Universidad Autónoma de Entre Ríos
Facultad de Ciencia y Tecnología
Sede: Oro Verde

FUNDAMENTOS DE PROGRAMACIÓN

RESUMEN DE CONTENIDOS N° 11

FUNCIONES

RESUMEN DE CONTENIDOS N° 11

Funciones

Introducción

Para hallar la solución de un problema complejo, es conveniente dividirlo en pequeños problemas más simples y buscar la solución de cada uno de ellos en forma independiente.
En el diseño de algoritmos computacionales y programas, esta subdivisión en segmentos o módulos - que llamaremos ***subprogramas*** - constituye una herramienta muy importante que permite modularizar problemas grandes o complejos.

Un ***subprograma*** es un conjunto de acciones, diseñado generalmente en forma separada y cuyo objetivo es resolver una parte del problema.

Estos ***subprogramas*** pueden ser invocados desde diferentes puntos de un mismo programa, desde otros ***subprogramas***, e incluso desde otros programas (a través de lo que se denomina ***"Librerías"***).

La finalidad de los ***subprogramas***, es simplificar el diseño, la codificación y la posterior depuración de los programas.

Las ventajas de usar subprogramas

- Reducir la complejidad del programa y lograr mayor modularidad.
- Permitir y facilitar el trabajo en equipo. Cada diseñador puede atacar diferentes módulos o subprogramas.
- Facilitar la prueba de un programa, ya que cada subprograma puede ser probado previamente y en forma independiente.
- Optimizar el uso y administración de memoria.
- Crear librerías de subprogramas para su posterior reutilización en otros programas.

Cuándo emplear subprogramas?

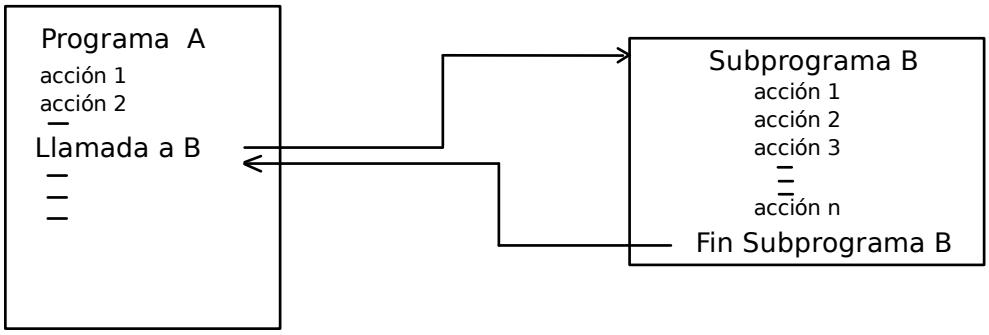
Es conveniente emplear subprogramas cuando:

- Existe un conjunto de operaciones que se utilizan más de una vez en un mismo programa.
- Existe un conjunto de operaciones útiles que pueden ser utilizadas por otros programas.
- Se desea agrupar procesos para lograr una mayor claridad en el código del programa.
- Se pretende crear bibliotecas que permitan lograr mayor productividad en el desarrollo de futuros programas.

Al plantear la solución a un problema que se quiere resolver, se diseña un programa al que se llamará ***programa principal*** **(main),** que incluirá entre sus acciones una sentencia especial que permitirá *llamar* al subprograma.

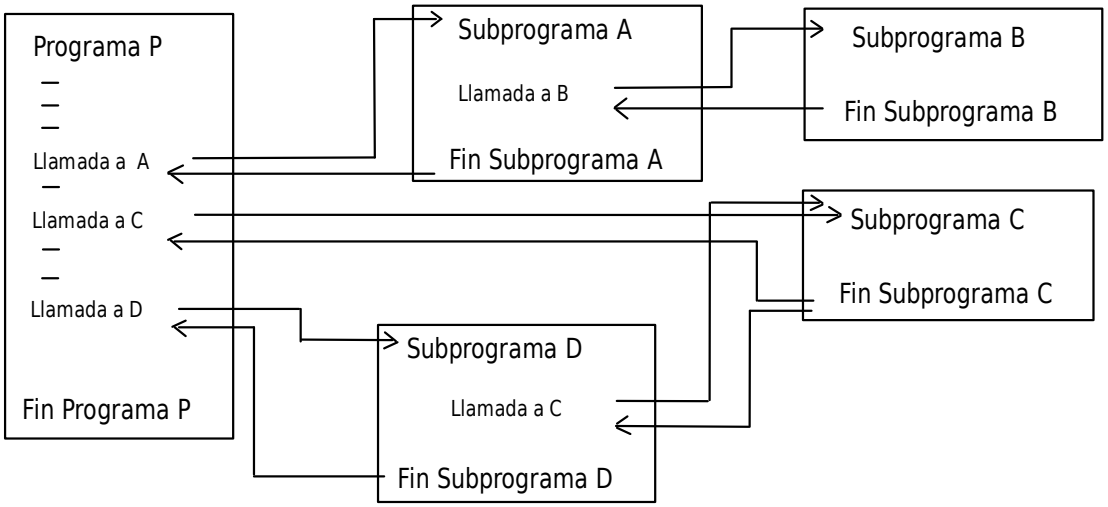
En la etapa de ejecución del programa, al encontrar la llamada al ***subprograma***, se transfiere el control de ejecución a éste y comienzan a ejecutarse las acciones previstas en él. Al finalizar la ejecución del ***subprograma*** y obtenidos los resultados planeados, el control retorna al programa que produjo la llamada, y continúa la ejecución del programa principal.

Se puede observar lo anterior gráficamente:



En el esquema, **A** se visualiza un programa que contiene una acción o *llamada* al ***subprograma*** **B**. Cuando el control de ejecución llega a la llamada, comienzan a ejecutarse las acciones descritas en **B**. Al finalizar **B**, el control vuelve al programa principal **A**, para continuar con las acciones restantes. Se dice que ***A*** es *cliente del subprograma* ***B.***

Este esquema simple: ***programa principal - subprograma****,* puede adquirir mayor complejidad con la existencia de otros ***subprogramas***. El control puede pasar del programa principal a cualquier subprograma, o de un subprograma a otro, pero siempre se retorna al lugar que produjo el llamado.



En el gráfico se ha representado el programa **P** que contiene 3 llamadas a subprogramas diferentes, **A**, **C**, **D**. A su vez, los subprogramas **A** y **D** son clientes de otros subprogramas: el subprograma **A** llama al **B**, y el **D** al **C**.

Durante la ejecución de **P**, se encuentra la acción de llamada a **A**, el control pasa a dicho subprograma y comienzan a ejecutarse las acciones que él describe hasta encontrar la llamada a **B**; en este momento, el control pasa al subprograma **B**, se ejecutan sus acciones y al finalizar éste retorna al punto desde donde fue llamado en el subprograma **A**. Luego se continúan ejecutando las acciones de A, y al finalizar, vuelve el control al punto de llamada en el programa principal **P**.

Continúa la ejecución de P hasta encontrar la llamada a **C**, pasando el control a este subprograma, se ejecutan sus acciones y retorna a **P** en el punto de llamada.
Continúa **P** hasta hallar la llamada a **D**; pasa el control a **D**, se ejecutan sus acciones hasta encontrar la invocación al subprograma **C**; comienzan a ejecutarse las acciones de **C**, y al terminar el control retorna a **D** en el mismo lugar donde se llamó a **C**. Continúa la ejecución de **D**, para luego retornar al programa principal **P**.

Observación: notar que el mismo subprograma **C** fue llamado desde el programa principal **P** y desde el subprograma **D**. En otras palabras: **P** y **D** son clientes del subprograma **C**.

Tipos de Subprogramas

Todos los lenguajes de programación admiten subprogramas. Se los denomina **funciones**, **procedimientos**, **subrutinas**. C++ emplea el subprograma **función**.

Funciones en C++

En C++ siempre se emplearán funciones. Todo programa C++ consta de una o más funciones y, una de ellas debe llamarse **main.**

La ejecución de un programa C++ comienza por las acciones planteadas en **main.**

Si un programa C++ contiene varias funciones estas pueden definirse en cualquier lugar del programa pero en forma independiente una de otra (no incluir una función en otra).

Como se indicó anteriormente, es posible acceder (llamar) a una función desde cualquier lugar del programa y hacerlo varias veces en un mismo programa. Al llamar a una función el control de ejecución pasa a la función y se llevan a cabo las acciones que la componen; luego el control retorna al punto de llamada.

Puede existir intercambio de información entre el programa o módulo que llama a la función y ésta. La función devuelve un solo valor (o ninguno) a través de la sentencia **return** y se verá más adelante que puede recibir y devolver información a través de sus parámetros o argumentos.

```
//Ejemplo: calcular el promedio entre 3 valores
//enteros que se ingresan como datos de entrada.

#include <iostream>
#include<iomanip>
using namespace std;

float promedio3(int x,int y,int z);

int main( )
    { int d1, d2, d3;
      cout <<"Ingrese el primer dato:" ; cin >> d1;
      cout <<"Ingrese el segundo dato:"; cin >> d2;
      cout <<"Ingrese el tercer dato:" ; cin >> d3;
      float p = promedio3(d1, d2, d3);
      cout <<setprecision(3)<<"El promedio es:" << p;
      return 0;
    }

float promedio3(int x,int y,int z)
    { float w=(x+y+z)/3.0 ;
      return(w);
     }
```

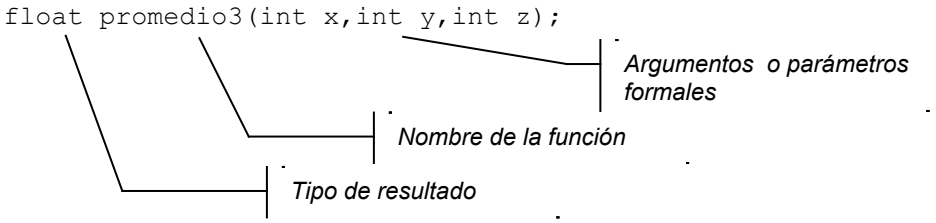
Prototipo de la función ***promedio3***

Llamada a la función ***promedio3***

Definición de función ***promedio3***

Declarar y definir funciones en C++.

C++ exige declarar una función antes de que sea utilizada. Para ello se debe escribir la cabecera de la función; indicando el tipo de resultado que devuelve la función, su nombre y sus argumentos. Esta cabecera recibe el nombre de **prototipo** de la función. Se puede tomar como ejemplo la función *promedio3* empleada en el recuadro anterior.



Usualmente se escribe el prototipo de la función antes de **main{ }** pero se debe recordar que en C++ es posible efectuar la declaración de un elemento en cualquier lugar del programa con la condición de hacerlo antes de invocar dicho elemento.

El esquema general de la *definición de una* función es el siguiente

TipoResultado *nombreFunción*(*parámetros)*
{
Declaración de Variables de la función

Secuencia de sentencias
*(sentencia "**return**" para devolver el resultado)*
}

Puede haber situaciones en las que la función cuente con más de una sentencia ***return***. Esto es posible, siempre y cuando las mismas se encuentren en una estructura condicional, donde sólo una de ellas se ejecute en un determinado momento (véase más adelante el ejemplo del cálculo del factorial que utiliza una función recursiva).

Resultados de una función C++

Si una función devuelve un resultado, se debe especificar su tipo antes del nombre o identificador de la función; y en el cuerpo, se debe emplear una variable o expresión de igual tipo como argumento de la sentencia **return**().

```
float promedio3(int x,int y,int z)
    { float w=(x+y+z)/3.0 ;
      return w;
    }
```

Observar en el ejemplo que el tipo de la función **promedio3** y el tipo de la variable **w** que será retornada coinciden. También puede plantearse de la siguiente manera:

```
float promedio3(int x,int y,int z)
   {
    return((x+y+z)/3.0) ;
   }
```

En el ejemplo anterior, no es necesario definir a la variable **w**, puesto que la acción **return()** admite variables, constantes y expresiones, siempre que el resultado de las mismas coincida con el tipo de la función (en este caso, **float**).

Es posible que una función no devuelva resultados; entonces se especifica el tipo nulo **void** en su prototipo y en la correspondiente definición.

```
void promedio3(int x,int y,int z)
    { float w=(x+y+z)/3.0 ;
      cout << "el promedio es:" << w << endl;
     }
```

Una función puede no devolver valores si se la define de tipo nulo (void). En este caso, la función no incluye la sentencia **return**. Para funciones no nulas, **return** debe emplearse en el cuerpo de la función para devolver un resultado del mismo tipo que la función.

Intercambio de información desde funciones C++

El empleo de funciones permite diseñar rutinas que pueden ser reutilizadas dentro del mismo programa o desde otros programas.

A menudo es necesario enviar información a la función desde el punto de llamada para que complete la tarea asignada. Esto se hace a través de sus **parámetros o argumentos**.

En el prototipo y en la definición de la función se plantean los **parámetros formales o de diseño** y, cuando se invoca a la función, se utilizan **parámetros actuales o de llamada**. Si una función no requiere parámetros de entrada se la define con el tipo void entre paréntesis (o directamente con paréntesis vacíos) y se la invoca con paréntesis vacíos y sin asignarla a ninguna variable.

```
void funcion_nula( );

void main()
{.....
 funcion_nula( );
 .....
}

void funcion_nula( )
{    .....    }
```

Función sin parámetros de resultado

Llamada a la función

El pasaje o intercambio de información entre parámetros puede hacerse por **valor** o por **referencia**.

Pasaje de parámetros por valor

En este caso, al producirse la llamada a la función los **parámetros formales** son asignados en forma correspondiente (en posición y tipo) con los valores contenidos en los **parámetros actuales** o de llamada.

```
.....
int main( )
{ .....
  float p = promedio3(d1, d2, d3);
  cout << "Datos:"<<d1<<" "<<d2<<" "<<d3;
  cout << "Promedio:"<<p;
  .....
  return 0;}


float promedio3(int x,int y,int z)
  { float w=(x+y+z)/3.0 ;
    return w; }
```

d1, d2, d3: parámetros actuales o de llamada

x, y , z: parámetros formales. Son asignados en la llamada: x=d1, y=d2, z=d3

Al finalizar las acciones de la función **promedio3()** se devuelve el control a la función principal **main()** retornándose el valor obtenido en **w**. Si en el ejemplo anterior los datos asignados a d1, d2, d3 son 10, 20, 45, la salida a través de los flujos **cout** será:

Datos: 10 20 45
Promedio: 25.00

El número de parámetros formales debe coincidir en **cantidad** y **tipo** con el número de parámetros actuales empleados en la llamada (excepto en el caso de parámetros por defecto que se verá más adelante).

Pasaje de parámetros por referencia

La referencia consiste en utilizar como parámetro formal una referencia a la posición de memoria del parámetro actual o de llamada. Esto puede hacerse a través del operador referencia **&** o empleando punteros. Con el operador referencia es posible definir alias de una variable y emplear los parámetros actuales o de llamada para obtener resultados.

C++ emplea el operador **&** para realizar esta referencia. Observar el siguiente ejemplo:

```
int  m=10;
int  &q = m;  // q es definido como alias de
q++;          // se incrementa q en 1 y también
cout << m; // se obtiene 11 como salida
```

Posiciones de Memoria

	m / q	
	10	

Es decir que la expresión **&q** permite hacer referencia a la variable **m** a través de otro nombre (**q** es el alias de **m**). Se verá qué ocurre en el ejemplo del cálculo del promedio al usar alias para pasaje por referencia:

```
.....
void promedio3(int x,int y,int z,float &p)

int main( )
{
int d1,d2,d3; float prom;
.....
promedio3(d1, d2, d3, prom);
cout << "Datos:"<<d1<<" "<<d2<<" "<<d3;
 cout << "Promedio:"<<prom;
.....
```

Parámetros actuales o de llamada

```
 return 0;
}

void promedio3(int x, int y, int z, float &p)
    { p=(x+y+z)/3.0 ; }
```

*El parámetro formal **p** es un alias de **prom***

*Modificación del parámetro formal **p** y consecuente modificación de **prom** en **main***

Ahora, la información obtenida a través de la variable **prom** en **main,** se obtuvo a través de **p** en la función. Como **p** es una referencia de **prom** (comparten la misma posición de memoria) al modificar **p** en la función, se modifica automáticamente **prom.**

El uso de alias a través del operador **&** permite a una función obtener otros resultados además del correspondiente a la sentencia **return( ).**

Parámetros por defecto

Es posible proponer, en el prototipo de la función, parámetros formales inicializados con valores. Estos valores serán asumidos por defecto en el cuerpo de la función **si no se indican parámetros actuales** para tales argumentos.

```
float promedio3(int x,int y,int z=10)
.....
void main( )
  {
     .....
     float p=promedio3(d1, d2);
     .....
     float q=promedio3(d1,d2,d3);
  }
```

Parámetro formal con valor 10 por defecto

Llamada a la función con solo 2 parámetros actuales (el tercero se asumirá por defecto)

Llamada a la función con 3 parámetros actuales. Aquí no se empleará el valor por defecto en la función.

La única restricción sintáctica para estos parámetros por defecto, es el hecho de que deben figurar al final (a la derecha) de la lista de parámetros formales.
De acuerdo a esto último, el siguiente prototipo de función C++ sería causa de error en una compilación:

```
float promedio3(int x,int y=5,int z) //ERROR!!
```

En el lenguaje C++ la función de tipo nulo (que devuelve void) es lo que en otros lenguajes se conoce como procedimiento.

Sobrecarga de Funciones

C++ admite que dos funciones diferentes puedan tener el mismo nombre. Para que el compilador pueda distinguirlas es necesario que difieran sus parámetros. Esto significa que se puede emplear el mismo identificador en dos o más funciones si estas funciones tienen distinta cantidad de parámetros o diferentes tipos de parámetros. Por ejemplo,

```
// Ejemplo de sobrecarga de funciones
```

```
#include <iostream>
using namespace std;

int dividir (int a, int b);

float dividir (float a, float b);

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << dividir (x,y);
  cout << "\n";
  cout << dividir (n,m);
  return 0;
}

int dividir (int a, int b)
{  return (a/b);  }

float dividir (float a, float b)
{  return (a/b); }
```

La salida del programa será:

2
2.5

En este caso se han definido dos funciones con el mismo nombre, pero una de ellas acepta parámetros de tipo **int** y la otra acepta parámetros de tipo **float**. El compilador *sabe* cual debe emplear pues los tipos de parámetros son considerados previamente.
Por simplicidad, ambas funciones tienen el mismo código, pero esto no es estrictamente necesario. Se pueden hacer dos funciones con el mismo nombre pero con comportamientos totalmente diferentes.

Observar un caso erróneo de aplicación de sobrecarga en funciones:

```
// Ejemplo erróneo de sobrecarga
int dividir(int x, int y);

float dividir(int a, int b);
```

En la primera función se propone una división entera y en la segunda una división entre enteros que arroja un flotante, pero ambas funciones están sobrecargadas y con el mismo tipo de parámetros. Cuando el programa cliente invoque a `division()` no podrá discernir a cuál de las 2 funciones se refiere la llamada.

Operaciones de entrada y salida en funciones

No es conveniente emplear operaciones de entrada y salida interactiva en funciones. Es mejor operar a través de parámetros y que la entrada y salida la realice el cliente de la función.

Esto es para independizar la función del tipo de entorno en que se ejecutará el programa cliente que la utilizará. Por ejemplo: si en una función se incluyen operaciones de salida empleando el modo consola en C++ a través del objeto **cout**, no se podrá emplear esta función en un programa C++ que opere en un entorno gráfico (como Windows), donde la

entrada y salida se realizan a través de componentes visuales situados en formularios (ventanas). En cambio, no representa un inconveniente utilizar entrada y salida a dispositivos que almacenan archivos.

Observar en el siguiente ejemplo la diferencia entre una función que realiza una salida y otra que sólo devuelve el resultado.

La función **volumen_cilindro1()** calcula el volumen de un cilindro y produce una salida con ese resultado

```
void volumen_cilindro1(float radio, float altura)
{
   float vol;
   vol= 3.14*radio*radio*altura;
   cout<<"El volumen del cilindro es:"<<vol;
}
```

La función **volumen_cilindro2()** sólo devuelve el resultado obtenido al cliente que invoque la función.

```
float volumen_cilindro2(float radio, float altura)
{
   float vol;
   vol= 3.14*radio*radio*altura;
   return vol;
}
```

La función **volumen_cilindro2()** puede ser reutilizada en programas cuya entrada y salida se realice a través de componentes visuales de un entorno gráfico. La función **volumen_cilindro1()** sólo puede emplearse en programas C++ que operen en modo consola.

Recursividad

La recursividad es una técnica que permite definir una función en términos de sí misma. En otras palabras: una función es recursiva cuando se invoca a sí misma.

C++ admite el uso de funciones recursivas; cualquier función C++ puede incluir en su código una invocación a sí misma, a excepción de **main().**

Para su implementación los compiladores utilizan una pila (stack) de memoria temporal, la cual puede causar una interrupción del programa si se sobrepasa su capacidad (stack overflow).

Como ventaja de esta técnica se puede destacar que permite en algunos casos resolver elegantemente algoritmos complejos. Como desventaja, los procedimientos recursivos son menos eficientes –en términos de velocidad de ejecución- que los no recursivos.

Los algoritmos recursivos surgen naturalmente de muchas definiciones que se plantean conceptualmente como recursivas. Observar el caso del factorial de un número: por definición es el producto de dicho número por todos los factores consecutivos y decrecientes a partir de ese número, hasta la unidad:

n!= n*.(n-1)*(n-2)*…*2*1.
Por ejemplo: 5!=5*4*3*2*1= 120
Pero el producto 4*3*2*1 es 4!

Por lo tanto podemos escribir: 5!= 5* 4!

Como se observa en la línea anterior, se ha expresado el factorial en función de sí mismo. Es exactamente lo que se puede plantear algorítmicamente usando funciones C++. La solución recursiva del factorial de un número puede expresarse de la siguiente forma:

```
long factorial(unsigned int x)
{
   if (x==0)
         return 1;
             else
          return x*factorial(x-1);
}
```

Observar que en la función recursiva existe una condición (x==0) que permite abandonar el proceso recursivo cuando la expresión relacional arroje verdadero; de otro modo el proceso sería infinito.

Condiciones para que una función sea recursiva

Toda función recursiva debe.

1. Realizar llamadas a sí misma para efectuar versiones reducidas de la misma tarea.
2. Incluir uno o más casos donde la función realice su tarea sin emplear una llamada recursiva, permitiendo detener la secuencia de llamadas (condición de detención o stop)

Analizando el ejemplo del factorial de un número, se puede ver que la expresión **`x*factorial(x-1)`** corresponde al requisito 1 y la expresión **`x==0`** al segundo requisito.

Arreglos como parámetros de funciones

Es posible utilizar arreglos como parámetros de funciones. En C++ no es posible pasar por valor un bloque de memoria completo como parámetro a una función, aún si está ordenado como un arreglo, pero está permitido pasar su dirección de memoria, lo cuál es mucho más rápido y eficiente. Al pasar una dirección de memoria se está efectuando un *"pasaje por referencia".*
Para admitir arreglos lineales como parámetro de una función, lo único que se debe hacer al declarar la función es especificar en el argumento el tipo de dato que contiene el arreglo, un identificador y un par de corchetes vacíos **[ ]**. Por ejemplo, observar la siguiente función:

```
void  leer_arreglo(int arg[])
```

La función **leer_arreglo( )** admite un parámetro llamado **arg** que es un arreglo de enteros (int). ¿Dónde está el pasaje por referencia o la dirección de memoria del inicio del arreglo? La respuesta es: en el nombre del arreglo.

El nombre del arreglo identifica a una variable que contiene la dirección de memoria donde comienza el bloque donde se aloja el arreglo. En otras palabras: el nombre del arreglo es una variable que contiene la dirección de memoria del elemento 0 del arreglo.

Para llamar a la función **leer_arreglo()** del ejemplo anterior, deberá utilizarse como parámetro actual un arreglo; para ello es suficiente utilizar el identificador del arreglo:

```
int miarreglo [30];
leer_arreglo(miarreglo);
```

En el siguiente ejemplo se invoca 2 veces a la función **muestra_arreglo( )** pasándole en cada caso un arreglo y su longitud.

```
// Ejemplo: arreglos como parámetros
//-------------------------------------------------------------
#include <iostream>
#include <iomanip>
using namespace std;

void muestra_arreglo (int lista[], int largo);

int main ()
        {
                int v1[] = {35, 41, 22};
                int v2[] = {12, 4, 6, 15, 10};
                muestra_arreglo(v1,3);
                muestra_arreglo(v2,5);
                return 0;
        }

void muestra_arreglo (int lista[], int largo)
{
        for (int i=0; i<largo; i++)
                cout<<setw(4)<<lista[i];
        cout<<endl;
}
```

La salida del programa será:

```
35  41  22
12   4   6  15  10
```

Observar que en el ejemplo se invoca dos veces a la función **muestra_arreglo()** empleando como argumentos arreglos de diferente dimensión.

Arreglos multidimesionales

Si se trata de pasar como parámetro un arreglo de más de una dimensión, el parámetro formal debe tener los corchetes correspondientes al primer índice vacíos, pero establecer explícitamente las otras dimensiones.

```
void  leer_tabla(int t[][10], int num_filas);
```

El modificador const y parámetros de tipo arreglo

Se ha visto que al pasar un arreglo como parámetro, se pasa la dirección de memoria del inicio del arreglo y por lo tanto, se está efectuando un pasaje de parámetros por referencia. Esto implica que cualquier modificación efectuada en uno o más elementos del arreglo durante la ejecución de una función, se producirá la automática modificación del arreglo utilizado como parámetro actual o de llamada (se trata en realidad de un único arreglo pues se está trabajando sobre esa única dirección de memoria).

Para evitar que un arreglo sea modificado al pasarlo como parámetro, debe utilizarse la etiqueta *const* precediendo al parámetro formal en el prototipo de la función.
Observar las dos funciones siguientes:

```
float permite_cambiar(int lista[], int n);
{ ......
  lista[3]= 255;  //modifica el parámetro de llamada
  .....
}

float no_permite_cambiar(const int lista[], int n);
{ ......
  lista[10]= 320;  //error de compilación
  .....
}
```

En la primera función, el cambio efectuado en **lista[3]**se reflejará en el arreglo que se utilice como parámetro para llamar a esta función. La segunda función producirá un error de compilación, pues se establece con **const** que la posición de memoria a la que se quiere acceder es de sólo lectura y no puede modificarse el valor allí alojado.

Variables Locales y globales

Las variables declaradas dentro de una función (incluida la función main) se denominan **variables locales** a esa función. Los parámetros formales, también son variables locales.
Una variable local sólo puede ser accedida dentro de la función que la declara y a partir del punto donde se declara.

Una variable local se crea cuando la función es llamada y se destruye cuando la ejecución de la misma termina.

C++ permite también declarar variables fuera de cualquier función. Estas variables se denominan **variables globales** y son accesibles por cualquier función que aparezca a partir de su declaración y hasta el final del programa.

Síntesis

1. Los subprogramas permiten modularizar el diseño de programas, con las ventajas de posibilitar la reutilización de código, facilitar el trabajo en equipo, agilizar la depuración de errores y optimizar la administración de recursos en una computadora.

2. En C++ los subprogamas se expresan a través de funciones. Pueden incluirse en el programa cliente que las empleará o en archivos separados que funcionan como bibliotecas de funciones.

3. El prototipo de una función C++ incluye el tipo de resultado que devuelve, el nombre de la función y entre paréntesis la lista de parámetros o argumentos declarados. El código de la función tiene la estructura de un programa, sólo que se programa separadamente.

4. Los programas intercambian información con las funciones a través de los parámetros. Los parámetros de diseño de una función se llaman *parámetros formales*. Al llamar a una función se utilizan los *parámetros actuales* o de llamada. El número de parámetros

formales debe coincidir en cantidad y tipo con el número de parámetros actuales empleados en la llamada (excepto en el caso de parámetros por defecto).

5. El pasaje de información a través de parámetros puede realizarse por valor o por referencia. *Por valor*: asignando los valores de los parámetros actuales a los formales. *Por referencia*: definiendo alias de los parámetros actuales con el operador &, el cual debe preceder a los parámetros formales en el prototipo de la función. El pasaje por referencia permite realizar cálculos y asignarlos a los parámetros formales que se verán reflejados en los parámetros actuales.

6. Las variables locales declaradas dentro de la función y los parámetros formales de la misma, sólo son reconocidos dentro del ámbito de la función. Al completar la ejecución de una función, estos elementos liberan recursos y pierden toda la información que guardaban.

7. Una función puede no devolver valores si se la define de tipo nulo (void). En este caso, la función no incluye la sentencia return. Para funciones no nulas, return debe emplearse en el cuerpo de la función para devolver un resultado del mismo tipo que la función.

8. Se recomienda no realizar operaciones de entrada/salida dentro del código de las funciones. De este modo la función puede ser reutilizada en diferentes entornos y no depende la de la interfaz de usuario empleada.

9. Dos o más funciones pueden en C++ tener igual nombre y realizar distintas tareas. Para ello deben operar con diferentes parámetros. Esta propiedad se denomina sobrecarga de funciones. Al llamar a una función sobrecargada, el compilador C++ reconoce cuál es la que debe ejecutar en base a los parámetros utilizados en la llamada.

10. C++ admite que una función incluya en su código una o más llamadas a sí misma para efectuar versiones reducidas de la misma tarea. En ese caso se dice que la función es recursiva. Una función recursiva, además de incluir la/s llamada/s a sí misma, debe proponer en su código uno o más casos donde la función realice su tarea sin emplear una llamada recursiva, permitiendo detener la secuencia de llamadas (condición de detención o parada).

11. Al pasar arreglos como parámetros de funciones, la relación entre parámetro formal-parámetro actual es similar al pasaje por referencia. Para evitar cambios en el arreglo enviado en la llamada a una función se debe preceder con el modificador **const** a la variable arreglo establecida como parámetro formal en el prototipo de la función.