parámetros constituyen lo que llamamos firma de la función.
- **Parámetros:** Un parámetro es un contenedor de valor. Cuando una función es llamada se pasa un valor para cada parámetro declarado, este valor es referido como argumento. La lista de parámetros se refiere al tipo, orden y número de parámetros de una función. Los parámetros son opcionales; es decir una función puede no tener parámetros.
- **Cuerpo de la Función:** El cuerpo de la función contiene el conjunto de sentencias que define lo qué se ejecuta al invocar la función.

Si revisamos los ejemplos de código de unidades anteriores veremos que en realidad ya estábamos usando una función, que es la función principal o main. Recordemos el ejemplo del Para de la unidad anterior

```
1  int main() {
2      for (int i = 2; i <= 12; i++) {
3          if (i % 2 == 0) {
4              cout << “Es par:” << i;
5          }
6      }
7      return 0;
8  }
```

Cómo podemos ver, la función main no recibe parámetros de entrada y tiene un tipo de retorno int, por lo cual al final siempre tenemos un return.

Cabe destacar que si la función define un tipo de retorno entonces el algoritmo de la función SIEMPRE debe retornar un valor de ese tipo.

Veamos otro ejemplo, la siguiente función devuelve el valor máximo de los dos valores recibidos por parámetro.

```
1   // funcion que retorna el máximo de dos valores
2
3   int max(int num1, int num2) {
4      // declaracion de variable local
5      int result;
6
7      if (num1 > num2)
8          result = num1;
9      else
10         result = num2;
11
12     return result;
13  }
```

En este caso si recibimos parámetros que son los dos números a comprar, y retornaremos el valor que tengamos almacenado en result.

La variable result, está definida a nivel local, esto quiere decir que la variable solo se puede utilizar dentro de esta función, y al retornar la variable en realidad lo que estamos retornando es solo el valor actualmente almacenado en la misma.

Es decir, **las variables locales sólo pueden ser accedidas, leídas y modificadas por el módulo que las declara**. Los otros módulos no conocen su existencia ni pueden consultarlas.

Pero entonces, ¿cómo es que alguien podría obtener el valor que devuelve la función? Justamente a través de la invocación de la función con los parámetros necesarios. El resultado de una función puede ser utilizado en expresiones, asignaciones, etc.

El objetivo de una función es “extender” las instrucciones primitivas con las que ya cuenta el lenguaje.

Invocando funciones

Mientras que al definir o declarar una función, se debe indicar qué hace (las instrucciones que la definen). Para usar una función deberás llamarla o invocarla.

Cuando en un programa se llama a una función, el control del programa o el hilo de ejecución es transferido a la función llamada. La función llamada realizará las tareas definidas y al llegar a la instrucción return o al llegar a la llave que finaliza la función en caso de no tener retorno (luego veremos que estos módulos son llamados “procedimientos”), retorna el control al programa principal.

Para llamar una función simplemente se debe pasar los parámetros requeridos junto con el nombre de la función, y si la función retornara un valor, se puede almacenar en una simple asignación. Por ejemplo

```
    Ejemplo declarando toda la función antes
    del main.

1   #include <iostream>
2   using namespace std;
3
4   // decalracion completa de la funcion
5   int max(int num1, int num2) {
6      int result;
7
8      if (num1 > num2)
9          result = num1;
10     else
11         result = num2;
12     return result;
13  }
14
15  int main () {
16     // variables locales:
17     int a = 100;
18     int b = 200;
19     int ret;
20
21     // llamado al a funcion.
22     ret = max(a, b);
23     cout << "Maximo: " << ret << endl;
24
25     return 0;
26  }
27
```

```
Ejemplo declarando la firma antes del
main y cuerpo luego.

#include <iostream>
using namespace std;

// decalracion de la firma de la funcion
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // llamado a la funcion.
   ret = max(a, b);
   cout << "Maximo: " << ret << endl;

   return 0;
}

// declaracion de la funcion
int max(int num1, int num2) {

   if (num1 > num2)
       return num1;
   else
       return num2;
}
```

Como se puede ver en los ejemplos, la declaración de la función debe estar antes del main para poder utilizarla. En el caso de la izquierda se declara la función completa, mientras que en el caso de la derecha solamente se declara la firma, y la definición completa al final.

Declarar las firmas de las funciones y no las funciones completas es algo que permite separar la definición de la función de la implementación y permitirá más adelante separar en archivos las declaraciones en archivos header o .h y las implementaciones en archivos .cpp. Esto permite tener el más visible la función main y no tener que ir hasta muchas líneas de código más abajo.

En el cuerpo de la función de la derecha también se muestra otra forma que podría definir la búsqueda del mayor, sin necesidad de utilizar otra variable local sino retornando directamente los valores recibidos.

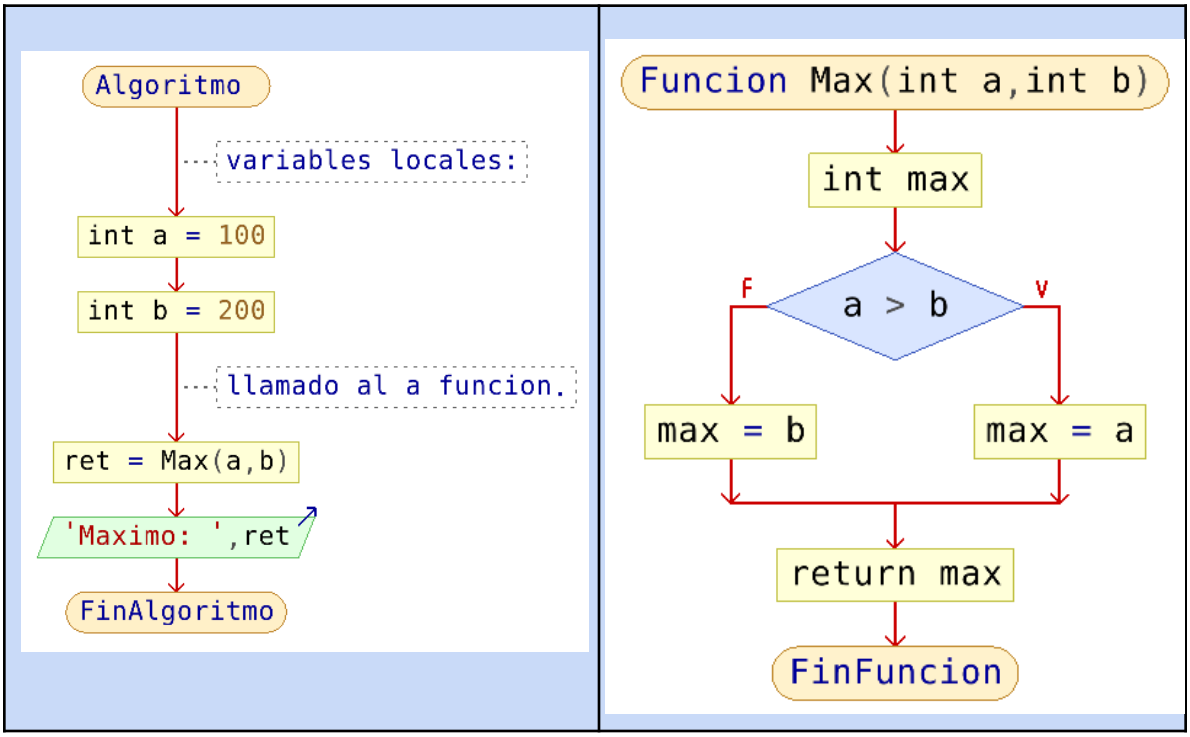
Cabe destacar que en el programa principal las variables locales son a y b, y al llamar a la función estas son las variables utilizadas, entonces a y b serán lo que llamamos “argumentos”, es decir la función en sus parámetros num1 y num2 recibirá los valores de a y b respectivamente. Vale la insistencia, lo que se pasa como parámetro en este caso es solo el valor y no la variable. Esto es lo que permite el ocultamiento y protección de la información: La función main no conoce

ni puede acceder a las variables locales de max, y viceversa. Toda la comunicación se realiza a través de los parámetros recibidos.

Diagramas

Para realizar los diagramas, lo que se hará es un diagrama independiente para cada módulo, y en el algoritmo principal (main) se mostrarán las invocaciones como si fueran expresiones.

Veamos el diagrama del algoritmo anterior:



Ámbito de las variables

Como vimos en la Unidad 2, una **variable es un espacio en memoria asociado a un nombre lógico**.

La **memoria total** que se encuentra disponible en un equipo informático está dividida en distintas áreas o sectores destinados a diferentes componentes del sistema. Por ejemplo, habrá un área de memoria para el sistema operativo, otra área de memoria para administrar las aplicaciones, y también habrá un área de memoria específica dedicada exclusivamente para nuestro programa.

Es por eso que las variables y todos los componentes de nuestro algoritmo utilizarán ese espacio de memoria. Estas secciones son conocidas como ámbitos de ejecución.

Las **variables, constantes, módulos y todo el algoritmo** serán almacenados en el ámbito del programa. Pero del mismo modo que te explicamos el concepto de dividir por módulos un

algoritmo, el área de memoria se encuentra, también, dividida generando los distintos ámbitos de memoria de cada rutina.

Por lo tanto, tendremos un ámbito de memoria global para nuestro algoritmo principal, y cada rutina tendrá definido un ámbito de memoria local, el cual será de acceso exclusivo dentro de esa rutina.

La división de estos ámbitos nos permite **clasificar las variables en globales y locales**.

Variables Globales

Las Variables Globales son aquellas que se definen en el programa principal (afuera del main). Si bien estas variables están accesibles desde el algoritmo principal y desde todos los módulos, es un error grave acceder directamente a ellas desde un módulo.

Para acceder a una variable global desde un módulo se debe pasar la variable como un parámetro. De esta manera logramos mantener la abstracción, ya que desde el módulo no se tendrá acceso directamente a un componente que no se encuentra definido en él.

A las Variables Globales se debe acceder solamente desde el programa principal, de lo contrario no estaremos realizando una abstracción de los módulos.

Declaración de variables globales en C / C++

```
1     #include <iostream>
2     using namespace std;
3
4     // Declaracion de variable global:
5     int g;
6
7     int main () {
8        // Decalracion de variable local:
9        int a, b;
10
11       // inicializacion
12       a = 10;
13       b = 20;
14       g = a + b;
15       cout << g;
16       return 0;
17    }
```

Un programa podría tener una variable local y otra global con el mismo nombre, pero la variable declarada en forma local tendrá preferencia y será la utilizada. Ejemplo

```
1     #include <iostream>
2     using namespace std;
3
4     // Variable global:
5     int g = 20;
6
7     int main () {
8        // Variable local:
9        int g = 10;
10
11       cout << g;
12
13       return 0;
14    }
```

Cuando el código anterior se compile y ejecute, se produce el siguiente resultado:

```
10
```

Variables Locales

Las variables locales son aquellas que se definen dentro de un módulo. A estas variables se pueden acceder solamente dentro del mismo módulo, ya que se encuentran almacenadas en su ámbito. Cuando un módulo se invoca, se crea en memoria el ámbito de ese módulo, que es donde se almacenan los parámetros y las variables locales.

Una vez que finaliza, se elimina su ámbito de memoria, borrándose todos los parámetros y variables locales.

A las Variables Locales solo se puede acceder desde el módulo que las define

En C / C++ existen dos formas de declarar variables que llamamos locales:

- Dentro de un bloque de Código o de una función.
- En la definición de una función como parámetros.

Retomando el ejemplo de la función max, vemos que la función tendrá las siguientes variables locales:

- Definidas por parámetros: **num1** y **num2**
- Definidas localmente: **result**

Mientras que la función main tiene las siguientes variables locales:

- Definidas localmente: **a**, **b** y **ret**

Analizando lo que hemos aprendido vemos que la función main no puede acceder a num1, num2 ni ret. Y de igual manera la función max no puede acceder a a, b y ret. Para que main pueda decirle a max cuáles son los valres a comprar tiene que pasar sus variables locales como argumentos al llamarla max(a, b).

Esto lo que generará es que al iniciar el ámbito local de la función max, se copiará **los valores** a sus variables locales establecidas como parámetros num1 y num2. El orden es lo que definirá qué argumento cae en qué parámetro.

En la declaración ➜ max(int num1, int num2)

En la invocación ➜ max(a , b)

Es por esta razón que las variables al ser de ámbitos distintos, podrían llamarse igual, o diferente, sin importar al ejecutarse.

Veamos cómo sería la definición de una función para calcular el promedio de notas en nuestro algoritmo:

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2           int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4           return promedio;
5   }
6
7   int main() {
8           int prom = 0;
9           int primerNota, segundaNota;
10
11          // Obtener Notas de Un alumno
12
13          prom = promedio(primerNota, segundaNota);
14
15          // Mostrar Promedio de Notas
16
17  }
```

Vemos que aún sin saber cómo se realizarán los módulos Obtener Notas de Un Alumno, ni Mostrar Promedio de Notas, ya podemos completar la parte que sí conocemos y definimos. Este también es un beneficio de la Modularización de los algoritmos, me permite poder avanzar por partes sin tener claras las implementaciones aún.

Procedimientos

Analicemos en el ejemplo anterior los módulos restantes:

Obtener Notas de Un alumno: Sabemos sólo por el nombre qué es lo que debería realizar, cuál es el objetivo de dicho módulo. Tomar las dos notas del alumno.

Pero si intentáramos realizarlo con una función, nos encontraremos con un inconveniente. Las funciones sólo devuelven un único resultado, de hecho, así las habíamos definido. Por lo cual no podemos hacer una función que devuelva dos notas, es decir esto es **completamente inválido:** *int, int obtenerNotas()*

Para estos casos existe el otro tipo de módulo, que se llama Procedimiento.

Definición	Un Procedimiento es un módulo que puede devolver **uno, muchos o incluso ningún resultado**

Los procedimientos, al igual que las funciones utilizan los parámetros para comunicarse con otros módulos, la diferencia es que la salida de un procedimiento no está limitada a un solo valor como en las funciones, sino que pueden ser muchos o incluso ningún valor.

Parámetros de Salida / Pasaje por Referencia

Para solucionar el problema que planteamos (tomar dos notas y devolverlas al programa principal) definiremos un procedimiento que tiene dos “parámetros de salida”.

Los parámetros de salida se declaran casi igual que los parámetros que hemos usado en las funciones (y que eran de entrada) solamente se debe agregar el símbolo & al parámetro. Veamos cómo será la definición:

void obtenerNotas (int **&**nota1, int **&**nota2);

El tipo de retorno es void dado que el procedimiento no retorna ningún valor a diferencia de la función, y los parámetros en lugar de ser contenedores de valores de entrada, ahora son **referencias** a variables y por lo tanto debemos especificar el símbolo &.

Si releemos la definición de los Ámbitos de variables recordaremos que las variables pertenecen a un ámbito específico y sólo las instrucciones de ese ámbito pueden acceder a sus valores. Entonces ahora lo que agregamos es que si deseamos que algún módulo modifique **una variable que no le pertenece** lo que deberá recibir como parámetro ya no es el valor de la variable sino la **referencia a la variable a modificar**. Esto en C/C++ es declarado con el símbolo &

Veamos cómo queda ahora el código de nuestro ejemplo:

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2           int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4           return promedio;
5   }
6
7   void obtenerNotas(int &val1, int &val2) {
8           cout << “Ingrese primer nota:”;
9           // Por ser pasado por referencia el valor modificado es el de la variable
10          //primerNota del main
11          cin >> val1;
12
13          cout << “Ingese segunda nota:”;
14          cin >> val2;
15  }
16
17  int main() {
18          int prom = 0;
19          int primerNota, segundaNota;
20
21          obtenerNotas(primerNota, segundaNota);
22
23          prom = promedio(primerNota, segundaNota);
24
25          // Mostrar Promedio de Notas
26
27  }
```

Lo que recibe el procedimiento obtener notas ya no es una copia de los valores de las variables primerNota y segundaNota, sino que recibe sus **referencias** para que el módulo pueda modificarlas.

Se debe tener especial cuidado en el uso de este tipo de parámetros dado que las acciones ejecutadas en el módulo tendrán impacto en el estado de las variables del algoritmo que lo llamo, es decir se pierde un poco de la protección de datos que habíamos mencionado en las funciones.

Parámetros de Entrada / Pasaje por Valor

El único módulo que nos resta programar en nuestro ejemplo es el de *Motrar Promedio de Notas*, que deberá informar por consola el resultado obtenido por el módulo **promedio** por lo tanto sabemos que precisará un dato de entrada. Los parámetros de entrada son los que hemos usado en toda la explicación de funciones, por lo cual ya sabemos como funcionan, se copia el valor de la variable pasada como argumento a una variable local del módulo que la recibe.

La única diferencia que veremos entre un Procedimiento que sólo recibe parámetros de entrada y una función, es que una función retorna un dato, mientras que este procedimiento no debe retornar nada. Es decir el módulo Mostrar Promedio de Notas no necesita devolver ningún valor

al programa principal, solamente imprimir por consola, por esto diremos que es un Procedimiento y no una función:

void mostrarPromedio (float promedio);

El parámetro promedio será un parámetro de entrada por lo cual en él recibiremos el valor de la variable usada en la invocación. Cualquier acción que hagamos sobre su valor, no modificará el valor en el programa principal (igual que en las funciones).

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2          int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4          return promedio;
5   }
6
7   void obtenerNotas(int &val1, int &val2) {
8          cout << “Ingrese primer nota:”;
9          // Por ser pasado por referencia el valor modificado es el de la variable
10         //primerNota del main
11         cin >> val1;
12
13         cout << “Ingese segunda nota:”;
14         cin >> val2;
15
16         return;
17  }
18
19  void mostrarPromedio(float promedio) {
20         cout << “El promedio del alumno es:” << promedio << endl;
21         return;
22  }
23
24
25  int main() {
26         int prom = 0;
27         int primerNota, segundaNota;
28
29         obtenerNotas(primerNota, segundaNota);
30
31         prom = promedio(primerNota, segundaNota);
32
33         mostrarPromedio(prom);
34  }
```

El procedimiento mostrarProcedimiento simplemente imprime por consola el valor recibido. En este caso hemos agregado la sentencia return; para marcar la finalización del procedimiento, como es void no se especifica ni valor ni variable, simplemente se fuerza la finalización. Esto puede ser utilizado en ciclos, condiciones, etc. Para forzar la finalización del procedimiento.

Combinando todo

Existe algunos casos donde podemos necesitar que un procedimiento reciba parámetros de entrada, pero también al tener más de una salida necesitará tener parámetros de salida. Esto lo podemos hacer simplemente combinando ambos tipos de parámetros en la declaración.

Veamos un ejemplo, supongamos que deseamos un procedimiento que realice una división entre dos números y calcule la parte entera y el resto.

```
1   void division (int dividendo, int divisor, int &cociente, int &resto) {
2           cociente = dividendo / divisor;
3           resto = dividendo % divisor;
4   }
5
6   int main() {
7           int divisor, dividend, c, r;
8           cout << “Ingrese el dividendo” << endl;
9           cin >> dividendo;
10          cout << “Ingrese el divisor” << endl;
11          cin >> divisor;
12
13          division(dividendo, divisor, c, r);
14
15          cout << “La división entera es:” << c << endl;
16          cout << “El resto de la división es:” << r << endl;
17  }
```

Ahora supongamos que queremos agregar la validación de que no se puede dividir por 0, entonces tenemos dos alternativas: o ponemos cociente y resto en 0 o lo informamos a través de algún retorno que indique si se pudo realizar la operación. Si así es, aunque sea un procedimiento también puede tener un valor de retorno, entonces si lo hacemos de esa forma la función quedaría

```
1   bool division (int dividendo, int divisor, int &cociente, int &resto) {
2           if (divisor == 0) {
3           return false;
4           }
5           cociente = dividendo / divisor;
6           resto = dividendo % divisor;
7           return true;
8   }
9
10  int main() {
11          int divisor, dividend, c, r;
12          cout << “Ingrese el dividendo” << endl;
13          cin >> dividendo;
14          cout << “Ingrese el divisor” << endl;
15          cin >> divisor;
16
17          if (division(dividendo, divisor, c, r)) {
18              cout << “La división entera es:” << c << endl;
19              cout << “El resto de la división es:” << r << endl;
20          } else {
21              cout << “No se pudo realizar la división” << endl;
22          }
23  }
```

Consejos para Modularizar

La técnica de modularizar para poder resolver un problema siempre es de gran utilidad y la debemos utilizar aunque luego no codifiquemos módulos independientes. Es decir, al plantearnos cómo resolver un problema siempre debemos “pensar en módulos”, luego dependiendo de si es de utilidad llevaremos el código a un módulo independiente o al algoritmo principal.

Por ejemplo, revisemos nuestro algoritmo para el cálculo de los promedios, desde un inicio identificamos y pensamos en 3 módulos. Obtener las notas, Calcular Promedio, Imprimir el Promedio. En consecuencia realizamos 2 procedimientos y 1 función para tener esos módulos codificados de forma independiente al programa principal. Para fines didácticos esto estuvo muy bien, pero si analizamos los dos procedimientos que generemos (Obtener notas e Imprimir Promedio) no nos significaron mucha ganancia a nivel de abstracción y reusabilidad.

El módulo Obtener notas, para que imprima mensajes que tengan sentido para el usuario quedo menos abstracto de lo que necesitaríamos para poder reutilizarlo, es decir no me va a servir para tomar cualquier dato, solo para tomar notas. De igual forma el módulo Imprimir promedio no me aporta mucho más que sacar las sentencias de impresión por consola, de igual forma el mensaje es muy específico como para que ese módulo sea reutilizado.

En cambio, la función promedio, si podremos reutilizarla siempre que deseemos calcular el promedio entre dos números. Para esto es muy importante que **tanto el nombre de la función como los nombres de parámetros no sean específicos del problema que estamos resolviendo sino lo más generales posibles**, si por ejemplo hubiéramos hecho la función

float promedioNotas(int nota1, int nota2)

operativamente es equivalente pero es menos abstracta dado que si deseamos calcular el promedio entre dos valores que representan distancias quedaría por lo menos raro.

Otro punto a considerar es que los módulos deben hacer una sola cosa para potenciar la abstracción y la reusabilidad. Por ejemplo, a alguien podría habérsele ocurrido hacer un procedimiento que además de obtener las notas ya hiciera la acumulación y la retornara, pero de esa forma nos hubiéramos perdido la posibilidad de usar un módulo genérico y reutilizable como es la función **promedio**.

Recursividad

¿Qué es la recursividad? La respuesta simpe es, cuándo una función se llama a sí misma. ¿Pero cómo pasa esto? Por qué pasaría esto, ¿y qué usos tiene?

Cuando hablamos de recursividad, en realidad estamos hablando de crear un bucle de repetición. Empecemos viendo un bucle básico

```
for(int i=0;  i<10; i++) {
    cout << "El número es: " << i <<
    endl;
}
```

Ya deberíamos saberlo, pero por las dudas aclaremos, esto debería generar la siguiente salida:

```
El número es: 0
El número es: 1
El número es: 2
El número es: 3
El número es: 4
El número es: 5
El número es: 6
El número es: 7
El número es: 8
El número es: 9
```

Ahora modifiquémoslo para que, en lugar de imprimir directamente, llame a una función que se encargue de eso

```
1  #include <iostream>
2  using namespace std;
3
4  void numberFunction(int i) {
5    cout << "El número es: " << i << endl;
6  }
7
8  int main() {
9
10     for(int i=0; i<10; i++) {
11       numberFunction(i);
12     }
13     return 0;
14 }
15
```

Declaramos un procedimiento (porque no tiene valor de retorno), que recibe un parámetro de entrada entero “int i” y lo único que hace es imprimir por consola el número recibido con la misma sentencia que antes estaba dentro del For. La función es llamada mientras el valor de i sea menor que 10.

Usando recursividad ya no necesitamos el ciclo For en el programa principal, porque haremos que nuestra función se llame a sí misma siempre que el número sea menor que 10 (misma condición que antes).

```
1  #include <iostream>
2  using namespace std;
3
4  void numberFunction(int i) {
5    cout << "El número es: " << i << endl;
6    i++;
7    if(i<10) {
8         numberFunction(i);
9    }
10 }
11 int main() {
12        int i = 0;
13        numberFunction(i);
14        return 0;
15 }
```

Vemos que, aunque hay una sola llamada a la función 'numberFunction' en el main, la función se invocará siempre que el valor de i recibido sea menor que 10.

Esto se produce porque cada llamada que se realiza desde la misma función se irá apilando, y al llegar a la llamada en la cual el valor de i es 10, no seguirá apilando llamadas y comenzará a volver el control de programa a la función que lo llamo, hasta llegar nuevamente al main.

Los algoritmos recursivos siempre tienen 2 partes, una es la regla trivial o caso base, que es lo que permite realizar el corte de la recursiviad, y luego la llamada a la misma función, pero con alguna modificación en sus parámetros.

Concepto de recursividad:

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea

Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.

Define el problema en términos de un problema más simple de la misma naturaleza. Debe disminuir el espacio del problema en cada llamada recursiva

Hay una instancia particular que se conoce como caso base o caso degenerado

Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- Una ecuación de recurrencia, en función de términos anteriores Tn = F(Tn-1, Tn-2, T0).
- Uno o varios términos particulares que no dependen de los anteriores. Ti = G(i) (base)

Veamos algunos ejemplos de problemas que pueden resolverse recursivamente

Función Factorial

- Ecuación de recurrencia : n! = n * (n-1)!
- Condiciones particulares: 0! = 1

```
1 double factorial(int n)
2 {
3        if (n==0)                    // Condiciones particulares: 0! = 1
4         return 1;
5        else
6          return n * factorial(n-1); // Ecuación de recurrencia :  n! = n * (n-1)!
7 }
```

Función PotenciaNatural

- Ecuación de recurrencia : an = a (n-1) * a si n > 1
- Condiciones particulares: a0 = 1

```
1 double potencia(int b, int p)
2 {
3         if(p == 0) {
4         return 1       // Condiciones particulares a0 = 1
5         } else {
6     return base * potencia(base, exponente -1)  // Ecuación de recurrencia
7 }
```

Sucesión de Fibonacci

Una pareja de conejos tarda un mes en alcanzar la edad fértil y a partir de aquí un mes en engendrar otra pareja que al alcanzar la fertilidad engendraran otra pareja, entonces ¿Cuántos conejos habrá al término de N meses? Si los animales no mueren cada mes hay tantas parejas como la suma de los dos meses anteriores

- Ecuación de recurrencia : Fib(n) = Fib(n-1) + (n-1)!
- Condiciones particulares: Fib(0) = 1 y Fib (1) = 1

Meses	Padres	Hijos	Nietos	Cant Parejas
0	I			1
1	M			1
2	M	I		2
3	M	M I		3
4	M	M M I	I	5
5	M	M M M I	M I I	8

¿Cuál sería su algoritmo?

Bibliografía utilizada

Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.

● Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)

● Debugging with CodeBlocks: http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks