Funciones II: Parámetros por valor y por referencia.

La forma en que hemos declarado y pasado los parámetros de las funciones hasta ahora es la que normalmente se conoce como "por valor". Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a "variables" locales de la función, estas "variables" son de hecho los propios parámetros.

Lo veremos mucho mejor con un ejemplo:

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

int funcion(int n, int m);

int main() {
   int a, b;
   a = 10;
   b = 20;

   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(a,b) ->"
        << funcion(a, b) << endl;
   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(10,20) ->"
        << funcion(10, 20) << endl;

   cin.get();
   return 0;
}

int funcion(int n, int m) {
   n = n + 5;
   m = m - 5;
   return n+m;
}
```

Bien, ¿qué es lo que pasa en este ejemplo?. Empezamos haciendo a = 10 y b = 20, después llamamos a la función "funcion" con las variables a y b como parámetros. Dentro de "funcion" los parámetros se llaman n y m, y cambiamos sus valores, sin embargo al retornar a "main", a y b conservan sus valores originales. ¿Por qué?.

La respuesta es que lo que pasamos no son las variables a y b, sino que copiamos sus valores a las variables n y m.

Piensa, por ejemplo, en lo que pasa cuando llamamos a la función con parámetros constantes, es lo que pasa en la segunda llamada a "funcion". Los valores de los parámetros no pueden cambiar al retornar de "funcion", ya que son valores constantes. Si no fuese así, no sería posible llamar a la función con estos valores.

Referencias a variables:

Las referencias sirven para definir "alias" o nombres alternativos para una misma variable. Para ello se usa el operador de referencia (&).

Sintaxis:

```
<tipo> &<alias> = <variable de referencia>
<tipo> &<alias>
```

La primera forma es la que se usa para declarar variables de referencia, la asignación es obligatoria ya que no pueden definirse referencias indeterminadas.

La segunda forma es la que se usa para definir parámetros por referencia en funciones, en las que las asignaciones son implícitas.

Ejemplo:

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

int main() {
    int a;
    int &r = a;

    a = 10;
    cout << r << endl;

    cin.get();
    return 0;
}
```

En este ejemplo las variables a y r se refieren al mismo objeto, cualquier cambio en una de ellas se produce en ambas. Para todos los efectos, son la misma variable.

Pasando parámetros por referencia:

Si queremos que los cambios realizados en los parámetros dentro de la función se conserven al retornar de la llamada, deberemos pasarlos por referencia. Esto se hace declarando los parámetros de la función como referencias a variables. Ejemplo:

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

int funcion(int &n, int &m);

int main() {
   int a, b;

   a = 10; b = 20;
   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(a,b) ->" << funcion(a, b) <<
endl;
   cout << "a,b ->" << a << ", " << b << endl;
   /* cout << "funcion(10,20) ->"
           << funcion(10, 20) << endl; (1)
   es ilegal pasar constantes como parámetros
cuando
   estos son referencias */

   cin.get();
   return 0;
}

int funcion(int &n, int &m) {
   n = n + 5;
   m = m - 5;
   return n+m;
}
```

En este caso, las variables "a" y "b" tendrán valores distintos después de llamar a la función. Cualquier cambio que realicemos en los parámetros dentro de la función, se hará también en las variables referenciadas. Esto quiere decir que no podremos llamar a la función con parámetros constantes, como se indica en (1), ya que no se puede definir una referencia a una constante

Punteros como parámetros de funciones:

Cuando pasamos un puntero como parámetro por valor de una función pasa lo mismo que con las variables. Dentro de la función trabajamos con una copia del puntero. Sin embargo, el objeto apuntado por el puntero sí será el mismo, los cambios que hagamos en los objetos apuntados por el puntero se conservarán al abandonar la función, pero no será así con los cambios que hagamos al propio puntero.

Ejemplo:

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

void funcion(int *q);

int main() {
   int a;
   int *p;

   a = 100;
   p = &a;
   // Llamamos a funcion con un puntero
funcion(p);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;
   // Llamada a funcion con la dirección de "a"
(constante)
   funcion(&a);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;

   cin.get();
   return 0;
}

void funcion(int *q) {
   // Cambiamos el valor de la variable apuntada
por
   // el puntero
   *q += 50;
   q++;
```

```
}
```

Dentro de la función se modifica el valor apuntado por el puntero, y los cambios permanecen al abandonar la función. Sin embargo, los cambios en el propio puntero son locales, y no se conservan al regresar.

Con este tipo de parámetro para función pasamos el puntero por valor. ¿Y cómo haríamos para pasar un puntero por referencia?:

```
void funcion(int* &q);
```

El operador de referencia siempre se pone junto al nombre de la variable.

Arrays como parámetros de funciones:

Cuando pasamos un array como parámetro en realidad estamos pasando un puntero al primer elemento del array, así que las modificaciones que hagamos en los elementos del array dentro de la función serán validos al retornar.

Sin embargo, si sólo pasamos el nombre del array de más de una dimensión no podremos acceder a los elementos del array mediante subíndices, ya que la función no tendrá información sobre el tamaño de cada dimensión.

Para tener acceso a arrays de más de una dimensión dentro de la función se debe declarar el parámetro como un array Ejemplo:

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

#define N 10
#define M 20

void funcion(int tabla[][M]);
// recuerda que el nombre de los parámetros en los
// prototipos es opcional, la forma:
// void funcion(int [][M]);
// es válida también.

int main() {
   int Tabla[N][M];
...
   funcion(Tabla);
```

```
...
   return 0;
}

void funcion(int tabla[][M]) {
...
   cout << tabla[2][4] << endl;
...
}
```

Estructuras como parámetros de funciones:

Las estructuras también pueden ser pasadas por valor y por referencia.

Las reglas se les aplican igual que a los tipos fundamentales: las estructuras pasadas por valor no conservarán sus cambios al retornar de la función. Las estructuras pasadas por referencia conservarán los cambios que se les hagan al retornar de la función.

Funciones que devuelven referencias:

También es posible devolver referencias desde una función, para ello basta con declarar el valor de retorno como una referencia.

Sintaxis:

```
<tipo>
&<identificador_función>(<lista_parámetros>);
```

Esto nos permite que la llamada a una función se comporte como un objeto, ya que una referencia se comporta exactamente igual que un objeto, y podremos hacer cosas como asignar valores a una llamada a función. Veamos un ejemplo:

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

int &Acceso(int*, int);

int main() {
   int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
   Acceso(array, 3)++;
   Acceso(array, 6) = Acceso(array, 4) + 10;

   cout << "Valor de array[3]: " << array[3] <<
endl;
   cout << "Valor de array[6]: " << array[6] <<
endl;

   cin.get();
   return 0;
}

int &Acceso(int* vector, int indice) {
   return vector[indice];
}
```

Esta es una potente herramienta de la que disponemos, aunque ahora no se nos ocurra ninguna aplicación interesante.

Veremos en el capítulo sobre sobrecarga que este mecanismo es imprescindible.