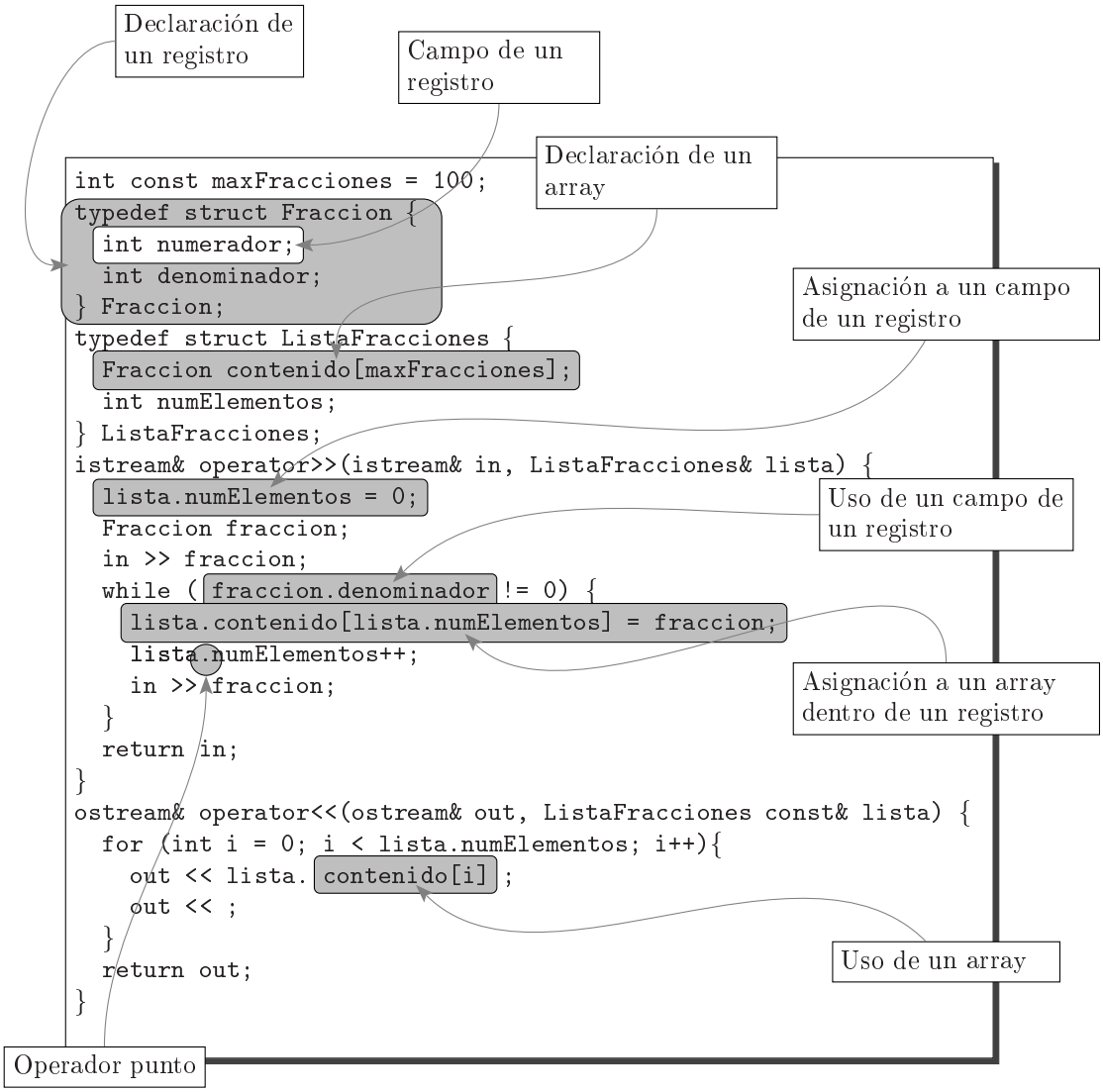
4 Capítulo CUATRO

Definición de tipos



4444444444444444444444444444444444444444 En este capítulo

Todo programa no trivial exige la definición de nuevos tipos que servirán para aglutinar y almacenar los datos de trabajo. Estos tipos, según se añadan nuevas operaciones en forma de subprogramas que los manejen, irán constituyendo el centro de piezas que tomarán identidad por sí mismas y que podrán ser de utilidad en otros programas.

En este capítulo practicamos con los aspectos más básicos de la definición de tipos en C++. Esta visión crecerá y se completará en los dos siguientes capítulos.

4.1 Arrays

Un *array* es una ristra de elementos del mismo tipo. Mediante la declaración

```
Tipo arr[N];
```

se introduce el array `arr`, formado por `N` elementos de `Tipo`. La expresión `N` ha de ser constante en el momento de la compilación. Los elementos de este array tienen índices del 0 al $N-1$, ambos incluidos. Se accede a los elementos con el operador de indexación: `arr[indice]`. Una indexación puede aparecer en la parte izquierda de una asignación.

El bucle `for` es la forma más fácil de recorrer un array. Supongamos que tenemos el array

```
double pesos[N];
```

que va a contener pesos. Para definir este array de forma que todos los pesos sean cero, escribimos

```
for (int i = 0; i < N; i++) pesos[i] = 0;
```

Sumamos todos los pesos así:

```
pesoTotal = 0;
for (int i = 0; i < N; i++) pesoTotal = pesoTotal + pesos[i];
```

No sólo se pueden declarar variables de tipo array, sino también definirlas; los distintos valores del array se separan por comas y se rodean con unas llaves:

```
double ejeY[3] = {0, 1.0, 0};
```

Se puede omitir el tamaño, y el compilador contará los elementos dados:

```
double ejeZ[] = {0, 0, 1.0};
```

Se pueden definir arrays de más de una dimensión; basta añadir, entre corchetes, tantas longitudes como dimensiones queramos. Por ejemplo, una matriz de `N` × `M` reales se define así:

```
double mat[N][M];
```

Para acceder a la posición (`i`, `j`) hay que escribir `mat[i][j]`. Es correcto escribir `mat[i]`: estaremos accediendo a la fila `i` de la matriz, que es un array unidimensional de longitud `M`.

Los parámetros de tipo array en un subprograma tienen ciertas prescripciones curiosas. La primera y más importante es que siempre se pasan por referencia; no hay cabida ni para `const` ni para `&`. La segunda es que también se devuelven por referencia cuando son el resultado de una función; por tanto, una función nunca debe devolver un array que sea una variable local. La tercera es que la longitud del primer índice se puede omitir; esto permite definir subprogramas independientes del tamaño de los arrays. Pero C++ no pasa al subprograma información relativa al tamaño real del array; es responsabilidad nuestra añadir un parámetro adicional con esa información si queremos escribir subprogramas realmente independientes del tamaño. Por ejemplo, la función que suma todos los elementos de un array se puede

escribir como sigue:

```
double suma(double arr[], int const longitud) {
  double laSuma = 0;
  for (int i = 0; i < longitud; i++) laSuma = laSuma + arr[i];
  return laSuma;
}
```

4.2 De cómo nombrar tipos

Es interesante poder dar nombre a un nuevo tipo. Se consigue con la palabra reservada `typedef`, que tiene una sintaxis cuanto menos curiosa: debe preceder a una declaración correcta de variable. Pero no definirá una nueva variable, sino un nuevo tipo. Así, para definir un `Vector` como un array de tamaño `N` debemos escribir:

```
typedef double Vector[N];
```

Sin el `typedef` estaríamos declarando la variable `Vector`; pero con el `typedef` estamos definiendo el tipo `Vector` como equivalente a un array con `N` `double`s.

Como colofón de este apartado y del anterior, y a modo de ejemplo ilustrativo, implementaremos algunas operaciones tradicionales del álgebra matricial. Tendremos vectores de dimensión `N` y matrices $\texttt{N} \times \texttt{N}$:

```
typedef double Vector[N];
typedef Vector Matriz[N];
```

El producto escalar de dos vectores se puede escribir así:

```
double productoEscalar(Vector v1, Vector v2) {
  double pe = 0;
  for (int i = 0; i < N; i++) pe = pe + v1[i]*v2[i];
  return pe;
}
```

La operación para sumar dos vectores no puede ser una función porque (1) la suma habrá que guardarla en un array local, (2) los arrays se devuelven por referencia y (3) nunca se debe devolver una variable local por referencia. Por tanto, no queda más remedio que escribir esta operación como una acción con tres argumentos: los dos primeros serán los vectores que queremos sumar, y el tercero el vector donde vamos a dejar el resultado:

```
void suma(Vector v1, Vector v2, Vector v3) {
  for (int i = 0; i < N; i++) v3[i] = v1[i] + v2[i];
}
```

Los tres parámetros, por ser arrays, se pasan por referencia. Que los dos primeros argumentos son de entrada y el tercero de salida es algo que no se puede plasmar directamente en el código y habría que hacerlo en forma de comentario al programa.

Terminamos con el producto de una matriz por un vector. Igual que la operación anterior, ésta también será una acción que tendrá como tercer parámetro un vector donde dejaremos el resultado:

```
void producto(Matriz mat, Vector vec, Vector result) {
  for (int i = 0; i < N; i++) result[i] = productoEscalar(mat[i], vec);
}
```

4.3 Registros

Con los arrays podemos juntar sólo elementos del mismo tipo. Con los registros es posible juntar diferentes tipos para formar otro. Cada uno de los elementos componentes se llama *campo* y ha de tener un nombre. La sintaxis para declarar una variable que sea un registro es muy simple: cada campo se declara a su vez como si fuera una variable, todos ellos se rodean con unas llaves y se preceden con la palabra clave `struct`; el nombre de la variable termina esta construcción. Por ejemplo, con el siguiente registro podríamos representar un número complejo:

```
struct {
  double parteReal;
  double parteImaginaria;
} complejo;
```

Y con este otro podríamos guardar información de una persona:

```
struct {
  char nombre[100];
  int edad;
  int dni;
  double altura;
} persona;
```

Casi nunca se utilizan los registros para definir una variable directamente, porque generalmente se necesita más de una variable de ese mismo tipo registro y repetir la definición es muy costoso. Lo normal es definir un tipo con `typedef`:

```
typedef struct Complejo {
  double parteReal;
  double parteImaginaria;
} Complejo;
```

Que se repita el nombre del tipo también detrás de `struct` es una costumbre que viene de C, que es necesaria en ciertas ocasiones y que en este libro siempre respetaremos.

A un campo de una variable de tipo registro se accede poniendo un punto entre la variable y el campo:

```
Complejo c;
c.parteReal = 1;
c.parteImaginaria = 2;
cout << "Parte real:  " << c.parteReal
     << ", parte imaginaria:  " << c.parteImaginaria
     << endl;
```

Al igual que con los arrays, una variable de tipo registro no sólo se puede declarar, sino que también se puede definir. Los valores de los campos se especifican entre llaves, separados por comas y respetando el orden de aparición dentro del registro:

```
Complejo c_i = {0.0, 1.0};
```

Los registros no tienen ninguno de los comportamientos extraños de los arrays cuando son parámetros de subprogramas. Se pasan por valor o por referencia dependiendo de si aparece el calificador `const` o el calificador `&`. Esto es así incluso si el registro contiene un array; de hecho, una buena forma de hacer que los arrays se *comporten* es encerrarlos dentro de un registro:

```
int const N = 3;
typedef struct Vector {
  double datos[N];
} Vector;
typedef struct Matriz {
  Vector datos[N];
} Matriz;
```

Las operaciones ahora se pueden escribir como funciones porque los registros se devuelven por valor:

```
double productoEscalar(Vector const& vec1, Vector const& vec2) {
  double pe = 0;
  for (int i = 0; i < N; i++) pe = pe + vec1.datos[i]*vec2.datos[i];
  return pe;
}
Vector suma(Vector const& vec1, Vector const& vec2) {
  Vector result;
  for (int i = 0; i < N; i++) result.datos[i] = vec1.datos[i] + vec2.datos[i];
  return result;
}
Vector producto(Matriz const& mat, Vector const& vec) {
  Vector result;
  for (int i = 0; i < N; i++) {
    result.datos[i] = productoEscalar(mat.datos[i], vec);
  }
  return result;
}
```

Terminaremos este apartado retomando el ejemplo de los complejos. La práctica matemática utiliza los mismos operadores para manipular números complejos que para números reales o enteros. Por tanto, es razonable sobrecargar los operadores aritméticos para que se puedan usar con números complejos. Por ejemplo, la suma quedaría así:

```
Complejo operator+(Complejo const& c1, Complejo const& c2) {
  Complejo result = {c1.parteReal + c2.parteReal,
                     c1.parteImaginaria + c2.parteImaginaria};
  return result;
}
```

Seguir con esta tarea hasta completar la definición de todas las operaciones sobre complejos es una actividad interesante que dejamos como ejercicio. Pero para que la biblioteca de funciones resultante sea realmente útil, también habrá que añadir operaciones para manipulación cruzada; por ejemplo, para sumar un real y un complejo:

```
Complejo operator+(double const r, Complejo const& c) {
  Complejo result;
  result.parteReal = r + c.parteReal;
  result.parteImaginaria = c.parteImaginaria;
  return result;
}
```

Si además contamos con una sobrecarga cruzada del producto y la definición del número i ($= \sqrt{-1}$),

```
Complejo const I = {0, 1};
```

podremos escribir cosas tan elegantes como `2 + 3*I` para representar al número complejo $2 + 3i$. Esta capacidad para hacer que un tipo nuevo se comporte casi como si estuviera predefinido en el lenguaje es una de las características distintivas de C++.

4.4 Enumeraciones

Las enumeraciones son la vía para definir constantes que queremos utilizar en contextos restringidos. Cuando se define una enumeración, se crea a la vez un tipo y los únicos valores que contiene. Por ejemplo, si queremos marcar que las tres posibles `Direcciones` de movimiento en un espacio tridimensional son `ejeX`, `ejeY` y `ejeZ`, damos la siguiente definición:

```
typedef enum Direcciones {ejeX, ejeY, ejeZ} Direcciones;
```

La sintaxis es muy similar a la de los registros: las enumeraciones se marcan con `enum` y lo que aparece rodeado entre llaves son sus valores. A los tipos definidos como una enumeración se los califica de *enumerados*.

Un tipo enumerado es un tipo entero; sus valores son constantes enteras. En muchas circunstancias podemos ignorar la asignación de valores que realiza C++. Pero en otras, cuando se ven involucrados arrays o bucles, es importante saber que C++ respeta el orden que hemos dado y asigna consecutivamente números a partir del 0.

```
typedef double Caja[ejeZ+1][2];
double volumen(Caja caja) {
  double elVolumen = 1;
  for (int dir = ejeX; dir <= ejeZ; dir++) {
    elVolumen *= caja[dir][1] - caja[dir][0];
  }
  return elVolumen;
}
```

Pero si tenemos alguna preferencia distinta, la podemos imponer:

```
typedef enum Nota {
  ut = 1, re = 3, mi = 5, fa = 6, sol = 8, la = 10, si = 12, silencio = 0
} Nota;
```

Los tipos enumerados no son compatibles entre sí. Son compatibles con los tipos enteros, pero para el paso inverso hay que hacer una conversión anteponiendo el nombre del tipo:

```
Nota n = (Nota)(mi+1);
```

No hay ninguna operación predefinida para los tipos enumerados; siempre que se hace aritmética con ellos es porque se están tratando como algún tipo entero. Por eso, en la función `volumen` se declara como `int` la variable con la que se recorren las direcciones.