UNIVERSIDAD DEL VALLE, FACULTAD DE CIENCIAS, DEPARTAMENTO DE FISICA

CAPÍTULO II

Proveer las declaraciones que controlan el flujo de información que tiene que desarrollar el programa, especificando cuando y bajo qué circunstancias debe ser realizado.

2. Declaraciones y Control de Flujo

Una declaración en C++, es cada línea del código que se finaliza con un (;). Pero los programas no estan constituidos sólo por declaraciones simples, en muchas ocaciones es necesario que un grupo de declaraciones o lo que es lo mismo un segmento de código se repinta varias veces o tomar decisiones y dividir la información. Varios de los controladores de flujo requieren como parte de su sintaxis un grupo de declaraciones encerradas en corchetes ({ })

```
{ statement1; statement2; statement3; }
```

2.1 Declaración de Selección: if y else

La palabra clave if, es usada para ejecutar una declaración o un grupo si y sólo si una condición se cumple.

```
if (condicion) declaracion
```

Aquí `condicion` es la expresión que debe ser evaluada. Si la `condicion` es verdadera entonces la `declaracion` es ejecutada, pero si es falsa la `condicion` entonces la `declaracion` no es ejecutada, es simplemente ignorada y el programa continúa justo después de la declaración o que se cierra el corchete .

Por ejemplo en el momento que x valga 100 se imprime en pantalla x es 100, si nunca es 100 esto es ignorado y el programa continúa sin evaluar la declaración.

```
1 if (x == 100)
2   cout << "x is 100";
```

Si hay más de una declaración los corchetes son indispensables para formar el bloque.

```
1 if (x == 100)
2 {
3    cout << "x is ";
4    cout << x;
5 }
```

Tenga en cuenta que diferentes declaraciones las puede escribir en un sólo renglón. Declaraciones y renglones no significan lo mismo, pero lo óptimo es escribir declaración por renglón para facilidad de lectura.

```
if (x == 100) { cout << "x is "; cout << x; }
```

Con la declaración de selección if también se puede especificar en el caso que la expresión evaluada no es cumplida usando la palabra clave `else` para introducir una alternativa en las declaraciones:

`if (condicion) declaracion1 else declaracion2`

donde `declaracion1` es ejecutada si `condicion` es verdadera y en caso de que no entonces `declaracion2` es ejecutada.

Por ejemplo:

```
1 if (x == 100)
2   cout << "x es 100";
3 else
4   cout << "x no es 100";
```

Aquí se imprime x es 100 si la condición x==0 se cumple, de lo contrario se imprime x no es 100.

Varias estructuras `if+else` pueden ser concatenadas, con la intención de revisar un rango de valores por ejemplo:

```
1 if (x > 0)
2   cout << "x es positiva";
3 else if (x < 0)
4   cout << "x es negativa";
5 else
6   cout << "x es cero";
```

Este código imprime si x es positiva, negativa o cero usando dos escructuras if-else. Recuerde que puede ejecutar más de una declaración si agrupa usando { } varias declaraciones en cada una de las estructuras.

Código fuente 14:

```
// conditional operator
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c,temp;
  cout<<"Ingrese un numero: ";cin>>a;
  cout<<"Ingrese un segundo numero: ";cin>>b;
  cout<<"Ingrese un tercer numero: ";cin>>c;
  if (a>b){
    temp=a;
    a=b;
    b=temp;
  }
  if (a>c){
    temp=a;
    a=c;
    c=temp;
  }
  if (b>c){
    temp=b;
    b=c;
    c=temp;
  }
  cout<<"Ordenando "<<a<<" , "<<b<<" y "<<c<<" de menor a mayor seria:";
```

```
  cout<<"\n"<<a<<" , "<<b<<" , "<<c<<"";
  return 0;
}
```

2.2 Declaraciones de iteración (loops)

Los loops o bucles repiten una declaración una cierta cantidad de veces o mientras una condición se cumple. Son realizados mediante las palabras claves: `while`, `do`, y `for`.

2.2.1 El loop for

El loop `for` es diseñado para iterar un cierto número de veces, su sintaxis es:

`for (inicializacion; condicion; incremento) declaracion;`

este loop repite la `declaracion` mientras `condicion` se cumple. Adicionalmente el loop provee el punto exacto donde comienza la iteración especificada en `inicializacion` y una expresión que define el incremento, la cual es ejecutada antes de que comienza el primer loop y después de cada iteración, por eso es muy útil usar variables de conteo como `condicion.`

El for funciona de la siguiente manera:

1. `initializacion` es ejecutada. Generalmente se declara la variable de conteo aquí y se le asigna un valor inicial el cual es ejecutado una única vez en el inicio del loop.

2. `condicion` es revisada. Si es verdadera, el loop continúa, de lo contrario el loop termina y la declaración es saltada e ignorada y se va directo al paso 5.

3. `declaracion` es ejecutada. Puede ser una sola declaración o un bloque encerrado entre corchetes {}.

4. `incremento` es ejecutado y el loop comienza de nuevo volviendo al paso 2.

5. El loop termina: la ejecución del programa continúa con la declaración que sigue justo al loop.

Un ejemplo del loop for realizando un conteo regresivo:

Código fuente 15:

```
1  // countdown using a for loop
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--) {
8      cout << n << ", ";
9    }
10   cout << "liftoff!\n";
11 }
```

Salida

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

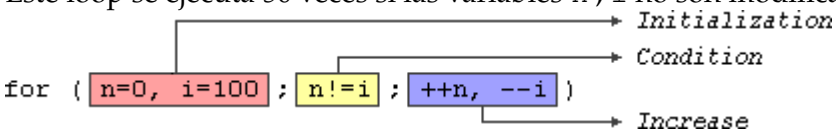
Existen casos en los que los tres campos en un bucle for `for (inicializacion; condicion; incremento)` no se llenan y pueden dejarse vacíos, pero siempre los punto y comas requeridos entre ellos se tienen que colocar. Por ejemplo usted puede usar `for(;n<10;++n)` donde tiene incremento

pero no inicialización porque la dicha inicialización la realizó por fuera del loop o usted también puede usar `for(;n<10;)` donde no coloca inicialización ni incremento porque dicho incremento se encuentra dentro del loop a manera de declaración (equivalente a un blucle de while, ver más adelante). En todos los casos tenga en cuenta que un loop sin `condicion` significa que tiene `true` como condición y por lo tanto es un loop infinito y puede llevar a errores.

Cada uno de los tres campos del bucle son ejecutados en un tiempo particular en un ciclo del bucle, y le puede ser utilizar más de una expresión individual o declaraciones a modo de inicialización, condición o incremento pero esto no es posible. Lo único que se puede hacer es usar el operador coma (,) para incluir más de una expresión en los campos. Por ejemplo, puede utilizar dos variables de conteo, inicializandolas e incrementandolas al tiempo mientras una única condición es cumplida:

```
1 for ( n=0, i=100 ; n!=i ; ++n, --i )
2 {
3    // whatever here...
4 }
```

Este loop se ejecuta 50 veces si las variables `n`, `i` no son modificadas dentro bucle:



en este ejemplo `n` comienza en 0, e `i` en 100, la condición es `n!=i` que `n` no puede ser igual a `i`, mientras ésto sea así `n` es incrementada por uno e `i` es disminuida en uno en cada iteración, la condición del loop será falsa despúes de la cincuentava iteración donde `n` e `i` son iguales a 50.

2.2.1.1. Loop `for` basado en rangos

El loop de for tiene otra sintaxis, la cual es exclusivamente utilizada si incluye el rango de validez, esto es:

```
for ( variable : rango ) declaracion;
```

Este bucles itera la `variable` sobre todos los elementos en un `rango` determinado, tenga en cuenta que la variable declarada debe ser capaz de tomar el valor de un elemento del conjunto que determina. Los rangos son secuencias de elementos que incluyen arreglos, contenedores o cualquier otro tipo que contenga un inicio y un final. Estos tipos de no se han incluido aún (se hará en el capítulo 3), sin embargo hay una clase de éstos que si hemos usado y es el `string`, el cual es una secuencia de caracteres, con lo que podemos usarlo para un ejemplo:

Código fuente 16

```
1 // loop for basado en rangos
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8   string str {"Hola!"};
```

Salida

```
[H][o][l][a][!]
```

```
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

Note que la variable usada es un `char` ya que los elementos del rango `string` son tipo `char`. Luego se usa la varialbe c en la declaración del bloque del for para representar cada uno de los elementos del rango. En este caso, no se necesita ninguna declaración de una variable de conteo ya que es claro cuando el loop comienza y termina. Los loops basados en rangos usualmente pueden realizar una auto detección del tipo de elemento que se encuentra dentro del rango, así que se puede usar el tipo de dato `auto` para declar la variable que se utiliza y el `for` anterior se puede escribir:

```
for (auto c : str)
  std::cout << "[" << c << "]";
```

donde el tipo de dato de c es automaticamente detectado como el tipo de dato de los elementos en `str`.

2.2.1.2 Declaraciones de salto

Estas declaraciones permiten alterar el flujo de un programa realizando saltos a una ubicación específica.

2.2.1.2.a Break

`break` interrumpe un loop, aún cuando la condición para terminar el for no se ha cumplido. Puede ser usado para terminar un loop infinito o para forzar un final antes del establecido. Por ejemplo puede terminar un conteo antes del final predeterminado:

Código fuente 17:

```
// interrupción de un bucle for
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "conteo abortado!";
      break;
    }
  }
}
```

Salida

```
10, 9, 8, 7, 6, 5, 4, 3, conteo abortado!
```

2.2.1.2.b Continue

`continue` causa que el programa salte la iteración en que se encuentra en el loop como si llegara a su final, haciendo que el ciclo comience de nuevo. Por ejemplo puede saltar un número dentro de un conteo:

Código fuente 18: Salida

```
// Ejemplo de continue en un for
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "despegue!\n";
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, despegue!
```

2.2.1.2.c Goto

`goto` permite realizar un salto a cualquier punto del programa. Este salto incondicional ignora niveles de anidamiento en las iteraciones y no causa ninguna conflicto en el flujo del programa. Es necesario tener especial cuidado con ésta declaración especialmente dentro de un bloque de declaraciones que tienen variables locales.

El punto de destino está identificado con una etiqueta, la cual debe ser usada en el argumento de la declaración `goto`. Una etiqueta debe estar bien identificada y seguida de dos puntos (:) como se muestra en el ejemplo.

En general los programadores EVITAN el uso de `goto` ya que es una herramienta considerada ser no óptima.

Código fuente 19: Salida

```
// ejemplo de goto
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
etiqueta:
  cout << n << ", ";
  n--;
  if (n>0) goto etiqueta;
  cout << "despegue!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, despegue!
```

2.2.2. El loop while

Es un loop simple y su sintaxis es:

```
while (expresion) declaracion
```

El loop while simplemente repite una declaración mientras la expresion es verdadera. Si después de cualquier ejecución de la declaración, la expresión deja de ser verdadera, el loop termina, y el programa continua justo después del loop. Por ejemplo:

Código fuente 20: Salida

```
//cuenta regresiva usando while
#include <iostream>
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, despegue!
```

```
3  using namespace std;
4
5  int main ()
6  {
7    int n = 10;
8
9    while (n>0) {
10     cout << n << ", ";
11     --n;
12   }
13
14   cout << "despegue!\n";
15 }
```

En el ejemplo, la primera declaración, antes de que comience el loop, le asigna el valor de 10 a la variable n. Éste será el primer número de la cuenta regresiva. Cuando el loop while comienza, si el valor de n cumple con la condición n>0, entoncees el bloque con declaraciones que le sigue a la expresión de condición se ejecuta, y se repite mientras la condición sea verdadera.
Todo el proceso del programa sigue los siguientes pasos de la prueba de escritorio, comenzando en la función `main`:
1. Se le asigna un valor a n.
2. Se evalúa la expresión de condición del while (n>0). Hay dos posibles respuestas:
    a) la condición es verdadera: en cuyo caso la declaración es ejecutada (primera línea dentro del bloque, paso 3).
    b) la condición es falsa: en cuyo caso la declaración se ignora y el programa continúa justo despúes que se acaba el bloque del while (paso 5).
3. En caso de ser verdadera la condición: Se evalúa la declaración:
    `cout << n << ", ";`
    `--n;`
    esto es, se imprime el valor de n y se disminuye n en una unidad.
4. Se finaliza el bloque y se vuelve automáticamente al paso 2.
5. Continúe el programa justo después que acaba el bloque donde encuentra la declaración imprima despegue! Y finaliza el programa

Algo importante para tener en cuenta con los ciclos while es que el loop debe terminar en algún momento, y por lo tanto las declaraciones dentro del bloque deben alterar los valores que se revisan en la condición para llevarla a ser falsa en algún momento. De otro modo, el loop será infinito. En este caso particular la declaración incluye `--n`, que reduce el valor de la variable hasta que eventualmente hará que la condición de n>0 sea falsa en algún momento finalizando el loop while.

2.2.3. El loop do-while

Un bucle similar al while es el do-while, cuya sintaxis es:

```
do statement while (condition);
```

Este loop se comporta como un loop while, exeptuando que la condición es evaluada después de la ejecución de la declaración en lugar de antes. Lo que garantiza por lo menos UNA ejecución de la declaración inclusive si la condición nunca se cumple. El siguiente ejemplo repite cualquier texto que

el usuario introduzca y lo hace hasta que el usuario escriba adios:

Código fuente 21:

Salida

```
// máquina repetidora
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Escriba un texto: ";
    getline (cin,str);
    cout << "Usted escribio: " << str << '\n';
  } while (str != "goodbye");
}
```

```
Escriba un texto: hola
Usted escribio: hola
Escriba un texto: como
esta?
Usted escribio: como esta?
Escriba un texto: adios
Usted escribio: adios
```

El loop do-while es preferido sobre el loop while cuando se necesita que la declaración sea ejecutada por lo menos una vez. En el ejemplo, lo que el usuario escribe es lo que determina si el loop continúa o sigue. Así que si el usuario desea terminar con el loop lo antes posible solo debe escribir adios, pero la declaración debe ser ejecutada mínimo una vez completa para poder detener el mismo loop.

2.2.4. Declaración de selección: switch

La sintaxis de la declaración switch es un poco particular. El propósito es revisar un valor dentro de un conjunto de posibles expresiones constantes. En algunas ocaciones puede parecerce al `if-else`, pero limitado a una expresión constante. La sintaxis típica es:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

Esto funciona de la siguiente manera: `switch` evalúa `expression` y revisa si es equivalente a `constant1`; si esto es así ejecuta `group-of-statements-1` hasta que éste encuentra el `break` . Cuando se encuentra su declaración `break`, el programa salta hasta el final del `switch` (hasta el corchete que cierra).

Si la expresión no fue igual a `constant1`, entonces es revisada con respecto a `constant2`. Si es igual a ésta entonces es ejecutado hasta que se encuentra el `break`, entonces se salta hasta el final del

`switch`.
Finalmente si el valor de la expresión no coincide con ninguna de las constantes especificadas (puede haber cualquier cantidad de éstas), el programa ejecuta la declaración incluida en bajo el rótulo `default:` si es que existe ya que éste es opcional.
Los dos fragmentos de código que se muestran a continuación tienen el mismo comportamiento mostrando que el if-else es equivalente a la declaración switch:

Ejemplo de switch	if-else equivalente

`switch (x) {`
`  case 1:`
`  cout << "x es 1";`
`  break;`
`  case 2:`
`  cout << "x es 2";`
`  break;`
`  default:`
`  cout << "valor de x desconocido";`
`  }`
	`if (x == 1) {`
	`  cout << "x es 1";`
	`}`
	`else if (x == 2) {`
	`  cout << "x es 2";`
	`}`
	`else {`
	`  cout << "valor de x desconocido";`
	`}`

La declaración `switch` tiene una sintaxis particular heredada de la primera era de los compiladores de C que usaban rótulos en lugar de bloques. El uso más común es el mostrado anteriormente donde la declaración `break` es usada después de cada grupo de declaraciones para un rótulo particular. Si `break` no es incluido, todas las declaraciones siguientes al caso son también ejecutadas, incluyendo las de otros rótulos hasta llegar al final del switch o hasta llegar a cualquier otra declaración de salto.

Si en el ejemplo anterior no hubiese break luego del primer caso (primer grupo), el programa no salta automáticamente al final del switch luego de imprimir "x es 1", sino que en lugar seguiría ejecutando las declaraciones del caso dos siguiente, esto es imprimiendo "x es 2" y hará esto hasta que encuentre un break o el final del switch. Esto hace inecesario el uso de corchetes pues se ejecuta línea tras línea y puede ser útil para ejecutar el mismo grupo de declaraciones para diferentes valores posibles. Por ejemplo:

```
1 switch (x) {
2   case 1:
3   case 2:
4   case 3:
5     cout << "x is 1, 2 or 3";
6     break;
7   default:
8     cout << "x is not 1, 2 nor 3";
9   }
```

Note que el `switch` es limitado para comparar el valor de una expresión contra rótulos que son expresiones constantes. No es posible usar variables o rangos como porque estos no son expresiones constantes válidas en C++.

Para revisar rangos o valores que no son constantes es mucho mejor usar declaraciones concatenadas

de `if` y `else if`.

2.3 Funciones

Las funciones permiten estructurar programas en segmentos de código para realizar tareas individuales.

En C++, una función es un grupo de declaraciones a las cuales se les da un nombre y pueden ser llamadas desde algún punto del programa. La sintaxis más común para definir una función es:

`type nombre ( parametro1, parametro2, ...) { declaraciones }`

donde:

`type` es el tipo de variable del valor que devuelve la función.

`nombre` es el identificador por el cual la función es llamada.

`parametros` (tantos como sean necesarios): cada parámetro consiste de un tipo de variable seguido por un identificador, cada parámetro es separado del siguiente por una coma. Cada parámetro luce mucho como una declaración normal de variable (por ejemplo: `int x`) y en efecto actúa dentro de la función como una variable normal la cual es local a la función. El propósito de los parámetros es permitir el paso de argumentos a la función desde el lugar donde ésta es llamada.

`declaraciones` es el cuerpo de la función. Es un bloque de declaraciones encerrados en corchetes { } que especifican lo que la función hace.

Ejemplo de funciones:

Código fuente 22:

```
1  // Ejemplo de función
2  #include <iostream>
3  using namespace std;
4
5  int addition (int a, int b)
6  {
7    int r;
8    r=a+b;
9    return r;
10 }
11
12 int main ()
13 {
14   int z;
15   z = addition (5,3);
16   cout << "El resultado es " <<
17 z;
   }
```

Salida:

```
El resultado es 8
```

Este programa se divide en dos funciones: `addition` y `main`. Recuerde que no importa el orden en que son definidas, un programa en C++ siempre comienza llamando a `main`. De hecho, `main` es la única función llamada automáticamente y el código escrito en cualquier otra función es ejecutado únicamente si su función es llamada desde `main` (directa o indirectamente).

En el ejemplo anterior, `main` comienza por la declaración de la variable `int z` , y justo después realiza el primer llamado de función: llama la función `addition`. El llamado a la función sigue una estructura muy similar a su declaración. En el ejemplo, el llamado a `addition` puede ser comparado con su definición justo unas lineas antes.

```
int addition (int a, int b)
```

Los parámetros en la declaración de la función tienen una clara correspondencia a los argumentos pasados en el llamado de la función. El llamado pasa dos valores, 5 y 3, a la función; éstos corresponden a los parámetros `a` y `b` declarados por la función `addition`.

En el punto en el cual la función es llamada desde main, el control se pasa a la función `addition`: aquí la ejecución de `main` para y se retoma sólo cuando la función `addition` termina. En el momento del llamado de la función los valores de ambos argumentos (5 y 3) se copian a las variables locales (`int a` y `int b`) dentro de la función.

Dentro de `addition`, otra variable local se declara (`int r`), y por medio de la expresión `r=a+b`, el resultado de la suma es asignado a r.

La declaración final dentro de la función es:

```
return r;
```

Esto finaliza la función `addition` y retorna el control al punto donde la función fue llamada, en este caso en la función `main`. En este justo momento, el programa continúa con su curso en `main` retornando exactamente en el mismo punto en el cual fue interrumpido por el llamado de `addition`. Pero adicionalmente, debido a que `addition` retorna un resultado, el cual es evaluado para tener un valor, y dicho valor es que es especificado por la declaración `return` al final de la función: en este ejemplo, es el valor de la variable local r, la cual en el momento de la declaración return tiene el valor 8.

```
int addition (int a, int b)
 8
z = addition (  5  ,  3  );
```

Por lo tanto, el llamado a `addition` es una expresión con el valor retornado por la función, y en este caso el valor de 8 es asignado a la variable z. Esto es como si el llamado de la función (`addition(5,3)`) fuera reemplazado por el valor que ésta retorna (es decir 8).

La función `main` simplemente imprime este valor llamando:

```
cout << "The result is " << z;
```

Una función puede ser llamada multiples veces dentro de un programa y sus argumentos pueden generalizarse:

Código fuente 23:

```
1 // Ejemplo de función
2 #include <iostream>
3 using namespace std;
```

Salida:

```
El primer resultado es 5
```

```
4  int substract (int a, int b)
5  {
6    int r;
7    r=a-b;
8    return r;
9  }
10
11 int main ()
12 {
13   int x=5, y=3, z;
14   z = substract (7,2);
15   cout << "El primer resultado es " << z << '\n';
16   cout << "El segundo resultado es " << substract (7,2) <<
17 '\n';
18   cout << "El tercer resultado es " << substract (x,y) <<
19 '\n';
20   z= 4 + substract (x,y);
21   cout << "El cuarto resultado es " << z << '\n';
   }
```

```
El segundo resultado es 5
El tercer resultado es 2
El cuarto resultado es 6
```

De manera similar a la función `addition` en el ejemplo anterior, este ejemplo define la función `substract`, que retorna la diferencia entre sus dos parámetros. En esta ocación `main` llama la función en varias ocaciones, mostrando diferentes maneras en las cuales una función puede ser llamada. (descripción detallada de la función `substract`)

2.3.1 Funciones sin tipo de dato. El uso de void

La sintaxis mostrada anteriormente para las funciones:

```
type name ( argument1, argument2 ...) { statements }
```

requiere que la declaración este iniciada con un tipo de variable. Este es el tipo del valor retornado por la función. Pero en el caso en que la función no retorna ningún valor, el tipo de dato usado es `void`, el cual es un tipo especial para representar la ausencia de tipo. Por ejemplo una función que sólo imprime un mensaje no necesita retornar ningún valor:

Código fuente 24: Salida:

```
1  // Ejemplo de función void
2  #include <iostream>
3  using namespace std;
4
5  void imprimir_mensaje ()
6  {
7    cout << "Soy una funcion!";
8  }
9
10 int main ()
11 {
12   imprimir_mensaje ();
13 }
```

```
Soy una funcion!
```

`void` también puede ser usada en la lista de los parámetros de una función para especificar

explicitamente que la función no toma parámetros cuando es llamada. Por ejemplo la función imprimir_mensaje puede declararse así:

```
1 void imprimir_mensaje (void)
2 {
3   cout << "Soy una función!";
4 }
```

En C++, una lista de parámetros vacía puede ser usada en lugar de void con el mismo significado, pero el uso de `void` en la lista de argumentos fue popularizada por el lenguaje C donde era indispensable.

Algo que es **obligatorio** es el uso de paréntesis () justo después del nombre de la función, inclusive si la función no tiene parámetros, allí un par de paréntesis vacios tienen que seguir al nombre de la función, como en el ejemplo anterior:

```
printmessage ();
```

Los paréntesis son los que diferencian las funciones de cualquier otro tipo de declaración.

2.3.2 El valor de retorno de la función `main`

Hasta ahora el tipo de dato que debe retornar `main` es `int`, pero la mayoría de los ejemplos anteriores realmente no hay ningún valor que retorne `main`.

Hay algo a tener en cuenta: si la ejecución de la función `main` termina normalmente sin encontrarse una declaración de `return`, el compilador asume que la función termina con una declaración de retorno implicita:

```
return 0;
```

Note que esto sólo aplica a la función `main` por razones históricas. Todas las otras funciones con un tipo de dato para retornar terminan con la declaración explicita `return` que devuelve el valor, inclusive si este nunca es usado.

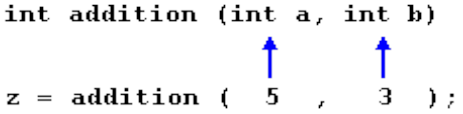
Cuando `main` retorna cero (implicitamente o explicitamente), es interpretado por el compilador como que el programa terminó con éxito. Otros valores pueden ser retornados por `main` pero sólo algunos valores son interpretados en todas las plataformas (o por todos los compiladores):

valor	descripción
0	El programa fue exitoso
`EXIT_SUCCESS`	El programa fue exitoso (igual que el anterior). Este valor es definido en el encabezado `<cstdlib>`.
`EXIT_FAILURE`	El programa ha fallado Este valor es definido en el encabezado `<cstdlib>`.

Debido a que la declaración de retorno implícito `return 0;` es una excepción con truco en la función `main`, algunos autores consideran una buena práctica escribir la declaración explicitamente siempre justo antes de terminar la función `main`.

2.3.3. Argumentos pasados por valor o por referencia

En las funciones vistas anteriormente, los argumentos han sido siempre pasados *por su valor*. Lo que significa, que cuando la función es llamada, lo que pasa a los argumentos de la función son copias de los valores que tienen las variables en el momento de ser llamada la función.



En ciertos casos puede ser útil tener acceso a una variable externa desde dentro de una función. Para hacer ésto, los argumentos deben ser pasados *por referencia* y no por valor. Por ejemplo en el siguiente código, la función duplicate en el código duplica el valor de sus tres argumentos, causando que las variables que son usadas como argumentos sean realmente modificadas en el llamado de la función:

Código fuente 25:

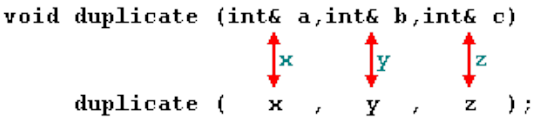
```
1  // Pasando parámetros por referencia
2  #include <iostream>
3  using namespace std;
4
5  void duplicate (int& a, int& b, int& c)
6  {
7    a*=2;
8    b*=2;
9    c*=2;
10 }
11
12 int main ()
13 {
14   int x=1, y=3, z=7;
15   duplicate (x, y, z);
16   cout << "x=" << x << ", y=" << y << ", z=" << z;
17   return 0;
18 }
```

Salida:

```
x=2, y=6, z=14
```

Para ganar acceso a estos argumentos, la función declara sus parámetros como *referencias*. En C++, las referencias son indicadas con un ampersand (&) justo después del tipo del parámetro, como en el ejemplo anterior.

Cuando una variable es pasada *por referencia*, lo que es pasa no es una copia, sino la variable en si misma. La variable identificada por el parámetro de la función biene a estar de alguna manera asociada con el arguemento pasado a la función y cualquier modificación en sus correspondientes variables locales dentro de la función son reflejadas en las variables pasadas como argumentos en la llamada.



En efecto `a`, `b` y `c` se convierten en alias (sobrenombres) de los argumentos pasados en el llamado de la función (x, y, z) y cualquier cambio en `a` dentro de la función modifica realmente a `x` fuera de la función.

Si en lugar de definir la función duplicate como se hizo en el ejemplo:

```
void duplicate (int& a, int& b, int& c)
```

Se hubieran definido los argumentos sin el signo de ampersand de la siguiente manera:

```
void duplicate (int a, int b, int c)
```

Las variables no se pasarían *por referencia*, sino *por valor*, creando copias de sus valores que entran a la función. En este caso, la salida del programa serían los valores de x, y, z sin ser modificadas, ,esto es 1, 3, y 7.

2.3.4. Recursividad

Es la propiedad que tienen las funciones de ser llamadas por ellas mismas. Esto es útil en alguas tareas tales como organizar elementos o calcular el factorial de números. Por ejemplo, para obtener el factorial de un número (n!) la formula matemática debe ser:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

Una función recursiva para calcular esto en C++ podría ser:

Código fuente 26:

```
1  // calculo de factorial
2  #include <iostream>
3  using namespace std;
4
5  long factorial (long a)
6  {
7    if (a > 1)
8     return (a * factorial (a-1));
9    else
10    return 1;
11 }
12
13 int main ()
14 {
15   long numero = 9;
16   cout << numero << "! = " << factorial (numero);
17   return 0;
18 }
```

Salida:

```
9! = 362880
```

Note como en la función factorial se incluye la función nuevamente, pero sólo si el argumento es mayor que 1, de otra manera la función entraría en un loop infinito en el cual luego que se llegue a cero se continuaría multiplicando por todos los números negativos provocando un desbordamiento en algún punto durante el proceso.

2.3.5. Funciones tipo Inline

Llamar una función puede causar sobrecostos en un cálculo, es por ésto que para funciones muy cortas, puede ser más eficiente simplemente incertar el código de la función en el sitio en que se llamó, en lugar de estar haciendo el proceso formal de llamar una función.

Cuando se precede la declaración de una función con `inline`, se informa al compilador que dicha

función debe ser preferida sobre la función llamada con el mecanismo usual, para una función específica. Ésto no cambia en nada el comportamiento de la función, es sólo una sugerencia al compilador que el código generado en el cuerpo de la función debe ser insertado en el momento que la función es llamada en lugar de ser invocado con un llamado regulr de la función.

Por ejemplo, la función `addition` del código fuente 22, puede ser escrita de manera abreviada como:

```
int addition (int a, int b)
{
  return a+b;
}
```

Pero también puede ser declarada como `inline` de manera:

```
inline int addition (int a, int b)
{
  return a+b;
}
```

Ésto informa al compilador que cuando `addition` es llamada, el programa prefiere expandir como en una línea de código la función en lugar de realizar el llamado usual. `inline` sólo se especifica en la declaración y no cuando se llama la función.

2.3.6. Valores por defecto en los parámetros

En C++, las funciones también pueden tener parámetros opcionales, para los cuales no se necesitan argumentos en el llamado. De tal menera, por ejemplo, una función con dos parámetros puede ser llamado sólo con uno. Para ésto, la función debe incluir un valor por defecto para su segundo parámetro, el cuál es usado por la función cuando ésta es llamada con pocos argumentos.

Código fuente 27:

```
// valores por defecto en funciones
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

```
6
5
```

En éste ejemplo, hay dos llamados a la función `divide`. El primer llamad es:

```
divide (12)
```

El llamado pasa sólo un argumento a la función a pesar que la función tiene dos parámetros. En éste

caso, la función asume que el segundo parámetro es 2 (el cual está declarado en la definición de la función como segundo parámetro `int b=2`). Por tanto el resultado es 6.

El segundo llamado es:

```
divide (20,4)
```

El llamado pasa dos argumentos a la función. Por tanto, el valor por defecto para `b` (`int b=2`) es ignorado, y `b` toma el valor pasado como argumento, el cual es 4 y ésto lleva al resultado de 5.

2.3.7. Declarando funciones

En C++, los idetificadores puede ser sólo usados en expresiones una vez éstos hayan sido declarados. Por ejemplo, una variable x no puede ser usada antes de ser declarada así:

```
int x;
```

Lo mismo apliaca para funciones. Las funciones no pueden ser llamadas antes de ser declaradas. Ésta es la razón por la cual en ejemplos anteriores las funciones fueron definidas antes de la función `main`, la cual es la función de donde finalmente son llamadas todas las otras. Si `main` fuera definida antes de las otras funciones, se rompería la regla que las funciones deben ser declaradas antes de ser usadas y por tanto el código no compila.

El prototipo de una función puede ser declarada sin una definición real y completa de la función, dando únicamente algunos detalles que le permiten a los tipos de datos involucrados ser reconocidos en el llamado de la función. Naturalmente la función debe ser definida en algún otro sitio, por ejemplo después de `main`. Pero declarada de ésta manera puede ser llamada.

La declaración debe incluir los tipos de datos de los argumentos y el tipo de dato que retorna la función. Se usa la misma sintaxis usada en la definición de una función, pero reemplazando el cuerpo de la función (el bloque de declaraciones) con un punto y coma para terminar.

La lista de parámetros no necesita incluir (pero si puede incluirlos si lo desea, es opcional) los nombre de los parámetros, sólo sus tipos de datos. Y los nombres no necesariamente deben ser los mismo que usa en la definición de la función. Por ejemplo, una función llamada `protofunction` con dos parámetros `int` puede ser declarada con cualquiera de las fos formas siguientes:

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

De todas maneras incluir nombres para cada parámetro da legibilidad a las declaraciones.

Código fuente 28:

```
1  // declarando funciones prototipo
2  #include <iostream>
3  using namespace std;
4
5  void odd (int x);
6  void even (int x);
7
8  int main()
9  {
10   int i;
11   do {
```

```
Please, enter number (0 to
exit): 9
It is odd.
Please, enter number (0 to
exit): 6
It is even.
Please, enter number (0 to
exit): 1030
It is even.
Please, enter number (0 to
exit): 0
```

```
12      cout << "Please, enter number (0 to exit): ";
13      cin >> i;
14      odd (i);
15    } while (i!=0);
16    return 0;
17  }
18
19  void odd (int x)
20  {
21    if ((x%2)!=0) cout << "It is odd.\n";
22    else even (x);
23  }
24
25  void even (int x)
26  {
27    if ((x%2)==0) cout << "It is even.\n";
28    else odd (x);
29  }
```

```
It is even.
```

Éste código, no es un ejemplo de eficiencia, quizá lo pueda escribir ud con muchas menos líneas. Sin embargo, éste ejemplo ilustra cómo las funciones pueden ser declaradas antes de ser definidas.

En las siguientes líneas:

```
1  void odd (int a);
2  void even (int a);
```

Se declaran las funciones prototipo. Ellas contienen todo lo necesario para llamarlas, sus nombres, los tipos de sus argumentos y los tipos de los valores que retornan (en éste caso son `void`). Con éstos prototipos de declaraciones antes del main, las funciones pueden ser llamadas antes de ser definidas del todo.

Pero declarar funciones antes de definirlas es útil no solo para reorganizar las funciones en el código. En algunos casos, como el presentado en código fuente 28, por lo menos una de las declaraciones es requerida ya que `odd` y `even` se llaman mutuamente. Hay un llamado a `even` en `odd` y un llamado a `odd` en `even`. Así que es necesrio definir `even` antes de `odd` y definir `odd` antes de `even` lo cual no es posible al tiempo. Por lo tanto no hay otra forma que declararlas ambas antes y definirlas después.

2.4 rand()

`rand()` es realmente un mal generador de números aleatoriosc hasta el punto que está a punto de ser reportado para ser dado de baja. Adicionalmente `rand()` se torna peor cuando se toma el módulo de éste (estadísticamente hablando). Para evitar éstos problemas se utiliza el nuevo generador de C++ de la librería números aleatorios de la siguiente manera:

```
//g++ -std=c++11 random.C -o random
//g++ -std=c++0x random.C -o random
//g++ -std=gnu++0x random.C -o random
```

```
#include <iostream>
#include <random>
#include <ctime>

int main() {
 const int size = 7;
 std::default_random_engine generator; //generador de números aleatorios
 generator.seed( time(0) );            //semilla del generador
 std::uniform_int_distribution<int> distribution(0,9);//note que el max y min
están aquí incluidos
 for(int i = 0; i < size; i++){
    std::cout << distribution(generator) << std::endl;
 }
 return 0;
}
```