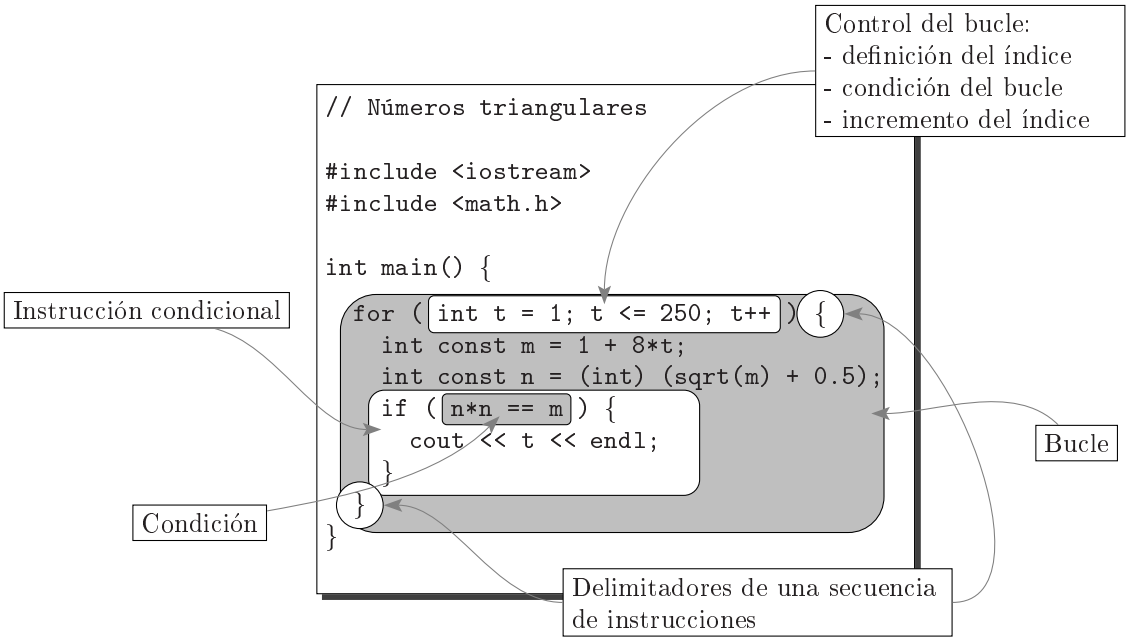
2 Capítulo DOS

Instrucciones estructuradas



En este capítulo

2.1 Composición secuencial

Para indicar que queremos ejecutar varias instrucciones seguidas, basta con escribirlas en ese orden. La secuencia

```
int x = 0;
x = x+1;
x++;
```

realiza por orden las siguientes instrucciones: define la variable entera `x` con valor inicial cero, calcula el sucesor del valor en `x` y lo guarda en esta misma variable, y vuelve a incrementar en una unidad la variable `x`, esta vez con el operador `++`, de incremento.

Una secuencia de instrucciones no es una instrucción. Esta afirmación puede parecer trivial, pero en breve se verá por qué es importante poder tratar una secuencia de instrucciones como si fuera una sola instrucción. En todo caso, para lograrlo basta con rodear con unas llaves toda la secuencia. El ejemplo anterior se convierte en una sola instrucción compuesta, así:

```
{
  int x = 0;
  x = x+1;
  x++;
}
```

Las llaves, además de agrupar, introducen un *bloque*. Las declaraciones sólo sobreviven hasta que se alcanza el final del bloque donde se han declarado. Si intentáramos compilar esto,

```
{
  int x = 0;
  x = x+1;
}
x++;
```

obtendríamos un error en la última línea: el compilador se quejaría de que en ese punto el identificador `x` no tiene una declaración asociada.

2.2 Selección condicional

Se puede condicionar la ejecución u omisión de una instrucción con la instrucción de selección condicional `if`. Su forma es la siguiente:

`if (`⟨*condición*⟩`)` ⟨*Cuerpo del* `if`⟩

La *condición* debe ser una expresión con tipo `bool`. Siempre debe ir rodeada con unos paréntesis que no forman parte de la condición sino de la sintaxis del `if`. El *cuerpo* del `if` debe ser una sola instrucción; para ejecutar condicionalmente más de una instrucción, hay que convertirlas en un bloque rodeándolas con unas llaves.

Cuando la ejecución alcanza el condicional, primero se evalúa la condición; si el resultado es `true`, se ejecuta el cuerpo, pero se obvia si es `false`. Por ejemplo,

```
if (n % 2 == 1) p = p + x;
```

incrementa `p` en `x` si `n` es un número impar. Hay una diferencia muy importante entre

```
if (n % 2 == 1) {
  p = p + x;
  n = n - 1;
}
```

y

```
if (n % 2 == 1)
  p = p + x;
  n = n - 1;
```

En el primer caso el decremento de `n` forma parte del cuerpo de la selección condicional, mientras que en el segundo caso está fuera del `if`. En términos prácticos, en el segundo caso el decremento de `n` siempre se ejecuta, mientras que en el primero sólo se ejecuta si `n` es impar. Es importante no olvidar las llaves. También es importante no dejarse engañar por los espacios o la división en líneas; C++ los ignora completamente y sólo se rige por la estructura de agrupación que inducen las llaves. En todo caso, nunca debemos ser nuestros propios enemigos: conviene utilizar la división en líneas y la estructura visual para ayudarnos. Recomendamos los siguientes estilos. Para un condicional con una sola instrucción corta en el cuerpo:

`if (`⟨*condición*⟩`)` ⟨*Instrucción corta*⟩

Cuando hay más de una instrucción es necesario poner llaves; la llave para abrir se debería poner en la misma línea que la condición y ser lo último; la llave para cerrar se debería poner sola en una línea, justo debajo de la "i" del `if`:

```
if (⟨condición⟩) {
  ⟨Instrucción 1⟩
  ⟨Instrucción 2⟩
}
```

Cuando hay una sola instrucción pero es larga, es mejor ponerla en la siguiente línea; para que el código sea lo más inteligible posible y, para que sea fácil aplicarle cambios, se deberían añadir unas llaves:

```
if (⟨condición⟩) {
  ⟨Instrucción larga que requiere al menos una línea⟩
}
```

Una variante de esta instrucción sirve para elegir condicionalmente entre dos instrucciones:

```
if (⟨condición⟩) ⟨Cuerpo a ejecutar si el predicado es cierto⟩
else ⟨Cuerpo a ejecutar si el predicado es falso⟩
```

Igual que ocurría antes, los cuerpos de uno u otro caso deben ser una sola instrucción. Se deberían aplicar los mismos criterios para decidir cuándo poner llaves. La forma de combinar el `else` con la llave que cierra el bloque de instrucciones de la parte cierta puede ser ésta,

```
if (⟨condición⟩) {
  ⟨Instrucción 1⟩
  ⟨Instrucción 2⟩
}
else {
  ⟨Instrucción 3⟩
  ⟨Instrucción 4⟩
}
```

o, preferiblemente, así:

```
if (⟨condición⟩) {
  ⟨Instrucción 1⟩
  ⟨Instrucción 2⟩
} else {
  ⟨Instrucción 3⟩
  ⟨Instrucción 4⟩
}
```

En algunas ocasiones el código de una bifurcación es largo mientras que el de la otra es corto. Por ejemplo:

```
if (x % 2 == 1) {
  p = p + x;
  n = (n-1) / 2;
} else n = n-1;
```

En estos casos, es mejor rodear ambas ramas con llaves, verbigracia:

```
if (x % 2 == 1) {
  p = p + x;
  n = (n-1) / 2;
} else {
  n = n-1;
}
```

Algunas veces hay que distinguir condicionalmente entre tres o más casos. Lo propio es primero escindir un caso, luego otro y así hasta haberlos completado todos. Por ejemplo, el signo de un número se puede calcular así:

```
if (n < 0) signo = -1;
else if (n > 0) signo = 1;
else signo = 0;
```

En general, esta situación se expresa así:

```
if (⟨condición 1⟩) {
  ⟨Instrucciones 1⟩
} else if (⟨condición 2⟩) {
  ⟨Instrucciones 2⟩
} else if (⟨condición 3⟩) {
  ⟨Instrucciones 3⟩
}
  ⟨Otros casos⟩
} else {
  ⟨Instrucciones para el último caso⟩
}
```

La estructura anterior resuelve el problema con una búsqueda secuencial, porque recorre uno por uno los distintos casos hasta encontrar qué es lo que está ocurriendo. Hay veces que es lo único que se puede hacer; pero en ocasiones es mejor una búsqueda más estructurada que nos ahorre pasos. Un ejemplo interesante es el cálculo del número de días de un mes en un año no bisiesto. Se puede resolver secuencialmente:

```
if (mes == 1) dias = 31;
else if (mes == 2) dias = 28;
else if (mes == 3) dias = 31;
⟨Otros casos⟩
```

O más estructuradamente aprovechando que, hasta el séptimo mes, los meses con número par tienen 30 días (o 28 en el caso de febrero) y los de número impar tienen 31 días, y que, a partir del octavo mes, ocurre lo contrario:

```
if (mes <= 7) {
  if (mes % 2 == 1) dias = 31;
  else if (mes == 2) dias = 28;
  else dias = 30;
} else {
  if (mes % 2 == 1) dias = 30;
  else dias = 31;
}
```

Con frecuencia, una selección entre múltiples casos como la anterior se puede abreviar usando otra estructura, la instrucción `switch`, que estudiaremos en el apartado 2.5.

2.3 Iteración

C++ tiene varias instrucciones de repetición. Empezaremos con la más básica y potente. Es la instrucción `while`; tiene la forma siguiente:

`while` (⟨*condición*⟩) ⟨*Cuerpo del* `while`⟩

Sobre la *condición* y el *cuerpo* hay que hacer las mismas observaciones que en el caso de la instrucción de selección condicional. La condición debe ser una expresión con tipo `bool`; siempre debe ir rodeada con unos paréntesis que no forman parte de la condición sino de la sintaxis del `while`. El cuerpo del `while` debe ser una sola instrucción; si queremos ejecutar repetitivamente más de una instrucción, habrá que convertirlas en un bloque rodeándolas con unas llaves.

Cuando la ejecución alcanza una instrucción de esta forma, evalúa la condición; si el resultado es `true`, se ejecuta el cuerpo y se vuelve a empezar; si el resultado es `false`, se termina la ejecución de esta instrucción. Este tipo de instrucciones que ejecutan repetidas veces una misma instrucción reciben el apelativo de *iterativas* o *bucles*; a cada ejecución del cuerpo de un bucle se le suele llamar *iteración* o *vuelta*. Como ejemplo:

```
while (n != 0) {
  if (n % 2 == 1) p = p + x;
  n = n / 2;
}
```

Cada vez que se ejecuta el cuerpo de este bucle, en cada vuelta, el valor de la variable `n` se reduce al menos a la mitad. Como es un dato entero, inevitablemente terminará valiendo 0, y entonces la siguiente comprobación de la condición del bucle resultará falsa, con lo que el bucle terminará.

Es fácil escribir un bucle que no termina nunca:

```
while (true) {
  cout << "No quiero terminar." << endl;
}
```

Sin pretenderlo, a menudo se escriben bucles que no terminan, y no es fácil descubrir el error.

Hay un detalle importante sobre la ejecución de un bucle `while`: la condición no se está comprobando continuamente, sólo se comprueba cuando se llega al bucle por primera vez o cuando se acaba de ejecutar el cuerpo y se ha de decidir si hay que volver a ejecutarlo. Por esto, el siguiente bucle nunca termina:

```
x = 1;
while (x > 0) {
  x = x - 1;
  if (x == 0) x = 1;
}
```

2.4 Otras instrucciones de iteración

2.4.1 Bucle `for`

El bucle de la sección anterior es tan simple y expresivo que suele resultar un desperdicio para algunas tareas. En esas ocasiones también se pierde tiempo de programación que se ahorraría usando un bucle más sencillo.

A menudo simplemente se quiere repetir una instrucción un número de veces conocido antes de entrar en el bucle. En estas ocasiones también es normal tener una variable que sirve para indicar la *vez* por la que vamos. Muchos lenguajes tienen un bucle especial para expresar esta idea; suele llamarse `for`. Existe un bucle `for` en C++; pero en vez de ser una restricción del bucle `while` resulta ser un bucle más complejo de usar. Afortunadamente, no es necesario pensar cada vez cómo funciona este bucle, sino que basta con ajustarse a unas pocas plantillas que resuelven los problemas más usuales. El bucle `for` tiene la siguiente estructura:

`for (`⟨*Instrucción inicial*⟩`;` ⟨*condición*⟩`;` ⟨*Incremento*⟩`)` ⟨*Cuerpo*⟩

La mejor forma de explicar su funcionamiento es escribiendo su equivalente con un bucle `while`:

```
{
  ⟨Instruccion inicial⟩;
  while (⟨condición⟩) {
    ⟨Cuerpo⟩;
    ⟨Incremento⟩;
  }
}
```

Que toda esta equivalencia esté insertada en un bloque (las llaves más externas) es un detalle importante, porque la instrucción inicial puede contener declaraciones que no tienen efecto más allá del bucle.

Como se ha dicho, este bucle se suele usar para recorrer con una variable índice un rango de valores, a saber:

```
for (int i = ⟨valor inicial⟩; i < ⟨valor siguiente al final⟩; i++) {
  ⟨Cuerpo, en donde se puede utilizar la variable i⟩
}
```

La variable índice no tiene por qué llamarse `i`. En el ejemplo anterior se recorre un rango que contiene el valor inicial pero no el final; basta convertir la desigualdad estricta (`<`) en una no estricta (`<=`) para que también se incluya el valor final.

En el capítulo 6, dedicado a la memoria dinámica, veremos otras formas de utilización de este bucle.

Hay que tener cuidado con la interacción de todas las partes de este bucle. Un ejemplo radical es el siguiente bucle que no termina nunca:

```
for (int i = 1; i <= INT_MAX; i++) {
  if (i % 5 == 0) cout << "Número múltiplo de cinco:  " << i;
}
```

2.4.2 Bucle con una ejecución

Ocasionalmente, deseamos escribir un bucle cuyo cuerpo se ha de ejecutar al menos una vez. Esta contingencia se puede expresar con el bucle `do while`, que tiene la siguiente forma:

```
do {
  ⟨Cuerpo⟩
} while (⟨condición⟩);
```

Los elementos reciben los mismos nombres que en el caso del bucle `while`. Es equivalente a ésta:

```
{
  ⟨Cuerpo⟩
}
while (⟨condición⟩) {
  ⟨Cuerpo⟩
}
```

Es decir, primero se ejecuta el cuerpo y luego se entra en un ciclo de comprobación de la condición y ejecución del cuerpo.

2.4.3 Interrupción de un bucle

En cambio, sí ocurre a menundo que ni el bucle `while` ni el `do while` son adecuados porque hay una parte del bucle que debe ejecutarse al menos una vez pero otra parte puede que no tenga que ejecutarse nunca. Ante esa situación, se suele recurrir a una estructura como la siguiente:

```
⟨Instrucciones que hay que ejecutar al menos una vez⟩
while (⟨condición⟩) {
  ⟨Instrucciones que puede que no haya que ejecutar nunca⟩
  ⟨Instrucciones que hay que ejecutar al menos una vez⟩
}
```

Si el conjunto de instrucciones que hay que ejecutar al menos una vez es sencillo, esta solución es adecuada. Pero si es complejo o grande, es una solución que complica el mantenimiento del código porque una mejora o la corrección de algún error necesita dos cambios. Entonces se suele recurrir a esta otra estructura,

```
bool continuar = true;
while (continuar) {
  ⟨Instrucciones que hay que ejecutar al menos una vez⟩
  if (⟨condición⟩) {
    ⟨Instrucciones que puede que no haya que ejecutar nunca⟩
  } else {
    continuar = false;
  }
}
```

que resulta particularmente embrollada y oculta la verdadera naturaleza de nuestro problema: tenemos un bucle con el punto de salida en su interior en vez de en un extremo. En C++ hay una instrucción que, al ejecutarse, interrumpe el bucle más inmediato a donde aparezca anidada; es la instrucción `break`. La solución más habitual de nuestro problema tiene la siguiente forma:

```
for (;;) {
  ⟨Instrucciones que hay que ejecutar al menos una vez⟩
  if (!⟨condición⟩) break;
  ⟨Instrucciones que puede que no haya que ejecutar nunca⟩
}
```

Obsérvese que esta construcción utiliza el bucle `for` de una manera un tanto extraña. Es una herencia de la tradición de C, aunque es igualmente válido y bastante usual utilizar un bucle `while`:

```
while (true) {
  ⟨Instrucciones que hay que ejecutar al menos una vez⟩
  if (!⟨condición⟩) break;
  ⟨Instrucciones que puede que no haya que ejecutar nunca⟩
}
```

2.5 Selección con tipos ordinales

Hay una instrucción especial para ejecutar código dependiendo del valor de una expresión que tenga un tipo ordinal. (Los tipos ordinales son los enteros y los valores de verdad.) La forma de esta instrucción es la siguiente:

```
switch (⟨expresión con tipo ordinal⟩) {
case ⟨valor 1⟩:
  ⟨Qué hacer para este valor⟩
  break;
case ⟨valor 2⟩:
  ⟨Qué hacer para este otro valor⟩
  break;
⟨Otros casos⟩
default:
  ⟨Qué hacer cuando el valor no corresponde a ninguno de los anteriores⟩
  break;
}
```

La expresión que sigue a la palabra `switch` es la expresión selectora. Los distintos valores que siguen a la palabra `case` deben ser expresiones constantes en tiempo de compilación. Todos estos valores y la expresión selectora deben ser del mismo tipo.

El funcionamiento de esta instrucción es muy sencillo. Primero se evalúa la expresión selectora. Luego se busca si se ha especificado un caso para el valor que hemos obtenido. Si así ocurre, se ejecutarán las instrucciones de ese caso: lo que haya entre los correspondientes `case` y `break`. Si no se ha especificado un caso para ese valor, se ejecutarán las instrucciones para el caso *residual*: lo que haya entre `default` y el siguiente `break`.

El caso residual se puede omitir. Entonces, si el resultado de la expresión selectora no coincide con ningún caso, la instrucción `switch` no hará nada.

En C++ no es obligatorio terminar las instrucciones asociadas a cada caso con un `break`. Un caso que no termine con un `break` se prolonga a los siguientes hasta que aparezca el primer `break`. Es una capacidad de C++ muy discutible con una semántica muy extraña. En este libro no se utilizará; no es en absoluto una pérdida porque en la práctica hay muy pocas ocasiones en donde se pueda aprovechar.

La instrucción de selección `switch` no es fundamental porque siempre se puede conseguir su efecto recurriendo a una serie de selecciones condicionales secuenciadas. El ejemplo anterior quedaría así:

```
{
  int const v = ⟨expresión con tipo ordinal⟩;
  if (v == ⟨valor 1⟩) {
    ⟨Qué hacer para este valor⟩
  } else if (v == ⟨valor 2⟩) {
    ⟨Qué hacer para este otro valor⟩
  }
  ⟨Resto de casos (traducidos a ifs anidados)⟩
  else {
    ⟨Qué hacer cuando el valor de v no responde a ninguno de los anteriores⟩
  }
}
```