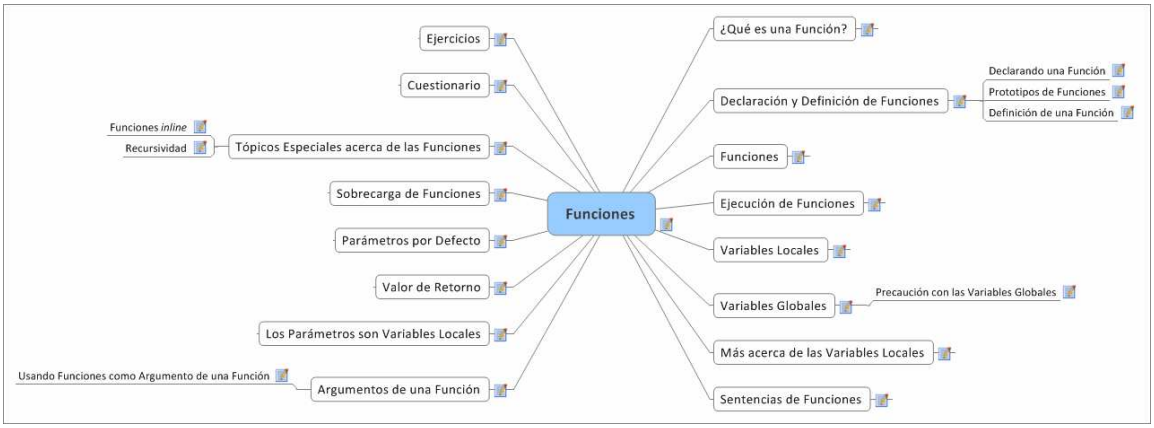
FUNCIONES



Aunque la programación orientada a objetos ha desviado la atención de las funciones hacia los objetos, las funciones son sin embargo el componente central de cualquier programa.

¿Qué es una Función?

Una función es un subprograma que actúa sobre datos y devuelve un valor. Cada programa escrito en C o C++ tiene al menos una función: main(). Cuando se inicia la ejecución del programa, automáticamente se ejecuta la función main(), la cual puede o no invocar otras funciones, las cuales a su vez podrían invocar a otras.

Cada función tiene su propio nombre, y cuando éste es encontrado, la ejecución del programa se transfiere al cuerpo de dicha función. Cuando finaliza la función, la ejecución del programa retorna a la línea siguiente desde donde fue invocada.

Existen dos tipos de funciones, las definidas por el usuario y las propias del compilador.

Declaración y Definición de Funciones

El uso de funciones en un programa exige su declaración y su definición. La declaración le indica al compilador el nombre, tipo de dato que retorna y parámetros que recibe la función. La definición le dice al compilador, cómo la función trabaja. Ninguna función puede ser invocada a menos que haya sido previamente declarada. A la declaración de una función se le llama prototipo.

Declarando una Función

Existen tres maneras de declarar una función:

Escribir el prototipo de la función en un archivo, y luego usar la directiva #include para incorporarla en su programa

Escribir el prototipo de la función en el archivo donde la función es usada

Definir la función antes de que sea invocada por cualquier otra función. En este caso, la definición funciona también como declaración

Aunque es posible definir una función antes de que sea usada, eliminando la necesidad de crear un prototipo, en general el uso de prototipos representa una mejor práctica de programación que no hacerlo. Existen al menos tres razones para ello:

Definir las funciones antes de que sean usadas obliga a establecer un orden específico para la escritura de las funciones, lo que puede dificultar el mantenimiento del código

Es posible que una función A() requiera invocar a una función B(), y que a su vez B(), bajo ciertas circunstancias, requiera a su vez invocar a A(). Resulta imposible definir A() antes de B() y simultáneamente definir B() antes de A().

Los prototipos de funciones son una buena herramienta de depuración. Si la declaración de una función define un conjunto específico de parámetros a ser usados por la función o un tipo particular de valor de retorno y luego la definición no coincide con el prototipo, el compilador señala el error más temprano que en el caso de usar sólo definiciones de funciones.

Prototipos de Funciones

Muchas de las funciones que vienen incluidas en las librerías del compilador tienen sus prototipos escritos en archivos que pueden ser incorporadas en los programas usando la directiva #include. Para las funciones definidas por el usuario, es necesario escribir los prototipos.

El prototipo de una función es una sentencia que consiste del tipo de dato de retorno, nombre de la función y lista de parámetros, terminada en punto y coma.

La lista de parámetros es una enumeración de todos los parámetros y sus tipos, separados por comas.

```
int suma (int op1, int op2);
```

En el ejemplo dado, la primera parte (int) representa el tipo del dato retornado por la función. En este caso, suma constituye el nombre de la función. Finalmente, entre paréntesis, está la lista de parámetros. En este ejemplo, la función suma recibe dos parámetros llamados op1 y op2, ambos de tipo entero (int).

El prototipo de la función y su definición deben concordar exactamente en tipo de valor retornado, nombre y lista de parámetros. Si no es así, el compilador señala el error. Sin embargo, el prototipo no necesita incluir el nombre de los parámetros, sólo sus tipos. Un ejemplo de prototipo perfectamente válido puede ser:

```
long area (int, int);
```

Este prototipo declara una función llamada area, que retorna una valor de tipo long y que tiene dos parámetros, ambos enteros (int). Aunque es legal, sin embargo constituye una mejor práctica de programación señalar el nombre de los parámetros ya que esto mejora la legibilidad del código. La misma función indicando los nombres de los parámetros luce de la siguiente manera:

```
long area (int largo, int ancho);
```

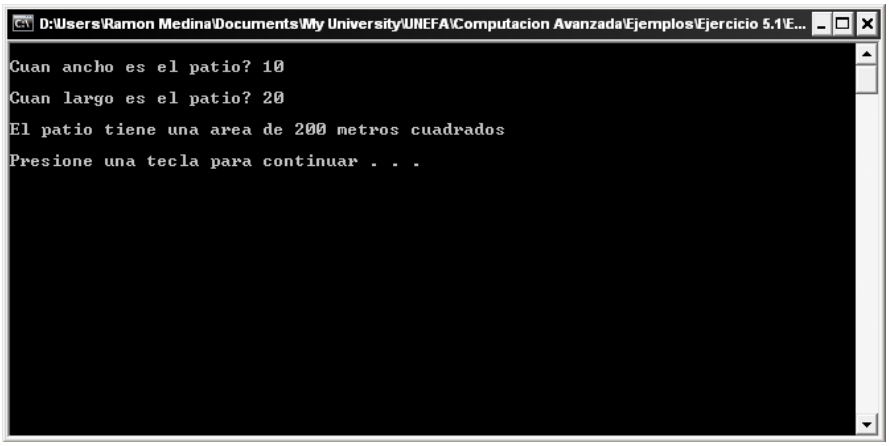
Con esta definición es obvio que función hace y cuál es la naturaleza de sus parámetros.

Todas las funciones tienen un tipo de dato de retorno. Cuando no se especifica una de manera explícita, se asume que es de tipo entero (int). Sin embargo, los programas serán más fáciles de leer si se especifica el tipo de dato de retorno de manera explícita en todos los casos.

```
1:  // Demuestra el uso de prototipos de una funcion
2:
3:  #include <iostream.h>
4:  #include <stdlib.h>
5:
```

```
6:  typedef unsigned short USHORT;
7:
8:  USHORT CalcularArea (USHORT largo,USHORT ancho);  // prototipo de funcion
9:
10: void main ()
11: {
12:     USHORT largoDelPatio;
13:     USHORT anchoDelPatio;
14:     USHORT areaDelPatio;
15:
16:     cout << "\nCuan ancho es el patio? ";
17:     cin >> anchoDelPatio;
18:     cout << "\nCuan largo es el patio? ";
19:     cin >> largoDelPatio;
20:
21:     areaDelPatio= CalcularArea(largoDelPatio,anchoDelPatio);
22:
23:     cout << "\nEl patio tiene una area de " << areaDelPatio << " metros
cuadrados\n\n";
24:
25:     system("PAUSE");
26: }
27:
28: USHORT CalcularArea (USHORT largo,USHORT ancho)
29: {
30:     return largo*ancho;
31: }
```

El prototipo de la función CalcularArea está en la línea 8. Compare el prototipo con el encabezado de la función en la línea 28. Note que se usa el mismo nombre, los mismos parámetros y el tipo de valor de retorno.



Definición de una Función

La definición de una función consiste en el encabezado y cuerpo de la función. El encabezado es idéntico al prototipo, con la excepción de que los parámetros deben obligatoriamente tener nombre (lo que puede ser omitido en el prototipo) y que no tiene punto y coma de terminación. El cuerpo de la función es el conjunto de sentencia incluidas entre las llaves

Funciones

La sintaxis para la construcción del prototipo de una función es la siguiente:

```
tipo_de_valor_de_retorno nombre_de_la_función (lista_de_parametros);
```

La sintaxis para la definición de la función es:

```
tipo_de_valor_de_retorno nombre_de_la_función (lista_de_parametros)
{
    sentencias;
    ...
}
```

El prototipo de una función le indica al compilador el tipo de valor de retorno, el nombre y la lista de parámetros de la función. No es indispensable que una función tenga parámetros, pero si los tiene, no es obligatorio que el prototipo indique sus nombres (sólo sus tipos). Un prototipo siempre termina con un punto y coma (;). La definición de la función debe concordar en tipo de valor de retorno y lista de parámetros con su prototipo. La definición de la función debe proporcionar los nombres de los parámetros y el cuerpo de la función debe estar confinado por llaves. Todas las sentencias dentro del cuerpo de la función deben terminar en punto y coma, pero la función en si no debe hacerlo; termina en una llave cerrada. Si la función retorna un valor, debe finalizar con una instrucción return. Toda función tiene un tipo de valor retornado. Si no se especifica de manera explícita, el tipo de valor retornado es int. Asegúrese de especificar el tipo valor retornado por la función. Si una función no retorna un valor, su tipo de valor retornado debe ser void.

Ejemplos de prototipos de funciones

```
long CalcularArea (long largo,long ancho);       //retorna long y tiene dos parámetros
```

```
void MostrarMensaje (int numeroDeMensaje);  // retorna void (nada) y tiene un
                                            // parámetro
int LeerOpcion ();                          // retorna int y no tiene parámetros
FuncionMala();                              // retorna int y no tiene parámetros
```

Ejemplos de definiciones de funciones

```
long Area (long l,long a)
{
    return l*a;
}
```

```
void MostrarMensaje (int mensaje)
{
    if (mensaje==0)
        cout << "Hola\n";
    if (mensaje==1)
        cout << "Adiós\n";
    if (mensaje>1)
        cout << "Estoy confundido\n");
}
```

Ejecución de Funciones

Cuando una función es invocada, la ejecución comienza en la primera sentencia luego de la llave abierta ({). Es posible hacer bifurcaciones en la ejecución de la función mediante el empleo de sentencias if y otras estructuras de control. Las funciones pueden también invocar a otras funciones o inclusive a ellas mismas (recursión).

Variables Locales

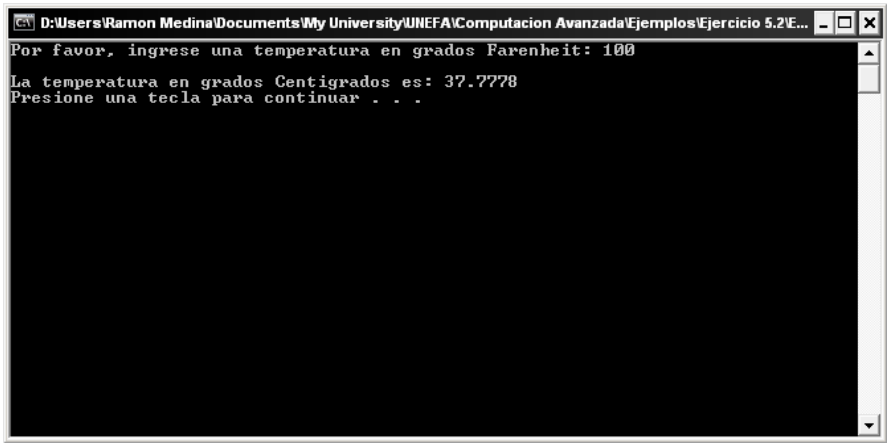
Es posible declarar variables dentro de una función. Estas reciben el nombre de variables locales, porque sólo existen localmente dentro de la función. Cuando la función termina su ejecución, las variables locales desaparecen

Las variables locales son definidas de la misma manera en la que lo son todas las variables. Los parámetros transferidos a una función pueden ser también

considerados como variables locales y pueden ser usados tal cual y como si hubiesen sido definidos dentro de la función.

```
1:  // Uso de variables locales y parametros
2:
3:  #include <iostream.h>
4:  #include <stdlib.h>
5:
6:  float Convertir (float temperaturaFarenheit);
7:
8:  void main ()
9:  {
10:     float tempFar;
11:     float tempCen;
12:
13:     cout << "Por favor, ingrese una temperatura en grados Farenheit: ";
14:     cin >> tempFar;
15:
16:     tempCen= Convertir(tempFar);
17:
18:     cout << "\nLa temperatura en grados Centigrados es: " << tempCen << endl;
19:
20:     system ("PAUSE");
21: }
22:
23: float Convertir (float temperaturaFarenheit)
24: {
25:     float temperaturaCentigrado;
26:
27:     temperaturaCentigrado= (temperaturaFarenheit-32)*5/9;
28:     return temperaturaCentigrado;
29: }
```

En las líneas 10 y 11 se declaran dos variables reales de simple precisión (float), una para almacenar la temperatura en grados Farenheit y otra para grados Centígrados. En la línea 13 se le pide al usuario que ingrese una temperatura en grados Farenheit y en la línea 16 se invoca a la función Convertir para transformar los grados Farenheit a grados Centígrados. Nótese que en la línea 25 se declara una variable local llamada temperaturaCentigrado, que sólo existen dentro de la función Convertir. Asimismo, en la línea 27 se emplea el parámetro temperaturaFarenheit que es utilizado en la expresión como si fuera una variable



Nuevo Término. Una variable tiene un alcance, que determina cuándo está disponible para ser 'accesada' dentro del programa. Las variables declaradas dentro de un bloque, tienen como alcance a dicho bloque; solo pueden ser manipuladas dentro del bloque y 'dejan de existir' cuando la ejecución sale de dicho bloque. Las variables globales tienen un alcance global y están disponible a o largo de todo el programa.

Variables Globales

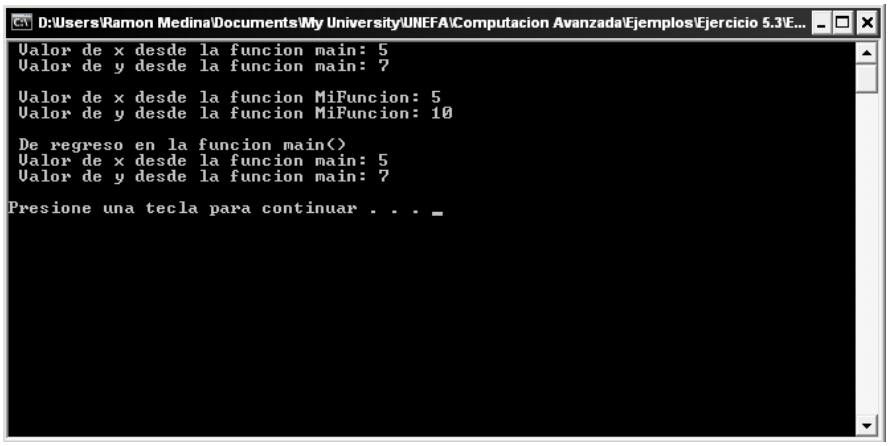
Las variables globales son declaradas fuera de las funciones por lo que tienen alcance global y están disponibles para cualquier función, incluyendo a main(). Las variables locales que tengan el mismo nombre que las globales, no las modifican. Una variable local con el mismo nombre que una global, la esconde. Si una función tiene una variable local con el mismo nombre que una global, dentro de la función el nombre siempre se referirá a la variable local.

```
1:  // Demostracion del uso de variables globales y locales
2:
3:  #include <iostream.h>
4:  #include <stdlib.h>
5:
6:  void MiFuncion();      // Prototipo
7:
8:  int x= 5, y=7;         // Variables globales
9:
10: int main ()
11: {
12:     cout << " Valor de x desde la funcion main: " << x << endl;
13:     cout << " Valor de y desde la funcion main: " << y << endl << endl;
14:
```

```
15:     MiFuncion();
16:
17:     cout << "\n De regreso en la funcion main()\n";
18:
19:     cout << " Valor de x desde la funcion main: " << x << endl;
20:     cout << " Valor de y desde la funcion main: " << y << endl << endl;
21:
22:     system ("PAUSE");
23: }
24:
25: void MiFuncion ()
26: {
27:     int y= 10;
28:
29:     cout << " Valor de x desde la funcion MiFuncion: " << x << endl;
30:     cout << " Valor de y desde la funcion MiFuncion: " << y << endl;
31: }
```

Este sencillo programa ilustra varios puntos claves (y potencialmente confusos) acerca del uso de variables globales y locales. En la línea 8 se declaran dos variables globales que son inicializadas con los valores 5 y 7. En las líneas 12 y 13 dentro de la función main(), se despliegan dichos valores por pantalla.

Cuando MiFuncion es invocada en la línea 15, la ejecución del programa se transfiere a la línea 27, donde se declara una variable local con igual nombre que una de las variables globales, que sin embargo es inicializada con un valor distinto. En las líneas 29 y 30 dentro de MiFuncion, se despliegan los contenidos de las variables x y y, en este caso el contenido de la variable x global y de la variable y local. Cuando la función termina su ejecución retorna a la línea 16 de la función main(). En las líneas 19 y 20 se despliegan nuevamente los contenidos de las variables globales. Nótese que el contenido de la variable global y no resultó afectado por la definición y uso de una variable local con el mismo nombre.



Precaución con las Variables Globales

En C y C++ las variables globales son legales pero son poco usadas. Las variables globales son peligrosas, porque almacenan información compartida y una función puede modificar su contenido de maneras que lo pueden hacer inadecuado para otras funciones. Esto puede ocasionar errores de funcionamiento en el programa que sean difíciles de identificar.

Más acerca de las Variables Locales

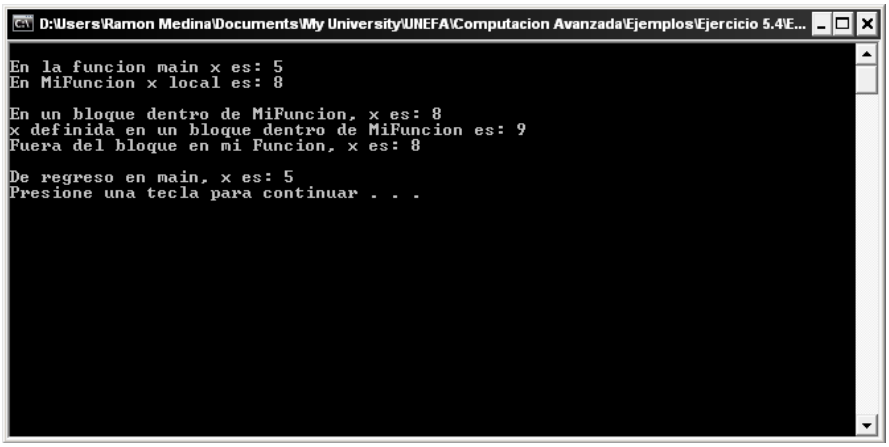
Se dice que las variables declaradas dentro de una función, tienen un alcance local. Esto significa, que son visibles y utilizables sólo dentro de la función donde fueron definidas. Inclusive, en C++ es posible definir variables en cualquier lugar dentro de la función y no sólo al comienzo; el alcance de la variable será el bloque donde haya sido definida. De hecho, si la variable es definida dentro de un par de llaves dentro de la función, su alcance será el bloque de código allí establecido.

```
1:  // Demostracion del uso de variables
2:  // con alcance dentro de un bloque de codigo
3:
4:  #include <iostream.h>
5:  #include <stdlib.h>
6:
7:  void MiFuncion();
8:
9:  void main()
10: {
11:     int x= 5;
12:     cout << "\nEn la funcion main x es: " << x;
13:
14:     MiFuncion();
15:
16:     cout << "\nDe regreso en main, x es: " << x << endl;
17:
18:     system("PAUSE");
19: }
20:
21: void MiFuncion()
22: {
23:     int x= 8;
24:     cout << "\nEn MiFuncion x local es: " << x << endl;
25:
26:     {
27:         cout << "\nEn un bloque dentro de MiFuncion, x es: " << x;
28:
29:         int x= 9;
```

```
30:
31:         cout << "\nx definida en un bloque dentro de MiFuncion es: " << x;
32:     }
33:
34:     cout << "\nFuera del bloque en mi Funcion, x es: " << x << endl;
35: }
```

El programa comienza inicializando la variable x en la línea 11 con el valor 5. El comando cout en la línea 12 permite corroborar con qué valor fue inicializada. MiFuncion es invocada en la línea 14, y dentro de ella, en la línea 23 se define una variable local x que inicializada con el valor 8. La sentencia en la línea 24 escribe 8 en lugar de 5, porque la variable local tiene precedencia dentro de la función. En la línea 26 se abre un bloque de código, dentro del cual la línea 27 muestra el contenido de la variable x, que sigue siendo 8 (definición local). Sin embargo, en la línea 29 se redeclara la variable x, en esta ocasión inicializada con el valor 9. La sentencia de la línea 31 muestra 9 en lugar de 8, porque la última definición de la variable x tiene precedencia incluso ante la variable local. En la línea 34, habiendo salido del bloque, la definición de x retorna a la variable local declarada en la línea 23.

Está demás decir, que aunque el programa anterior muestra que dichas definiciones son 'legales' y válidas, su utilización produce más confusión que ayuda, por lo que este tipo de 'redefiniciones' de una variable, no constituye una buena práctica de programación.



Sentencias de Funciones

Virtualmente no existe límite al número o tipo de sentencias que pueden estar dentro del cuerpo de una función. Aunque no es posible definir una función dentro de otra, es sin embargo posible invocar otra función. Las funciones

pueden inclusive invocarse a si mismas, lo que es llamado recursión o recursividad.

Aunque no existe para el tamaño de una función en C o C++, las bien diseñadas tienden a ser pequeñas. Muchos programadores recomiendan mantener el tamaño de las funciones lo suficientemente pequeño como para que quepa en una sola pantalla.

Cada función debe llevar a cabo una tarea sencilla y fácil de entender. Si las funciones del programa comienzan a ser muy largas, es momento de buscar cómo dividirlas en funciones más simples.

Argumentos de una Función

Los argumentos de una función no tienen porque ser del mismo tipo. Es perfectamente razonable que una función tome como argumentos un entero, dos enteros largos y un caracter.

Cualquier expresión válida en C y C++ puede ser usada como argumento de una función, incluyendo constantes, expresiones lógicas y matemáticas, así como otras funciones que retornen valores.

Usando Funciones como Argumento de una Función

Aunque es legal que una función tome como argumento a otra función que retorne un valor, esto puede hacer que el código sea más difícil de entender. Como ejemplo, supongamos que se tienen cuatro funciones llamadas doble(), triple(), cuadruple() y quintuple() y que todas ellas devuelven un valor. Esto hace posible escribir una sentencia como la siguiente:

```
miRespuesta= double( triple( cuadruple( quintuple( miVariable ) ) ) );
```

Esta sentencia toma el contenido de miVariable y se lo pasa como argumento a la función quintuple. El valor que retorna la función quintuple es a su vez pasado como argumento a la función cuadruple. El que esta devuelve es pasado a la función triple y el de esta a su vez, a la función doble. El valor retornado por la función doble es finalmente almacenado en la variable miRespuesta.

En una sentencia como es difícil seguir el curso de las acciones lo que complica la corrección de errores.

Los Parámetros son Variables Locales

Los argumentos pasados a una función son locales a la misma. Por lo tanto, los cambios que se haga a los contenidos de dichos argumentos, no afectan los valores de la función 'invocadora'. Esto es conocido como 'pase por valor', lo que significa que se dentro de la función, se hace una copia local de cada argumento. Esas copias son tratadas como cualquier otra variable local.

```
1:  // Demostracion de pase por valor
2:
3:  #include <iostream.h>
4:  #include <stdlib.h>
5:
6:  void Intercambiar(int x,int y);
7:
8:  void main()
9:  {
10:     int x= 5, y= 10;
11:
12:     cout << "Funcion main(). Antes del intercambio, x: " << x << " y: " << y << endl;
13:     Intercambiar(x,y);
14:     cout << "Funcion main(). Despues del intercambio, x: " << x << " y: " << y << endl;
15:
16:     system ("PAUSE");
17: }
18:
19: void Intercambiar (int x,int y)
20: {
21:     int temporal;
22:
23:     cout << "Funcion Intercambiar. Antes del intercambio, x: " << x << " y: " << y <<
endl;
24:
25:     temporal= x;
26:     x= y;
27:     y = temporal;
28:
29:     cout << "Funcion Intercambiar. Despues del intercambio, x: " << x << " y: " << y <<
endl;
30: }
```

Este programa inicializa dos variables en la función main() con los valores 5 y 10 respectivamente y los muestra en la línea 12. En la línea 13 invoca a la función Intercambiar() la cual en la línea 23 muestra el contenido de los argumentos, intercambia los valores en las líneas 25 a la 27 y los muestra, intercambiados, en la línea 29. Al retornar a la función main(), en la línea 14 se muestran nuevamente los contenidos de las variables que permanecen inalterados, ya

que el intercambio se realizó en los argumentos de la función Intercambiar() y no en las variables de la función main().

Valor de Retorno

Las funciones retornan un valor o retornan void. Void le indica al compilador, que la función no retorna valor alguno.

Para que la función devuelva un valor, es necesario escribir la return seguido del valor que se desea retornar. Dicho valor puede ser a su vez una expresión que devuelva un valor. Algunos ejemplos son:

```
return 5;
return (x > 5);
return (MiFuncion());
```

Todas las anteriores son sentencias válidas, suponiendo que la función MiFuncion() devuelva un valor. El valor de la segunda sentencia será cero si x no es mayor que 5 y uno si lo es. En ese caso devolver 0 (falso) o 1 (verdadero) pero no el valor de x.

Cuando se ejecuta la instrucción return, el valor que le acompaña es retornado como valor de la función. La ejecución del programa regresa de inmediato a la función invocadora, así que no será ejecutada cualquier sentencia que esté escrita luego de return. Es legal tener más de una sentencia return dentro de una función.

Parámetros por Defecto

La función invocadora debe proporcionar un valor para cada parámetro que sea declarado en el prototipo de la función. Los valores proporcionados deben ser del tipo declarado. Así, si se tiene una función declarada como:

```
long MiFuncion (int x)
```

la función debe tomar una variable entera. Si la definición de la función difiere o si no se proporciona el valor correspondiente al argumento, el compilador señala un error.

La única excepción a esta regla es cuando el prototipo declara un valor por defecto para el parámetro. El valor por defecto es usado cuando la función invocadora no lo proporciona. La declaración previa puede ser rescrita como:

```
long MiFuncion (intx=50);
```

El prototipo dice que MiFuncion devuelve un entero largo y que toma un entero como parámetro. Si el argumento no es suministrado, usará 50 como valor por defecto. No es necesario modificar la definición de la función por el hecho de haber declarado un valor por defecto.

Cualquiera de los parámetros de una función, puede tener valor por defecto. La única restricción es que si alguno de los parámetros no lo tiene, entonces ninguno previo lo puede tener. En un prototipo como el siguiente:

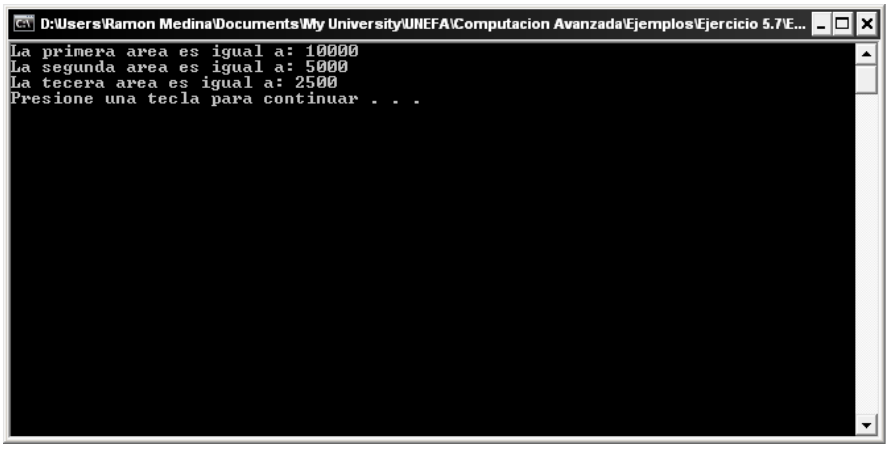
```
long MiFuncion  (int Param1,int Param2,int Param3);
```

es posible asignar un valor por defecto a Param2, siempre que también se le asigne uno a Param3. Se le podrá asignar un valor por defecto a Param1, siempre que Param2 y Param3 a su vez lo tengan.

```
1:  // Demostracion del uso de valor por defecto en los parametros
2:
3:  #include <iostream.h>
4:  #include <stdlib.h>
5:
6:  int AreaDelCubo(int largo, int ancho = 25, int alto = 1);
7:
8:  void main()
9:  {
10:     int largo = 100;
11:     int ancho = 50;
```

```
12:     int alto = 2;
13:     int area;
14:
15:     area = AreaDelCubo(largo, ancho, alto);
16:     cout << "La primera area es igual a: " << area << endl;
17:
18:     area = AreaDelCubo(largo, ancho);
19:     cout << "La segunda area es igual a: " << area << endl;
20:
21:     area = AreaDelCubo(largo);
22:     cout << "La tecera area es igual a: " << area << endl;
23:     system("PAUSE");
24: }
25:
26: int AreaDelCubo(int largo, int ancho, int alto)
27: {
28:     return (largo * ancho * alto);
29: }
```

La función AreaDelCubo es invocada en las líneas 15, 18 y 21. En la primera ocasión se le proporcionan los tres argumentos, así que se calcula el área de un cubo de 100 por 50 por 2. La segunda vez se omite el último argumento, así que la función asume el valor 1 para la altura. En la última ocasión se omiten los dos últimos parámetros así que la función trabajo con un ancho de 25 y una altura de 1.



ADVERTENCIA. Recuerde que los parámetros de la función son tratados como variables locales dentro de esta. No trate de asignar un valor por defecto al primer parámetro si el segundo no lo tiene. No olvide que los argumentos son pasados por valor y que no afectan las variables de la función invocadora. No olvide que los cambios hechos sobre una variable global por una función, modifica su contenido para todas las funciones.

Sobrecarga de Funciones

El C++ permite crear más de una función con el mismo nombre. Esto recibe el nombre de sobrecarga de funciones. Las funciones deben sin embargo diferir en la lista de parámetros, bien sea con parámetros de distinto tipo, distinta cantidad de parámetros o ambos. He aquí algunos ejemplos:

```
int MiFuncion (int a,int b);
int MiFuncion (long x,long y);
int MiFuncion (long z);
```

MiFuncion() es sobrecargada con tres listas de parámetros diferentes. La primera y la segunda versión difieren en el tipo de parámetros. La tercera difiere en la cantidad de parámetros.

El valor de retorno puede ser igual o diferente en las funciones sobrecargadas. Dos funciones con el mismo nombre y la misma lista de parámetros, aunque con diferente tipo de valor retornado, generarán un error del compilador.

NUEVO TÉRMINO. La sobrecarga de funciones es también llamada polimorfismo. Polimorfismo viene de poli que significa muchos y morfismo que significa forma. Una función polimórfica es aquella que adopta muchas formas.

El polimorfismo se refiere a la habilidad de 'sobrecargar' una función con más de un significado. Al cambiar el número de parámetros, es posible darle a dos o más funciones, el mismo nombre. La función correcta será ejecutada en función del número y tipo de argumentos utilizados en la invocación. Esto permite por ejemplo, crear una función que calcule promedios de enteros, reales de doble precisión y otros valores sin tener que crear nombres individuales para cada función.

Tópicos Especiales acerca de las Funciones

Debido a la importancia de las funciones en la programación, surgen algunos tópicos de interés cuando se confrontan problemas inusuales. Por ejemplo, las funciones inline permiten sacar el máximo provecho de cada bit de rendimiento. La recursividad es uno de esos aspectos 'esotéricos' de la programación, que de vez en cuando, ayudan a resolver problemas muy complejos con relativa sencillez.

Funciones inline

Cuando se define una función, el compilador crea un conjunto de instrucciones en la memoria. Cuando la función es invocada, la ejecución del programa 'salta' hasta donde se encuentran localizadas las instrucciones, y cuando la función retorna, la ejecución regresa a la línea siguiente donde se invocó. Si la función es invocada 10 veces, el programa 'saltará' al mismo conjunto de instrucciones las 10 veces. Significa que existe una única copia del conjunto de instrucciones.

El hecho de que el programa de bifurcar su ejecución secuencial hacia y desde la función, añade tiempo de ejecución que afecta el rendimiento del programa. En los casos en que las funciones son muy pequeñas, por ejemplo un par de líneas de código, es posible ganar eficiencia de ejecución (velocidad) evitando que el programa haga los saltos. Si la función es declarada con la palabra inline, el compilador no crea una función real, sino que copia el código de la función directamente donde ésta es invocada. Por lo tanto, no se ejecuta ningún salto. De hecho, resulta equivalente a haber escrito las líneas de código justo donde se invoca a la función.

Las funciones inline sin embargo, puede tener un costos alto. Si la función es invocada 10 veces, el código de la función es copiado 10 veces. La pequeña mejora en velocidad puede traer como consecuencia un aumento considerable del tamaño del programa.

Recursividad

Una función puede invocarse a si misma. Esto es llamado recursividad y esta puede ser directa o indirecta. Es directa cuando la función se invoca a si misma, y es indirecta cuando la función llama a otra función que a su vez la invoca a ella. Existen algunos problemas que son más fácilmente resueltos utilizando recursividad.

Es importante tomar en cuenta que cuando una función se invoca a si misma, se ejecuta una nueva copia de la función. Esto significa que las variables locales de la segunda versión, son independientes de la primera y que no las pueden modificar directamente.

Para ilustrar un ejemplo de recursividad, considérese la serie de Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Cada número después del segundo, es el producto de la suma de los dos anteriores. Un problema Fibonacci puede ser por ejemplo, determinar cuál es el décimo segundo término de la serie.

Una manera de resolver el problema planteado es examinar cuidadosamente la serie. Los dos primeros números son 1. Cada número subsiguiente es la suma de los dos anteriores. Por lo tanto, el séptimo número es la suma del quinto y el

sexto. Generalizando, el n-ésimo número es la suma del n-ésimo -1 y el n-ésimo -2, siempre que n sea mayor que 2.

Las funciones recursivas necesitan una condición de parada. Algo debe suceder para que la función deje de invocarse recursivamente, o nunca finalizará. En la serie de Fibonacci, la condición de parada es n menor que 3.

El algoritmo a usar sería algo como esto:

Preguntar al usuario qué término (n) de la serie desea

Invocar a la función fibonacci(), pasando el valor que ingresó el usuario

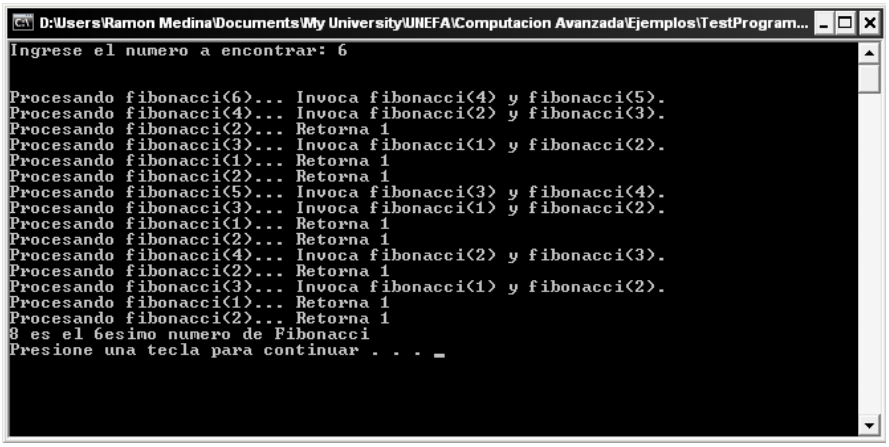
La función fibonacci() examina el argumento. Si es menor que 3, la función retorna 1; en caso contrario, la función se invoca a si misma pasando como argumento n-1, luego de nuevo pasando como argumento n-2 y retornando el resultado de ambas invocaciones.

Si se invoca fibonacci(1) o fibonacci(2), retornará 1. Si se invoca fibonacci(3), retornará la suma de los valores devueltos por fibonacci(2) y fibonacci(1). Si se invoca fibonacci(4), retornará el valor correspondiente invocando a su vez a fibonacci(3) y fibonacci(2) y así sucesivamente.

```
1:  // Demostracion de una funcion recursiva
2:  // Serie de Fibonacci
3:  // Usa el algoritmo fibonacci(n)= fibonacci(n-1) + fibonacci(n-2)
4:  // La condicion de parada es n = 2 || n = 1
5:
6:  #include <iostream.h>
7:  #include <stdlib.h>
8:
9:  int fibonacci(int n);
10:
11: int main()
12: {
13:     int n, respuesta;
14:     cout << "Ingrese el numero a encontrar: ";
15:     cin >> n;
16:
17:     cout << "\n\n";
18:
19:     respuesta = fibonacci(n);
20:
21:     cout << respuesta << " es el " << n << "esimo numero de Fibonacci\n";
22:
23:     system("PAUSE");
24:     return 0;
25: }
26:
27: int fibonacci (int n)
28: {
```

```
29:     cout << "Procesando fibonacci(" << n << ")... ";
30:
31:     if (n < 3 )
32:     {
33:         cout << "Retorna 1\n";
34:         return (1);
35:     }
36:     else
37:     {
38:         cout << "Invoca fibonacci(" << n-2 << ") y fibonacci(" << n-1 << ").\n";
39:         return (fibonacci(n-2) + fibonacci(n-1));
40:     }
41:}
```

En línea 15 al usuario que ingrese el número de Fibonacci que desea encontrar y en la línea 19 se invoca a la función fibonacci suministrando como argumento el valor suministrado por el usuario. La ejecución es entonces transferida a la función fibonacci, y en la línea 29 muestra su argumento. En línea 31 la función verifica si se dio la condición de parada (n < 3). Si es así, muestra el mensaje Retorna 1 y devuelve un 1. Si no lo es, en la línea 38 muestra otro mensaje donde señala a quien invoca y devuelve la suma de los valores devueltos por fibonacci(n-1) y fibonacci(n-2).



La recursividad no es una herramienta de uso común en C y C++. Sin embargo, ocasionalmente proporciona una manera potente y elegante de resolver ciertos problemas.

NOTA: la recursividad es una técnica complejo de programación avanzada Se presenta aquí, porque puede ser de mucha utilidad conocer su funcionamiento.

Cuestionario y Ejercicios

1. ¿Por qué no es conveniente que todas las variables sean globales?
2. ¿Cuándo debe usarse el comando inline al definir una función?
3. ¿Por qué no se reflejan los cambios que la función hace sobre sus argumentos, en las variables de la función invocadora?
4. Si los argumentos son pasados por valor, ¿cómo hago si necesito reflejar cambios de los argumentos en las variables de la función invocadora?
5. ¿Qué sucede si se declaran dos funciones como las presentadas a continuación?
6. int Area (int ancho,int longitud=1);
7. int Area (int tamano);
8. ¿Cuáles es la diferencia entre el prototipo de una función y su definición?
9. Si la función no retorna un valor, ¿cómo debe declararse?
10. Si no se declara explícitamente el tipo del valor de retorno, ¿qué valor se asume por defecto?
11. ¿Qué es una variable local?
12. ¿Qué significa el alcance de una variable?
13. ¿Qué es recursividad?
14. ¿Cuándo deben usarse variables globales?
15. ¿Qué significa sobrecarga de una función?
16. ¿Qué es polimorfismo?
17. Escriba el prototipo de una función llamada Perimetro que retorne un entero largo sin signo y reciba dos parámetros ambos enteros cortos sin signo

18. Escriba la definición de la función Perimetro como se describió en el ejercicio anterior. Los parámetros representan la longitud y ancho de un rectángulo. La función debe retornar el perímetro del rectángulo

19. ¿Qué error tiene la función que se muestra a continuación?

```
#include <iostream.h>
void miFuncion(unsigned short int x);
int main()
{
    unsigned short int x, y;

    y = miFuncion(int);
    cout << "x: " << x << " y: " << y << "\n";
}
void miFuncion (unsigned short int x)
{
    return (4*x);
}
```

20. ¿Qué error tiene la función que se muestra a continuación?

```
#include <iostream.h>
int miFuncion(unsigned short int x);
int main()
{
    unsigned short int x, y;

    y = miFuncion(x);
    cout << "x: " << x << " y: " << y << "\n";
}
int miFuncion (unsigned short int x)
{
    return (4*x);
}
```

21. Escriba una función que tome dos enteros cortos como argumento y que devuelva el resultado de dividir el primero por el segundo. SI el segundo número es cero, la función debe retornar -1.

22. Escriba un programa que pida al usuario dos números y llame a la función escrita en el ejercicio anterior. Muestre el resultado o un mensaje de error si la respuesta es -1.

23. Escriba un programa que pida un número y una potencia. Escriba una función recursiva que calcule el número elevado a la potencia.