

SUBPROGRAMAS

FUNCIONES y

PROCEDIMIENTOS

Beneficios de usar Funciones

- ❑ **Permitir la programación modular:** divide el problema en partes más simples
- ❑ **Hacer el programa más legible:** permite entender mejor el código al agrupar varias sentencias bajo un nombre de función, el código queda más organizado, posibilita encontrar más rápido los errores
- ❑ **Reuso de código:** permite ganar tiempo al no tener que escribir las funciones reiteradamente, evita la repetición de código y reduce el tamaño del programa
- ❑ **Trabajar con librerías:** permite hacer uso de las funciones que ya están codificadas, reduce el tiempo de codificación
- ❑ **Mejor uso de la memoria:** al no tener que reservar espacio para una misma función reiteradas veces en un programa

Diseño Modular

- Construir un programa mediante pequeñas piezas o componentes
 - Esas piezas más pequeñas son llamadas **módulos**
- Cada pieza es más manejable que el programa original

Módulos o Subprogramas

- **Subprogramas**

- **Modularizan los programas**

- Las variables definidas dentro de una función son **variables locales**

- Conocidas solamente dentro de la función definida

- **Parámetros**

- Información comunicada entre funciones

- Variables locales

Subprogramas

- Un Subprograma es un mini programa **englobado dentro de otro** programa
- Su objetivo es **agrupar una serie de sentencias** que realizan una tarea concreta, bajo un nombre
- Permite que las tareas que puedan ser repetidas dentro de un programa **se implementen una sola vez** y se llamen tantas veces como haga falta
- Producen **ahorro** de espacio y **aumento** en la legibilidad del código
- Se recomiendan incluso cuando se llamen solo una vez, ya que **aumentan** la abstracción y **facilitan** el mantenimiento de programas

Tipos de Subprogramas

1) Funciones:

Encargadas de realizar un cálculo computacional y generar un resultado único.

Ejemplos:

Menor (a,b,c)

Factorial (n)

CantDigitos (numero)

2) Procedimientos:

Encargados de resolver un problema computacional general.

Ejemplos:

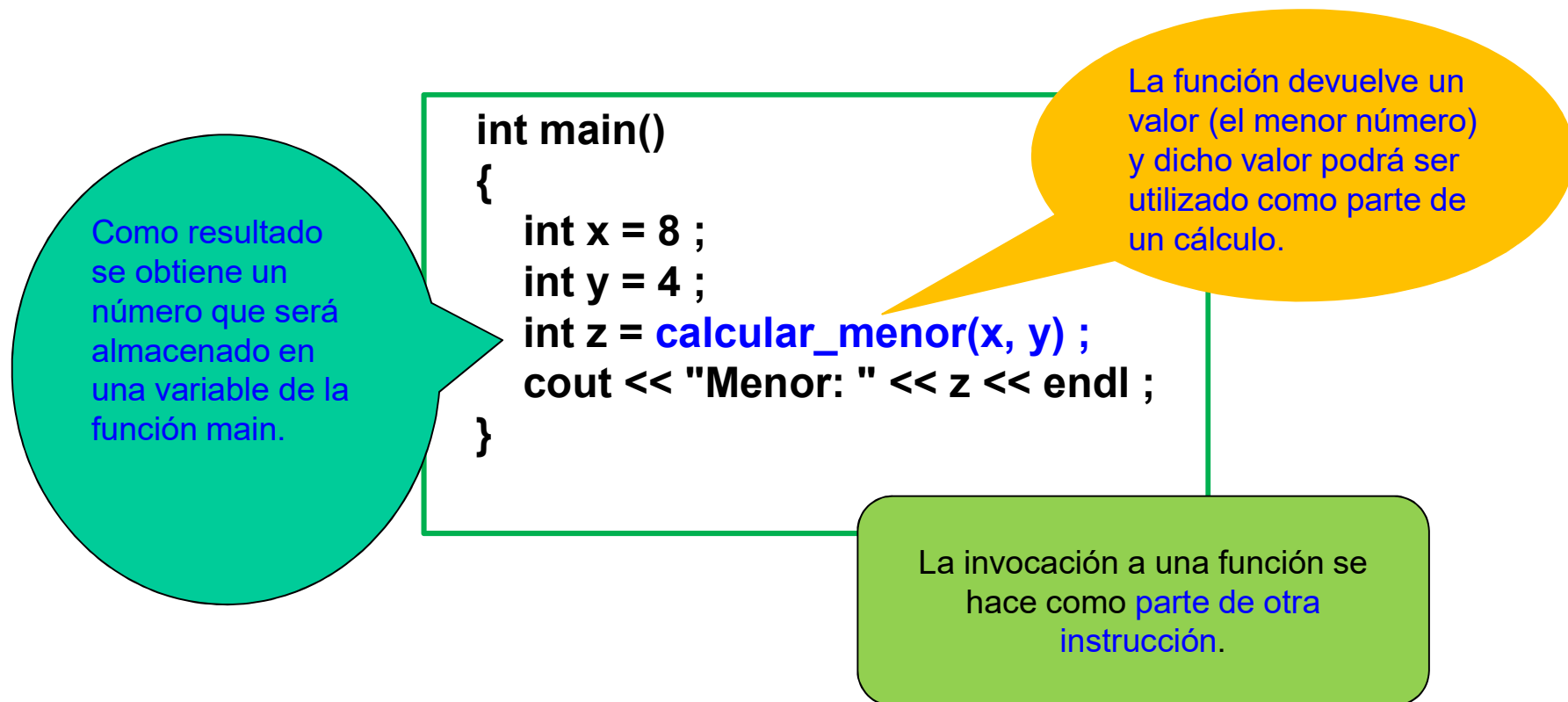
Limpiar pantalla

PonerFilaEn0 (A,n,i)

Saludar (frase)

Tipos de Subprogramas: 1) Funciones

En este ejemplo, la función *calcular_menor* recibe dos números como parámetros y calcula el menor de ellos.



La información se intercambia no solo a través de los parámetros sino también mediante el valor devuelto por la función.

Estructura de una Función en C++

- Una función es un conjunto de sentencias que se puede llamar desde cualquier parte del programa.
- Es un fragmento de programa parametrizado, que efectúa unos cálculos y:
 - Devuelve un valor como resultado o,
 - Tiene efectos laterales (modificación de variables globales o argumentos, volcado de información en pantalla, etc.) o,
 - Ambas cosas.
- Para trabajar con funciones se requiere:
 - Prototipo
 - Definición
 - Llamada (activación, ejecución)

Prototipos de Funciones

- Se usa para validar funciones
 - Nombre de la función
 - Parámetros que la función recibe
 - Tipo de retorno: tipo de dato que la función retorna
 - Termina con un punto y coma
- El prototipo sólo se necesita si la definición de la función viene después de su uso en el programa
- Ejemplo: Prototipo de una función *max*:
`int max(int x, int y, int z);`
 - Recibe 3 **enteros**
 - Retorna un **entro**

Prototipos de Funciones

```
#include <stdio.h>
```

```
int cuadrado (int x);  
int sumar (int a, int b);  
float entrada (void);
```

```
int main()  
{  
    .....  
    return 0;  
}
```

Prototipos de funciones


Estructura de una Función

- **Formato de definición de funciones**

```
tipo-de-retorno nombre-de-funcion ( lista-parametros )  
{  
    declaraciones y sentencias;  
}
```

- **Nombre de función:** cualquier identificador disponible
- **Tipo de retorno:** tipo de dato devuelto como resultado
 - **void** – indica que la función no retorna nada (**es un procedimiento**)
- **Lista de parámetros:** parámetros declarados, separados por coma.
 - Para cada parámetro, debe indicarse un tipo explícitamente y un nombre.
 - Para indicar que una función no tiene argumentos se usa paréntesis vacíos **()**.

Estructura de una Función

tipo-de-retorno nombre-de-funcion (lista-parametros)
{
declaraciones y sentencias;  **Cuerpo**
}

→ El **cuerpo** del subprograma especifica la secuencia de acciones necesarias para resolver el subproblema especificado

- ☐ Podrán definirse tantas variables locales como se necesiten.
- ☐ En el caso de una función, el valor que devuelva será indicado en el cuerpo con una expresión en la sentencia **return**.

→ **Resultado** de una función: **Se retorna con la sentencia *return***

- **return (expresión);**
- **return expresion;**
- **return;**

Llamada de Funciones

- Las funciones para ejecutarse, necesitan ser llamadas o invocadas.
- Cuando una función es llamada:
 - ✓ Se ejecuta desde el principio, y
 - ✓ Cuando termina retorna el control a la función que la llamó.

Composición de Funciones

Ejemplo: Se dispone de la función `maxim2` que obtiene el máximo entre 2 valores. Emplearla en un programa para calcular el máximo entre 4 valores enteros `a`, `b`, `c` y `d`.

```
int maxim2(int x, int y);

int main() {
    int a, b, c, d, w, z;
    cout << "Ingrese dos enteros: ";
    cin >> a >> b >> c >> d;
    w = maxim2(maxim2(a, b), maxim2(c, d));
    cout << endl << "El maximo es: " << w;
    z = maxim2(a, maxim2(b, maxim2(c, d)));
    cout << endl << "El maximo es: " << z;
    return 0;
}

int maxim2(int x, int y) {
    int max = x;
    if (y > max)
        max = y;
    return max;
}
```

```
Ingrese dos enteros: 12 7 23 4
El maximo es: 23
El maximo es: 23

<< El programa ha finalizado: codigo de salida:
<< Presione enter para cerrar esta ventana >>
```

Llamadas
anidadas

`maxim2 (maxim2(A, B), maxim2(C,D))` o bien,
`maxim2 (A, maxim2(B, maxim2(C,D)))`

Ejercicio 1

Escribir un **programa** que reciba un valor entero positivo *numero* y, mediante una función *cantDigitos()*, obtenga la cantidad de dígitos que posee el valor ingresado.

El programa deberá informar dicha cantidad.

Referencias

- ❑ Son una manera de **bautizar con otro nombre** a una determinada variable.
- ❑ Nos permiten referirnos a una variable con otro identificador, es decir, sin tener que copiarla.

- ❑ Ejemplo:

`int x = 10;` // declara una variable entera x con valor 10

`int & z = x;` // declara z como referencia a x

- ✓ Toda modificación que se haga a **z** se hará también a **x**.


Por ejemplo, si hacemos:

`z = 30;`

`cout << x;`

Se mostrará 30 por pantalla, pues se asignó 30 a x (a través de z).

```
int main() {  
    int x = 10;  
    int & z = x;  
  
    cout << x << endl;  
    cout << z << endl;  
  
    z = 30;  
    cout << x << endl;  
    cout << z << endl;  
  
    return 0;  
}
```



Tipos de Subprogramas: 2) Procedimientos

En este ejemplo se invoca al procedimiento *ordenar* para conseguir que, como resultado, el menor de dos números quede almacenado en una variable x y el mayor en una variable y.

```
int main()
{
    int x = 8 ;
    int y = 4 ;
    ordenar(x, y) ;
    cout << x << " " << y << endl ;
}
```

La función *main* se comunica con el procedimiento *ordenar* mediante una **llamada**.

Hay un intercambio de información entre *main* y *ordenar* a través de los parámetros x e y.

La invocación a un **procedimiento** es una **instrucción independiente**.

Procedimientos y Funciones

→ Tanto los procedimientos como las funciones se utilizan para **realizar un cálculo** y simplifican el diseño del programa principal.

→ En ambos casos **hay una invocación al subprograma** correspondiente y un **intercambio de información** entre el que llama y el subprograma llamado.

→ La única diferencia entre ambos tipos de subprogramas está en la forma de hacer **las llamadas**.

Procedimientos y Funciones

En C++ tanto las funciones como los procedimientos se definen todos como **funciones**, y se distinguirán *según tengan retorno o no*:

- **Funciones** → tienen un tipo de retorno
- **Procedimientos** → tienen retorno *void*

Parámetros de una Función

- El intercambio de información en la llamada a subprogramas no difiere del intercambio de información en una comunicación general entre dos entidades. La información puede fluir en uno u otro sentido, o en ambos. Dicho intercambio de información se realiza a través de los **parámetros**.
- Dependiendo del contexto en que se utilicen, se habla de **parámetros formales** (los que aparecen en la definición del subprograma), o de **parámetros actuales** (o reales, los que aparecen en la llamada al subprograma)
- El intercambio de información se analiza desde el punto de vista del subprograma llamado. Se tienen parámetros:
 - **de entrada**
 - **de salida**
 - **de entrada/salida.**

Parámetros de Entrada

- ➔ Son aquellos que se utilizan para **recibir la información necesaria** para realizar un proceso. Por ejemplo, los parámetros **x** e **y** de la función **calcular_menor** (diapositiva 7).
- ➔ Los **parámetros de entrada se definen mediante paso por valor**
 - ❑ Ello significa que los parámetros formales son variables independientes, que toman sus valores iniciales **como copias** de los valores de los parámetros actuales de la llamada en el momento de la invocación al subprograma.
- ➔ El parámetro actual puede ser **cualquier expresión** cuyo tipo sea compatible con el tipo del parámetro formal correspondiente.
- ➔ Se declaran especificando el tipo de dato y el identificador asociado.

Parámetros de Entrada

Paso de parámetros a funciones por VALOR (o por copia)

- ❑ Cuando se ejecuta la llamada a la función, ésta recibe una copia de los valores de los parámetros.
- ❑ Si se modifica el valor de un parámetro, el cambio sólo afecta a la función y no tiene efecto fuera de ella.
- ❑ Se usa cuando la función no necesita modificar el argumento.
 - Evita cambios accidentales.

Paso de parámetros por valor

Ejemplo 4: Calcular el factorial de un número.

```
long factorial (unsigned int N);

int main(){
    unsigned int N;

    cout << "Ingresa un numero natural: ";
    cin >> N;
    cout << "El factorial de " << N <<
        " es " << factorial(N) << endl;

    return 0;
}

long factorial (unsigned int N){
    if (N == 0)
        return 1;

    long resu = N;
    while (N > 1){
        resu *= --N;
    }
    return resu;
}
```

```
Ingresa un numero natural: 5
El factorial de 5 es 120

<< El programa ha finalizado: codigo de
<< Presione enter para cerrar esta venta
```

Parámetros de Salida

- Se utilizan para transferir al programa llamador información producida como parte de la solución realizada por el subprograma.
- Los parámetros de salida se definen mediante paso por referencia.



Se copia una dirección de memoria

- ☐ Significa que el parámetro formal es una referencia a la variable que se haya especificado como parámetro actual en el momento de la llamada al subprograma.
 - ☐ Exige que el *parámetro actual* correspondiente a un parámetro formal por referencia sea una variable.
- Cualquier acción dentro del subprograma que se haga sobre el parámetro formal se realiza sobre la variable referenciada, que aparece como parámetro actual en la llamada al subprograma.
 - Se declaran especificando el tipo de dato, el símbolo “ampersand” (&) y el identificador asociado.

Parámetros de Salida

Ejemplo 5: Obtener el cociente y el resto de una división.

```
void dividir (int dividendo, int divisor, int& coc, int& resto);  
  
int main(){  
    int cociente, resto;  
  
    dividir(7, 3, cociente, resto);  
    cout << "El resultado de dividir 7 entre 3 es: ";  
    cout << endl << "Cociente: " << cociente;  
    cout << endl << "Resto: " << resto;  
    return 0;  
}  
  
void dividir (int dividendo, int divisor, int& coc, int& resto){  
    coc = dividendo / divisor;  
    resto = dividendo % divisor;  
}
```

Parámetros de entrada

Parámetros de salida

Paso de parámetros por REFERENCIAS

```
El resultado de dividir 7 entre 3 es:  
Cociente: 2  
Resto: 1  
  
<< El programa ha finalizado: codigo de :  
<< Presione enter para cerrar esta venta
```

Ejercicio 2

Escribir una función *cuadrado()* que reciba 2 parámetros:

- *uno de entrada*, correspondiente a un **número entero**
- *el otro de salida*, que deberá transferir a main el **cuadrado** del número entero ingresado.

Parámetros de Entrada/Salida

- Son aquellos que se utilizan tanto **para recibir información** de entrada, necesaria para que el subprograma pueda realizar su computación, **como para devolver los resultados** obtenidos de la misma.
- **Se definen mediante paso por referencia** y se declaran igual que los parámetros de salida: especificando el tipo de dato, **el símbolo “ampersand” (&)** y el identificador asociado.
- Al momento de la llamada ingresan los valores a trabajar y, cuando acaba el subprograma, son utilizados para devolver los resultados.
- Dentro del subprograma los parámetros formales permiten acceder a las variables utilizadas en la llamada.
 - Si dentro del subprograma se intercambian los valores de los parámetros, indirectamente se intercambian también los valores de las variables de la llamada.
- Al terminar el subprograma el resultado está almacenado en las variables utilizadas en la llamada.

Parámetros de Entrada/Salida

Ejemplo 6: Intercambiar el valor de dos variables

```
void intercambio(int & x, int & y);
```

Parámetros de entrada/salida

```
int main() {  
    int a,b;
```

```
    a=3; b=5;
```

```
    cout << "a: " << a << " b: " << b << endl;
```

```
    intercambio(a,b);
```

```
    cout << "a: " << a << " b: " << b << endl;
```

```
    return 0;
```

```
}
```

```
void intercambio( int & x, int & y){
```

```
    int aux = x;
```

```
    x = y;
```

```
    y = aux;
```

```
}
```

```
a: 3 b: 5  
a: 5 b: 3
```

Paso de parámetros
por REFERENCIAS

Ver diferencia con:
void intercambio (int x, int y)

Parámetros de Entrada/Salida

Ejemplo 7: Multiplicar dos valores enteros

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

void multiplica(int & x, int & y);
```

```
int main() {
    int a = 2, b = -3;

    cout << "a: " << a << " b: " << b << endl;

    multiplica(a,b);

    cout << "a: " << a << " b: " << b << endl;

    return 0;
}
```

```
void multiplica(int & x, int & y) {
    x = x * y;
}
```

```
a: 2 b: -3
a: -6 b: -3
```

Paso de parámetros
por REFERENCIAS

Reglas en el paso de parámetros

En la llamada a un subprograma se deben cumplir las siguientes normas:

→ El número de parámetros actuales debe **coincidir** con el número de parámetros formales.

→ Cada parámetro formal **se corresponde** con aquel parámetro actual que ocupe la misma posición en la llamada.

→ El tipo del parámetro actual debe **estar acorde** con el tipo del correspondiente parámetro formal.

→ Un **parámetro formal de salida o entrada/salida (paso por referencia)** requiere que el parámetro actual sea una **variable**.

→ Un **parámetro formal de entrada (paso por valor)** permite que el parámetro actual sea una **variable, constante o cualquier expresión**.

Uso de la cláusula “const” en los parámetros

A veces se quiere pasar por referencia un determinado parámetro, pero a su vez se pretende que no se pueda cambiar dentro de la función porque no se requiere.

Para asegurar que dentro de una función **no se producen cambios** de un parámetro pasado por referencia solamente se tiene que usar la palabra reservada **const**, para modificar la referencia, dejándola como **referencia a constante** de la siguiente forma:

```
void referencias(int& a, const int& b)
{
    a = b + 3; // Correcto a es una referencia
    b = b + 1; // Incorrecto 'b' es una referencia a constante
}
```

En el primer caso la operación de asignación no tiene ningún problema: *a puede ser modificada y b puede ser consultada*, en cambio en el segundo caso, el compilador generará un error y nos informará de que estamos intentando modificar una constante.






Ejemplo 8: Uso de la cláusula “const” en los parámetros

```
void referencias(int & a, const int & b);

int main() {
    int x, y;
    x = 4;
    y = 7;
    referencias(x,y);
    cout << x;
    cout << y;
    return 0;
}

void referencias(int & a, const int & b){
    a = b + 3; //Correcto a es una referencia
    b = b + 1; //Incorrecto 'b' es una referencia a constante
}
```

Resultados de la Compilación

-  Compilación interrumpida!
-  Errores (1)
 -  sin_titulo.cpp:19:4: error: assignment of read-only reference 'b'
-  Advertencias (0)
-  Toda la salida

Funciones de números aleatorios

- **Función rand()**

- Definida en la biblioteca **<cstdlib>**
- Retorna un número “aleatorio” entre 0 y RAND_MAX (como máximo 32767)
- Ejemplo: `i = rand();`
- Pseudoaleatorio:
 - Secuencia preestablecida de números “random” pues la semilla del generador de números aleatorios es siempre la misma.
 - Siempre devuelve la misma secuencia para cada llamada a la función.

- **Llevando a escala**

- **rand() % n** retorna un número entre 0 y $n - 1$
- Para obtener un número aleatorio entre 1 y n : **(rand() % n + 1)**
- Sumamos 1 para obtener un número aleatorio entre 1 y n :
 $(rand() \% 6 + 1)$ obtiene un número aleatorio entre 1 y 6.

Funciones de números aleatorios

- **Función srand (semilla)**
 - Definida en la biblioteca **<cstdlib>**
 - *Inicializa* el generador de números aleatorios.
 - Se utiliza para fijar el punto de comienzo para la generación de series de números aleatorios; este valor se denomina **semilla**.
 - **srand(time(NULL));** /* incluir **<ctime>** */
 - **time(NULL)**
 - Retorna la hora actual en segundos.
 - “Randomiza” la semilla

Generador de números aleatorios

Declaración de la función	Descripción	Archivo de encabezado
<code>int random(int);</code>	La llamada <code>random(n)</code> devuelve un entero pseudo-aleatorio mayor o igual que 0 y menor o igual que <code>n-1</code> . (No está disponible en todas las implementaciones. Si no está disponible, debe usar <code>rand</code> .)	<code>cstdlib</code>
<code>int rand();</code>	La llamada <code>rand()</code> devuelve un entero pseudoaleatorio mayor o igual que 0 y menor o igual que <code>RAND_MAX</code> . <code>RAND_MAX</code> es una constante entera predefinida que se define en <code>cstdlib</code> . El valor de <code>RAND_MAX</code> es dependiente de la implementación, pero será por lo menos 32767.	<code>cstdlib</code>
<code>void srand(unsigned int);</code> (El tipo <code>unsigned int</code> es un tipo entero que sólo permite valores no negativos. Puede pensar en el tipo de argumento como <code>int</code> con la restricción de que no debe ser negativo.)	Reinicializa el generador de números aleatorios. El argumento es la semilla. Si se llama a <code>srand</code> varias veces con el mismo argumento, <code>rand</code> o <code>random</code> (la que usted utilice) producirá la misma secuencia de números pseudoaleatorios. Si se llama a <code>rand</code> o a <code>random</code> sin llamar antes a <code>srand</code> , la secuencia de números producidos será la misma como si se hubiera llamado a <code>srand</code> con un argumento de 1.	<code>cstdlib</code>

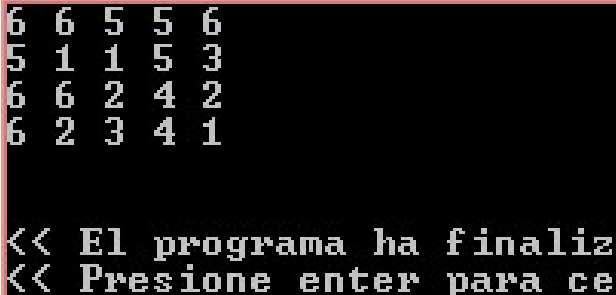

Ejemplo 9: Función que obtiene números aleatorios

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    for (int i=1; i<=20; i++) {
        /* números aleatorios entre 1 y 6 */
        cout << (rand() % 6 + 1) << " ";

        /* si el contador es divisible por 5, nueva línea */
        if (i % 5 == 0) {
            cout << endl;
        }
    }
    return 0;
}
```



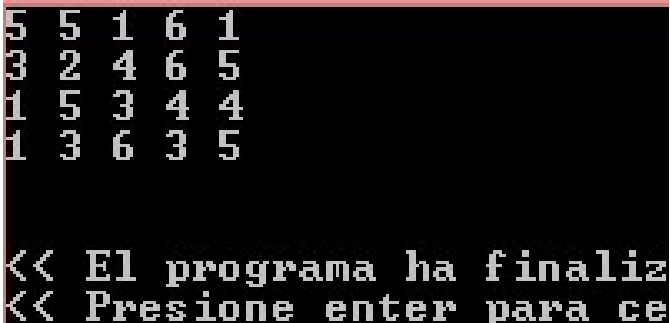
Ejemplo 9: Función que obtiene números aleatorios, cambiando la semilla

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    srand(time(NULL));
    for (int i=1;i<=20;i++){
        /* números aleatorios entre 1 y 6 */
        cout << (rand() % 6 + 1) << " ";

        /* si el contador es divisible por 5, nueva línea */
        if (i % 5 == 0){
            cout << endl;
        }
    }
    return 0;
}
```



```
5 5 1 6 1
3 2 4 6 5
1 5 3 4 4
1 3 6 3 5

<< El programa ha finaliz
<< Presione enter para ce
```

Ejercicio 3

Escribir un programa que genere aleatoriamente
5 valores enteros e informe:

- la cantidad de números impares, y
- la cantidad de números de 4 dígitos.

Para ello, deberá implementar 2 funciones:

- **esImpar(x)** que determine si x es impar, y
- **cantDig(x)** que determine la cantidad de dígitos de x (siendo x un **número entero**).

Leer Capítulo 5

Libro:

**Benjumea-Roldan
Univ-de-Málaga**