Objetivos de la práctica:
- Aplicar el método de diseño descendente para obtener programas modulares.
- Aprender el uso de funciones y la correcta modularización de programas.
- Adquirir destreza en el paso de parámetros a las funciones y reconocer la conveniencia de aplicar un tipo de paso de parámetros concreto.
- Reforzar el concepto de ámbito de las variables al aplicarlo en el diseño de subprogramas.
- Uso de funciones de librerías.

Programación modular

Los problemas más complejos se resuelven mejor cuando se descomponen en subproblemas ó módulos. Resolver por separado cada uno de estos subproblemas es más fácil que intentar resolver el problema como una unidad. Esta estrategia es lo que se llama programación modular o descendente.

En el caso de la programación, la técnica de diseño descendente ("Top-Down") de programas nos lleva a la construcción de subprogramas. Esta técnica de diseño que divide sucesivamente el problema inicial en problemas más sencillos para cuando esta subdivisión llega a un nivel de dificultad mínimo en el que ya es convertible dicho subproblema en un subprograma.

Funciones y procedimientos

Existen dos tipos de módulos independientes o subprogramas: los *procedimientos* y las *funciones*. Sin embargo, el C no hace tal distinción; para él sólo existen funciones e interpreta los procedimientos como un ejemplo particular de función que no devuelve (retorna) ningún valor.

A los subprogramas que devuelven un valor se les denomina FUNCIONES.

Si el subprograma no devuelve ningún valor se le denomina PROCEDIMIENTO.

Las funciones en C/C++

Un programa en C no es más que una colección de funciones, entre las cuales siempre ha de estar presente la función principal *main.*

¿Qué es una función en C/C++?

Una función es un módulo independiente de código, diseñado para realizar una tarea especifica y que tiene asignado un nombre como identificador, a través del cual puede ser llamada desde otras funciones, y "lanzar" su ejecución.

Definición de funciones

Para crear una nueva función hay que definirla. Dentro de la definición fijamos el nombre de la función, el tipo del resultado que devuelve, la lista de argumentos y el cuerpo de la función, siguiendo la siguiente sintaxis:

```
tipo_del_valor_devuelto nombre_de_la_función (lista_de_parámetros)
{
   cuerpo_de_la_función
}
```

Donde:

- **`Tipo_del_valor_retornado`** (tipo de la función) es el tipo del valor que devuelve la función como resultado, a través de la sentencia **`return`**. Si la función no devuelve ningún valor, es decir, sino existe ninguna sentencia **`return`** dentro de su cuerpo, el tipo de la función debe ser **`void`**.
- **`Nombre_de_la_función`** es el identificador de la función.
- **`Lista_de_parámetros_formales`** es una lista de variables precedidas de su tipo y separadas por comas. Estas variables copian los valores de los argumentos que utilizamos cuando llamamos a la función. Cuando la función no tiene argumentos se dejan vacíos los paréntesis, o bien se pone entre ellos la palabra reservada **`void`**, para indicar exactamente eso, que la función esta 'vacía' de argumentos.

- **`Cuerpo_de_la_función`** es el bloque de sentencias que definen lo que hace la función. Se encierra entre llaves.

Ejemplo de función y de procedimiento:

```
int mayor(float num1, float num2)
{
       float aux;
       if(num1 > num2)
          aux = num1;
       else
          aux = num2;
       return aux;
}
```

Un procedimiento tiene a `void` el tipo de dato que devuelve:

```
void saluda()
{
       cout << "Hola ¿Cómo estas?\n";
}
```

Llamada a las funciones

Para utilizar las funciones que hemos declarado, simplemente hemos de llamar a la función por su nombre, pasándole entre paréntesis tantos valores (argumentos) como parámetros formales hayamos definido. Las funciones se pueden llamar desde cualquier punto del programa, bien desde otro subprograma (o función) o bien desde la función principal.

Los valores que utilizamos en la llamada se van copiando sobre las variables definidas como parámetros; por lo tanto ha de existir una correspondencia, no sólo entre el número de valores y el número de parámetros, sino también entre sus tipos.

Ejemplo de llamada:

```
01 int main(){
02     float num1, num2, resul;
03     cout << "Introduce dos valores en punto flotante"\n";
04     cin >> num1 >> num2;
05     resul = mayor(num1,num2);
06     saluda();
07     return 0;
08 }
```

Parámetros Formales y Reales

Por parámetros formales entendemos los parámetros que utilizamos en la codificación del subprograma. Por ejemplo, en la función

```
int mayor(float num1, float num2)
{
.....
}
```

los parámetros `num1` y `num2` reciben el nombre de **parámetros formales .**

En cambio cuando la función es llamada en la línea 05 del código del apartado anterior, allí se les da un valor concreto y son denominados **parámetros reales**.

La sentencia return

Podemos devolver el valor dentro del cuerpo de la función a través de la sentencia **`return`**. Además, la sentencia **`return`** es una sentencia de salto que provocaba la salida inmediata de la función, independientemente de su posición relativa dentro del cuerpo de la función. Por tanto, una función acaba cuando se ejecuta la sentencia **`return`** o, en el caso de que no devuelva ningún valor, cuando se acaba el bloque de sentencias.

Sintaxis: **`return ( valor );`**

El valor que acompaña al **`return`** ha de ser del mismo tipo que "el tipo de la función".

Nada impide que dentro del cuerpo de una función existan varias sentencias **`return`**, aunque sólo una de ellas se ejecute, ya que con esta sentencia acaba la ejecución de la función.

Ámbito de las variables

Recordemos que definíamos a las funciones como un subprograma independiente que realiza una determinada tarea definida en su cuerpo. Estas funciones manejarán datos almacenados en variables, que podrán ser de diferentes clases según el lugar donde estén declaradas:

a) Variables globales: Están declaradas fuera del cuerpo de cualquier función y antes de que sea utilizada por ninguna de ellas (se recomienda declararlas al comienzo del programa antes de las definiciones de las funciones).
   El ámbito de estas variables es global, es decir, que son visibles por cualquier función y cualquiera de ellas puede modificar su valor. La vida de estas variables va ligada a la del programa, se crean cuando empieza la ejecución del programa y pueden ser utilizadas hasta que se acabe el mismo.
b) Variables locales: Están declaradas dentro de la función, bien dentro del cuerpo o bien como parámetros formales. El ámbito de la variable se reduce al de la función y fuera de esta no tienen presencia, es decir, sólo puede ser utilizada por la función donde esta declarada. Ningún otro subprograma es capaz de acceder a las variables locales de un subprograma. A esto se le llama "encapsulación" de los datos y es beneficioso para el programador tomar ventaja de esta propiedad.

Paso de parámetros por valor y paso por referencia.

Los parámetros de una función se pueden definir de dos maneras diferentes, lo que determina la forma en que se pasan los argumentos a las funciones. Se pueden pasar argumentos por valor o por referencia.

Paso por valor

Este método copia el valor de los argumentos sobre los parámetros formales, de manera que los cambios de valor de los parámetros no afectan a las variables utilizadas como argumentos en la llamada.

Lo importante en el paso por valor es el valor del argumento, por eso es indiferente si este argumento es una variable, una constante o una expresión.

Ejemplo:

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

/* Prototipos de las funciones */
float funcion_ejemplo ( float param1, int param2, int p3 );

/* Definición */
float ender( float param1, int param2, int p3 )
{
   p3 = param1 * param2 - 5;
}
int main (void)
{
   int a = 5, b;
   float h = 9.0, k;

   b = -1;
   k = ender ( 2 * sqrt ( h ), 3*7 + a, b);
   /* Después de la llamada, b sigue siendo -1 */
   return 0;
}
```

***Nota**: En el ejemplo, en la llamada de la función ender se le pasan 3 valores; estos tres valores se copian sobre los tres parámetros de la función:*

```
param1 = 2 * sqrt ( h );        /* param1 = 6 */
param2 = 3 * 7 + a;             /* param2 = 26 */
p3 = b;                         /* p1 = -1 */
```

En el ejemplo, dentro de la función se altera el valor de uno de los parámetros (p3) pero esto no modifica el valor de la correspondiente variable utilizada como argumento.

Concluyendo, en el paso por valor los parámetros copian el valor de los argumentos. Si estos parámetros cambian de valor dentro de la función, esto no altera el valor de las variables utilizadas en la llamada. Dicho de otra manera, los parámetros sólo le sirven de entrada de datos a la función.

Otro ejemplo de paso por valor:

```
void funcion_A ( void )
{
   int a;

   a = 5;
   cout << "Soy la funcion_A() y voy a pasar un " << a << " a la funcion_B( ).\n";
   funcion_B ( a );
   cout << "Soy otra vez la funcion_A( ) y a vale " << a << endl;
}

void funcion_B ( int b )
{
   cout << "Soy la funcion_B( ) y me han pasado un " << b << endl;
   b = 2;
   cout << "Soy otra vez la funcion_B( ) y he cambiado el parametro al valor " <<
                                                  b << endl;
}
```

***Nota:** El ejemplo provocará la siguiente salida por pantalla:*

```
Soy la funcion_A( ) y voy a pasar un 5 a la funcion_B( ).
Soy la funcion_B( ) y me han pasado un 5.
Soy otra vez la funcion_B( ) y he cambiado el parametro al valor 2.
Soy otra vez la funcion_A( ) y a vale 5
```

En el ejemplo anterior hemos pasado la variable "a" como argumento por valor. Esto significa que la funcion_B puede conocer el valor de "a" y usarlo, pero no puede modificarlo.

Paso por referencia

A diferencia del paso por valor, en el paso por referencia los parámetros no copian el valor del argumento, sino que comparten su valor. Por lo que cuando cambia el valor del parámetro también cambia el valor de la variable utilizada como argumento en la llamada.
Entonces, los parámetros definidos por referencia se pueden utilizar tanto de entrada como de salida de datos. Este es el otro mecanismo que poseen las funciones, a parte del **`return`**, para devolver valores.
Una consecuencia evidente es que, en la llamada de la función, los argumentos por referencia han de ser siempre variables.

Ejemplo:

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

void intercambio ( float & x, float & y );

void intercambio (float & x, float & y)
{
   float t;

   t = x;
   x = y;
   y = t;
}

int main ( void )
{
   float h , j;

   cout << "Introduce dos numeros:" << endl;
   cin >> h >> j;
   cout << "la primera variable vale " << h << " y la segunda " << j << endl;
   intercambia( h , j );
   cout << "Ahora la primera vale " << h << " y la segunda " << j << endl;
   return 0;
}
```

Otro Ejemplo:

```
void funcion_A ( void )
{
   int a;

   a = 5;
   cout << "Soy la funcion_A( ) y voy a pasar un " << a << " a la funcion_B( )\n";
   funcion_B( a );
   cout << "Soy otra vez la funcion_A( ) y a vale " << a << endl;
}

void funcion_B ( int &b )
{
   cout << "Soy la funcion_B( ) y me han pasado un " << b << endl;
   b = 2;
   cout << "Soy otra vez la funcion_B() y he cambiado el parametro a " << b<<endl;
}
```

Nota: *El ejemplo provocará la siguiente salida por pantalla:*

```
Soy la funcion_A( ) y voy a pasar un 5 a la funcion_B()
Soy la funcion_B( ) y me han pasado un 5
Soy otra vez la funcion_B( ) y he cambiado el parametro a 2
Soy otra vez la funcion_A( ) y a vale 2
```

Como hemos pasado el argumento a la funcion_B por referencia, esta función puede cambiarlo.

***Resumen**: Cuando en la llamada a un subprograma pasamos a éste un parámetro por valor, estamos pasando una copia del valor a una variable local del subprograma. Esta copia se destruye cuando acaba su ámbito, es decir, cuando acaba la ejecución del subprograma y por tanto, al ser una copia, la variable que le dio el valor, no modifica su contenido.*

En el paso por referencia, en cambio, no pasamos una copia de la variable que da el valor, sino que pasamos la misma variable en sí, de tal manera que cuando acaba la llamada del subprograma, la variable que fue pasada como parámetro real conserva las modificaciones realizadas por el subprograma.

Ejemplo de sintaxis para cada uno de estos pasos:

```
void paso_parametros(int param1, int& param2)
{
        param1 = 5;
        param2 = 7;
}
```

El parámetro `param1` está pasado por valor. El parámetro `param2` está pasado por referencia. Observar que la diferencia está en que el de referencia tiene un '`&`' delante.

Si hiciésemos la siguiente llamada:

```
int main()
{
        int x,y;
        x=8;
        y=9;

        paso_parametros(x,y);
        cout<< "x = " << x <<endl << "y = " << y<< endl;
..... return 0;
}
```

La salida por pantalla sería:

```
x = 8
y = 7
```

Al estar `x` pasado por valor, el valor de la variable que hace de parámetro real no se modifica. En cambio la que es pasada por referencia si que lo hace.

Funciones de Librería del C++

Al igual que las funciones de entrada-salida por consola se encuentran en la librería "**iostream**", existen otras librerías en C++, cada una con sus propias funciones. La mejor forma de descubrirlas es echar un vistazo a un manual de C++.

Las cabeceras de algunas librerías de utilidad son:

"iostream": Contiene los objetos **cin**, **cout**, **cerr** y **clog** que corresponden al flujo de entrada y salida estándar, y flujo de error respectivamente. Esta librería nos proporciona funciones de entrada y salida con o sin formato.

"stdlib.h": Contiene muchas funciones útiles, entre ellas algunas para convertir números a cadenas de caracteres y a la inversa (funciones **itoa**, **ultoa**, **atoi**...), **system** o **exit**.

"string": Contiene las funciones de tratamiento de cadenas de caracteres (strings).

"ctype.h": Define funciones para operar sobre los caracteres y averiguar de qué tipo son mayúsculas, números (funciones **isupper**, **islower**, **isdigit**, …)

"math.h": Contiene funciones para cálculos matemáticos. Algunas de las más útiles son:

Funciones aritméticas:

Prototipo	**Descripción**	**Cabecera**
`int abs(int x);`	Valor absoluto de x.	stdlib.h
`long labs(long x);`	Valor absolute de x.	stdlib.h
`double fabs(double x);`	Valor absoluto de x.	math.h
`double sqrt(double x);`	Raíz cuadrada de x.	math.h
`double pow(double x, double y);`	Devuelve el primer argumento (x) elevado a la potencia del segundo argumento (y).	math.h
`double exp(double x);`	Devuelve *e* (base de los logaritmos naturales) elevado a la potencia de su argumento x.	math.h
`double log(double x);`	Logaritmo natural (ln) de x.	math.h
`double log10(double x);`	Logaritmo base 10 de x.	math.h
`double ceil(double x);`	Devuelve el entero más pequeño, mayor o igual que el argumento.	math.h
`double floor(double x);`	Devuelve el entero más grande, menor o igual que el argumento x.	math.h

Funciones trigonométricas:

Prototipo	**Descripción**	**Cabecera**
`double acos(double x);`	Arco coseno.	math.h
`double asin(double x);`	Arco seno.	math.h
`double atan(double x);`	Arco tangente.	math.h
`double cos (double x);`	Coseno de x en radianes.	math.h
`double cosh (double x);`	Coseno hiperbólico de x.	math.h
`double sin (double x);`	Seno de x en radianes.	math.h
`double sinh (double x);`	Seno hiperbólico de x.	math.h
`double tan (double x);`	Tangente de x	math.h
`double tanh (double x);`	Tangente hiperbólica de x.	math.h

Generador de números aleatorios:

Prototipo	**Descripción**	**Cabecera**
`int random (int);`	La llamada random (n) devuelve un entero pseudoaleatorio mayor o igual que 0 y menor o igual que n-1. (No está disponible en todas las implementaciones. Si no está disponible, debe utilizar rand).	stdlib.h
`int rand();`	La llamada rand() devuelve un entero pseudoaleatorio mayor o igual que 0 y menor o igual que RAND_MAX. RAND_MAX es una constante entera predefinida en stdlib.h. El valor de RAND_MAX depende de la implementación pero por lo menos es 32767.	stdlib.h
`void srand(unsigned int);`	Reinicializa el generador de números aleatorios. El argumento es la semilla. El uso de srand varias veces con el mismo argumento hace que rand o random (el que utilice) produzca la misma sucesión de números pseudoaleatorios. Si rand o random se invoca sin una llamada previa a srand, la sucesión de números que se produce es la misma que si se hubiera invocado srand con un argumento 1.	stdlib.h

Notas:

• Si queremos obtener un número aleatorio entre un subrango determinado [R1 …R2], podemos utilizar la función rand, y posteriormente modificar estos valores convenientemente

$$y = \frac{x \cdot (R2 - R1)}{RAND\_MAX} + R1 \quad \text{donde x= rand(),} \quad x \in [0 \ldots RAND\_MAX], \quad y \in [R1 \ldots R2]$$

• Si se quiere inicializar el generador de semillas de forma diferente sin necesidad de proporcionar explícitamente una semilla se puede hacer *srand(time (NULL))* ya que *time (NULL)* devuelve la hora actual en décimas de segundo.
• Cuando se utilice una función matemática, además de incluir el fichero **`math.h`**, hay que linkar la librería **`-lm`** a la hora de crear el ejecutable (en Dev-C++ se hace de forma automática).

COMENTARIOS:

1. Como documentación del programa, se deberá escribir para cada función su nombre, los parámetros de entrada y salida que posee y una breve descripción de qué hace. La descripción y los parámetros de entrada y salida se deberán poner después como comentarios a la función.

Ej.

Nombre Función	**Entradas**	**Salidas**
Factorial	n: entero	fact: real
	Descripción: Cálculo del factorial de un número.	

Comentarios de la función:

```
/*************************************************************
* Funcion Factorial
*
* Descripcion: Cálculo del factorial de un número.
*
* Parametros:
* Nombre      E/S   Descripcion
* ------      ----  -----------
*   n         E
*
* Valor devuelto:
*  double     Es de tipo real para evitar que se salga de rango
*************************************************************/

double factorial(int n)
{
    ....
}
```

La descripción de los parámetros deberá escribirse únicamente si no está claro su significado. En una función pueden existir varias salidas. La elección de cual de estas salidas corresponderá con el resultado de la función se decide en la implementación (al escribir el programa) y no en la documentación del programa.

2. Las funciones deben ser lo más independientes posible del resto del programa, por lo que sólo se pueden comunicar con el resto del programa mediante los parámetros. **NO pueden utilizar variables globales**.