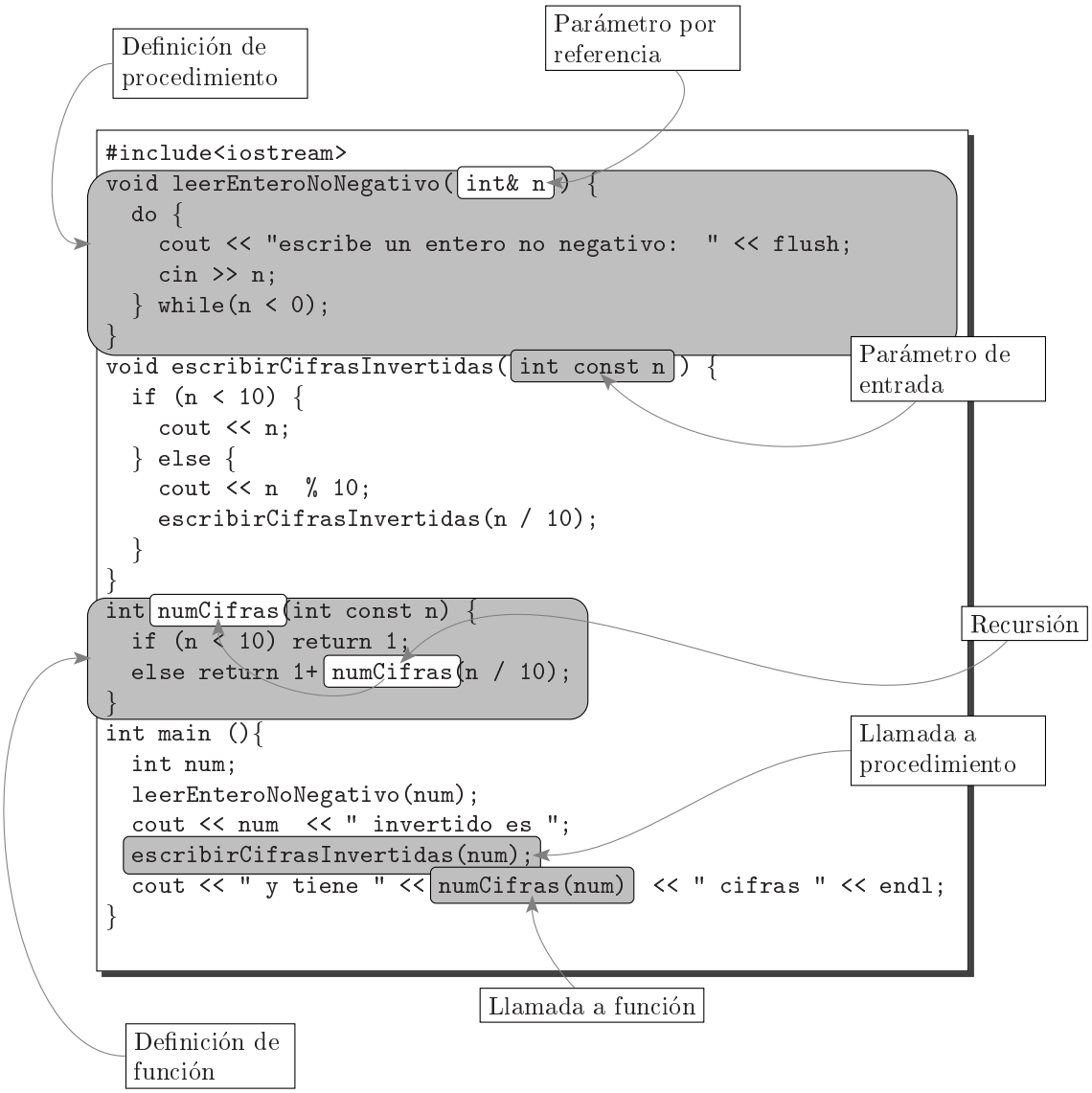
# 3 Capítulo TRES

## Subprogramas



# En este capítulo


Resumen 99

3.1 Funciones . . . 99
3.2 Acciones o procedimientos . . . 101
3.3 Elementos avanzados . . . 101

Enunciados 103

3.1 Incierta igualdad de números reales . . . 103 131
3.2 Triángulos rectángulos enteros . . . 103
3.3 Ponle tú el título . . . 103 131
3.4 Descomposición de un número . . . 104
3.5 Palíndromos . . . 104
3.6 Cuentaletras . . . 105 131
3.7 Movimiento de una partícula dentro de una superficie . . . 106
3.8 Generación de primos . . . 107
3.9 Conjetura de Goldbach . . . 107
3.10 El factorial en la sociedad del futuro . . . 108
3.11 Sucesión bicicleta . . . 109 134
3.12 Cotejo de $n$-gramas . . . 109
3.13 Simulación de variables aleatorias . . . 110
3.14 Aproximación hacia $\pi$ con dardos . . . 114
3.15 Conjetura para la formación de palíndromos . . . 114
3.16 Juegos perdedores ganan . . . 115
3.17 La tabla de Galton . . . 116
3.18 El retrato robot . . . 116
3.19 Número de ceros en que termina un factorial . . . 118 135
3.20 Representación de un número con palabras . . . 118
3.21 ¿Cuál es el mejor orden para recibir los datos de un polinomio? . . . 119 136
3.22 Calificación de oído . . . 120
3.23 Los cuadrados abominables y cáncer . . . 120
3.24 Codificaciones de plantas con cadenas . . . 121 138
3.25 Triángulos anidados . . . 122 143
3.26 Cálculo puntual de la matriz de mediotono de Judice-Jarvis-Ninke . . . 123 144
3.27 Dibujo de árboles mediante fractales . . . 124 146
3.28 Dragones y teselas . . . 126 148

Pistas 129

Soluciones 131

Un *subprograma* es un fragmento de código que resuelve una tarea claramente demarcada. Se llama *funciones* a los subprogramas que desempeñan el papel de una expresión: su resultado es un valor y no tienen ningún efecto ni sobre variables globales ni sobre el entorno. Se llama *acciones* o *procedimientos* a los subprogramas que desempeñan el papel de una instrucción: leen, escriben o modifican variables.

En C++, funciones y acciones se definen prácticamente de la misma forma, con unas diferencias sintácticas muy pequeñas. Esto no tiene nada de extraño. Es más desconcertante la escasa distinción semántica entre funciones y acciones. Por ejemplo, se puede usar una función como si fuera una acción y C++ ignorará su valor resultante sin la más mínima queja. No obstante, la distinción entre funciones y acciones no sólo es interesante desde el punto de vista expositivo o metodológico, sino también desde un punto de vista práctico, porque representan dos conceptos distintos que, también en su uso, están claramente diferenciados. Recomendamos escribir funciones o acciones tal y como se explica en los apartados 3.1 y 3.2.

C++ admite recursión, técnica que supondremos conocida y que usaremos sin más en algunos de los ejemplos de este resumen.

## 3.1 Funciones

Una función es una entidad que recibe datos mediante unos *parámetros de entrada* y devuelve como resultado un valor de un cierto tipo. Una función se define así:

⟨*Tipo resultado*⟩ `nombreDeLaFuncion(`⟨*parámetros de entrada*⟩`) {`
  ⟨*Cuerpo de la función*⟩
`}`

Un *parámetro de entrada* tiene la misma forma que la declaración de una constante; recibe el apelativo de *entrada* precisamente porque son, a todos los efectos, constantes de las que podremos coger su valor pero que no se pueden modificar. Los parámetros de una función deben ser siempre de entrada y se expresan separados por comas.

El cuerpo de la función puede ser arbitrariamente complejo, con asignaciones, bucles, llamadas a funciones o acciones, o lo que sea menester. En el cuerpo se pueden usar los parámetros como si fueran constantes. Justo antes de que acabe la ejecución de este cuerpo, es necesario indicar el resultado que se quiere devolver con la instrucción `return`. La forma de uso de esta instrucción es

`return` ⟨*expresión que calcula el resultado por devolver*⟩`;`

Como ejemplo, he aquí una función que calcula $x^n$:

```
double potencia(double const x, int const n) {
  double pot = 1;
  for (int i = 0; i < n; i++) pot = pot * x;
  return pot;
}
```

Las funciones que definimos se utilizan exactamente igual que las funciones de la librería estándar. No hay diferencia porque aquéllas suelen estar definidas también en C++.

En el cuerpo de la función puede haber más de una instrucción `return`; por ejemplo, si es necesario regir el cálculo por medio de un condicional:

```
int signo(int const v) {
  if (v < 0) return -1;
  else if (v > 0) return 1;
  else return 0;
}
```

La instrucción `return` termina inmediatamente la ejecución de la función, que tendrá como resultado el valor de la expresión que la acompaña. Por tanto, no tiene sentido que haya código después de un `return`; los compiladores de C++ razonables suelen informar de este desliz indicando que hay *código muerto*, que nunca se alcanzará. Esta terminación inmediata es útil en diversas circunstancias y permite escribir un código muy legible si se usa adecuadamente. Dedicaremos el resto de este apartado a explorar situaciones que aprovechan correctamente la terminación inmediata.

Supongámonos enfrentados a un problema que tiene casos particulares triviales, irrelevantes o imposibles de tratar. Por ejemplo, la parte entera de la raíz cuadrada de un número. Lo usual es desembarazarse de esos casos devolviendo un valor preacordado:

```
int raizCuadrada(int const n) {
  if (n < 0) return -1;
  int raiz = 0;
  while ((raiz+1) * (raiz+1) <= n) raiz++;
  return raiz;
}
```

Supongámonos enfrentados a un problema de búsqueda; por ejemplo, la determinación del menor factor primo de un número. Es necesario un bucle que recorra el espacio de alternativas. Podremos tener éxito en esta búsqueda o podremos fracasar. En caso de éxito, estaremos en medio de un bucle cuando hayamos encontrado nuestro objetivo; la terminación inmediata nos permite acabar la búsqueda y devolver lo encontrado. Si fracasamos, llegaremos al final del bucle y allí tendremos que devolver algún valor razonable:

```
int menorFactor(int const n) {
  int factor = 2;
  int const limite = raizCuadrada(n);
  while (factor <= limite) {
    if (n % factor == 0) return factor;
    factor++;
  }
  return n;
}
```

Veamos finalmente un caso que mezcla la terminación inmediata y la recursión. Supongámonos enfrentados esta vez a un problema que tiene dos casos, pero que uno se puede expresar en función del otro. Por ejemplo, la potencia, que hemos implementado antes de forma incompleta porque no hemos tenido en cuenta que el exponente puede ser negativo. Afortunadamente, $x^{-n} = 1/(x^n)$. La terminación inmediata nos permite reconvertir el problema, al principio de la función:

```
double potencia(double const x, int const n) {
  if (n < 0) return 1 / potencia(x, -n);
  double pot = 1;
  for (int i = 0; i < n; i++) pot = pot * x;
  return pot;
}
```

## 3.2 Acciones o procedimientos

Las acciones se diferencian de las funciones en que no devuelven un resultado. Sin embargo, les está permitido hacer muchas más cosas. Por ejemplo, pueden leer o escribir. Además, una acción puede tener parámetros de salida, por los que puede devolver resultados, o de entrada y salida, para modificar valores. Una acción se define así:

```
void nombreDeLaAccion(⟨parámetros⟩) {
  ⟨Cuerpo de la acción⟩
}
```

El tipo `void` es un tipo especial que no tiene valores y se utiliza para indicar la carencia de algo: en este caso, de un valor de retorno.

Los parámetros de una acción pueden ser de dos formas: primero, los parámetros *de entrada* que hemos visto para las funciones; segundo, los parámetros *por referencia*, o de (entrada y) salida, por los que la acción puede (recibir y) devolver datos; su sintaxis es ésta:

```
Tipo& nombreDelParametro
```

Cuando se llama a una acción, en el lugar de un parámetro de entrada se puede escribir una expresión cualquiera, mientras que en el lugar de un parámetro por referencia es necesario poner una variable. Dentro de la acción, el parámetro por referencia se convierte en otro nombre para la variable con la que se hizo la llamada; cuando accedemos al valor del parámetro, estamos consultando el valor de la variable; cuando modificamos el parámetro, estamos asignando a la variable. Como ejemplo, esta acción incrementa una variable en una cierta cantidad:

```
void incrementar(int& var, int const delta) {
  var = var + delta;
}
```

Dentro de una acción también se puede utilizar la instrucción `return`. Pero, como no hay que devolver ningún valor, no lleva expresión asociada. Su papel se restringe a la terminación inmediata. No suele tener tanto sentido en acciones como en funciones y, por eso, se utiliza mucho menos. Las situaciones que explicamos para funciones se pueden extrapolar a acciones.

## 3.3 Elementos avanzados

### 3.3.1 Referencias constantes

Un parámetro de entrada implica una copia del valor; por eso, también se conocen como parámetros por valor. Un parámetro por referencia implica el paso de un puntero (que se verán en el capítulo 6). Para los tipos predefinidos que hemos visto hasta ahora, pasar un dato por valor o por referencia tiene un coste similar, porque un puntero es aproximadamente del tamaño de un entero.

En el capítulo 4 veremos cómo se pueden definir nuevos tipos por aglomeración de los primitivos. Cuando estas aglomeraciones crecen, el paso por valor es bastante más caro que el paso por referencia. Sería interesante tener parámetros de entrada por referencia en vez de por valor. Un parámetro de este estilo se declara en C++ juntando en el orden adecuado los calificadores `&` y `const`, verbigracia:

```
Tipo const& parametroDeEntradaPorReferencia
```

Que un parámetro de entrada sea `const` o `const&` resulta indistinguible por el resto del programa; se puede cambiar de uno a otro por meras consideraciones de eficiencia, sin preocuparse porque el programa deje de compilar o cambie su funcionamiento.

### 3.3.2 Sobrecarga

C++ no sólo distingue entre subprogramas por su nombre sino también por el tipo de sus parámetros. Podemos definir varios subprogramas que se llamen igual siempre y cuando sus parámetros sean distintos, ya sea en número o en tipo. C++ distinguirá entre ellos y sabrá cuál utilizar en una cierta llamada analizando el tipo de los valores que pasamos a los parámetros. Es lo que se conoce como *sobrecarga* de identificadores. (Véase el apartado 5.1 para un ejemplo interesante de sobrecarga.)

### 3.3.3 Definición de operadores

En C++ los operadores son meros nombres para subprogramas y aceptan nuevas definiciones. Al dar una nueva definición a un operador (todos los operadores tienen ya uno o varios papeles dentro del lenguaje, así que una nueva definición siempre supone una sobrecarga) hay que prefijar su nombre con la palabra `operator`. (Véase el apartado 4.3 para un ejemplo interesante de definición de operadores.)