

CAPÍTULO 3

©F.J.Ceballos/RA-MA

ESTRUCTURA DE UN PROGRAMA

En este capítulo estudiaremos cómo es la estructura de un programa C++. Partiendo de un programa ejemplo sencillo analizaremos cada una de las partes que componen el mismo, así tendrá un modelo para realizar sus propios programas. También veremos cómo se construye un programa a partir de varios módulos fuente. Por último, estudiaremos los conceptos de ámbito y accesibilidad de las variables.

PARADIGMAS DE PROGRAMACIÓN

C++ es un lenguaje que ofrece un abanico tal de utilidades, que lo hacen cómodo para soportar distintos estilos de programación como son la *programación estructurada*, la *programación modular* y la *programación orientada a objetos*.

La unidad básica de la programación estructurada es el procedimiento o función. Desde este estilo de programación un programa C++ se diseña como un conjunto de funciones, las cuales se comunican entre sí mediante el paso de parámetros y la devolución de valores. El concepto de función supone una abstracción que se hace tanto más latente cuanto más grande es el programa.

Un conjunto de procedimientos relacionados, junto con los datos que manipulan, se denomina *módulo*. La aplicación de esta definición condujo a la programación modular, que va mucho más allá de la encapsulación en simples procedimientos; aquí se encapsula un conjunto de operaciones relacionadas que actuarán sobre un conjunto de datos, formando un módulo. El nivel de abstracción está ahora definido por la interfaz que muestra el módulo (funciones del módulo) al usuario para operar sobre el conjunto de datos.

Un módulo define una especie de caja negra que muestra una interfaz al usuario para actuar sobre un conjunto de datos; pero, vista como tal caja negra, no hay

forma de adaptarla a nuevos usos si no es modificando el propio módulo, lo que evidencia poca flexibilidad de adaptación a nuevas necesidades. Este problema tendría solución, en parte, mientras no se alterase el conjunto de datos, si fuera posible construir una jerarquía de módulos en la que cada uno de ellos heredara la funcionalidad de su predecesor; de esta forma añadir nuevas características sería sencillo. Estas ideas, reutilización del código, flexibilidad para añadir nueva funcionalidad, incluso modificar la estructura de datos sobre la que se opera, son las que caracterizan a la programación orientada a objetos que estudiaremos un poco más adelante, a partir del capítulo *Clases*.

C++ fue precisamente diseñado para apoyar la abstracción de datos y la programación orientada a objetos, pero sin forzar a los usuarios a utilizar este estilo de programación, conclusión que usted mismo obtendrá cuando finalice la lectura de este libro.

ESTRUCTURA DE UN PROGRAMA C++

Puesto que C++ es un lenguaje híbrido, vamos a pensar primeramente en un diseño bajo los términos que definen la *programación estructurada* y la *programación modular* y, como ya hemos dicho, más adelante nos centraremos en la *programación orientada a objetos*. Desde esta perspectiva, la solución de cualquier problema no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de elementos naturales del problema mismo, abstraídos de alguna manera, que darán lugar al desarrollo de las funciones mencionadas y a su agrupación en módulos. Muchas de las funciones que utilizaremos pertenecen a la biblioteca estándar de C++, por lo tanto ya están escritas y compiladas. Pero otras tendremos que escribirlas nosotros mismos, dependiendo del problema que tratemos de resolver en cada caso.

Para explicar cómo es la estructura de un programa C++ desde este punto de vista, vamos a plantear un ejemplo sencillo. Se trata de un programa que muestra una tabla de equivalencia entre grados centígrados y grados *Fahrenheit*, según se observa a continuación:

- 30 C	- 22 , 00 F
- 24 C	- 11 , 20 F

90 C	194 , 00 F
96 C	204 , 80 F

La relación entre los grados centígrados y los grados *Fahrenheit* viene dada por la expresión $grados\ Fahrenheit = 9/5 * grados\ centigrados + 32$. Los cálculos

los los vamos a realizar para un intervalo de -30 a 100 grados centígrados con incrementos de 6.

Según lo enunciado, vamos a analizar el problema propuesto. ¿Qué piden? Escribir cuántos grados *Fahrenheit* son -30 C, -24 C..., *n* grados centígrados..., 96 C. Y, ¿cómo hacemos esto? Aplicando la fórmula:

$$\text{GradosFahrenheit} = 9/5 * \text{nGradosCentígrados} + 32$$

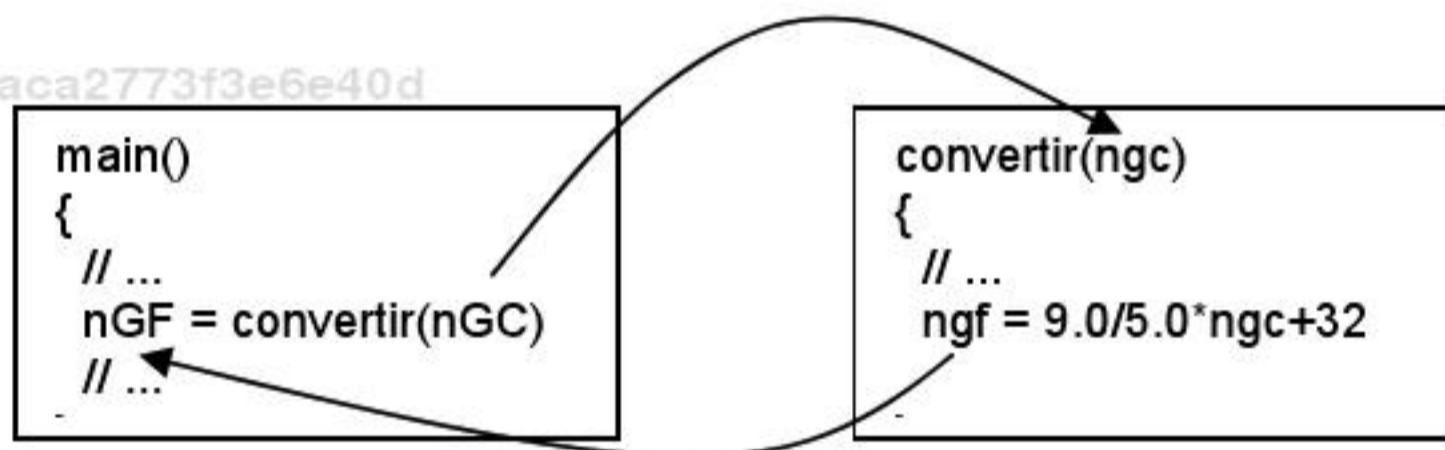
una vez para cada valor de *nGradosCentígrados*, desde -30 a 100 con incrementos de 6. Para entender con claridad lo expuesto, hagamos un alto y pensemos en un problema análogo; por ejemplo, cuando nos piden calcular el logaritmo de 2, en general de *n*, ¿qué hacemos? Utilizar la función *log*; esto es:

$$x = \log(n)$$

Análogamente, si tuviéramos una función *convertir* que hiciera los cálculos para convertir *n* grados centígrados en grados *Fahrenheit*, escribiríamos:

$$\text{GradosFahrenheit} = \text{convertir}(\text{nGradosCentígrados})$$

Sin casi darnos cuenta, estamos haciendo una descomposición del problema general en subproblemas más sencillos de resolver. Recordando que un programa C++ tiene que tener una función **main**, además de otras funciones si lo consideramos necesario, ¿cómo se ve de una forma gráfica la sentencia anterior? La figura siguiente da respuesta a esta pregunta:



Análogamente a como hacíamos con la función logaritmo, aquí, desde la función **main**, se llama a la función *convertir* pasándola como argumento el valor en grados centígrados a convertir. La función logaritmo devolvía como resultado el logaritmo del valor pasado. La función *convertir* devuelve el valor en grados *Fahrenheit* correspondiente a los grados centígrados pasados. Sólo queda visualizar este resultado y repetir el proceso para cada uno de los valores descritos. Seguramente pensará que todo este proceso se podría haber hecho utilizando solamente la función **main**, lo cual es cierto. Pero lo que se pretende es que pueda ver de una forma clara que, en general, bajo los términos que definen la *programación es-*

tructurada, un programa C++ es un conjunto de funciones que se llaman entre sí con el fin de obtener el resultado perseguido, y que la forma sencilla de resolver un problema es descomponerlo en subproblemas más pequeños y por lo tanto más fáciles de solucionar; cada subproblema será resuelto por una función C++.

Una vez analizado el problema, vamos a escribir el código. Inicie un nuevo proyecto y añada al fichero *main.cpp* las funciones *convertir* y *main* como se muestra a continuación:

```
/* Paso de grados Centígrados a Fahrenheit (F=9/5*C+32)
 */
#include <iostream>
#include <iomanip>
using namespace std;

// Declaración de la función convertir
float convertir(int c);

int main()
{
    const int INF = -30; // Límite inferior del intervalo de °C
    const int SUP = 100; // Límite superior */

    // Declaración de variables locales
    int nGradosCentigrados = 0;
    int incremento = 6; // iniciar incremento con 6
    float GradosFahrenheit = 0;

    nGradosCentigrados = INF;
    cout << fixed; // formato en coma flotante
    while(nGradosCentigrados <= SUP)
    {
        // Se llama a la función convertir
        GradosFahrenheit = convertir(nGradosCentigrados);
        // Se escribe la siguiente línea de la tabla
        cout << setw(10) << nGradosCentigrados << " °C";
        cout << setw(10) << setprecision(2)
            << GradosFahrenheit << " °F\n";
        // Siguiente valor a convertir
        nGradosCentigrados += incremento;
    }
    return 0;
}

// Función convertir grados centígrados a Fahrenheit
float convertir(int gcent)
{
    float gfahr;
```

```
    gfahr = 9.0 / 5.0 * gcent + 32;  
    return (gfahr);  
}
```

Como ya expusimos en un capítulo anterior, siempre que se invoque a una función de la biblioteca de C++, de cualquier otra biblioteca o de nuestros propios recursos, como ocurre con *convertir*, el compilador requiere conocer cómo fue declarada esa función que estamos utilizando, de ahí la declaración anticipada que hemos especificado antes de **main**. Análogamente, los ficheros de cabecera como *iostream* e *iomanip* anticipan las declaraciones de los elementos de la biblioteca de C++ que utilizamos en el código que hemos escrito.

No se preocupe si no entiende todo el código. Ahora lo que importa es que aprenda cómo es la estructura de un programa, no por qué se escriben unas u otras sentencias, cuestión que aprenderá más tarde en éste y en sucesivos capítulos.

A continuación vamos a realizar un estudio de las distintas partes que forman la estructura de un programa. En el ejemplo realizado podemos observar que un programa C++ consta de:

- Directrices **#include** (inclusión de declaraciones y/o definiciones).
- Directrices **using** (definir los espacios de nombres a los que se refiere nuestro programa).
- Función principal **main**.
- Otras funciones y/o declaraciones.

El orden establecido no es esencial, aunque sí bastante habitual. Así mismo, cada función consta de:

- Definiciones y/o declaraciones.
- Sentencias a ejecutar.

Los apartados que se exponen a continuación explican brevemente cada uno de estos componentes que aparecen en la estructura de un programa C++.

Directrices para el preprocessador

La finalidad de las directrices es facilitar el desarrollo, la compilación y el mantenimiento de un programa. Una directriz para el preprocessador se identifica porque empieza con el carácter **#**. Las más usuales son la directriz de inclusión, **#include**, la directriz de definición, **#define**, y las condicionales **#ifndef** y **#endif**. Las directrices son procesadas por el *preprocesador* de C++, que es invocado por el compilador antes de proceder a la traducción del programa fuente.

Inclusión incondicional

En general, cuando se hace uso de un elemento de C++ (constante, variable, clase, objeto, función, etc.), es necesario que dicho elemento esté previamente declarado. Por ejemplo, observe el objeto **cout** en el programa anterior. ¿Cómo sabe el compilador que **cout** admite los argumentos que se han especificado? Pues lo sabe por la información aportada por el fichero de cabecera *iostream*. Lo mismo podríamos decir de las funciones **setw** (ancho) o **setprecision** (precisión); la información respecto a qué parámetros aceptan estas funciones y qué valor retornan es proporcionada por el fichero de cabecera *iomanip*. Esto es así para todos los elementos utilizados en el programa, los pertenecientes a la biblioteca de C++ y los definidos por el programador. Las declaraciones de los elementos pertenecientes a la biblioteca de C++ se localizan en los ficheros de cabecera que generalmente se encuentran en el directorio predefinido *include* de la instalación C++.

Según lo expuesto en el apartado anterior, para incluir la declaración de un elemento de la biblioteca de C++ antes de utilizarlo por primera vez, basta con añadir el fichero de cabecera que lo contiene. Esto se hace utilizando la directriz **#include** de alguna de las dos formas siguientes:

```
#include <iostream>
#include "misfunc.h"
```

Si el fichero de cabecera se delimita por los caracteres `<>`, el preprocesador de C++ buscará ese fichero directamente en las rutas especificadas para los ficheros de cabecera (generalmente esta información la proporciona una variable de entorno del sistema, por ejemplo, *INCLUDE*). En cambio, si el fichero de cabecera se delimita por los caracteres `" "`, el preprocesador de C++ buscará ese fichero primero en el directorio actual de trabajo y si no lo localiza aquí, entonces continúa la búsqueda por las rutas a las que nos hemos referido anteriormente. En cualquier caso, si el fichero no se encuentra se mostrará un error.

Lógicamente, con esta directriz se puede incluir cualquier fichero que contenga código fuente, independientemente de la extensión que tenga.

Definición de un identificador

Mediante la directriz **#define** *identificador* se indica al preprocesador que declare ese *identificador*. Por ejemplo:

```
#define ID
```

Inclusión condicional

Utilizando las directrices **#ifndef** *id* (si no está definido *id*) y **#endif** (fin de si...) se puede especificar que un determinado código sea o no incluido en un programa. La forma de realizar esto es así:

```
#ifndef ID
#define ID
// Añadir aquí el código que se desea incluir.
#endif
```

Las directrices especificadas en el código anterior se ejecutan así: si el identificador *ID* aún no ha sido definido, lo cual ocurrirá la primera vez que se ejecute **#ifndef**, esta directriz devolverá un valor distinto de cero (verdadero) y se procesará todo el código que hay hasta la directriz **#endif**, incluida la directriz **#define** que definirá *ID*; si el *ID* ya está definido, lo cual ocurrirá las siguientes veces que se ejecute **#ifndef**, esta directriz devolverá un valor cero (falso) y no se procesará el código que hay hasta la directriz **#endif**. Esta es la técnica que emplea la biblioteca de C++ en sus ficheros de cabecera para que su contenido no sea incluido más de una vez en una unidad de traducción a través de la directriz **#include**, lo que durante la compilación provocaría errores por redefinición.

Definiciones y declaraciones

Una declaración introduce uno o más nombres en un programa. Por lo tanto, todo nombre (identificador) debe ser declarado antes de ser utilizado. ¿Dónde? Se aconseja hacerlo en el mismo lugar donde vaya a ser utilizado por primera vez. La declaración de un nombre implica especificar su tipo para informar al compilador a qué tipo de entidad se refiere. Algunos ejemplos son:

```
class CGrados;           // declaración previa de la clase CGrados
int nGradosCentigrados = 0; // definición de nGradosCentigrados
double sqrt(double);      // declaración de la función sqrt
extern int x;             // declaración de la variable x
float gfahr = 0;          // definición de la variable gfahr
```

Una declaración es una definición, a menos que no haya asignación de memoria como ocurre en las líneas siguientes:

```
double sqrt(double);
extern int x;
class CGrados;
```

La definición de una función (método o procedimiento) declara la función y además incluye el cuerpo de la misma. Por ejemplo:

```
float fx(int z)
{
    // Cuerpo de la función
}
```

La declaración o la definición de una variable puede realizarse a *nivel interno* (dentro del cuerpo de una función) o a *nivel externo* (fuera de toda definición de función), pero la definición de una función no puede realizarse dentro de otra función.

Sentencia simple

Una *sentencia simple* es la unidad ejecutable más pequeña de un programa C++. Las sentencias controlan el flujo u orden de ejecución. Una sentencia C++ puede formarse a partir de: una palabra clave (**for**, **while**, **if ... else**, etc.), expresiones, declaraciones o llamadas a funciones. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un punto y coma (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas una de otra por un punto y coma, aunque esta forma de proceder no es aconsejable porque va en contra de la claridad que se necesita cuando se lee el código de un programa.
- Una sentencia nula consta solamente de un punto y coma. Cuando veamos la sentencia **while**, podrá ver su utilización.

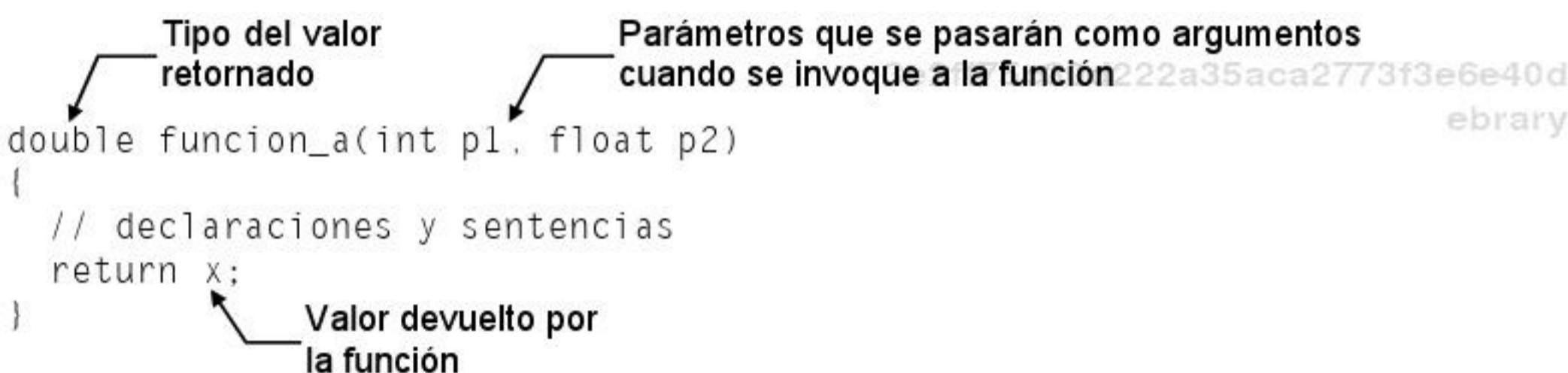
Sentencia compuesta o bloque

Una *sentencia compuesta* o bloque es una colección de sentencias simples incluidas entre llaves - { } -. Un bloque puede contener a otros bloques. Un ejemplo de una sentencia de este tipo es el siguiente:

```
{
    GradosFahrenheit = convertir(nGradosCentigrados);
    cout << setw(10) << nGradosCentigrados << " C";
    cout << setw(10) << setprecision(2)
        << GradosFahrenheit << " F\n";
    nGradosCentigrados += incremento;
}
```

Funciones

Una función (unidad de ejecución denominada también procedimiento o método) es un bloque de sentencias que ejecuta una tarea específica y al que nos referimos mediante un nombre. El bloque es el cuerpo de la función y el nombre del bloque es el nombre de la función. Cuando se escribe una función, además del cuerpo y del nombre de la misma, en general hay que especificar también los parámetros en los que se apoyan las operaciones que tiene que realizar y el tipo del resultado que retornará. Por ejemplo:



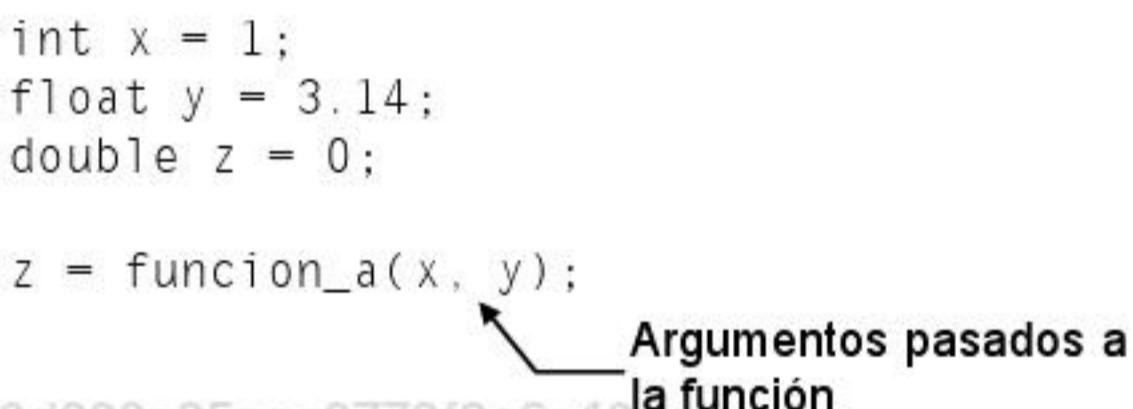
Este diagrama ilustra la declaración de una función en C++. Se muestra el siguiente código:

```
double funcion_a(int p1, float p2)
{
    // declaraciones y sentencias
    return x;
}
```

Con tres flechas dirigidas a diferentes partes del código:

- Una flecha apunta a "double" con la etiqueta "Tipo del valor devuelto".
- Otra flecha apunta a "p1" y "p2" con la etiqueta "Parámetros que se pasarán como argumentos cuando se invoque a la función".
- Una tercera flecha apunta a "x" con la etiqueta "Valor devuelto por la función".

Un argumento es el valor que se pasa a una función cuando ésta es invocada. Dicho valor será almacenado en el parámetro correspondiente de la función.



Este diagrama ilustra la invocación de la función "funcion_a" en el siguiente código:

```
int x = 1;
float y = 3.14;
double z = 0;

z = funcion_a(x, y);
```

Una flecha apunta a la llamada "funcion_a(x, y)" con la etiqueta "Argumentos pasados a la función".

Según lo estudiado hasta ahora, habrá notado que cuando necesitamos realizar un determinado proceso y existe una función de la biblioteca de C++ que lo puede hacer, utilizamos esa función; en otro caso, escribimos nosotros una función que lo realice. Evidentemente se trata de una unidad fundamental en la construcción de módulos y, en definitiva, en la construcción de programas. Por eso, vamos a describir cómo se declaran, cómo se definen y cómo se invocan. Posteriormente, en otros capítulos aprenderá más detalles.

Declaración de una función

La declaración de una función, también conocida como *prototipo de la función*, indica, además del nombre de la función, cuántos parámetros tiene y de qué tipo son, así como el tipo del valor devuelto. Su sintaxis es:

tipo - resultado nombre - función([lista de parámetros]);

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

El prototipo de una función es una plantilla que se utiliza para asegurar que una sentencia de invocación escrita antes de la definición de la función es correcta; esto es, que son pasados los argumentos adecuados para los parámetros especificados en la función y que el valor retornado se trata correctamente. Este chequeo de tipos y número de argumentos permite detectar durante la compilación si se ha cometido algún error.

Por ejemplo, la sentencia siguiente indica que cuando sea invocada la *funcion_a* hay que pasarla dos argumentos, uno entero y otro real, y que dicha función retornará un valor real cuando finalice su ejecución.

```
double funcion_a(int p1, float p2);
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

En conclusión, la declaración de una función permite conocer las características de la misma, pero no define la tarea que realiza.

La función **main** que hemos utilizado hasta ahora siempre devuelve un valor de tipo **int**. Quizás, se pregunte: ¿y por qué en este caso podemos prescindir de la sentencia **return**? Simplemente porque no hace falta, ya que no consta en el programa ninguna llamada a la misma.

La *lista de parámetros* normalmente consiste en una lista de identificadores con sus tipos, separados por comas. En el caso del prototipo de una función, se pueden omitir los identificadores por ser locales a la declaración. Por ejemplo:

```
double funcion_a(int, float);
```

En cambio, cuando se especifiquen, su ámbito queda restringido a la propia declaración; esto es, no son accesibles en otra parte (su presencia es sólo para aportar una sintaxis más clara).

De lo expuesto se deduce que los identificadores utilizados en la declaración de la función y los utilizados después en la definición de la misma no necesariamente tienen que nombrarse igual. Observe como ejemplo la declaración y la definición de la *funcion_a* mostradas a continuación:

```
double funcion_a(int x, float y); // declaración de la función  
  
int main()  
{  
    // ...  
  
    c = funcion_a(a, b); // se llama a la funcion_a  
    // ...  
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
double funcion_a(int p1, float p2) // definición de la función
{
    // Cuerpo de la función
}
```

Obsérvese que la *funcion_a* es llamada para su ejecución antes de su definición. Por lo tanto, el nombre de la función, el tipo de los argumentos pasados y el tipo del valor returned son verificados tomando como referencia la declaración de la función. Si la definición de la función se escribe antes de cualquier llamada a la misma, no es necesario escribir su declaración. Por ejemplo:

```
double funcion_a(int p1, float p2) // definición de la función
{
    // Cuerpo de la función
}

int main()
{
    // ...

    c = funcion_a(a, b); // se llama a la funcion_a
    // ...
}
```

La lista de parámetros puede también estar vacía. Por ejemplo:

```
float funcion_x(); // o bien
float funcion_x(void);
```

Así mismo, para indicar que una función no devuelve nada, se utiliza también la palabra reservada **void**. Por ejemplo:

```
void funcion_x(void)
```

Finalmente, cuando desde una función definida en nuestro programa se invoca a una función de la biblioteca de C++, ¿es necesario añadir su prototipo? Sí es necesario, exactamente igual que para cualquier otra función. Pero no se preocupe, esta tarea resulta sencilla porque las declaraciones de las funciones pertenecientes a la biblioteca estándar de C++, como **sqrt**, son proporcionadas por los ficheros de cabecera. Por eso, cuando un programa utiliza, por ejemplo, la función **sqrt**, observará que se incluye el fichero de cabecera *cmath* (en algunos compiladores basta con incluir *iostream*; a través de éste, se incluyen otros).

```
#include <cmath>
```

¿Cómo conocemos el fichero de cabecera en el que está el prototipo de una determinada función? Porque al especificar la sintaxis de las funciones de la biblioteca de C++, también se indica el fichero de cabecera donde está declarada.

Definición de una función

La definición de una función consta de una *cabecera* de función y del *cuerpo* de la función encerrado entre llaves.

```
tipo - resultado [ámbito::] nombre - función([parámetros formales])  
{  
    declaraciones de variables locales;  
    sentencias;  
    [return [()expresión[]]];  
}
```

Las variables declaradas en el cuerpo de la función son locales y por definición solamente son accesibles dentro del mismo.

El *tipo del resultado* especifica el tipo de los datos retornados por la función. Éste puede ser cualquier tipo primitivo o derivado, pero no puede ser una matriz o una función. Para indicar que una función no devuelve nada, se utiliza la palabra reservada **void**. Por ejemplo, la siguiente función no acepta argumentos y no devuelve ningún valor:

```
void mensaje(void)  
{  
    printf("Ocurrió un error al realizar los cálculos\n");  
}
```

Cuando una función devuelve un valor, éste es devuelto al punto donde se realizó la llamada a través de la sentencia **return**. La sintaxis de esta sentencia es la siguiente:

```
return [()expresión[]];
```

La sentencia **return** puede ser o no la última y puede aparecer más de una vez en el cuerpo de la función.

```
int funcion_y(int p1, float p2, char p3)  
{  
    // ...  
    if (a < 0) return 0;  
    // ...  
    return 1;  
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

En el ejemplo anterior, si $a < 0$ la función devuelve 0 dando su ejecución por finalizada; si $a \geq 0$ la ejecución continúa y devolverá 1.

Es un error especificar más de un elemento de datos a continuación de **return** (por ejemplo, *return x, y*) ya que el tipo del valor returned se refiere sólo a uno.

En el caso de que la función no retorne un valor (**void**), se puede especificar simplemente **return** cuando sea necesario, o bien omitir si se trata del final de la función. Por ejemplo:

```
void escribir(void)
{
    // ...
    if (a < 0) return;
    // ...
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

La *lista de parámetros formales* de una función está compuesta por las variables que reciben los valores especificados cuando es invocada. Consiste en una lista de cero, uno o más identificadores con sus tipos, separados por comas. El ejemplo siguiente muestra la lista de parámetros de *funcion_x* formada por las variables *p1*, *p2* y *p3*:

```
float [ámbito::]funcion_x(int p1, float p2, char p3)
{
    // Cuerpo de la función
}
```

Los parámetros formales de una función son variables locales a dicha función. Esto significa que sólo son visibles dentro de la función; dicho de otra forma, sólo tienen vida durante la ejecución de la función.

El *ámbito* es opcional. Indica el ámbito (clase, espacio de nombres, de los que hablaremos un poco más adelante, etc.) al que pertenece la función. Cuando no se especifica, la función pertenece al ámbito global; esto es, se trata de una función global. El siguiente ejemplo incluye una función *f* perteneciente a un espacio de nombres personalizado (*miesnom*) y otra función *f* perteneciente al espacio de nombres global. El espacio de nombres sólo contiene la declaración de la función, aunque podría haber contenido su definición. Por eso, al escribir la definición de esta función fuera del cuerpo del espacio de nombres hay que indicar el ámbito al que pertenece, de lo contrario sería tomada como una función global.

```
// Ámbito de una función
#include <iostream>
using namespace std;
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
namespace miesnom // espacio de nombres
{
    void f(); // declaración de la función f de miesnom
}

void miesnom::f() // definición de la función f de miesnom
{
    cout << "función f perteneciente al espacio de nombres miesnom\n";
}

void f() // definición de la función f global
{
    cout << "función f perteneciente al ámbito global\n";
}

int main()
{
    f();           // se invoca a la función f global
    miesnom::f(); // se invoca a la función f de miesnom
}
```

Llamada a una función

Llamar o invocar a una función es sinónimo de ejecutarla. La llamada se hará desde otra función o, como veremos más adelante, incluso desde ella misma. Dicha llamada está formada por el nombre de la función seguido de una lista de argumentos, denominados también *parámetros actuales*, encerrados entre paréntesis y separados por comas. Por ejemplo, las siguientes líneas son llamadas a distintas funciones:

```
c = funcion_a(a, b);
f();
miesnom::f();
```

Los argumentos *a* y *b* son las variables, lógicamente definidas previamente, que almacenan los datos que se desean pasar a la *funcion_a*. Una vez finalizada la ejecución de la *funcion_a*, el resultado devuelto por ésta es almacenado en la variable *c* especificada.

Función main

Todo programa C++ tiene una función denominada **main** y sólo una. Esta función es el punto de entrada al programa y también el punto de salida (en condiciones normales de ejecución). Su definición es como se muestra a continuación:

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
int main(int argc, char *argv[])
{
    // Cuerpo de la función
}
```

Como se puede observar, la función **main** está diseñada para devolver un entero (**int**) y tiene dos parámetros que almacenarán los argumentos pasados en la línea de órdenes cuando desde el sistema operativo se invoca al programa para su ejecución, concepto que estudiaremos posteriormente en otro capítulo.

Así mismo, C++ permite escribir **main** sin argumentos:

```
int main()
{
    // Cuerpo de la función
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

PASANDO ARGUMENTOS A LAS FUNCIONES

Cuando se llama a una función, el primer argumento en la llamada es pasado al primer parámetro de la función, el segundo argumento al segundo parámetro y así sucesivamente. Por defecto, todos los argumentos, excepto las matrices, son pasados *por valor*. Esto significa que a la función se pasa una copia del valor del argumento. Esto supone que la función invocada trabaje sobre la copia, no pudiendo así alterar las variables de donde proceden los valores pasados.

En el siguiente ejemplo, puede observar que la función **main** llama a la función *intercambio* y le pasa los argumentos *a* y *b*. La función *intercambio* almacena en *x* el valor de *a* y en *y* el valor de *b*. Esto significa que los datos *a* y *b* se han duplicado.

```
// Paso de parámetros por valor
#include <iostream>
using namespace std;

void intercambio(int, int); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(a, b); // a y b son pasados por valor
    cout << "a vale " << a << " y b vale " << b << endl;
}

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

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```

int z = x;
x = y;
y = z;
}

```

Veámoslo gráficamente. Supongamos que la figura siguiente representa el segmento de la memoria de nuestro ordenador utilizado por nuestro programa. Cuando se inicia la ejecución de la función **main** se definen las variables *a* y *b* y se inician con los valores 20 y 30, respectivamente. Cuando **main** llama a la función *intercambio*, se definen dos nuevas variables, *x* e *y*, las cuales se inician con los valores de *a* y *b*, respectivamente. El resultado es el siguiente:

a	b	x	y
20	30	20	30

Continúa la ejecución de *intercambio*. Se define una nueva variable *z* que se inicia con el valor de *x*. A continuación, en *x* se copia el valor de *y*, y en *y* el valor de *z* (el que tenía *x* al principio). Gráficamente el resultado es el siguiente:

a	b	x	y
20	30	30	20
	z		
	20		

Los parámetros *x* e *y* de la función *intercambio* son variables locales a dicha función; lo mismo sucede con *z*. Esto significa que sólo son accesibles dentro de la propia función. Esto se traduce en que las variables locales se crean cuando se ejecuta la función y se destruyen cuando finaliza dicha ejecución. Por lo tanto, una vez que el flujo de ejecución es devuelto a la función **main**, porque *intercambio* finalizó, el estado de la memoria podemos imaginarlo como se observa a continuación, donde se puede observar que *a* y *b* mantienen intactos sus contenidos, independientemente de lo que ocurrió con *x* y con *y*:

a	b
20	30

Si lo que se desea es alterar los contenidos de los argumentos especificados en la llamada, entonces hay que pasar dichos argumentos *por referencia*. Esto es, a la función hay que pasarla la *dirección* de cada argumento y no su valor, lo que exi-

ge que los parámetros formales correspondientes sean punteros. Para pasar la dirección de un argumento utilizaremos el operador &.

Aclaremos esto apoyándonos en el ejemplo anterior. Si lo que queremos es que el intercambio de datos realizado por la función *intercambio* suceda sobre las variables *a* y *b* de **main**, la función *intercambio* lo que tiene que conocer es la posición física que ocupan *a* y *b* en la memoria; de esta forma podrá dirigirse a ellas y alterar su valor. Esa posición física es lo que llamamos dirección de memoria.

Recuerde, en el capítulo 2 se expuso que para que una variable pueda contener una dirección (una dirección es un valor ordinal) hay que definirla así: *tipo *var*. Esta variable recibe el nombre de puntero (apunta al dato) porque su contenido es la posición en la memoria de un determinado dato, no el dato.

Atendiendo a lo expuesto, modifiquemos el ejemplo anterior como se muestra a continuación:

```
// Paso de parámetros por referencia
#include <iostream>
using namespace std;

void intercambio(int *, int *); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(&a, &b); // a y b son pasados por referencia
    cout << "a es " << a << " y b es " << b << endl;
}

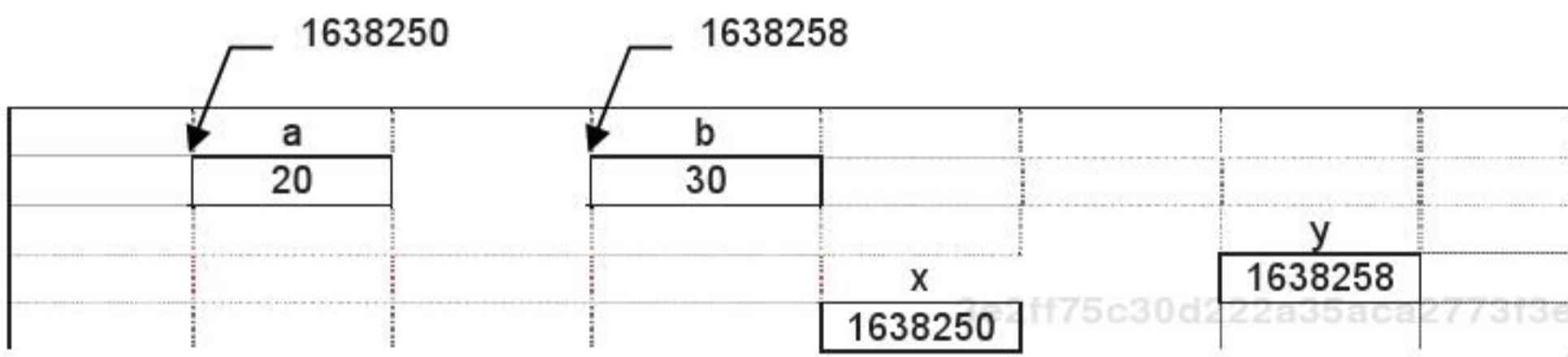
void intercambio(int *x, int *y)
{
    int z = *x; // z = contenido de la dirección x
    *x = *y;    // contenido de x = contenido de y
    *y = z;     // contenido de y = z
}
```

En el ejemplo expuesto podemos ver que la función *intercambio* tiene dos parámetros *x* e *y* de tipo *puntero a un entero* (*int **), que reciben las direcciones de *a* y de *b*, respectivamente (*&a* y *&b*). Esto quiere decir que cuando modificuemos el contenido de las direcciones *x* e *y* (**x* y **y*), indirectamente estamos modificando los valores *a* y *b*.

Cuando ejecutemos el programa, la función **main** definirá las variables *a* y *b* y llamará a la función *intercambio* pasando las direcciones de dichas variables como argumento. El valor del primer argumento será pasado al primer parámetro y

el valor del segundo argumento, al segundo parámetro. Suponiendo que esas direcciones son 1638250 y 1638258, respectivamente, lo que ocurre es lo siguiente:

```
x = &a; // x = 1638250
y = &b; // y = 1638258
```

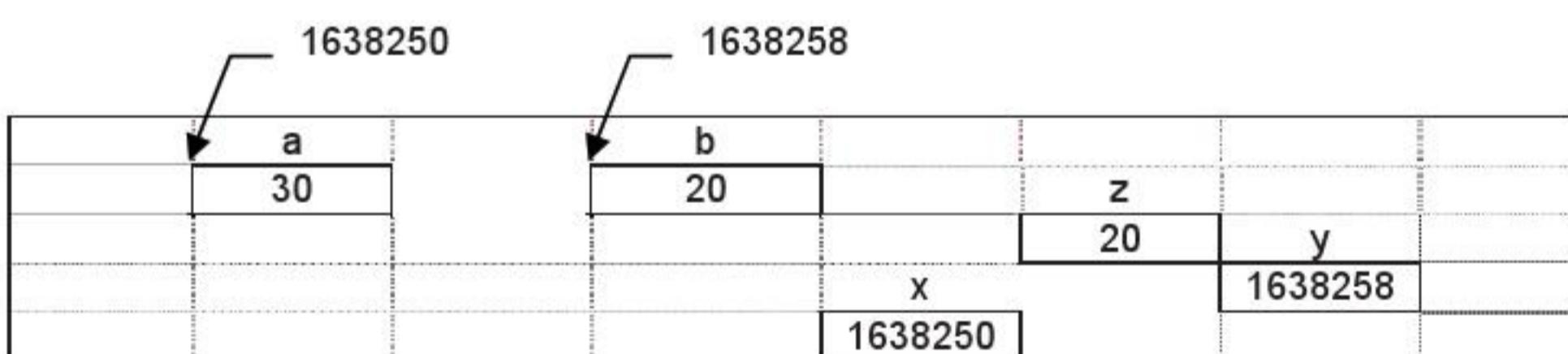


Ahora *x* apunta al dato *a*; esto es, el valor de *x* especifica el lugar donde se localiza *a* en la memoria. Análogamente, diremos que *y* apunta a *b*.

Observe que **x* (contenido de la dirección 1638250; esto es, lo que hay en la casilla situada en esta posición) hace referencia al mismo dato que *a* y que **y* hace referencia al mismo dato que *b*. Dicho de otra forma **x* y *a* representan el mismo dato, 20. Análogamente, **y* y *b* también representan el mismo dato, 30.

Según lo expuesto, cuando se ejecuten las sentencias de la función *intercambio* indicadas a continuación, el estado de la memoria se modificará como se indica en el gráfico mostrado a continuación:

```
int z = *x; /* z = contenido de la dirección x */
*x = *y;      /* contenido de x = contenido de y */
*y = z;       /* contenido de y = z */
```



Cuando la función *intercambio* finaliza, los valores de *a* y *b* han sido intercambiados y las variables *z*, *x* e *y*, por el hecho de ser locales, son destruidas.

Un ejemplo de la vida ordinaria que explique esto puede ser el siguiente. Un individuo A realiza un programa en C++. Posteriormente envía por correo electrónico a otro individuo B una copia para su corrección. Evidentemente, las correcciones que el individuo B realice sobre la copia recibida no alterarán la que tiene A. Esto es lo que sucede cuando se pasan parámetros por valor:

Ahora, si en lugar de enviar una copia, A le dice a B que se conecte a su máquina 190.125.12.78 y que corrija el programa que tiene almacenado en tal o cual carpeta, ambos, A y B, están trabajando sobre una única copia del programa. Esto es lo que sucede cuando se pasan los argumentos por referencia.

Otra forma de resolver el problema anterior es utilizando referencias en lugar de punteros. Recordemos que una referencia es un nombre alternativo (un sinónimo) para una variable.

Atendiendo a lo expuesto, modifiquemos el ejemplo anterior como se muestra a continuación:

```
// Paso de parámetros por referencia
#include <iostream>
using namespace std;
void intercambio(int&, int&); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(a, b); // a y b son pasados por referencia
    cout << "a es " << a << " y b es " << b << endl;
}

void intercambio(int& x, int& y)
{
    int z = x; // z = a
    x = y; // a = b
    y = z; // b = z
}
```

En el ejemplo expuesto podemos ver que la función *intercambio* tiene dos parámetros *x* e *y* que son referencias, esto es, *x* e *y* son sinónimos de *a* y de *b*, respectivamente. Esto quiere decir que cuando modificuemos *x* e *y*, estamos modificando los valores *a* y *b*. Esta forma de proceder es más sencilla, sin embargo presenta un inconveniente: observando solamente la llamada a la función *intercambio*, no se sabe si los argumentos son pasados por valor o por referencia.

Resumiendo, pasar parámetros por referencia a una función es hacer que la función acceda indirectamente a las variables pasadas, y a diferencia de cuando se pasan los parámetros por valor, no hay duplicidad de datos.

Cuando se trate de funciones de la biblioteca de C++, también se le presentarán ambos casos.

PROGRAMA C++ FORMADO POR VARIOS MÓDULOS

Según lo que hemos estudiado hasta ahora, no debemos pensar que todo programa tiene que estar escrito en un único fichero *.cpp*. De hecho en la práctica no es así, ya que además del fichero *.cpp*, intervienen uno o más ficheros de cabecera. Por ejemplo, en el programa inicial (conversión de grados) está claro que intervienen los ficheros *main.cpp* e *iostream*, pero, ¿dónde está el código de los elementos de la biblioteca de C++ invocados? Por ejemplo, el programa definido por *main* utiliza el objeto **cout** e invoca a funciones de la biblioteca de C++ como **setw** y **setprecision**, que no están en el fichero *main.cpp*. Estos elementos están definidos en otro fichero separado que forma parte de la biblioteca de C++, al que se accede durante el proceso de enlace para obtener el código correspondiente. Pues bien, nosotros podemos proceder de forma análoga; esto es, podemos optar por escribir el código que nos interese en uno o más ficheros separados (llamados también módulos o unidades de traducción) y utilizar para las declaraciones y/o definiciones uno o más ficheros de cabecera.

Un fichero fuente puede contener cualquier combinación de directrices para el compilador, declaraciones y definiciones. Pero, una función o una clase, en general un bloque, no puede ser dividido entre dos ficheros fuente. Por otra parte, un fichero fuente no necesita contener sentencias ejecutables; esto es, un fichero fuente puede estar formado, por ejemplo, solamente por definiciones de variables que son utilizadas desde otros ficheros fuentes.

Como ejemplo de lo expuesto, vamos a escribir el programa de conversión de grados de otra forma. Esto es, el código correspondiente a dicho programa lo escribiremos en tres ficheros:

- El fichero de cabecera *misfunciones.h* contendrá la declaración de la función *convertir*.
- El fichero *misfunciones.cpp* contendrá la definición de la función *convertir*.
- El fichero *main.cpp* contendrá, además de la función **main**, otras declaraciones que sean necesarias.

Según lo expuesto, escriba el código siguiente en un fichero llamado *misfunciones.h*.

```
#ifndef MISFUNCIONES_H_INCLUDED
#define MISFUNCIONES_H_INCLUDED

// Declaración de la función convertir
float convertir(int c);

#endif // MISFUNCIONES_H_INCLUDED
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

A continuación, escriba este otro código en un fichero llamado *misfunciones.cpp*.

```
// Función convertir grados centígrados a Fahrenheit
float convertir(int gcent)
{
    float gfahr;

    gfahr = 9.0 / 5.0 * gcent + 32;
    return (gfahr);
}
```

Y, finalmente, escriba el código siguiente en el fichero *main.cpp*.

```
/* Paso de grados Centígrados a Fahrenheit (F=9/5*C+32)
 */
#include <iostream>
#include <iomanip>
#include "misfunciones.h"
using namespace std;

int main()
{
    const int INF = -30; // límite inferior del intervalo de °C
    const int SUP = 100; // límite superior */

    // Declaración de variables locales
    int nGradosCentígrados = 0;
    int incremento = 6; // iniciar incremento con 6
    float GradosFahrenheit = 0;

    nGradosCentígrados = INF;
    cout << fixed; // formato en coma flotante
    while (nGradosCentígrados <= SUP)
    {
        // Se llama a la función convertir
        GradosFahrenheit = convertir(nGradosCentígrados);
        // Se escribe la siguiente línea de la tabla
        cout << setw(10) << nGradosCentígrados << " °C";
        cout << setw(10) << setprecision(2)
            << GradosFahrenheit << " °F\n";
        // Siguiente valor a convertir
        nGradosCentígrados += incremento;
    }
    return 0;
}
```

Observe estos ficheros en la figura siguiente (para añadir un nuevo fichero al proyecto ejecute la orden *File* del menú *File*):

```

1  /* Paso de grados Centígrados a Fahrenheit (F=9/5*C+32)
2  */
3  #include <iostream>
4  #include <iomanip>
5  #include "misfunciones.h"
6  using namespace std;
7
8  int main()
9  {
10     const int INF = -30; // límite inferior del intervalo de °C
11     const int SUP = 100; // límite superior */

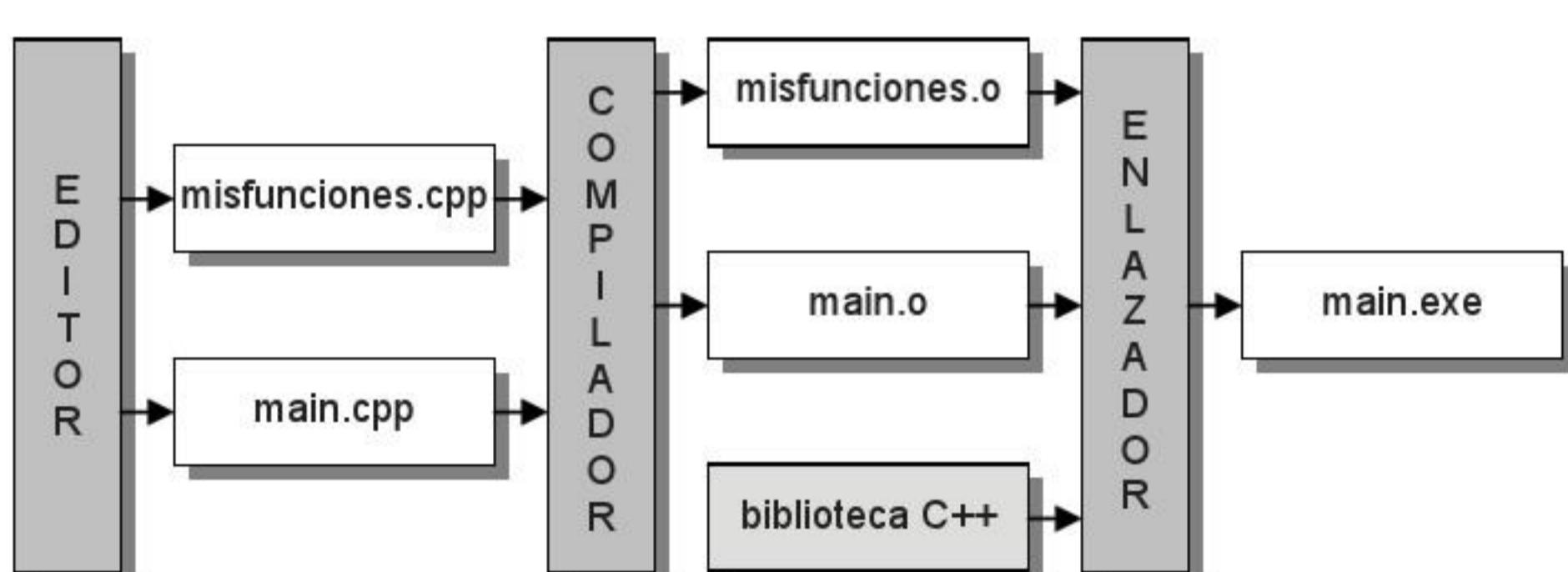
```

Para compilar un programa formado por varios ficheros, la orden *Build* del menú *Build* del EDI ejecutará una orden análoga a la siguiente:

```
g++ main.cpp misfunciones.cpp -o main.exe
```

En este caso, la orden *g++* junto con la opción *-o*, correspondiente al compilador C++ de GCC (licencia GNU), sería la que tendríamos que escribir en la línea de órdenes para compilar por separado cada uno de los ficheros *.cpp* (los ficheros de cabecera son incluidos por el preprocesador de C++) y, a continuación, enlazarlos para generar el programa ejecutable *main.exe*.

La siguiente figura muestra paso a paso la forma de obtener este fichero. En ella se indica que una vez editados los ficheros *misfunciones.cpp* y *main.cpp*, se compilan obteniéndose como resultado los ficheros objeto *misfunciones.o* y *main.o* (la extensión de estos ficheros depende también de la plataforma sobre la que estemos trabajando), los cuales se enlazan con los elementos necesarios de la biblioteca de C++, obteniéndose finalmente el fichero ejecutable *main.exe*.



ÁMBITO DE UN NOMBRE

Una declaración introduce un nombre (un identificador) en un ámbito, entendiendo por ámbito la parte de un programa donde dicho nombre puede ser referenciado. Un nombre puede ser limitado a un bloque, a un fichero, a una declaración o definición de función, a una clase, a un espacio de nombres, etc.

Nombres globales y locales

Cuando un nombre se declara en un programa fuera de todo bloque (`{ ... }`), es accesible desde su punto de definición o declaración hasta el final del fichero fuente. Este nombre recibe el calificativo de *global*.

Un *nombre global* existe y tiene valor desde el principio hasta el final de la ejecución del programa.

Si la declaración de un nombre se hace dentro de un bloque, el acceso al mismo queda limitado a ese bloque y a los bloques contenidos dentro de éste por debajo de su punto de declaración. En este caso, el nombre recibe el calificativo de *local* o *automático*.

Un *nombre local* existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definido. Cada vez que se ejecuta el bloque que lo contiene, el nombre local es nuevamente definido, y cuando finaliza la ejecución del mismo, el nombre local deja de existir. Por lo tanto, un nombre local es accesible solamente dentro del bloque al que pertenece.

El siguiente ejemplo muestra el ámbito de las variables `var1` y `var2`, dependiendo de si están definidas en un bloque o fuera de todo bloque.

En este ejemplo, analizando el ámbito de las variables, distinguimos cuatro niveles:

- El nivel externo (fuera de todo bloque).

Las variables definidas en este nivel son accesibles desde el punto de definición hasta el final del programa.

- El nivel del bloque de la función `main`.

Las variables definidas en este nivel solamente son accesibles desde la propia función `main` y, por lo tanto, son accesibles en los bloques 1 y 2.

- El nivel del bloque 1 (sentencia compuesta).

Las variables definidas en este nivel solamente son accesibles en el interior del bloque 1 y, por lo tanto, en el bloque 2.

- El nivel del bloque 2 (sentencia compuesta).

Las variables definidas en este nivel solamente son accesibles en el interior del bloque 2.

```
/* Variables globales y locales
 */
#include <iostream>
using namespace std;

// Definición de var1 como variable GLOBAL
int var1 = 50;

int main()
{ // COMIENZO DE main Y DEL PROGRAMA

    cout << var1 << endl; // se escribe 50

    { // COMIENZO DEL BLOQUE 1

        // Definición de var1 y var2 como variables
        // LOCALES en el BLOQUE 1 y en el BLOQUE 2
        int var1 = 100, var2 = 200;

        cout << var1 << " " << var2 << endl; // escribe 100 y 200

        { // COMIENZO DEL BLOQUE 2
            // Redefinición de la variable LOCAL var1
            int var1 = 0;

            cout << var1 << " " << var2 << endl; // escribe 0 y 200
        } // FINAL DEL BLOQUE 2

        cout << var1 << endl; // se escribe 100
    } // FINAL DEL BLOQUE 1

    cout << var1 << endl; // se escribe 50
} // FINAL DE main Y DEL PROGRAMA
```

En el ejemplo anterior se observa que una variable *global* y otra *local* pueden tener el mismo nombre, pero no guardan relación una con otra, lo cual da lugar a que la variable *global* quede ocultada (anulada) en el ámbito de la *local* del mis-

mo nombre. Como ejemplo observe lo que ocurre en el programa anterior con *var1*. No obstante, una variable *global* oculta se puede referenciar utilizando el operador de resolución de ámbito ::. Por ejemplo, modifique el programa anterior como se indica a continuación y observe los resultados.

```
// FINAL DEL BLOQUE 2

cout << var1 << endl; // se escribe 100
cout << ::var1 << endl; // se escribe 50
```

No hay ninguna forma de utilizar una variable *local* oculta.

CLASES DE ALMACENAMIENTO DE UNA VARIABLE

Por defecto, todas las variables llevan asociada una clase de almacenamiento que determina su accesibilidad y existencia. Los conceptos de accesibilidad y de existencia de las variables pueden alterarse por los calificadores:

auto	almacenamiento automático
register	almacenamiento en un registro
static	almacenamiento estático
extern	almacenamiento externo

Los calificadores **auto** y **register** pueden ser utilizados solamente con variables locales; el calificador **extern** puede ser utilizado sólo con variables globales; y el calificador **static** puede ser utilizado con variables locales y globales.

Calificación de variables globales

Una variable declarada a nivel global es una *definición* de la variable o una *referencia* a una variable definida en otra parte. Las variables definidas a nivel global son iniciadas a 0 por omisión.

Una variable global puede hacerse accesible antes de su definición (si esto tiene sentido) o en otro fichero fuente, utilizando el calificador **extern**. Esto quiere decir que el calificador **extern** se utiliza para hacer visible una variable global allí donde no lo sea.

El siguiente ejemplo formado por dos ficheros fuente, *uno.cpp* y *dos.cpp*, muestra con claridad lo expuesto. El fichero fuente *uno.cpp* define la variable *var* de tipo **int** y le asigna el valor 5. Así mismo, utiliza la declaración **extern int var** para hacer visible *var* antes de su definición.

```
// uno.cpp
#include <iostream>
using namespace std;

void funcion_1();
void funcion_2();

extern int var; // declaración de var que hace referencia a la
                // variable var definida a continuación
int main()
{
    var++;
    cout << var << endl; // se escribe 6
    funcion_1();           3e2ff75c30d222a35aca2773f3e6e40d
}                   ebrary

int var = 5; // definición de var

void funcion_1()
{
    var++;
    cout << var << endl; // se escribe 7
    funcion_2();
}
```

El fichero fuente *dos.cpp* utiliza la declaración **extern int var** para poder acceder a la variable *var* definida en el fichero fuente *uno.cpp*.

```
// dos.cpp
#include <iostream>
using namespace std;
3e2ff75c30d222a35aca2773f3e6e40d
ebrary extern int var; // declaración de var. Referencia a la variable var
                    // definida en el fichero uno.cpp
void funcion_2()
{
    var++;
    cout << var << endl; // se escribe 8
}
```

Observe que en el programa anterior, formado por los ficheros fuente *uno.cpp* y *dos.cpp*:

1. Existen tres declaraciones externas de *var*: dos en el fichero *uno.cpp* (una definición es también una declaración) y otra en el fichero *dos.cpp*.
2. La variable *var* se define e inicia a nivel global una sola vez; en otro caso se obtendría un error.

3. La declaración **extern** en el fichero *uno.cpp* permite acceder a la variable *var* antes de su definición. Sin la declaración **extern**, la variable global *var* no sería accesible en la función **main**.
4. La declaración **extern** en el fichero *dos.cpp*, permite acceder a la variable *var* en este fichero.
5. Si la variable *var* no hubiera sido iniciada explícitamente, C++ le asignaría automáticamente el valor 0 por ser global.

Si se utiliza el calificador **static** en la declaración de una variable a nivel global, ésta solamente es accesible dentro de su propio fichero fuente. Esto permite declarar otras variables con el mismo nombre en otros ficheros correspondientes al mismo programa. Como ejemplo, sustituya, en el fichero *dos.cpp* del programa anterior, el calificador **extern** por **static**. Si ahora ejecuta el programa observará que la solución es 6, 7, 1 en lugar de 6, 7, 8, lo que demuestra que el calificador **static** restringe el acceso a la variable al propio fichero fuente.

Calificación de variables locales

Una variable local declarada como **auto** (variable automática) solamente es visible dentro del bloque donde está definida. Este tipo de variables no son iniciadas automáticamente, por lo que hay que iniciarlas explícitamente. Es recomendable iniciarlas siempre.

Una variable local declarada **static** solamente es visible dentro del bloque donde está definida; pero, a diferencia de las automáticas, su existencia es permanente, en lugar de aparecer y desaparecer al iniciar y finalizar la ejecución del bloque que la contiene.

Una variable declarada **static** es iniciada solamente una vez, cuando se ejecuta el código que la define, y no es reiniciada cada vez que se ejecuta el bloque que la contiene, sino que la siguiente ejecución del bloque comienza con el valor que tenía la variable cuando finalizó la ejecución anterior. Si la variable no es iniciada explícitamente, C++ la inicia automáticamente a 0.

Una variable local declarada **register** es una recomendación al compilador para que almacene dicha variable, si es posible, en un registro de la máquina, lo que producirá programas más cortos y más rápidos. El número de registros utilizables para este tipo de variables depende de la máquina. Si no es posible almacenar una variable **register** en un registro, se le da el tratamiento de automática. Este tipo de declaración es válido para variables de tipo **int** y de tipo puntero, debido al tamaño del registro.

Una variable declarada **extern** a nivel local hace referencia a una variable definida con el mismo nombre a nivel global en cualquier parte del programa. La finalidad de **extern** en este caso es hacer accesible una variable global, en un bloque donde no lo es.

El siguiente ejemplo clarifica lo anteriormente expuesto.

```
// Variables locales: clases de almacenamiento
#include <iostream>
using namespace std;

void funcion_1();

int main()
{
    // Declaración de var1 que se supone definida en otro sitio
    // a nivel global
    extern int var1;
    // Variable estática var2: es accesible solamente
    // dentro de main. Su valor inicial es 0.
    static int var2;

    // var3 se corresponderá con un registro si es posible
    register int var3 = 0;

    // var4 es declarada auto, por defecto
    int var4 = 0;

    var1 += 2;

    // Se escriben los valores 7, 0, 0, 0
    cout << var1 << " " << var2 << " " << var3 << " " << var4 << endl;
    funcion_1();
}

int var1 = 5;

void funcion_1()
{
    // Se define la variable local var1
    int var1 = 15;

    // Variable estática var2; accesible sólo en este bloque
    static int var2 = 5;

    var2 += 5;
    // Se escriben los valores 15, 10
    cout << var1 << " " << var2 << endl;
}
```

En este ejemplo, la variable global *var1* se define después de la función **main**. Por eso, para hacerla accesible dentro de **main** se utiliza una declaración **extern**. La variable *var2* declarada **static** en **main** es iniciada, por defecto, a 0. La clase de almacenamiento de la variable *var3* es **register** y la de *var4*, **auto**.

En *funcion_1* se define la variable local *var1*; esta definición oculta a la variable global del mismo nombre haciéndola inaccesible dentro de este bloque. La variable *var2*, declarada **static**, es iniciada a 5. Esta definición no entra en conflicto con la variable *var2* de la función **main**, ya que ambas son locales. A continuación, *var2* es incrementada en 5. Entonces, la próxima vez que sea invocada *funcion_1*, iniciará su ejecución con *var2* igual a 10, puesto que las variables locales **static** conservan el valor adquirido de una ejecución para la siguiente.

ESPACIOS DE NOMBRES

El programa ejemplo que acabamos de presentar al comienzo de este capítulo consta de una función, *convertir*, y de la función **main**. Pero en la realidad, los programas son más grandes y, lógicamente, su código está distribuido en múltiples ficheros, cada uno de los cuales puede ser construido y mantenido por diferentes personas o grupos. Esto quiere decir que los desarrolladores deben tener un cuidado especial para no utilizar los mismos identificadores para sus variables, funciones, clases, etc., lo cual resulta difícil y generalmente termina en un gasto de tiempo especial para solucionar estas colisiones entre nombres.

C++ estándar tiene un mecanismo para prevenir este problema: los espacios de nombres. Un *espacio de nombres* es un concepto muy básico y simple: es un ámbito. Por lo tanto, los ámbitos locales y los globales, de los que hemos hablado anteriormente, son espacios de nombres. No obstante, aunque los nombres puedan pertenecer a funciones o a clases, los nombres de funciones globales, los nombres de clases o los nombres de variables globales pertenecen todavía a un único espacio de nombres global. En un proyecto grande, la falta de control sobre estos nombres puede causar problemas. Para adelantarnos a estos problemas, es posible subdividir el espacio de nombres global en espacios personalizados utilizando la palabra reservada **namespace**.

La creación de un espacio de nombres es muy sencilla:

```
namespace nombre_del_espacio
{
    // Declaraciones
}
```

Esto produce un espacio de nombres que empaqueta las declaraciones especificadas en el bloque que define. No es necesario poner un punto y coma después de la llave de cierre.

Los espacios de nombres pueden aparecer solamente en un ámbito global. Por ejemplo:

```
#include <iostream>

namespace LibXXX
{
    int x;
    void f1();
    int f2(bool b);
}

int main()
{
    // ...
}
```

Un espacio de nombres puede estar definido en varios ficheros de cabecera. Por ejemplo:

```
// Espacio de nombres definido en fcab01.h
namespace LibXXX
{
    int x;
    void f1();
    int f2(bool b);
}

// Añadir más nombres a LibXXX en fcab02.h
namespace LibXXX
{
    extern int x;
    int y;
    void f3();
}
```

También es posible declarar un alias de un espacio de nombres:

```
namespace miespnom = LibXXX;
```

En las bibliotecas de clases, los espacios de nombres se utilizan también con frecuencia para empaquetar clases relacionadas entre sí de alguna forma, o bien simplemente para incluir a otros espacios de nombres. Por ejemplo, el código siguiente crearía el espacio de nombres **System** que empaqueta las clases A y B y el

espacio de nombres **Windows**, que a su vez empaqueta el espacio **Forms** que incluye las clases C y D:

```
namespace System
{
    class A {};
    class B {};
    namespace Windows
    {
        namespace Forms
        {
            class C {};
            class D {};
        }
    }
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Para referirnos a un elemento de un espacio de nombres, tenemos que hacerlo utilizando su nombre completo, excepto cuando el espacio de nombres haya sido declarado explícitamente, como veremos a continuación. Por ejemplo, *System::Windows::Forms::C* hace referencia a la clase *C* del espacio de nombres *System::Windows::Forms*.

```
// ...
int main()
{
    System::Windows::Forms::C obj;
    // ...
    std::cout << LibXXX::x << std::endl;
    // ...
}
```

Toda la biblioteca estándar de C++ está definida en un único espacio de nombres llamado **std** (estándar).

Directriz using

Un programa puede hacer referencia a un nombre de un espacio de nombres de dos formas:

1. Utilizando como prefijo el nombre del espacio en todas las partes del código donde haya que referirse a él. Así, para referirnos a **cout** y **endl** de **std** y a *x* de *LibXXX* escribiríamos:

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

```
std::cout << LibXXX::x << std::endl;
```

2. Indicando al compilador el espacio de nombres donde está el nombre referenciado, lo que posibilita referirse a él sin el nombre de su espacio. Para ello utilizaremos la directriz **using**. Por ejemplo:

```
using namespace std; // usar el espacio de nombres std  
  
int main()  
{  
    using namespace LibXXX; // usar el espacio de nombres LibXXX  
    // ...  
    cout << x << endl;  
    // ...  
}
```

3e2ff75c30d222a35aca2773f3e6e40d

ebrary

Como se puede comprobar en el ejemplo anterior, declarar un espacio de nombres permite al programa referirse a sus nombres más tarde sin utilizar el nombre del espacio. Esto es, la directriz **using** sólo indica al compilador C++ dónde encontrar los nombres, no trae nada dentro del programa C++ actual.

En el caso concreto del ejemplo expuesto, si eliminamos la directriz *using namespace std*, el compilador mostrará dos errores para indicar que no puede encontrar los nombres **cout** y **endl**.

La expresión *using namespace miespacio* hace referencia a todos los elementos del espacio de nombres *miespacio*. Para permitir el uso de un elemento determinado, por ejemplo de la variable *x* de *LibXXX*, utilizaremos una declaración **using** así:

```
void f1()  
{  
    int x;  
    using LibXXX::x; // error: x ya está declarado  
    // ...  
}
```

Una declaración **using** añade un nombre a un ámbito local; una directriz **using** no, simplemente hace accesibles los nombres de un espacio de nombres.

La directriz **using** puede utilizarse también dentro de un bloque (incluso en el de un espacio de nombres) para hacer disponibles dentro de éste todos los nombres de un espacio de nombres. Esta práctica se recomienda frente al uso de las directrices **using** globales, dejando el uso de éstas últimas para casos excepcionales como, por ejemplo, para la migración de código ya escrito.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

EJERCICIOS RESUELTOS

Escriba un programa que utilice:

1. Una función llamada *par_impar* con un parámetro de tipo **int**, que visualice “par” o “impar” respecto del valor pasado como argumento.
2. Una función llamada *positivo_negativo* con un parámetro de tipo **int**, que visualice “positivo” o “negativo” respecto del valor pasado como argumento.
3. Una función llamada *cuadrado* con un parámetro de tipo **int**, que devuelva el cuadrado del valor pasado como argumento.
4. Una función llamada *cubo* con un parámetro de tipo **int**, que devuelva el cubo del valor pasado como argumento.
5. Una función llamada *contar* sin parámetros, que devuelva el siguiente ordinal al último devuelto; el primer ordinal devuelto será el 1.

La función **main** llamará a cada una de estas funciones para un valor determinado y finalmente, utilizando la función *contar*, realizará una cuenta hasta 3.

```
// funciones.cpp - Cómo es un número. Contar.  
#include <iostream>  
using namespace std;  
  
void par_impar(int);  
void positivo_negativo(int);  
int cuadrado(int);  
int cubo(int);  
int contar(void);  
  
int main()  
{  
    int n = 10;  
  
    par_impar(n);  
    positivo_negativo(n);  
    cout << "cuadrado de " << n << " = " << cuadrado(n) << endl;  
    cout << "cubo de " << n << " = " << cubo(n) << endl;  
    cout << "\nContar hasta tres: ";  
    cout << contar() << " ";  
    cout << contar() << " ";  
    cout << contar() << endl;  
}  
  
void par_impar(int n)  
{  
    cout << n << " es " << ((n % 2 == 0) ? "par" : "impar") << endl;  
}
```

```
void positivo_negativo(int n)
{
    cout << n << " es " << ((n >= 0) ? "positivo" : "negativo") << endl;
}

int cuadrado(int n)
{
    return n * n;
}

int cubo(int n)
{
    return n * n * n;
}

int contar(void)
{
    static int n = 1;
    return n++;
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Ejecución del programa:

```
10 es par
10 es positivo
cuadrado de 10 = 100
cubo de 10 = 1000
```

Contar hasta tres: 1 2 3

Este programa, partiendo de un valor *n* visualiza, invocando a las funciones correspondientes, si este valor es par o impar, positivo o negativo, su cuadrado, su cubo y finalmente realiza una cuenta hasta 3.

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

- 1) Un programa C++ compuesto por dos o más funciones, ¿por dónde empieza a ejecutarse?
 - a) Por la primera función que aparezca en el programa.
 - b) Por la función **main** sólo si aparece en primer lugar.
 - c) Por la función **main** independientemente del lugar que ocupe en el programa.
 - d) Por la función **main** si existe, si no por la primera función que aparezca en el programa.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary