Capítulo 1

Funciones

1.1. Introducción.

Las funciones nos permiten encapsular un trozo de código como una única operación. Son una herramienta fundamental para la programación modular, que nos permite resolver problemas complejos de manera más sencilla[1].

La forma de la definición de una función es la siguiente

```
tipo_función nombre_función(parámetros formales)
{
... código función (cuerpo) ...
}
```

donde

- *tipo_función* es el tipo del dato devuelto al finalizar la función.
- *nombre_función* identifica a la función. Este nombre es un identificador válido que nos permite referenciarla.
- *parámetros formales* es una lista de cero o más declaraciones separadas por comas en las que se especifica, uno a uno, los identificadores y tipos de las variables que se pasan desde el código que llama a la función.
- *código función* es el conjunto de sentencias que implementan la acción para la que se crea la función. También se denomina el cuerpo de la función. Como vemos es un bloque de sentencias delimitado, como sabemos, por el par de caracteres {}. Las variables declaradas en este bloque son locales a la función, y por tanto no se conocen fuera de ella (se crean al entrar en la función y se destruyen al salir).

A la primera linea, también se denomina la cabecera o declaración de la función y especifica, sintácticamente, la forma en que se puede realizar una llamada a la función.

Para realizar una llamada a una función, se deberá escribir

```
nombre_función(parámetros actuales)
```

en una expresión o como una sentencia independiente. El nombre de la función es idéntico al de la defínción, para que el compilador identifique la función a la que se llama, mientras que los *parámetros actuales* son los datos que se pasan a la función para que conozca los valores de los parámetros formales.

[1] Una discusión más extensa sobre el uso de funciones en la programación se ofrece más adelante, en el capítulo dedicado a la abstracción (ver pág **??**)

Un ejemplo es la siguiente función

```
float maximo3floats (float f1, float f2, float f3)
{
   float f;

   if (f1>=f2 && f1>=f3)
      f= f1;
   else if (f2>=f1 && f2>=f3)
            f= f2;
         else f= f3;

   return f;
}
```

que tiene como objetivo el cálculo del máximo valor de entre tres flotantes. Esta función se puede utilizar en otro trozo de código independiente como el siguiente

```
float maximo3floats (float f1, float f2, float f3);

....
float f,g,h,res;
...
f= 1.0;
g= 2.0;
h= 3.0;
res= 2*maximo3floats (f,g,h);
...
```

de manera que decimos que en este código hay una llamada a la función *maximo3floats*.
En este ejemplo podemos observar que

- Antes de realizar la llamada a la función, se presenta una primera línea, que corresponde a su cabecera. En un programa *C++* es necesario informar al compilador de la sintaxis de la función. De esta forma, cuando el compilador analiza la línea donde se realiza la llamada, puede comprobar si el número de argumentos y su tipo son válidos. Nótese que si la definición de la función aparece antes de esta llamada, el compilador ya tiene la información para las comprobaciones y no es necesaria la cabecera.

- El tipo de la función es *float*, es decir, el valor que devuelve como resultado será de ese tipo. El resultado se devuelve mediante la sentencia *return expresión*. Por lo tanto, esta sentencia tiene dos objetivos, en primer lugar determina que la función ha finalizado, y en segundo lugar indica la expresión que corresponde al valor que se devuelve como resultado.

  Un caso especial es el de una función de tipo *void* que no devuelve ningún valor. Éste tipo de función finaliza cuando se ejecute la última sentencia del cuerpo (llega hasta la llave que indica fin del bloque de código de la función) o mediante una sentencia *return* sin especificar ningún valor a devolver.

- El nombre es *maximo3floats*, que como vemos se utiliza en el trozo de código para hacer referencia a la función.

- Los parámetros formales son *f1,f2,f3* de tipo *float*. Estos identificadores se utilizan en la función para describir la forma en que se debe llevar a cabo la acción implementada. Sin embargo, y como es lógico, las variables que se utilizan desde el trozo de código que llama a la función son distintas (*f,g,h*). En este caso hablamos de parámetros actuales ya que son los que en ese momento se quieren utilizar en la función. La correspondencia es mediante la asignación uno a uno y de izquierda a derecha de las dos listas de parámetros. La forma en que se pasan los valores se estudia en detalle en las secciones siguientes.

- El código compara los valores de los tres parámetros formales y calcula el máximo, almacenándolo en *f*, una varialbe local al módulo (no se conoce fuera de éste). La sentencia *return* es la última que se ejecuta, devolviendo como resultado el valor de esta variable. Como vemos en el trozo de código que llama a la función, podemos entender que el valor devuelto por una función toma el lugar de la llamada de manera que en nuestro ejemplo, el máximo se multiplica por dos y el resultado se asigna a la variable *res*.

Es importante comentar la forma en que una función lleva a cabo la acción para la que ha sido diseñada. Inicialmente, es posible pensar que el objetivo último de una función es la devolución de un dato mediante la sentencia *return*. Sin embargo, el conjunto de sentencias que componen su código pueden tener como efecto otro tipo de resultados como:

- Tienen acceso a zonas de memoria que conoce el código que llama a la función y, por tanto, los valores que escriba en ellas se podrán usar como resultados por el trozo de código que la llama.
- Tienen acceso a otros dispositivos del sistema, por tanto, el efecto que se obtiene en la llamada no es en forma de dato. Por ejemplo, pueden rebobinar una cinta, inicializar una impresora, o escribir en pantalla.

Estos ejemplos, justifican claramente la posibilidad de declarar una función de tipo *void* cuando no se quiere devolver ningún valor en especial. En este caso, como es de esperar, el código que llama tiene una sentencia que es, únicamente, la llamada a la función[2].

Finalmente, podemos considerar funciones que llevan a cabo alguna acción y además la devolución de algún valor. Consideremos por ejemplo, una función que tiene como misión escribir cierto dato en la pantalla y a la vez devolver algún valor indicando si ha habido éxito. Un trozo de código que llame a esta función puede tener garantías de que la operación se realizará con éxito y no necesita utilizar el valor devuelto. En *C++* es posible una sentencia que tenga únicamente la llamada a la función, de forma que el compilador entiende que se quiere obviar el valor devuelto.

1.1.1. La función *main*.

La ejecución de un programa en *C++* consiste en la ejecución de las sentencias que componen una función específica, con nombre *main*. Así el programa comienza cuando el sistema operativo transfiere el control a esta función y finaliza, salvo terminación anormal, por ejemplo, con funciones como *exit* y *abort* (véase la librería estándar de *C*), cuando la función *main* acaba[3].

La forma y uso de esta función es idéntica al resto de funciones, excepto que es el sistema operativo el que llama a esta función, y es al sistema operativo al que se devuelve el resultado de esta función. La cabecera es la siguiente

```
int main (int argc, char *argv[])
```

donde podemos observar que

- El tipo devuelto es un entero. Este entero corresponde a un valor que informa, al sistema operativo o programa que ha lanzado su ejecución, sobre la terminación.

  Por ejemplo, si devolvemos un cero, indicará que el programa ha realizado todas las acciones con éxito y ha finalizado correctamente. Un valor distinto puede indicar alguna causa de error en la ejecución y por tanto una terminación fallida (por ejemplo, no existe algún fichero de entrada, falta espacio en disco o en memoria principal, etc.).

- Los datos de entrada se ofrecen a través de los parámetros *argc* y *argv*. Cuando lanzamos un programa desde un intérprete de comandos, escribimos el nombre y, posiblemente, una serie de cadenas que corresponden a los datos que ofrecemos directamente en la línea de comandos.

  Por ejemplo, cuando queremos hacer una copia de un fichero a un directorio escribimos en primer lugar el comando de copia (*cp*), después una cadena que corresponde al nombre del fichero a copiar

[2] En algunos lenguajes se diferencia entre módulos que devuelven algún valor y los que no, a los primeros se denomina *funciones* y a los últimos *procedimientos*

[3] En realidad, hay funciones que pueden ejecutarse antes y después de *main* (constructores y destructores de objetos globales). A pesar de ello, por ahora podemos asumir que el programa comienza y termina con la función *main*.

y finalmente otra con el nombre del directorio donde copiar. Cuando se ejecuta el programa de copia debe tener acceso a las dos cadenas adicionales. Esto se lleva a cabo mediante *argc* y *argv*.

El primero de ellos es un entero con el número de cadenas, incluída la del nombre del programa. El segundo es una matriz de *argc* cadenas, que corresponden a las cadenas que se han escrito en la línea de comandos para lanzar el programa. Así, en nuestro ejemplo de copia de un fichero *argc* valdría 3 y *argv* tendría 3 cadenas (desde 0 a 2) con el nombre del programa, del fichero a copiar y el directorio donde copiar, respectivamente.

Parámetros de main y tipo string.

Es importante destacar que, aunque *C++* incluye el tipo de dato estándar *string*, mantiene la sintaxis y semántica en los parámetros de la función *main* de *C* . Es por ello, que las cadenas que se pasan a esta función se representan como en este lenguaje, es decir, como una matriz de tipo *char* que termina en el carácter especial de terminación de cadena '\0'.

A pesar de ello, y si es conveniente, es muy fácil transformarlos al tipo *string*. Recordemos que la asignación de un vector de caracteres (*char **) terminado en el carácter de fin '\0' se puede asignar al tipo *string* si ningún problema. Así, como ejemplo de comportamiento de la función *main*, proponemos

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
   string par;

   cout << ``Parámetros:'' << endl;
   for (int i=0;i<argc;++i) {
      par= argv[i];
      cout << par << endl;
   }
   return 0;
}
```

Como podemos observar, en un simple ejemplo, en el que se muestra como podemos realizar la transformación. Por supuesto, obtenemos un resultado equivalente si eliminamos el tipo *string* y pasamos directamente el argumento *argv[i]* a *cout*.

1.1.2. La responsable de que todo funcione: La Pila.

Un programador debe conocer perfectamente la forma en que funcionan y se gestionan las llamadas a funciones. En principio puede parecer un tema que, correspondiendo a detalles internos que resuelve el compilador, no tiene que ser necesario para escribir en un lenguaje de programación. Sin embargo es muy importante, para implementar, optimizar y saber depurar los programas. En esta sección se ofrece una pequeña introducción para que el lector pueda leer fácilmente el resto del capítulo, a lo largo del cual se irán mostrando y aclarando los detalles del mecanismo que nos ofrece la posibilidad de estructurar el código en funciones.

En un trozo de código de una función, se tiene acceso a un conjunto de variables, globales o externas y locales o internas a la función. Las variables globales existen independientemente de la función, pero las locales empiezan a existir cuando comienza su ejecución y dejan de existir cuando termina[4]. Al conjunto de variables formales, en la cabecera de la función, se las considera locales, ya que existen únicamente cuando la función se está ejecutando.

Consideremos que tenemos un trozo de código que realiza una llamada a una función. Antes de la llamada, podemos considerar que existen un conjunto de identificadores que denominamos *contexto*, que

[4]Estas variables se dicen de persistencia automática y no sólo tienen sentido a nivel de función, sino también de bloque, aunque en este momento nos centraremos en el primer caso.

son accesibles en ese punto del programa. Cuando se llega al punto de la llamada a función, el programa debe detener la ejecución de ese trozo de código ya que ha de resolverse el conjunto de acciones que corresponden a la función, para poder seguir avanzando.

En el momento de la llamada, podemos decir que el programa transfiere los datos desde los parámetros actuales a los parámetros formales (más adelante se describe detalladamente la forma en que se realiza esta operación) y cambia el contexto. Ahora no se conocen los mismos identificadores que en el trozo que llama, sino que algunos ya no son accesibles (por ejemplo, las variables locales) y aparecen nuevas variables para las que es necesario reservar un trozo de memoria donde almacenarán los valores correspondientes. Si dentro de esta función llamada aparece una nueva llamada a una segunda función, vuelve a ser necesario dejar "aparcado" el trabajo de la primera, cambiar de nuevo el contexto a la segunda, reservando de nuevo espacio para las nuevas variables si es necesario.

Por tanto, cada vez que aparece una llamada a una función, es necesario guardar la información sobre el punto donde nos encontrábamos y buscar espacio para poder ejecutar la función que se llama. Para poder gestionar este funcionamiento, el programa dispone de una zona de memoria que se denomina *Pila*, y cuyo funcionamiento es dinámico (crece y decrece) de forma que el programa cuando tiene que almacenar el contexto en un punto y reservar espacio para las variables de una nueva función, utiliza la pila. Así podemos imaginarla como una "pila de platos"(en nuestro caso, corresponden a *contextos*) en la que las operaciones básicas consisten en almacenar datos en el tope (apilar nuevos platos) o eliminarlos, pudiendo así acceder por tanto a los últimos que se apilaron. Nótese que cuando una función termina, el espacio reservado en la pila para su gestión se destruye y se pasa el control a la función que la llamó, es decir, la última que se "apiló".

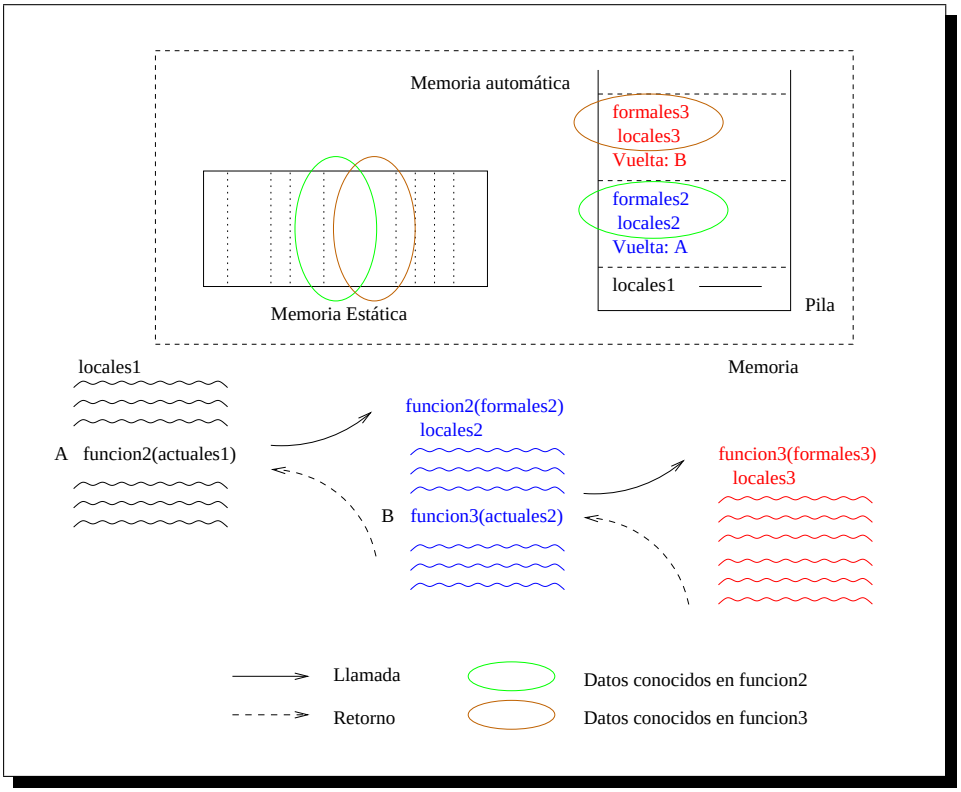

Figura 1.1: Funcionamiento de la pila.

En la figura 1.1 se muestra gráficamente el funcionamiento de la pila que acabamos de discutir. Se representa la memoria estática y automática así como un esquema de los trozos de código involucrados en las llamadas.

El contenido de la pila indica que corresponde a un momento de ejecución de la *funcion3* ya que podemos ver que las variables de su contexto están creadas en la pila.

La secuencia de pasos podría ser:

1. Estamos en un punto situado en el primer bloque de código, donde aparecen las variables *locales1*

y otras que no aparecen representadas. La pila no contiene las representadas en azul y rojo.

2. Llamamos a *funcion2*. En la pila se crea la información que hemos representado en azul. Los parámetros *actuales1* se pasan a los *formales2*, utilizando los mecanismos que veremos en la siguiente sección. En esta función, se conocen tanto variables globales como las que aparecen en su contexto (elipses verdes).

3. Llamamos a *funcion3*. En la pila se crea la información que hemos representado en rojo. Los parámetros *actuales2* se pasan a los *formales3*. En esta función, se conocen tanto variables globales como las que aparecen en su contexto (elipses marrones).

4. Volvemos de la *funcion3*. Se destruye la información de la pila representada en rojo y se recupera el lugar desde el que se hizo la llamada. (en este caso, el punto *B*). Se eliminan los datos (decrece el tamaño de la pila). Nótese que podríamos utilizar la pila para almacenar un valor devuelto por la función, de forma que la función que llama *funcion2* recupere el resultado desde esta.

5. Volvemos de la *funcion2*. Se llevan a cabo operaciones similares.

A continuación se estudia detenidamente los detalles sobre las distintas posibilidades en el paso de parámetros y devolución de resultados, haciendo referencia a la pila como la estructura que soporta el funcionamiento de las llamadas a la función. Más adelante, se estudiará el tipo de dato *Pila* y el lector podrá darse cuenta de la forma en que podemos construir programas que utilicen pilas gestionadas por nosotros mismos.

1.2. Paso de parámetros y devolución de resultados

En esta sección vamos a estudiar la forma en que se puede pasar parámetros y devolver resultados en *C++* . Es interesante destacar que los mecanismos que ofrece *C* están presentes con un funcionamiento idéntico, añadiéndose algunos más en *C++* para establecer, finalmente, un conjunto cómodo y potente de posibilidades. Dado que todos ellos son igualmente válidos, los presentamos todos dividiéndolos en dos conjuntos, los que ya existían en *C* y los que, adicionalmente, incorpora *C++* .

Representaremos gráficamente la situación de la memoria y el efecto que tienen las operaciones sobre ella. Para ello, simplificamos mostrando dos zonas diferenciadas:

1. En negro, el conjunto de datos accesible desde el trozo de código que llama a la función. Nótese, que este conjunto está compuesto tanto por variables que están en la pila como en la zona de memoria estática. Sin embargo, para entender el paso y devolución es recomendable considerarlo una zona de memoria diferenciada.

2. En rojo, el conjunto de datos en la pila. Se muestran solo los datos relevantes, es decir, los que aparecen como efecto de la llamada concreta a la función.

1.2.1. Paso de parámetros y devolución de resultados en C

Supongamos una función *máximo* que toma dos valores enteros y devuelve como resultado otro entero. Consideremos un trozo de código que llama a esta función y en el que se declaran 3 variables enteras *max,ant,nue* de manera que en *max* almacenamos el resultado de multiplicar 5 por el máximo de las otra dos. En la figura 1.2 representamos esta situación.

Podemos observar la representación del código principal y la llamada a la función *máximo*. Además representamos:

- En la parte superior una zona de memoria con 3 localizaciones que almacenan las variables *max, ant, nue* (en las direcciones *0x20, 0x30, 0x40*).

- Encima de la fución, otra zona de memoria[5](en rojo), donde encontramos también 3 localizaciones de las variables locales que son necesarias para ejecutar la función. Esta zona empieza a existir cuando se comienza la ejecución de la función y deja de existir cuando se devuelve el control al trozo de código que la ha llamado.

[5]Esta zona se localiza en la pila.

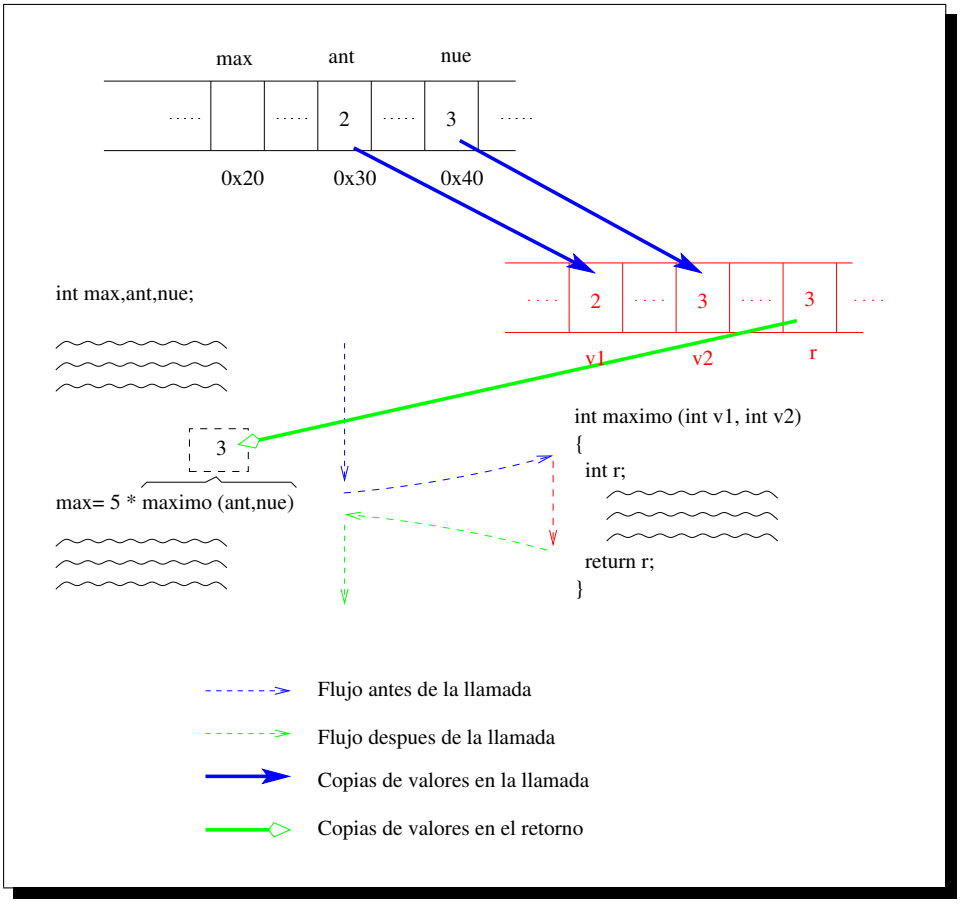

Figura 1.2: Paso por valor y retorno de valor.

- Encima de la llamada a la función una localización temporal de memoria donde se almacena el valor devuelto por la función y que es recogido por el trozo de código que la ha llamado.
- Finalmente un conjunto de líneas de distinto trazo y que vienen descritas en la parte inferior de la figura.

En este ejemplo, se muestra el paso por valor y la devolución del resultado de una función tal como se conoce en *C*. Es interesante ver que en la llamada a la función se crea la zona de memoria para almacenar tanto los valores de los parámetros como las variables locales. Los valores se duplican y por tanto dentro de la función, la modificación de *v1,v2* no afectará a los valores de las variables *ant,nue* del trozo que llama.

A lo largo de la ejecución de la función, se carga el valor de la varible local *r*. Cuando termina la función, se devuelve como resultado su valor y se almacena en una zona temporal[6] desde donde la recoge el código que llama a la función. Nótese que se indica con una flecha de copia tras la ejecución. Después de esta copia la memoria local de la ejecución de la función deja de existir y por tanto los valores ahí almacenados dejan de poder usarse. Una vez finalizada, el valor temporal (3) se multiplica por 5 y se almacena en *max* (que acaba con el valor 15)

Por tanto, en la llamada como en la vuelta se ha realizado la copia de los **valores** que se pasan. Esta forma de comunicación es la que se denomina **"Paso por valor"**.

Ahora bien, si se desea modificar uno de los parámetros actuales, es necesario buscar un mecanismo para que la función acceda a la variable original. En este segundo caso, mostramos el uso de punteros para poder modificar dichos valores[7]. En la figura 1.3 se muestra esta situación.

[6] Normalmente, también en la pila.

[7] Aunque este mecanismo es habitual en los programas *C* , la programación en *C++* minimiza su uso al ofrecer un sistema de paso por referencia mucho más simple (véase más adelante sección 1.2.2)

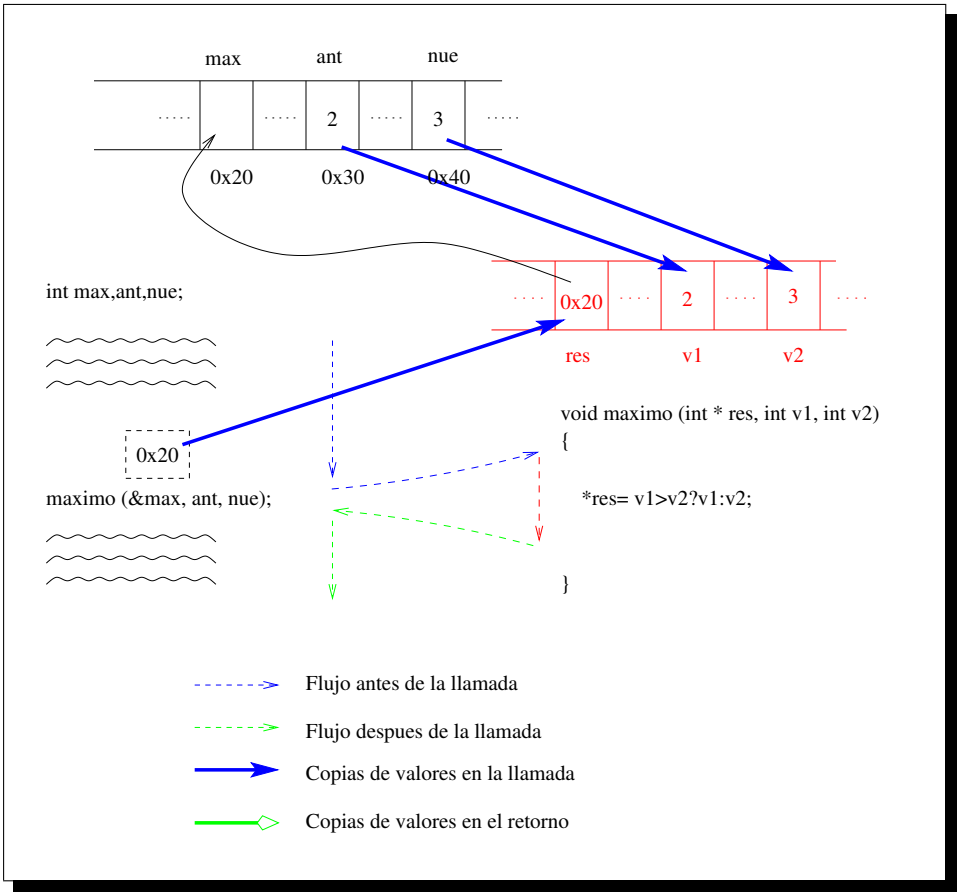

Figura 1.3: Paso de la dirección(puntero) de un entero.

Realmente, el paso sigue siendo por valor, pues ahora es un puntero lo que se transfiere[8]. Por tanto, en esta situación no debemos encontrar nada nuevo en lo que respecta a la forma en que se pasan los parámetros. Como vemos, en la llamada se calcula el valor de una expresión (el operador de obtención de dirección & aplicado a la variable *max*) que devuelve *0x20* (de tipo puntero a entero). Este valor es el que se pasa como primer parámetro y por tanto se copia a la zona donde se sitúa la variable local *res*. Es en la asignación, dentro de la función, cuando se modifica el valor de *max* ya que *res* es un puntero a dicha variable.

En tercer lugar, podemos considerar una situación distinta, en la que uno de los parámetros es un vector de cinco enteros. Esta situación se representa en la figura 1.4

En este caso, no es buena idea realizar un paso por valor propiamente dicho, ya que sería necesario primero conocer cúantos componentes tiene el vector y después copiarlos en la llamada. En este caso, el compilador simplifica la operación pues cuando se pasa un *"array"* no realiza el paso por valor, sino que se pasa la dirección del primer elemento del *"array"* . Esto tiene dos interpretaciones equivalentes

- La estrecha relación que existe entre un *"array"* y un puntero a su primer elemento[9], permite al compilador interpretar el paso a la función como el paso de un puntero, copíandose el valor de la dirección del *"array"* al valor que almacena el *"array"* (o puntero) en la función. Podemos decir que en el paso de vectores el compilador opta por una interpretación como puntero y realmente el paso sigue siendo por valor.

- La dirección del parámetro actual se copia como la dirección donde se sitúa el parámetro formal. Esto significa que acceder al contenido del *"array"* en la función es acceder al mismo sitio que el

[8] Se puede decir que se realiza el paso por referencia de forma manual.

[9] Nótese que esta relación no es más que la posibilidad de flexibilizar el comportamiento del programa por medio de una adecuada interpretación de la variable como un vector (por ejemplo, al acceder con el operador []) o como un puntero (por ejemplo, al sumar un entero al vector).

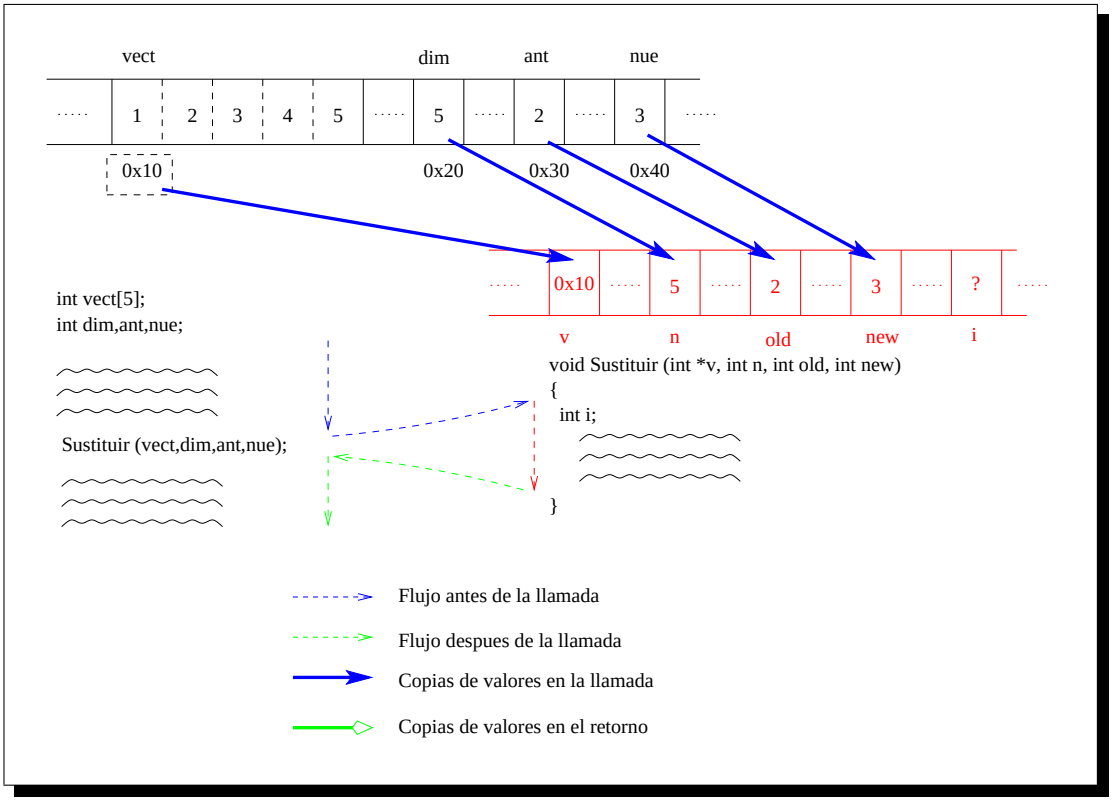

Figura 1.4: Paso de un vector de enteros.

"array" original. Esto es, el paso de matrices en C se realiza por **referencia**.

En cualquier caso, el lector deberá tener en cuenta que al pasar una matriz a una función implica que se usarán los valores originales y por tanto no existe copia del contenido de la matriz.

Por otro lado, es importante insistir en que la interpretación del parámetro actual y formal debe ser compatible para que el programa sea correcto (véase problema 1.8).

Finalmente, es importante destacar el comportamiento "especial" en el paso de un *"array"* . Como hemos podido comprobar, el compilador sustituye el sentido de *paso por valor* en éste caso por el de paso de la dirección del primer elemento. Nótese que este comportamiento puede ser conveniente pues el paso por valor de un *"array"* de un tamaño muy grande significaría realizar una copia para no modificar el original, y por tanto una pérdida de eficiencia en tiempo y espacio. Además, la posibilidad de intercambiar el papel de punteros y matrices en *C* permite obtener funciones que utilizan mecanismos de muy bajo nivel, y por tanto muy eficientes.

Obviamente, también puede ser útil el disponer del paso por valor y evitar los mecanismos de bajo nivel de *C* para obtener códigos más sencillos de construir, leer y mantener. Para ello, *C++* incorpora en sus librerías estándar el tipo *vector*[10] que, ofreciendo más operaciones, se comporta igual que cualquier tipo simple en el momento de pasarlo o devolverlo en una función.

1.2.2. Paso de parámetros y devolución de resultados en C++. Tipo referencia.

Es muy importante entender el sistema que implementa el lenguaje C++ para pasar parámetros y devolver resultados. En primer lugar, el paso (y devolución) por valor que corresponde a C, se mantiene en C++ con un comportamiento idéntico. Por tanto, la sección anterior sigue siendo válida para este lenguaje. Veamos las nuevas capacidades.

En C++, aparece un nuevo mecanismo que nos va a permitir nuevas posibilidades en la transferencia de información. Es el uso de una *referencia*. La declaración se lleva a cabo añadiendo el caracter &. Por

[10] Este tipo pertenece a la STL de *C++* y por tanto se estudiará más adelante.

ejemplo *int& ref* indica que *ref* es una referencia a un entero.

Cuando un programa se ejecuta, se almacenan "cosas" en distintas regiones de memoria. Así, al declarar una variable

```
int a;
```

indicamos al compilador que busque una zona de la memoria que almacenará un **objeto** de tipo entero. Además, el identificador *"a"* se **refiere** a esa zona de memoria (a ese objeto). Cuando escribimos líneas de código, utilizamos esas referencias a objetos para indicar un lugar de almacenamiento[11]. Por ejemplo, si escribimos:

```
a= 5;
```

el compilador entiende que se quiere almacenar el valor 5 en el objeto (lugar de almacenamiento) al que se refiere *a*.

Cuando se declara una variable, el compilador se encarga de que exista un objeto(lugar contiguo de memoria) al que se refiere. Cuando se declara una referencia, se indica que sólo se desea una nueva referencia, pero el compilador no debe encontrar ningún objeto al que se refiera. Eso significa que es similar al caso anterior, pero la referencia no nos sirve de nada hasta que no se le indica el lugar de la memoria al que se refiere. Somos nosotros los que, al inicializarla, hacemos que se refiera a una zona de memoria. Podemos decir que una referencia en un nombre alternativo a un objeto. Por ejemplo, si realizamos la siguiente declaración

```
int a=0;
int &ref= a;
```

declaramos un nuevo objeto de tipo entero (una zona que almacena un entero), donde se guarda el valor 0 e indicamos que *a* se refiere a esa zona. Cuando declaramos *ref* indicamos que será una referencia a un objeto entero. Cuando lo inicializamos con *a*, el compilador considera que es la misma referencia y por tanto ambas variables se pueden considerar la misma pues son dos referencias al mismo objeto (zona de memoria). Son dos nombres para un mismo objeto.

La aplicación de este tipo es en el paso de parámetros y devolución de resultados de funciones. En primer lugar, nos permite realizar un paso por variable a una función. Si uno de los parámetros formales de una función lo declaramos de tipo referencia, al llamar a la función, el parámetro actual que se pasa inicializa la referencia al mismo objeto y por tanto la utilización del nombre de la referencia dentro de la función será igual que si se usara dicho parámetro actual. Un ejemplo se muestra en la figura 1.5

Como podemos observar, en la zona de memoria local que utiliza la función, se ha buscado espacio para almacenar las dos variables *v1, v2*, que se pasan por valor. En cambio, *res* es una referencia y por tanto no tiene localización asignada. Cuando se llama a la función, se pasa la variable *max* como parámetro actual de la referencia y por tanto, la variable *ref* se inicializa con el objeto *max*, es decir, es una referencia al mismo lugar de memoria. Por consiguiente, tienen la misma dirección, y el paso del parámetro se ha realizado por **referencia**[12].

En la asignación interior a la función *máximo*, se indica el almacenamiento de un valor en *res* que se refiere al mismo objeto que *max*, y por tanto su modificación implica un cambio en el valor de *max*.

Por último, el concepto de tipo referencia que hemos explicado nos permite fácilmente implementar un mecanismo de devolución de valores por referencia. La sintaxis para esta devolución es muy simple pues sólo debemos de añadir el caracter & al tipo devuelto por la función.

La interpretación de esta operación es idéntica al caso de paso de un parámetro. De la misma forma que al pasar un parámetro el tipo referencia (parámetro formal) se inicializaba con la referencia del objeto que se pasaba como parámetro actual, cuando se devuelve un objeto, la referencia resultado se inicializa con la referencia de dicho objeto. Podríamos pensar, que el resultado de la función es una "variable" del tipo referenciado y que se localiza en el sitio que indique la sentencia de retorno en la función. En la figura 1.6 se muestra un ejemplo.

[11] En la literatura, el lector puede encontrar a menudo el concepto de *lvalue* como una expresión que se refiere a un objeto. El nombre proviene de "algo que se puede colocar a la izquierda de una asignación" aunque si declaramos una constante, es un *lvalue* pues se refiere a una zona de memoria pero no se puede poner a la izquierda de una asignación.

[12] Nótese la similitud con el caso anterior de paso de un *"array"* a una función en C.

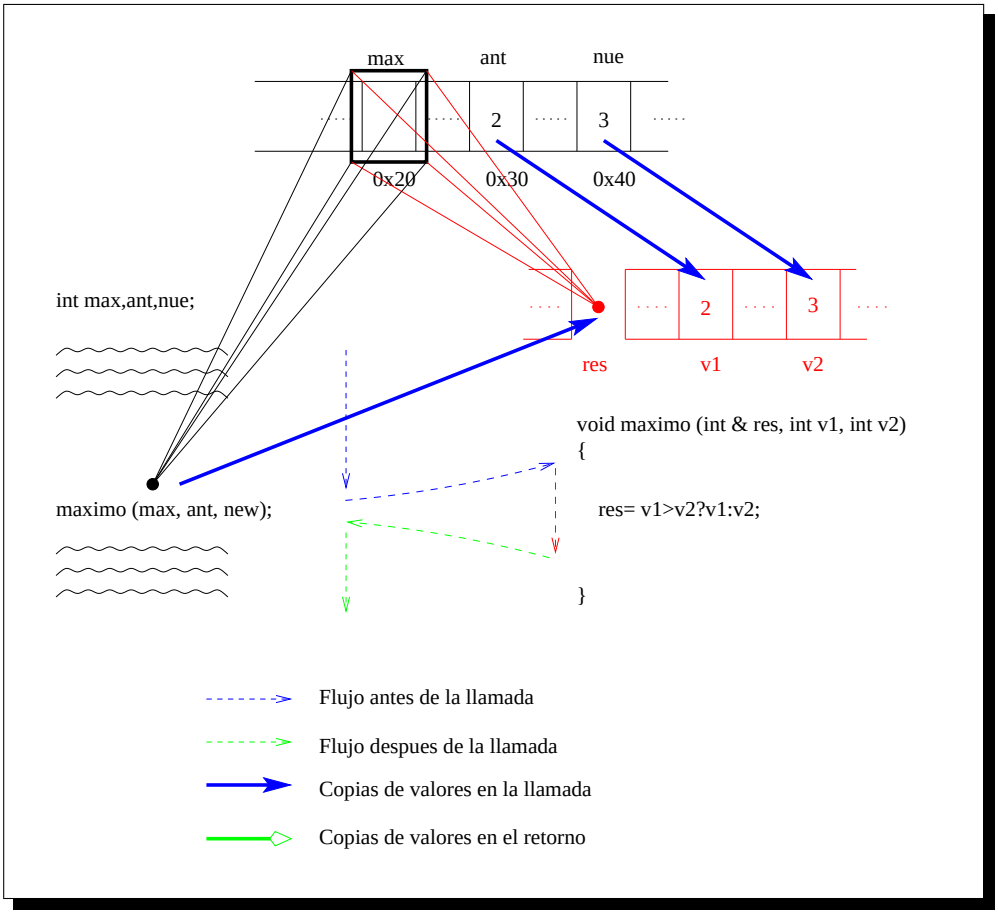

Figura 1.5: Paso de un entero por referencia.

En esta figura se muestra una función que realiza una operación muy simple, la devolución de una referencia al elemento i de un vector de enteros. Es decir, la función

```
int &valor(int *v, int i)
{
   return v[i];
}
```

Como podemos ver, la llamada a la función se realiza tal como se ha mostrado en los apartados anteriores, mediante la copia de un entero y el paso por referencia del vector. Esto significa que el entero *v[i]* es directamente el valor original en el *"array" vector*.

Cuando se ejecuta la orden *return* se asigna como resultado de la función una referencia que se inicializa con el objeto que se indica. Es decir, el resultado es una referencia al objeto que también referencia *v[i]*. Por tanto son dos nombres para una misma localización en memoria.

Tal como indicábamos antes, podemos interpretar que cuando escribimos, en el trozo de código que llama, *valor(vector,ind)* es equivalente a un nombre de variable que se localiza en el lugar donde indica la orden *return*, es decir, la cuarta posición del *"array" vector*. Dicho de otra forma, las siguientes dos líneas serían equivalentes

```
v[4]= 3*5;
valor(v,4)= 3*5;
```

Podemos afirmar que cuando se devuelve una referencia, la llamada a la función es un nombre alternativo, un sinónimo, de la variable o zona de memoria que indicamos en la sentencia *return*. En el ejemplo anterior, es un sinónimo de un valor entero que se sitúa en el vector que se pasa. Es un objeto igual que

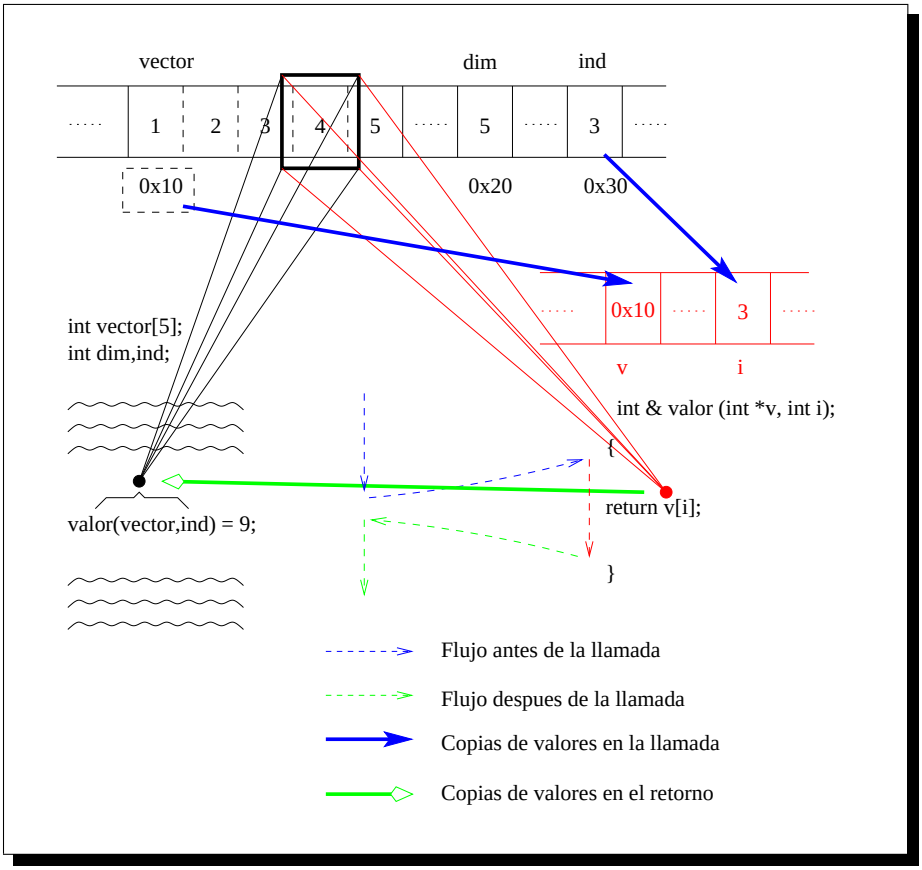

Figura 1.6: Devolución de un entero por referencia.

cualquier otro entero (con un nombre especial). Por ejemplo, podemos hacer la siguiente asignación:

```
valor(v,2)= valor(v,1)*valor(v,0);
```

1.2.3. Parámetros y *const*.

Cuando se utiliza el paso por valor, el programador no se tiene que preocupar de que la parte interna pueda modificar el valor del parámetro actual ya que se trabaja con una copia. Sin embargo, cuando consideramos el paso por referencia, es necesario tener en cuenta si el objeto se modifica. Para poder controlar esta situación, se puede usar la palabra reservada *const*.

Cuando se utiliza la palabra *const* se indica al compilador que no se puede modificar el valor del objeto. Así, la más inmediata aplicación que podemos considerar es la declaración de constantes en un programa. Estas constantes se inicializan, y ya no se vuelven a modificar sus valores.

De la misma forma, podemos usarla en la declaración de parámetros para indicar valores que no se pueden modificar. Podemos distinguir distintas situaciones:

- Supongamos que se desea una función que, a partir de un valor *float* realiza una serie de cálculos y devuelve un resultado del mismo tipo. La cabecera de la función puede ser

  ```
  float calculos (const float f)
  ```

  La semántica es idéntica a la declaración de constantes que hemos comentado. Lo que indicamos en este caso es que la variable es una constante, ahora con alcance de función, del tipo *float*.

  En este caso, el que usa la función no encuentra ninguna ventaja (normalmente este tipo de parámetros se declaran sin especificar *const*), sin embargo, puede tener sentido si el que programa quiere

garantizar que a lo largo de toda la función dichos parámetros no se modifican. La ventaja de esta declaración será por consiguiente que el usuario tiene garantizado en cualquier punto de la función que el valor que se almacena es exactamente el original que se pasó a la función. No puede cometer el error de modificar un parámetro en un punto de la función y usarlo posteriormente pensando que almacena el valor original.

Sin embargo, es probable que la función no sea demasiado compleja (si lo es, podría ser signo de un mal diseño) y no sea necesario que el parámetro sea constante. Por eso, muchos autores aconsejan evitar este uso "inútil" de *const*.

- Supongamos que se desea una función de búsqueda de un entero en un *"array"* ordenado de enteros. La cabecera se puede declarar como

```
int busqueda (int *v, int n, int el)
```

En este caso, sabemos que la modificación interna de los parámetros formales *n* y *el* no afectará al código que la llama, ya que se pasan por valor. En cambio, si la función modifica los valores a los que apunta *v*, ese código se verá afectado. La solución a este problema es indicar al compilador que los valores a los que apunta ese puntero son constantes y, por tanto, no pueden ser modificados. La forma correcta de declarar esta cabecera es

```
int busqueda (const int *v, int n, int el)
```

Es decir, *v* es un puntero (vector) de enteros constantes. Así, por un lado el que la programa no puede equivocarse ya que el compilador avisará de una operación que viole esta modificación y por otro, el que la usa sabrá que la matriz queda intacta.

- Por otro lado, en C++ aparecen el tipo referencia y por tanto la posibilidad de modificar el parámetro actual. Inicialmente, podemos pensar que si alguien declara un paso por referencia es porque desea tener la posibilidad de modificar dicha variable, sin embargo, no es así. Es posible desear pasar una variable a una función, que no la modifique, pero que tampoco haga una copia a causa del paso por valor.

  Considere por ejemplo que tiene una variable de un tamaño considerable y quiere pasarla a una función que no la modificará, pero que perderá bastante tiempo para realizar la copia. En este caso es ideal el paso por referencia ya que será mucho más eficiente, pero nos encontramos con el mismo problema que antes, cuando pasamos un *"array"* que no deseamos modificar. De nuevo, añadir la palabra *const* indica al compilador que no se debe modificar, a pesar de que pasamos una referencia y trabajamos, por tanto, con el objeto original.

  Por ejemplo, si tenemos el tipo *Gigante* que ocupa mucho espacio y deseamos una función para procesarlo sin que sea necesaria la copia, podemos usar una cabecera como

```
int Procesa (const Gigante& par)
```

en la que se pasa por referencia el parámetro pero se indica que no se modifica en el interior de la función.

- Por último, como es de esperar, *const* también se puede usar en los valores que se devuelven, en especial, cuando devolvemos un puntero o una referencia. El objetivo de ello, es indicar que el objeto referenciado en la devolución sólo se puede usar en modo de sólo lectura.

  Por ejemplo, la función *valor* que antes mostrábamos se puede reescribir para que devuelva una referencia a un entero que no se puede modificar. Esto implica que la instrucción de asignación que

antes veíamos es ilegal. Es decir, si modificamos la función y obtenemos

```
const int &valor(const int *v, int i)
{
  return v[i];
}
```

implica el siguiente comportamiento

```
// La siguiente línea es correcta,
// cambia la posición 4 de v

v[4]= 3*5;

// La siguiente línea es incorrecta, pues devuelve
// un entero que no se puede cambiar.

valor(v,4)= 3*5;

// La siguiente linea es correcta,
// usa el entero (lee) de la posición 4
res= valor(v,4)*3;
```

Otro ejemplo, ahora con el uso de punteros se muestra a continuación

```
// Función errónea
int * posicion5 (const int * p)
{
  return p+5;
}
```

En este caso vemos una cabecera de función incorrecta para la definición que se muestra. En principio, podemos pensar que es correcta pues los valores apuntados por *p* no se modifican. Sin embargo, el compilador no la acepta, ya que no está permitido convertir un tipo puntero a entero constante a un simple puntero a entero. Nótese que podríamos "hacer trampa", ya que pasamos un *const int ** a una variable *int ** y después podemos usar esta última para modificar los enteros.

A lo largo de este documento, nos encontraremos con más ejemplos sobre el uso de *const* en los tipos devueltos.

Usando *typedef*.

Es necesario insistir en un punto especialmente importante que se ha mostrado a lo largo de los ejemplos anteriores y que es muy útil para comprender y usar correctamente *const*: Las referencias o punteros a un tipo *const* son de tipo distinto a las mismas referencias o punteros sin *const*.Así, un *char ** se puede asignar a un *const char **, pero no al revés.

Un ejemplo especialmente interesante es el uso simultáneo de *typedef* y *const*. Esto es especialmente relevante, ya que en temas posteriores será necesario tenerlo en cuenta. Podemos preguntarnos por la semántica del siguiente ejemplo

```
typedef int * pInt;

...
int calculos (const pInt p)
{
  ...
}
```

En este caso, el significado no cambia, volvemos a tener una variable de un tipo y la palabra *const*

para indicar que p no se puede modificar. Por tanto, podemos modificar los enteros apuntados por p[13]

Si nuestra intención es indicar que p es un puntero a enteros constantes, nos hemos equivocado de tipo (recordemos que ¡Es otro tipo!). Se podría haber hecho

```
typedef int * pInt;
typedef const int * pConstInt;

...
int calculos (pConstInt p)
{
  ...
}
```

Nótese que hemos dejado la línea anterior, para indicar que si queremos utilizar ambos tipos con un nuevo nombre, debemos realizar ambas definiciones ya que son dos tipos distintos.

1.2.4. Ejemplo de paso de parámetros.

En esta sección mostramos un ejemplo de ordenación de un vector, para el que se particulariza la ordenación de dos o tres elementos. Además, complicamos un poco estos casos particulares.

El objetivo es presentar un algoritmo muy simple, en el que se incluyen distintos mecanismos de paso de parámetros. Por supuesto, un proyecto de programación real, donde se necesitaran estos algoritmos, no incluirían el código que se presenta.

La función principal se llama *ordena* y tiene como objetivo la ordenación de un vector de enteros de cierto tamaño n positivo.

El código es el siguiente,

```
void ordena (int *v, int n)
{
  switch (n) {
  case 2:
    ordena2(v,v+1);
    break;
  case 3:
    ordena3(v[0],v[1],v[2]);
    break;
  default:
    ordenaN(v,n);
  }
}
```

Mediante una selección con *switch*, distinguimos los casos de dos, tres y más elementos. Los dos primeros se implementan, y para el último se supone que existe otra función que lo resuelve.

[13] Esto nos ayuda a entender que con la palabra reservada *typedef* obtenemos un nuevo tipo sinónimo, no una macro que expande la definición donde aparezca el nombre.

Las funciones adicionales para implementar los casos básicos son

```
bool ordenados(int n, int m)
{
  return (n<=m);
}

void intercambia (int *p, int *q)
{
  int temp= *p;
  *p=*q;
  *q=temp;
}

void ordena2(int *p1, int *p2)
{
  if (!ordenados(*p1,*p2))
    intercambia (p1,p2);
}

void ordena3(int& n, int& m, int& l)
{
  ordena2(&n,&m);
  ordena2(&m,&l);
  ordena2(&n,&m);
}
```

1.3. Funciones *inline*.

El tiempo de ejecución de una función viene determinado, no sólo por el tiempo necesario para ejecutar todas las operaciones programadas en el cuerpo, sino también por el tiempo de llamada y retorno, incluyendo el tiempo necesario para realizar las copias de los parámetros y devolución de resultados en la pila.

Consideremos el caso de funciones cortas, en las que los tiempos de llamada y retorno pueden ser un porcentaje muy alto del tiempo de ejecución total. Además, estas funciones pueden ser responsables de gran parte del tiempo de ejecución del programa, por ejemplo, si se llaman muchas veces (por ejemplo, en una sentencia en el bucle más interno de varios anidados).

En estos casos, es tentador obviar la implementación de la función de manera que, en lugar de la llamada a la función, escribimos las sentencias que componen el cuerpo de ésta. Sin embargo, aunque ganamos en velocidad, la claridad en el programa se ve empeorada. Además, resulta aún menos aconsejable si existen multiples llamadas a la función.

Para resolver este tipo de situaciones, el lenguaje *C++* ofrece una nueva herramienta, las funciones en línea[14](*inline*).

La definición de una función en línea se realiza anteponiendo la palabra clave *inline* al tipo devuelto por la función. Un ejemplo,

```
inline bool NumeroPar (const int i)
{
  return (i%2==0);
}
```

Desde el punto de vista del usuario, seguimos teniendo una función con la misma sintaxis y semántica, y por tanto su uso es idéntico. Sin embargo, el compilador puede expandir el cuerpo de la función en cada llamada que se haga. Con este comportamiento se puede disminuir el tiempo de ejecución del programa.

El compilador interpreta el calificador *inline* como una recomendación. Por tanto, es posible que no

[14]En el nuevo estándar *C99*, el lenguaje C incorpora también las funciones *inline* con la misma sintaxis.

lleve a cabo las expansiones en cada llamada, aunque es de esperar que las funciones más simples y cortas si sean expandidas[15].

La desventaja es que el tamaño del ejecutable puede aumentar, ya que en lugar de tener una copia de la función a la que se pasa el control en cada llamada, existen multiples copias.

Finalmente, también es necesario algún comentario sobre el lugar donde se realiza la definición de la función. Si pensamos detenidamente en el proceso que debe realizar el compilador, cuando llega a un punto donde hay una llamada a la función, debe expandirla. Esto implica que el compilador debe de conocerla previamente, es decir, debe haber sido definida antes. Por tanto, la definición debe aparecer en algún punto por encima de ese mismo fichero, o en algún fichero incluido anteriormente (mediante la directiva del precompilador *include*).

1.4. Sobrecarga de funciones.

En las secciones anteriores, distintas funciones se han diferenciado, fundamentalmente, por tener nombres diferentes. Resulta lógico pensar que el compilador resuelve las llamadas a función, simplemente consultando su nombre. Por lo tanto, distintas funciones deberían tener distinto nombre. Este es el caso para algunos lenguajes de programación (entre ellos el C), pero no para el *C++* .

En *C++* , el compilador puede distinguir distintas funciones, no sólo por su nombre, sino también por sus parámetros. Esto nos permite utilizar el mismo nombre varias veces, es decir, sobrecargar las funciones.

Es posible que la semántica de distintas funciones sea similar, de manera que puede ser de interés sobrecargarlas. Por ejemplo, supongamos un programa para el dibujo de gráficos. Considerando que podemos dibujar distintas curvas (líneas, rectángulos, etc) se pueden definir distintos tipos de datos. Si queremos diseñar una función de dibujo para cada uno de ellos, podemos llamarla *dibujar*, de forma que el compilador será el que distinga la función a llamar dependiendo del tipo de los parámetros que se utilicen.

Esta discusión resulta de interés para insistir en la distinción de puntero o referencia a algo constante. Por ejemplo, recordemos que hay que distinguir entre punteros a un tipo constante o no constante. Una forma de ilustrarlo es mediante la sobrecarga de funciones cuya única diferencia sea ésta. Un ejemplo es

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

void funcion(char *m) {
  cout << ''Llamamos con puntero a char'' << endl;
}
void funcion(const char *m) {
  cout << ''Llamamos con puntero a const char'' << endl;
}

int main(){
  char cadena[5];
  const char * cadenaConstChar;

  funcion (''cadena constante'');
  funcion (cadena);
  funcion (cadenaConstChar);

  return 0;
}
```

En este ejemplo hemos sobrecargado la función de manera que la única diferencia es que en la primera, el puntero apunta a un tipo no constante y en la segunda a un tipo constante.

El resultado de la ejecución de este programa es

[15] Considere el caso de funciones recursivas, para las que resulta muy difícil, si no imposible, su expansión.

```
Llamamos con puntero a const char
Llamamos con puntero a char
Llamamos con puntero a const char
```

En donde también hemos incluido la llamada con una cadena literal. El compilador ha resuelta las llamadas de forma que, como vemos, la cadena literal es un puntero a *const char*, y las otras dos llamadas se resuelven como esperábamos.

Nótese que es posible distinguir cuando un parámetro es un puntero o referencia a algo constante. Sin embargo, no es posible hacerlo cuando la palabra *const* afecta únicamente a que el parámetro no puede modificarse. Por ejemplo, las dos cabeceras siguientes no se pueden distinguir

```
int funcion (float f);
int funcion (const float f);
```

Finalmente, es importante indicar que la discusión acerca de la sobrecarga de funciones puede ser mucho más extensa. Por ejemplo, es necesario decidir el comportamiento del compilador cuando existen distintas funciones compatibles con una llamada (mediante conversiones implícitas entre tipos).

No es nuestro objetivo llegar a ese detalle, aunque el lector interesado puede consultar, por ejemplo, Deitel[**?**] o Stroustrup[**?**].

Nosotros consideraremos ejemplos en los que la distinción es muy clara, y se llama a la función con mejor coincidencia. Por ejemplo, si tenemos una función que acepta el tipo *char* y otra el tipo *long*, al llamar con un tipo *int* parece que la promoción a *long* resulta más conveniente.

1.5. Parámetros por defecto.

En *C++* , es posible indicar el valor que tomarán los parámetros de una función cuando no se especifiquen. Esto significa que el programador puede seleccionar un conjunto de parámetros de una función y considerarlos opcionales.

Un ejemplo de una función con parámetros opcionales es

```
void imprimir_documento(string nombre, int copias= 1);
```

que nos permite llamar a la función sólo con el primer parámetro, en cuyo caso, asume que el segundo vale 1.

Para simplificar el proceso de compilación y evitar ambigüedades, el conjunto de parámetros opcionales deberán ser los últimos en la función. Además, si sólo se especifican un subconjunto de ellos, deberán ser los primeros.

De esta forma, podemos simplificar el conjunto de funciones que definimos. Por ejemplo, si queremos tener la posibilidad de llamar a la función del ejemplo anterior especificando o no el parámetro opcional, tendríamos que definir dos funciones (sobrecargar la función). Esta situación oscurece el código y lo hace más difícil de mantener. Más adelante veremos más ejemplos en los que el uso de este tipo de parámetros simplifica el conjunto de funciones a definir.

Por otro lado, también podemos encontrar ventajas en lo que a la reutilización del código se refiere. Por ejemplo, podemos tener un programa que hace uso de la función anterior, de imprimir documento, con un sólo parámetro (el nombre). Si actualizamos el módulo que se refiere a esta función, podemos decidirnos por una nueva cabecera, donde también se especifica el número de copias. Si asignamos un valor por defecto, permite que el código del programa que llama a esta función siga siendo válido. Es decir, creamos un nuevo interface, manteniendo el anterior compatible.

1.6. Variables locales *static*.

En todos los ejemplos que hemos comentado, se ha considerado que las variables locales a una función se creaban cuando se llamaban a la función, y se destruían cuando se terminaba. Sin embargo, es posible declarar una variable que no se destruya al terminar la función, es decir, una variable que se crea cuando

se llama a la función y no se destruye hasta que termina el programa. Esto significa que en sucesivas llamadas a la función, el valor de la variable se mantiene.

Este comportamiento resulta familiar. Podemos entender que una variable así se comporta como una variable global. Podemos llamar varias veces a la función y ésta puede acceder a la variable con el mismo valor que tenía la última vez que se usó. Sin embargo, existe una diferencia fundamental, y es que la variable es local a la función, es decir, no se conoce fuera de ella. De esta forma, podemos tener una variable con un comportamiento como una global pero que sólo se conoce localmente.

Para declarar una variable de este tipo se utiliza la palabra reservada *static*. Un ejemplo de este tipo de variable local es

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

float captura_minimo(float valor)
{
  static float actual=10;                  // El 10 solo al principio

  if (valor<actual) actual=valor;
  return actual;
}
int main()
{
  float v;

  do {
    cout ≪ ''Introduzca un valor de 0 a 10'' ≪ endl;
    cin ≫ v;
    cout ≪ ''El valor minimo hasta ahora es: ''
         ≪captura_minimo(v)≪endl;
  } while (v<=10);

  return 0;
}
```

donde se declara una variable local *actual* en la función *captura_minimo*.

Es importante destacar que la variable se crea la primera vez que se llama a la función. Desde ese momento, siempre existe hasta que termine el programa. El valor inicial (10) se asigna esa primera vez. A partir de entonces, cada vez que se llama a la función, la variable mantiene el último valor que contenía.

Un intento de acceder a la variable *actual* desde la función *main* no es posible, ya que esa variable es local a la función, en el sentido de que sólo se conoce ahí.

1.7. Problemas.

Problema 1.1 *Indique el efecto de la siguiente función*

```
void intercambia (int *p, int *q)
{
  int &temp= p;
  p=q;
  q=temp;
}
```

Problema 1.2 *Indique el efecto de la siguiente función*

```
void intercambia (int *p, int *q)
{
  int *temp= p;
  p=q;
  q=temp;
}
```

Problema 1.3 *Implemente una función* swap_puntero *que intercambie el valor de dos punteros usando paso por referencia. Muestre un ejemplo de llamada.*

Problema 1.4 *Implemente la función del problema anterior sin usar paso por referencia. Deberá usar un paso de parámetros al "estilo C", es decir, por medio de un puntero al valor a modificar. Reescriba el ejemplo de llamada.*

Problema 1.5 *Consideremos que en un proyecto, un programador tiene la tarea de crear una función que determine si un valor entero, indicado por un puntero, es par. Crea la siguiente función*

```
bool par(int *p)
{
  return (*p %2==0);
}
```

Considere otra función, realizada por otro programador, que usa la primera.

```
int numero_pares (const int *v, int n)
{
  int res=0;
  for (int i=0;i<n;i++)
    if (par(v+i)) res++;
  return res;
}
```

¿Qué problema se va a encontrar la persona encargada de integrar ambos códigos? Si es necesario, incluya las dos funciones en un fichero fuente e intente compilarlo. ¿Qué error es de esperar?

Problema 1.6 *Considere la siguiente función*

```
void copiar (float **m, int f, int c, float **& res)
{
  // modifica res, para apuntar a una nueva zona y
  // copia la matriz fxc m a res
}
```

- *¿Qué posible problema nos podemos encontrar al ejecutar* `Copiar(mat,f,c,mat)`*?*
- *¿Cómo podríamos asegurarnos de resolver ese problema?*

Problema 1.7 *Las cadenas de caracteres representan un ejemplo clásico en el uso de punteros. El tipo correspondiente para almacenarlas es un vector de caracteres* [16]*. Implemente las siguientes funciones:*

- *Función* copia_cadena*. Copia una cadena de caracteres en otra.*
- *Función* encadenar_cadena*. Añade una cadena de caracteres al final de otra.*

[16] Recordemos que un literal de cadena, es una secuencia de caracteres entre comillas. Por ejemplo, "Hola". El tipo de este ejemplo es const char [5] (incluye el '\0').

- *Función* longitud_cadena. *Devuelve un entero con la longitud (número de caracteres sin contar el nulo) de la cadena.*

- *Función* comparar_cadena. *Compara dos cadenas. Devuelve un valor negativo si la primera es más "pequeña", positivo si es más "grande" y cero si son "iguales".*

Teniendo en cuenta que se supone que hay suficiente memoria en las cadenas de destino y no es necesario pasar el tamaño de las cadenas (controlado por la terminación en caracter nulo).

Problema 1.8 *En la tabla 1.1 se muestran posibles pasos de parámetros. Indique si son correctos y por qué.*

	Declaración	Parámetro Actual	Parámetro formal
1	int v	v	float v
2	int m[]	m	int *mat
3	float mat[5]	mat	float *& mat
4	const int v[10]	v	int *mat
5	int m[]	m	int mat[10]
6	int m[3][5][7]	m	int mat[][5][7]
7	float v[5]	v+2	const float mat[]
8	int m[]	m	int mat[][5]
9	float f	f	double f
10	float f	&f	double& f
11	bool mat[5][7]	&mat[3][2]	const bool mat[]
12	char mat[3][5]	mat[0]	char *mat
13	int *m[10]	m	int **mat
14	const double v	&v	double v[]
15	int **m	m	int mat[][]
16	int mat[5][7][9][11]	&mat[0][0][0][0]	int *p
17	int *p	&p	int *mat[]
18	float *p	p	float *& p
19	int m[3][5][7]	m	int *mat[5][7]
20	int mat[5][7][9][11]	mat[2][4][3]	const int *m
21	float *p	p+5	float *& p
22	double *p	p+2	float *p
23	int m[5][10]	m	int *mat[]
24	int *mat[5]	&mat[5]	const int **p
25	const bool *mat	mat+4	bool *p
26		5	int& val

Cuadro 1.1: Problema 1.8. Lista de posibles pasos de parámetros

Problema 1.9 *Considere las siguientes funciones*

```
int& primera ()
{ int local; ... return local; }
int& segunda ()
{ static int local=0; ... return local; }
```

¿Cree que son correctas? Razone la respuesta.

Problema 1.10 *Suponga que ejecutamos el siguiente programa*

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

struct Par{
  int *p1,*p2;
};
void intercambia(Par p)
{
  int *paux;
  paux=p.p1; p.p1=p.p2; p.p2=paux;

  int aux;
  aux=*p.p1; *p.p1=*p.p2; *p.p2=aux;
}

int main()
{
  Par p;
  int x=0,y=1;
  p.p1=&x;
  p.p2=&y;
  intercambia(p);
  if (p.p1==&x)
    cout << ''Los punteros están igual''<<endl;
  if (*p.p1==0)
    cout << ''El primero sigue teniendo 0''<<endl;
}
```

¿Cuál cree que será la salida? Razone la respuesta.

Problema 1.11 *Suponga que ejecutamos el siguiente programa*

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

struct Par{
  int vec[2];
};
void intercambia(Par p)
{
  int aux;
  aux=p.vec[0]; p.vec[0]=p.vec[1]; p.vec[1]=aux;
}

int main()
{
  Par p;
  p.vec[0]=0;
  p.vec[1]=1;
  intercambia(p);
  if (p.vec[0]==0)
    cout << ''El primero sigue teniendo 0''<<endl;
}
```

¿Cuál cree que será la salida? Razone la respuesta.