OIFem 2021: Segunda semana

Funciones

En C++, una *función* es un conjunto de líneas de código con un nombre. Usando ese nombre podemos ejecutar esas líneas desde otro punto del programa.

Las funciones tienen este aspecto:

```
tipo-de-retorno nombre (tipo1 parametro_1, tipo2 parametro_2, ...) {
      ...
}
```

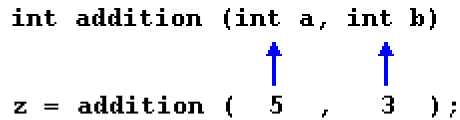
donde "tipo-de-retorno" nos indica qué tipo de dato devuelve la función. Los parámetros indican cuántos y de qué tipo son los valores que la función consume para realizar su labor.

Por ejemplo:

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

int addition (int a, int b) {
  int r;
  r = a + b;
  return r;
}

int main() {
  int z;
  z = addition(5, 3);
  cout << "El resultado es " << z;
}
```

Este programa se divide en *dos* funciones: `addition` y `main`. Recuerda que un programa C++ siempre empieza su ejecución en la función `main`, la cual es invocada automáticamente por el sistema operativo.

En este ejemplo, `main` comienza por declarar una variable, llamada z, de tipo int. Acto seguido le asigna el valor retornado por la función `addition` cuando ésta es invocada con los argumentos 5 y 3, que se hacen corresponder con los parámetros "a" y "b".



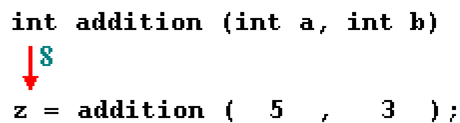
Al invocar una función, la ejecución salta al cuerpo de la función. Mientras `addition` se está ejecutando, `main` está parada esperando por el resultado. Al llamar a la función, los valores de ambos argumentos, 5 y 3, es copiado a las variables "a" y "b" utilizadas dentro de la función.

Dentro del mundo particular de `addition`, se declara la variable local "r", a la cual asignamos la suma de "a" y "b", que en este caso se correspondería con calcular 5 + 3. Por tanto a "r" se le asigna el valor 8.

Por último,

```
return r;
```

se encarga de devolver el valor de "r" al punto desde el que se invocó la función, en este caso a `main`. En cuanto `main` recibe este valor, continúa ejecutándose, en este caso asignando el valor obtenido, 8, a la variable "z":



Por supuesto, una función puede invocarse tantas veces como se quiera (una de las mayores razones de ser de las funciones: poder ser reutilizadas):

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

int subtraction(int a, int b) {
  int r;
  r=a-b;
  return r;
}

int main() {
  int x = 5, y = 3, z;
  z = subtraction(7, 2);
  cout << "El primer resultado es " << z << '\n';
  cout << "El segundo resultado es " << subtraction (7, 2) << '\n';
  cout << "El tercer resultado es " << subtraction (x, y) << '\n';
  z = 4 + subtraction (x,y);
  cout << "El cuarto resultado es " << z << '\n';
}
```

Funciones sin tipo: el uso de "void"

En la sección anterior hemos visto una función que devolvía un valor. También hemos visto que las funciones se declaraban como

```
tipo-de-retorno nombre (tipo1 parametro_1, tipo2 parametro_2, ...) {}
```

con "tipo-de-retorno" indicando de qué tipo es el valor devuelto. Sin embargo, las funciones pueden también **no** devolver nada. Para esto utilizamos el "tipo vacío", *void*. Por ejemplo, la siguiente función solamente muestra un mensaje, sin devolver nada:

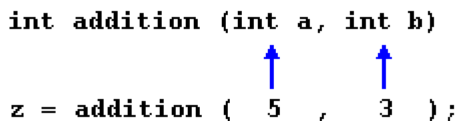
```
#include <iostream>
using namespace std;

void printmessage() {
  cout << "I'm a function!";
}

int main() {
  printmessage ();
}
```

Paso de argumentos por valor y por referencia

En las funciones que hemos visto, los argumentos ("a" y "b") han sido pasados *por valor*. Esto quiere decir que se crea una copia de los valores dados como argumentos. Por ejemplo, en:

```
int x=5, y=3, z;
z = addition ( x, y );
```

a la función `addition` se le pasan 5 y 3, *copias* de los valores contenidos en "x" e "y". Estos valores, 5 y 3, son utilizados para inicializar las variables utilizadas en la definición de `addition` ("a" y "b"). Ya que dentro de `addition` trabajamos sobre "a" y "b", que son copias, cualquier cambio sobre "a" o "b" no tiene efecto fuera de la función `addition`.



Sin embargo, en algunos casos puede ser útil permitir que modificaciones a las variables parámetro ("a" y "b" en este ejemplo) sí se propaguen. Para esto, podemos utilizar el *paso de argumentos por referencia*. En el siguiente ejemplo, la función `duplicate` multiplica por dos el valor de sus tres argumentos, haciendo que las variables utilizadas al invocarla también cambien de valor:

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

void duplicate(int& a, int& b, int& c) {
  a *= 2;  // lo mismo que "a = a * 2"
  b *= 2;
  c *= 2;
}

int main() {
  int x=1, y=3, z=7;
  duplicate(x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Para indicar que queremos utilizar *paso por referencia* (en vez de *por valor*), utilizaremos el símbolo "&", normalmente conocido por su nombre inglés *ampersand* (pero que no es más que una abreviatura de la conjunción latina "et", es decir, "y" en castellano). Lo colocamos a continuación del tipo del parámetro que queremos pasar por referencia, como puede verse en el ejemplo anterior.

Cuando usamos *paso por referencia*, lo que se pasa a la función ya no es una copia, sino la variable en sí misma (la misma posición de memoria), con lo cual cualquier modificación dentro de la función también tiene efecto fuera.

```
void duplicate (int& a,int& b,int& c)
                      x      y      z
      duplicate (   x  ,   y  ,   z  );
```

De hecho, "a", "b" y "c", son *alias* de las variables usadas al invocar la función ("x", "y", "z"). Es exactamente lo mismo modificar "a" dentro de la función que modificar "x" fuera, por ejemplo. Los nombres de las variables cambian, pero se refieren *a la misma posición de memoria*.

Si en vez de definir `duplicate` como

```
void duplicate (int& a, int& b, int& c)
```

La hubiésemos definido como:

```
void duplicate (int a, int b, int c)
```

(es decir, utilizando paso *por valor*), la función sería bastante inútil, ya que ni devuelve ningún valor, ni es capaz de modificar "el mundo exterior" desde la que es invocada.

Referencias y rendimiento

Cuando utilizamos paso *por valor*, realizar las copias tiene un coste. Este coste es bajo para tipos "simples", como int. Sin embargo, si el tipo del parámetro es más complejo, puede llevar tiempo copiarlo. Por ejemplo:

```
string concatenate (string a, string b) {
  return a+b;
}
```

Esta función toma dos cadenas de caracteres (strings) como parámetros, usando paso por valor. Debido a esto, cada vez que se llame a esta función se consumirá tiempo realizando copias para "a" y "b". Si los strings son largos, o si se invoca a la función muchas veces, puede llegar a tomar una porción considerable del tiempo de ejecución total del programa.

Lo más importante es que en este caso es *totalmente innecesario utilizar paso por valor*. Podemos utilizar paso *por referencia* y ahorrarnos todo el coste de realizar copias que no necesitamos:

```
string concatenate (string& a, string& b) {
  return a+b;
}
```

Ahora mismo el único problema es modificar las referencias "a" y "b" accidentalmente. Para evitar esto, podemos marcar las referencias como *constantes*:

```
string concatenate (const string& a, const string& b) {
  return a+b;
}
```

De esta manera el compilador nos avisará si intentamos modificar "a" o "b". Además, es posible que con la información adicional de que "a" y "b" no pueden modificarse, el compilador sea capaz de generar código más eficiente.

Reglas de alcance

Las "cosas con nombre" (llamémoslas *entidades*), tales como variables y funciones, son accesibles dentro de un *contexto*, que determina si el nombre existe o no. Es más sencillo verlo sobre un ejemplo:

```
int foo;        // variable global

int una_funcion() {
  int bar;      // variable local (a esta función/bloque)
  bar = 0;
}

int otra_funcion() {
  foo = 1;  // funciona: foo es una variable global (boooo)
  bar = 2;  // incorrecto: bar no es visible en este punto.
}
```

Las entidades declaradas fuera de un *bloque* (en general, lo que hay dentro de un `{ ... }`) se dice que tienen *alcance global*, es decir, que pueden referenciarse desde cualquier punto. El uso de variables globales es un síntoma de mal diseño, y suelen hacer un programa más difícil de entender, ya que es más difícil razonar acerca de su existencia y valor en un momento dado. Su uso debe evitarse.

Las cosas declaradas dentro de un bloque tienen *alcance local*, es decir, sólo existe y son visibles dentro de ese bloque. A las variables con alcance local se las conoce como *variables locales*.

En cada bloque, un nombre sólo puede referirse a una única cosa. Por ejemplo, no podemos utilizar el mismo nombre para dos variables diferentes *en el mismo bloque*:

```
int una_function() {
  int x;
  x = 0;
  double x;   // incorrecto: el nombre "x" ya existe en este bloque
  x = 0.0;
}
```

La *visibilidad* de los nombres locales se extiende hasta el final del bloque, e incluye también a bloques internos. Sin embargo, dado que un bloque interno sigue siendo un bloque diferente, puede reutilizar un nombre existente en el bloque que lo contiene. En este caso, el nombre reutilizado se referirá a una entidad diferente que sólo tiene alcance dentro del bloque interno, "ocultando" las entidades con el mismo nombre en bloques superiores. Por ejemplo:

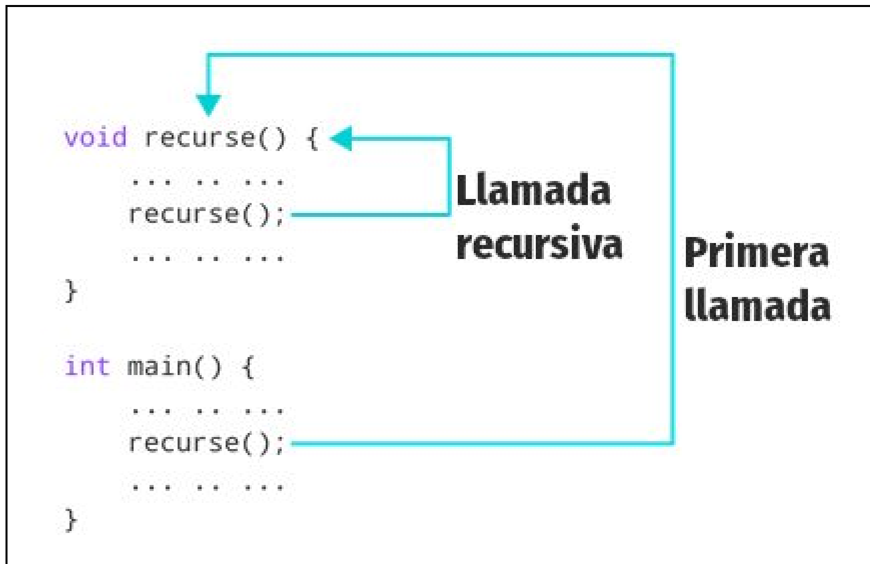
```
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // correcto, bloque interno.
    x = 50;  // asigna valor a la "x" del bloque interno.
    y = 50;  // asigna valor a la "y" del bloque padre/externo.
    cout << "bloque interno:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "bloque externo:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

Nótese como al acceder a "y" en el bloque interno nos estamos refiriendo a la "y" del bloque externo, ya que —al contrario que en el caso de "x"— su nombre no está siendo reutilizado, como en el caso de "x".

Las variables declaradas en instrucciones que introducen un bloque (como en `for(int i = ...)`) también son variables locales, al bloque que sigue a la instrucción.

Recursividad

Se llama recursividad a la propiedad de las funciones de invocarse a sí mismas.



Es útil, por ejemplo, para ordenar elementos, o cálculos como el del factorial de un número. El factorial de un número "n", representado por "n!" se calcula mediante la fórmula:

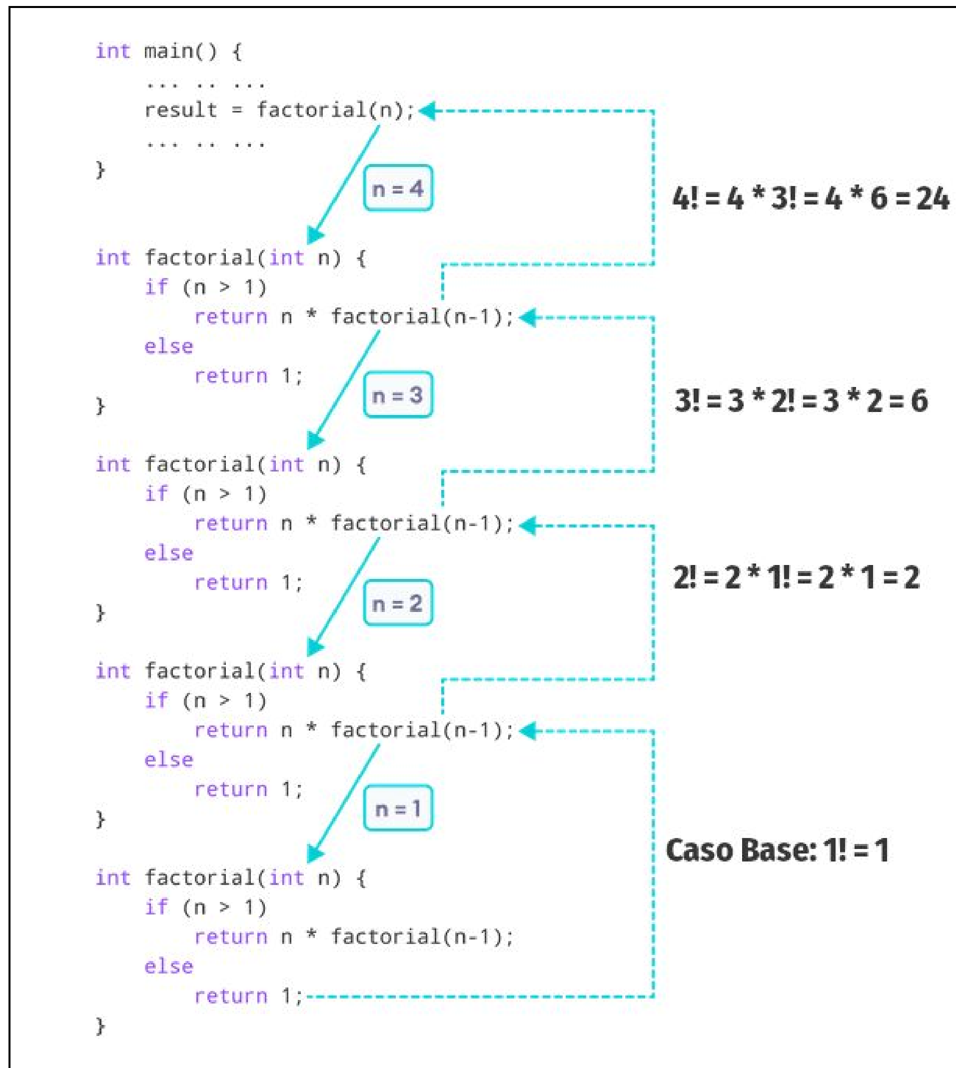
```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

Para n = 5, 5! sería:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Como veis, a medida que calculamos estamos haciendo lo mismo pero sobre un número más pequeño, hasta que llegamos a un número para el que conocemos la solución directamente (1! es 1). En C++, esto sería:

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

long factorial(long n) {
  if (n > 1) {
   return n * factorial(n-1);
  }
  else {
   return 1;
  }
}

int main() {
  long n = 4;
  long resultado = factorial(n);
  cout << number << "! = " << resultado;
  return 0;
}
```



Es muy importante que siempre exista una manera de escapar de la recursión, ya que si no caeríamos en un bucle infinito. En este caso, si "n" es menor o igual a 1 sabemos que su factorial es 1, y es de esa manera que dejamos de llamarnos recursivamente.