3 Procedimientos y funciones

Contenido

3.1. Justificación

Cuando un programa comienza a ser *largo y complejo* (la mayoría de los problemas reales se solucionan con programas de este tipo) no es apropiado tener un único texto con sentencias una tras otra. La razón es que no se comprende bien qué hace el programa debido a que se intenta *abarcar toda la solución* a la vez. Asimismo el programa se vuelve *monolítico* y *difícil de modificar.* Además suelen aparecer trozos de código muy similares entre sí *repetidos* a lo largo de todo el programa.

Para solucionar estos problemas y proporcionar otras ventajas adicionales a la programación, los lenguajes de alto nivel suelen disponer de una herramienta que permite estructurar el programa principal como compuesto de **subprogramas** (rutinas) que resuelven problemas parciales del problema principal. A su vez, cada uno de estos subprogramas puede estar resuelto por otra conjunción de problemas parciales, etc... Los **procedimientos** y las **funciones** son mecanismos de estructuración que permiten

ocultar los detalles de la solución de un problema y resolver una parte de dicho problema en otro lugar del código.

Supongamos que queremos calcular la media de votos que un número (dado por el usuario) de candidatos ha conseguido obtener en una votación. La entrada y salida de datos se pretende mejorar separándolas mediante una línea de guiones (Figura 3.1).

```
--------------------------------------------------
¿Cuántos candidatos hay? ... 3
--------------------------------------------------
Candidato número: 1
Teclee el número de votos para este candidato: ... 27
--------------------------------------------------
Candidato número: 2
Teclee el número de votos para este candidato: ... 42
--------------------------------------------------
Candidato número: 3
Teclee el número de votos para este candidato: ... 3
--------------------------------------------------
Número medio de votos conseguidos por candidato es: 2.4E+1
--------------------------------------------------
```

Figura 3.1. *Ejemplo de una entrada y salida de datos para el problema de la votación.*

Como podemos observar en la Figura 3.2, el código de esta solución se ve oscurecido por la aparición repetitiva de las sentencias que se usan para dibujar la línea. Si tenemos en cuenta que estas sentencias no son ni siquiera una parte fundamental de la solución del problema y que se mezclan con las otras visualmente y que repetimos código similar, llegaremos a la conclusión de que esta solución no es la mejor.

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. Usando procedimientos (Figura 3.3) la solución es más corta, comprensible y fácilmente modificable.

Los procedimientos no siempre realizan la misma función y pueden recibir parámetros como se verá en breve.

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.**

```
/*************************************************
* Autor:
* Fecha:              Versión:
*************************************************/
#include <iostream.h>

const int longitudLinea             = 65;
const int maximoNumeroCandidatos    = 50;

int main()
{
  int numeroDeVotos, totalDeVotos, numeroDeCandidatos;
  float media;
  int i;
  int numeroDeCandidato;

  for( i = 1; i <= longitudLinea; i++ )
  {
       cout << "-";
  }
  cout << endl;
  cout << "¿Cuántos candidatos hay? ...";
  cin >> numeroDeCandidatos;
  totalDeVotos = 0;

  for( numeroDeCandidato = 1;
       numeroDeCandidato <= numeroDeCandidatos;
       numeroDeCandidato++ )
  {
 for( i = 1; i <= longitudLinea; i++ )
 {
      cout << "-";
 }
 cout << endl;
 cout << "Candidato número:" << numeroDeCandidato;
 cout << endl;
 cout << "Teclee el número de votos para este candidato
";
 cin >> numeroDeVotos;
 totalDeVotos += numeroDeVotos;
  }

  for( i = 1; i <= longitudLinea; i++ )
  {
       cout << "-";
  }
  cout << endl;
```

```
  media = float (totalDeVotos) / float
(numeroDeCandidatos);
  cout << "Número medio de votos conseguidos por
candidatos es:";
  cout << media << endl;
  for( i = 1; i <= longitudLinea; i++ )
  {
    cout << "-";
  }
  cout << endl;

  return 0;
}
```

Figura 3.2. *Solución sin procedimientos del problema de la votación.*

3.2. Declaración y llamada de procedimientos

Como el resto de entidades (objetos) en C++, los procedimientos (y las funciones) *deben declararse antes de ser usados*. La declaración de un procedimiento se realiza en dos partes: prototipo e implementación. El prototipo de un procedimiento sirve para declarar el nombre del procedimiento y los parámetros que recibe. La implementación sirve para definir qué trabajo realiza el procedimiento y cómo lo lleva a cabo. La sintaxis de la declaración de un procedimiento se presenta en la Figura 3.4.

```
/*************************************************
* Autor:
* Fecha:              Versión:
*************************************************/
#include <iostream.h>

const int longitudLinea             = 65;
const int maximoNumeroCandidatos    = 50;

void dibujarLinea();

int main()
{
  int numeroDeVotos, totalDeVotos, numeroDeCandidatos;
  float media;
  int numeroDeCandidato;

  dibujarLinea();
  cout << "¿Cuántos candidatos hay? ...";
  cin >> numeroDeCandidatos;
  totalDeVotos = 0;
```

```
  for( numeroDeCandidato = 1;
       numeroDeCandidato <= numeroDeCandidatos;
       numeroDeCandidato++ )
  {
    dibujarLinea();
    cout << "Candidato número: " << numeroDeCandidato;
    cout << endl;
    cout << "Teclee el número de votos para este
candidato ";
    cin >> numeroDeVotos;
    totalDeVotos += numeroDeVotos;
  }

  dibujarLinea();

  media = float (totalDeVotos) / float
(numeroDeCandidatos);
  cout << "Número medio de votos conseguidos por
candidatos es: ";
  cout << media << endl;
  dibujarLinea();

  return 0;
}




void dibujarLinea()
{
  int i;

  for( i = 1; i <= longitudLinea; i++ )
  {
       cout << "-";
  }
  cout << endl;
}
```

Figura 3.3. *Solución con procedimientos del problema de la votación.*

Por el momento no vamos a presentar procedimientos con parámetros y por esa razón no vamos a describir el elemento <*ParámetrosFormales*>.

Dentro del código fuente de nuestro programa, el prototipo de cada procedimiento aparecerá antes del cuerpo principal de nuestro programa (función `main`), tras la declaración de constantes. Por otro lado, la implementación de cada procedimiento se

realizará al final del código fuente del programa tras el cuerpo principal del mismo. Un ejemplo de esto puede verse en la Figura 3.3.

En las declaraciones es conveniente separar los procedimientos por una o varias líneas en blanco para después poder encontrarlos rápidamente. Naturalmente todo procedimiento debería tener asociado un comentario respecto a lo que hace y el algoritmo que utiliza para ello.

El uso de procedimientos es un claro exponente de la programación estructurada y modular, en el sentido que los errores se pueden localizar en el procedimiento que los produce. La solución al problema es más comprensible y clara, y en el futuro, con una buena documentación, será posible realizar cambios en el programa de forma sencilla.

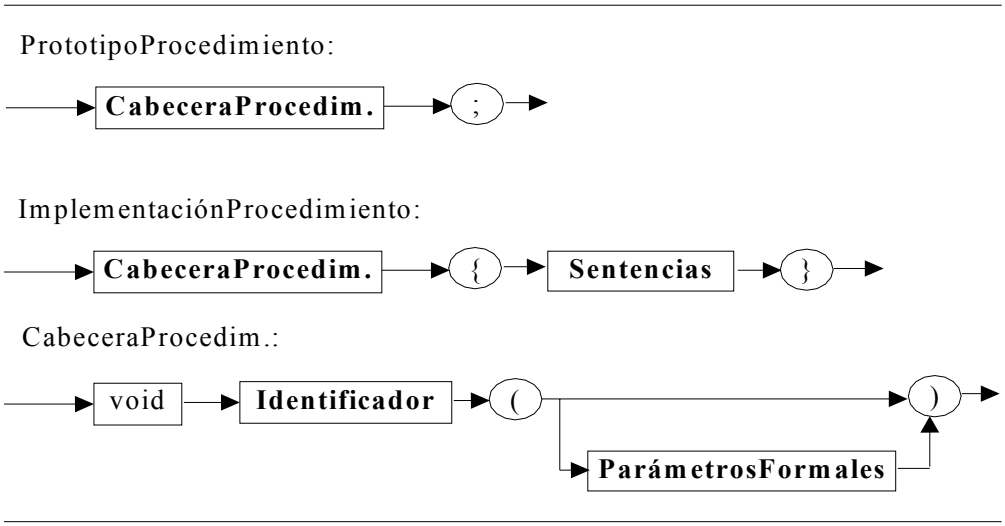


Figura 3.4. *Notación Conway para la sintaxis de la declaración de procedimientos (prototipo e implementación).*

Una vez declarado, el procedimiento puede ser *llamado* (invocado) en el programa. Para ello basta especificar su nombre (y parámetros en el caso de que los tuviese) como si se tratara de una sentencia más. Cuando se alcanza la llamada, el control pasa a la primera sentencia de dicho procedimiento y cuando éste acaba de ejecutar su cuerpo el control vuelve a la siguiente instrucción escrita tras la llamada al procedimiento.

3.2.1. La sentencia nula

Como se puede observar en la Figura 3.4, el *cuerpo de un procedimiento* (en la implementación del procedimiento) está formado por una secuencia de sentencias. Las sentencias más importantes de C++ se han descrito en los temas anteriores. En este

apartado se describirá una sentencia particular de C++ sintácticamente útil para la escritura de programas: la **sentencia nula**.

La sentencia nula es una sentencia de C++ que no realiza ninguna acción en el programa (no hace nada). Como ya se ha dicho, la principal utilidad de esta sentencia es de origen sintáctico, ya que se utiliza en aquellos puntos del código en C++ donde debe aparecer una sentencia pero no queremos que el programa realize ninguna tarea. Esta sentencia se representa en C++ mediante el carácter punto y coma (`;`).

Un ejemplo típico de utilización de la sentencia nula son los cuerpos de bucles en los que no hay que realizar ninguna acción. Por ejemplo, en la Figura 3.5 puede verse un procedimiento con un bucle cuyo cuerpo es la sentencia nula y cuya finalidad es esperar hasta que se pulse un retorno de carro.

```
void esperaRetorno()
{
  cout << "Pulsar retorno de carro para continuar...";

  while( cin.get() != '\n' );
}
```

Figura 3.5. *Ejemplo de bucle con sentencia nula como cuerpo.*

3.3. Localidad, anidamiento, ámbito y visibilidad

La introducción de procedimientos y funciones en un programa, permite tener varias zonas de declaraciones de entidades (variables, constantes, etc...) a lo largo del programa. Es necesario, por tanto, establecer ciertas reglas que definan las zonas de visibilidad o accesibilidad de los mismos en función del lugar donde aparecen declarados. Estas reglas son las reglas de ámbito. Las reglas de ámbito establecidas en la asignatura Elementos de Programación para el pseudolenguaje utilizado en la misma son totalmente válidas para C++ y, por tanto, aplicables a nuestros programas escritos en dicho lenguaje, por lo que no volverán a ser descritas en este tema.

La única aclaración que es necesario hacer a estas reglas de ámbito se refiere a la visibilidad de los procedimientos. Al no permitir C++ el anidamiento de procedimientos (declarar un procedimiento dentro de otro), las reglas de ámbito relativas a los procedimientos anidados no son aplicables.

3.4. Procedimientos con parámetros

Un procedimiento puede resultar mucho más útil si se puede *variar su comportamiento de una llamada a otra*. Esto se consigue añadiéndole **parámetros**. Por ejemplo, sería poco útil que el procedimiento *system* de la biblioteca *stdlib* ejecutase siempre el mismo comando de sistema. Por eso es posible pasarle un parámetro que se corresponde con el comando que queremos ejecutar en cada momento.

```
system( "dir" );
```

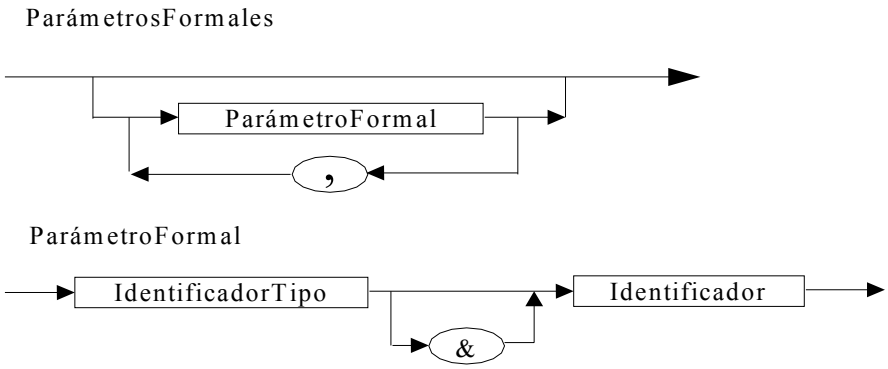
En C++ existen dos tipos de parámetros diferentes:

1. Parámetros de entrada (valor)
2. Parámetros de entrada/salida (referencia)

Los parámetros únicamente de salida no son implementados por C++.

Los parámetros que se declaran en la cabecera de un procedimiento se denominan **Parámetros Formales**, mientras que los parámetros que aparecen en una *llamada* al procedimiento se denominan **Parámetros Actuales o Reales**.

La Figura 3.6 muestra la sintaxis para los parámetros formales (escritos en la



cabecera del procedimiento).

Figura 3.6. *Sintaxis para declarar los Parámetros Formales*

3.4.1. 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**.

En la Figura 3.7 puede verse un ejemplo de paso de parámetros por valor. En esta figura puede apreciarse un procedimiento llamado `dibLineas` que recibe dos parámetros por valor: `anchura` y `altura`.

```
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;
  }
}
```

Figura 3.7. *Parámetros de entrada*

Como ejemplo, la llamada siguiente llamada a `dibLineas` genera una línea con 65 guiones:

```
dibLineas( 65, 1 )
```

Esta otra llamada a `dibLineas` dibuja 5 líneas en blanco:

```
dibLineas( 0, 5 )
```

Por último, la llamada a `dibLineas` que se muestra a continuación no hace nada:

```
dibLineas( 5, 0 )
```

3.4.2. 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 **cambie** 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 puede modificar 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**.

Un ejemplo de paso de parámetros por referencia puede verse en la Figura 3.8. En esta figura se muestra un procedimiento que recibe tres parámetros por valor (`a, b, c`) y utiliza dos parámetros por referencia (`R1` y `R2`) para devolver el valor de las raíces de un polinomio de segundo grado.

```
void raices( float a, float b, float c, float &R1,
float &R2 )
{
  float 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 );
}
```

Figura 3.8. *Parámetros de entrada/salida*

Una posible llamada a la función de la Figura 3.8 sería:

```
Raices( 1.0, 2.0, 1.0, a, b )
```

donde a y b son dos variables de tipo *float*.

Los parámetros formales reciben valores que se reflejarán en los parámetros actuales usados en la llamada al procedimiento. Los tipos de la variable formal y actual deben coincidir.

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

No existe ninguna relación entre los identificadores de los parámetros formales y actuales. 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.

3.5. Funciones

Mientras que un procedimiento ejecuta un grupo de sentencias, una función además d*evuelve 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.

En la Figura 3.9 se muestra un ejemplo de función que recibe dos parámetros de tipo **`float`** y devuelve un valor de este mismo tipo. Concretamente, la función de la Figura 3.9 devuelve el mayor valor que se le pase por parámetro.

```
float Maximo( float x, float y )
{
  float resultado;

  if (x>y)
  {
    resultado = x;
  }
  else
  {
    resultado = y;
  }

  return resultado;
}
```

Figura 3.9. *Ejemplo de una función que calcula el máximo de entre dos números.*

El uso de esta función puede ser algo como:

```
valor =  2.0 + 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 = 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*.