

## Práctica 5: Procedimientos y Funciones en C++.

En su concepción más simple, un **procedimiento** es una construcción que permite dar nombre a un conjunto de *sentencias* y *declaraciones* asociadas que se usan para resolver un subproblema dado. Los procedimientos no siempre realizan la misma función y pueden recibir parámetros. No repetir código no es la única razón para estructurar un programa usando procedimientos. Puesto que un subproblema puede codificarse como un procedimiento, un problema complejo puede dividirse en subproblemas más simples, quienes a su vez pueden ser de nuevo subdivididos hasta llegar a la descripción de subproblemas muy simples que se puedan codificar como procedimientos escritos en C++. La filosofía que acabamos de presentar se denomina **Refinamiento Progresivo o por pasos, diseño descendente, Programación Top-Down** o bien **Divide y Vencerás**.

En C++ no hay diferencias entre procedimientos y funciones: todo son funciones, con la diferencia de que un procedimiento es una función que no devuelve nada (void). Tanto procedimientos como funciones, deben ser declarados antes de poder ser usados. Esto es posible de dos formas: indicando su cabecera (nombre, parámetros y tipo de retorno) después de la zona de constantes y antes del programa principal, o bien escribir el procedimiento o la función entera en dicha zona. Nosotros recomendamos la primera opción, ya que evita problemas de orden de implementación para solucionar problemas de ámbito. Mientras que un procedimiento ejecuta un grupo de sentencias, una función además *devuelve un valor al punto donde se llamó*. Una llamada a una función puede aparecer como *operando de alguna expresión*. El valor de la función se usa, por tanto, para calcular el valor total de la expresión. El uso de una función puede ser algo como  $2.0 + \text{Maximo}(3.0, p) / 6.90$ , de forma que la llamada devolverá el mayor valor entre 3.0 y el contenido de la variable p y con dicho valor se evaluará el resto de la expresión. Otro ejemplo puede ser:  $p = \text{Maximo}(3.0, p)$ .

Obsérvese que el *tipo del resultado* que devolverá la función aparece declarado en la cabecera sustituyendo a la palabra *void* que identifica a un procedimiento. En este sentido, en C++, puede verse a un procedimiento como un tipo especial de función que devuelve un valor void (nulo).

- ❑ Las funciones en C++ pueden devolver cualquier tipo menos arrays.
- ❑ Toda función debe ejecutar una sentencia *return*.

### Sintaxis de un Función:

- Cabecera: `<tipo> <identificador> ( <parámetros> ) ;`  
Ejemplo: `int mi_funcion( int x );`
- Implementación: `<tipo> <identificador> ( <parámetros> )`  
`{`  
`<sentencias>`  
`return <valor>;`  
`}`

Ejemplo: `int mi_funcion( int x )`  
`{`  
`return 0;`  
`}`

**Parámetros de entrada (valor)**

Los parámetros de **entrada** (valor) se usan para proporcionar información de entrada a un procedimiento. Dentro de éste pueden considerarse como variables cuyo valor inicial es el resultado de evaluar los parámetros actuales. Como parámetro *actual* debe aparecer una **expresión** cuyo resultado sea un valor de un tipo *asignable* al correspondiente parámetro formal. Puesto que las variables usadas como parámetros formales de entrada no *sirven* para *cambiar a los parámetros actuales* (sólo para conocer su valor en el momento de la llamada y asignarle un nombre a ese valor dentro del procedimiento) se les suele denominar **Parámetros por valor**. Ejemplo:

```
void dibLineas( int anchura, int altura )
{
    int nFila;
    int nColumna;

    for( nFila = 1 ; nFila <= altura; ++nFila )
    {
        for( nColumna = 1; nColumna <= anchura; ++nColumna )
        {
            cout << "-";
        }
        cout << endl;
    }
}
```

**Parámetros de entrada/salida (referencia)**

Para usar parámetros de **entrada/salida**, el parámetro *formal* debe estar precedido por el símbolo & y el *parámetro actual debe ser una variable* (no una expresión cualquiera). Los parámetros de entrada/salida se usan cuando se desea que un procedimiento **MODIFIQUE** el contenido de la *variable actual*. El hecho de definir estos parámetros explícitamente como variables hace consciente al programador de los lugares donde un procedimiento cambia a una variable que se le pase como parámetro. El funcionamiento de los parámetros de entrada/salida está basado en pasar al procedimiento una referencia a la variable actual en lugar de su valor. Por ello, a estos parámetros también se los denominan **parámetros por referencia**. Ejemplo:

```
void raices( double a, double b, double c,
            double &R1, double &R2 )
{
    double DiscriminanteS;
    // Se supone un discriminante positivo
    DiscriminanteS = sqrt( b*b-4.0*a*c );
    R1 = (-b + DiscriminanteS) / (2.0*a);
    R2 = (-b - DiscriminanteS) / (2.0*a);
}
```

Para el compilador la *zona de memoria* representada por el parámetro formal y real es la *misma* y al terminar la ejecución del procedimiento el parámetro real *puede* haber cambiado su valor.

No existe ninguna relación entre los identificadores de los parámetros formales y reales. La transferencia de valores se efectúa **por posición** en la lista de parámetros y **no por el nombre** que éstos tengan. Por ejemplo, el resultado de ejecutar `a= 5; b = 3; p(a,b);` es el mismo que el de ejecutar `b = 5; a = 3; p(b,a);` siempre que el procedimiento `p` no intente cambiar el valor de las variables que recibe como parámetros.

### Enunciado de la Práctica

Se desea realizar un programa para el cálculo de operaciones con números complejos representados binomialmente ( $a + b i$ ). El programa deberá presentar un menú con las siguientes operaciones:

```
MENU
====
Elaborado Por : Nombre Apellidos
E.T.S.I Informatica 1ª Gestion
Fecha: 2 de Diciembre de 2.004

A. Sumar 2 Números Complejos.
B. Restar 2 Números Complejos.
C. Multiplicar 2 Números Complejos.
D. Dividir 2 Números Complejos.
E. Mostrar Módulo y Argumento de 1 Número Complejos.
X. Salir del Programa
```

### Descripción de Opciones:

**A. Sumar 2 Números Complejos.** Se solicita al usuario que introduzca la parte real y la parte imaginaria de ambos números y se muestra por pantalla el resultado.

Ejemplo:

```
Número 1
Parte Real: 2
Parte Imaginaria: 3
```

```
Número 2
Parte Real: 8
Parte Imaginaria: 2
```

$$(2+3i) + (8+2i) = (10+5i)$$

Presione una tecla para continuar . . .

**B. Restar 2 Números Complejos.** Se solicita al usuario que introduzca la parte real y la parte imaginaria de ambos números y se muestra por pantalla el resultado.

Ejemplo:

```
Número 1
Parte Real: 2
Parte Imaginaria: 3
```

```
Número 2
Parte Real: 8
Parte Imaginaria: 2
```

$$(2+3i) - (8+2i) = (-6+i)$$

Presione una tecla para continuar . . .

**C. Multiplicar 2 Números Complejos.** Se solicita al usuario que introduzca la parte real y la parte imaginaria de ambos números y se muestra por pantalla el resultado.

Ejemplo:

Número 1  
Parte Real: 2  
Parte Imaginaria: 3

Número 2  
Parte Real: 8  
Parte Imaginaria: 2

$$(2+3i) * (8+2i) = (10+28i)$$

Presione una tecla para continuar . . .

**D. Dividir 2 Números Complejos.** Se solicita al usuario que introduzca la parte real y la parte imaginaria de ambos números y se muestra por pantalla el resultado. Si el divisor es nulo (0+0i) se informará del error.

Ejemplo:

Número 1  
Parte Real: 2  
Parte Imaginaria: 3

Número 2  
Parte Real: 8  
Parte Imaginaria: 2

$$(2+3i) / (8+2i) = (0.323529+0.294118i)$$

Presione una tecla para continuar . . .

**E. Mostrar Módulo y Argumento de 1 Número Complejos.** Se solicita al usuario que introduzca la parte real y la parte imaginaria de 1 número y se muestra por pantalla su módulo y su argumento (en radianes).

Ejemplo:

Parte Real: 2  
Parte Imaginaria: 3

Módulo : 3.60555  
Argumento : 0.982794

Presione una tecla para continuar . . .

**X. Salir del Programa.** Se solicita confirmación y sólo en caso de sea afirmativa se sale del programa.