Copia, movimiento u omisión de la copia

Tecnología de la Programación de Videojuegos 1 – Curso 2023-2024

Se muestran algunos ejemplos para ilustrar la utilidad y las diferencias entre los constructores y asignaciones por copia y por movimiento.

1 ¿Constructor o asignación?

La clave para distinguir si se llama a un constructor o a una asignación por copia (al `operator=`) es si se aplica sobre un objeto que se está inicializando o sobre uno ya existente.

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

class A {
public:
        A() { cout << "ctor defecto\n"; }
        A(const A& otro) { cout << "ctor copia\n"; }
        A& operator=(const A& otro) { cout << "operator= copia\n" }
        A(A&& otro) { cout << "ctor movimiento\n"; }
};
```

En la declaración de la clase `A` se proporcionan un constructor por defecto, un constructor por copia, un operador de asignación por copia y un constructor por movimiento. Si ejecutamos el siguiente código

```
A a;              // ctor defecto
A b = A();        // ctor defecto
A c(a);           // ctor copia
A d = b;          // ctor copia
a = b;            // operator= copia
```

se mostrará lo que aparece a la derecha de cada línea. La presencia del símbolo `=` no debe confundir: en la declaración de un objeto se utilizan los constructores, independientemente de que en la sintaxis se usen los paréntesis, los corchetes o el signo igual.

2 `ifstream` no es copiable, pero sí movible

La clase `std::ifstream` de la STL (flujo de entrada desde archivo, de la cabecera `fstream`) no es copiable. El constructor por copia se ha eliminado explícitamente.

```
ifstream(const ifstream& rhs) = delete;
```

El estado que guarda el objeto `ifstream` (por ejemplo, un búfer con caracteres preleídos) está sincronizado con el archivo, pero si varios objetos operasen sobre el mismo flujo sería complicado mantenerlos en un estado consistente.

Sin embargo, el constructor `ifstream(ifstream&& rhs)` está definido y sí que es posible mover el flujo de archivo entre los ámbitos de distintas funciones u objetos. Por ejemplo, supongamos que queremos implementar una biblioteca para manipular imágenes, con una función `leeImagen` que recibe la ruta del archivo y devuelve el objeto imagen correspondiente.

```
Imagen leeImagen(const string& ruta) {
        ifstream archivo(ruta, ios::binary);
        char magic = archivo.get();  // Primer byte del archivo

        if (magic == 42) return BMPImage(std::move(archivo));
        else if (magic == 89) return PNGImage(std::move(archivo));
        else if (magic == 255) return JPGImage(std::move(archivo));
        else throw std::invalid_argument("formato desconocido");
}
```

Como se admiten diversos tipos de imágenes, `leeImagen` abre el archivo, lee los primeros bytes para detectar el formato y luego construye un objeto del subtipo específico de imagen (`BMPImage`, `PNGImage`, ...). La responsabilidad sobre el archivo se mueve a ese nuevo objeto para seguir leyendo la imagen.

Alternativamente, se podría haber creado el objeto `ifstream` en memoria dinámica con `new` y haber pasado el puntero. En ese caso tendríamos que asegurarnos de destruir el archivo manualmente en el destructor de las clases, mientras que al mover el objeto y conservarlo como atributo la destrucción es automática.

```
class BMPImage : public Image {
        std::ifstream data;                                          // [...]
public:
        BMPImage(std::ifstream&& archivo) : data(archivo) { /* ... */ }   // [...]
};
```

3 Lectura de múltiples cadenas en un vector

Supongamos que hemos ido apuntando en un archivo las series que hemos visto y queremos leer en un vector las 10 primeras de la lista. Implementaríamos típicamente la lectura con el siguiente bucle:

```
ifstream archivo("lista_series.txt");
vector<string> nombres;

for (int i = 0; i < 10; ++i) {
        string nombre;
        getline(archivo, nombre);
        nombres.push_back(nombre);
}
```

La instrucción `push_back` añade en cada iteración la línea leída `nombre` al vector. Entre esa instrucción y el fin de la iteración

1. Se copia el objeto `nombre` en `nombres[i]`.
   (a) Se reserva memoria nueva para el array dinámico de caracteres de `nombres[i]`.
   (b) Se copian todos los caracteres del array dinámico de `nombre` en el array dinámico de `nombres[i]`.
2. Se destruye el objeto `nombre` al salir de ámbito.
   (a) Se libera la memoria del array dinámico de caracteres de `nombre`.

Si el objeto `nombre` se va a destruir inmediatamente, hemos reservado memoria dinámica nueva y copiado los caracteres tontamente. ¿No sería mejor que `nombres[i]` reutilizara el array dinámico de `nombre`, que ya no le hace falta?

Con el depurador (por ejemplo, el de Visual Studio, aquí con GDB) podemos ver que los arrays internos de caracteres de `nombre` y de `posiciones[i]` (a los que se accede con el método `data`) están en posiciones de memoria distintas. Esto significa que se ha copiado la cadena.

```
(gdb) print nombre.data()
$1 = 0x55555556e4a0 "El Ministerio del Tiempo"
(gdb) print nombres[i].data()
$2 = 0x55555556e510 "El Ministerio del Tiempo"
```

Si se cambia el `push_back` por `nombres.push_back(std::move(nombre))` el resultado es distinto.

```
(gdb) print nombre.data()                                  # antes de ejecutar push_back
$1 = 0x55555556d4a0 "El Ministerio del Tiempo"
(gdb) next                                                 # ejecuta push_back
(gdb) print nombre.data()
$2 = 0x7fffffffe260 ""
(gdb) print nombres[i].data()
$3 = 0x55555556d4a0 "El Ministerio del Tiempo"
```

Véase que `nombres[i]` reutiliza el array dinámico de `nombre` y el objeto `nombre` ahora representa una cadena vacía (un estado válido, pero indeterminado). En efecto, la documentación del constructor de `string` afirma que el constructor por copia tiene complejidad lineal sobre la longitud de la cadena, mientras que el constructor por movimiento la tiene constante.

4 ¿Copia, movimiento o ninguna de las anteriores?

Extendiendo el ejemplo de la clase `A` en el apartado 1, definimos las siguientes funciones:

```
A devuelveA1() {
  A a;
  return a;
}

A devuelveA2() {
  A a[2];
  return a[1];
}

A devuelveA3() {
  A a[2];
  return std::move(a[1]);
}

struct SA {
  A a;
};

A devuelveA4() {
  SA s; return s.a;
}

A devuelveA5() {
  return SA{}.a;
}
```

y ejecutamos las siguientes llamadas:

```
A a1 = devuelveA1();     // (no se imprime nada)
A a2 = devuelveA2();     // ctor copia
A a3 = devuelveA3();     // ctor movimiento
A a4 = devuelveA4();     // ctor copia
A a5 = devuelveA5();     // ctor movimiento
```

El resultado de cada línea es lo que se muestra en el comentario de la derecha (obviando el constructor por defecto). Es llamativo que en la primera inicialización no se ha llamado a ningún constructor de copia o movimiento, como sí se ha hecho en los otros casos. ¿Por qué se hace una copia en esos otros casos si los objetos van a salir de ámbito inmediatamente?

El compilador de C++ puede (o incluso debe) omitir las copias completamente y construir los objetos directamente en el espacio de memoria de su destino final en algunas circustancias (*copy ellision*). No hay que preocuparse por eso porque un constructor bien definido podrá ser omitido sin problema alguno. La documentación de C++ indica en qué circunstancias se omite completamente la copia o se utiliza un movimiento en lugar de una copia, pero la casuística no es siempre fácil de desentrañar. Usando el depurador con el ejemplo anterior se puede ver que la variable `a` de `devuelveA1` y la variable `a1` de `main` tienen la misma dirección de memoria.

```
#2 in devuelveA1 ()
(gdb) print &a
$1 = (A *) 0x7fffffffe497
(gdb) up
#1 in main ()   A a1 = devuelveA1();
(gdb) print &a1
$2 = (A *) 0x7fffffffe497
```

Esto es que `devuelveA1` escribe directamente su resultado sobre la variable local `a1` del `main`. En el caso de `devuelveA2` no se puede aplicar la misma optimización, es decir, `a[1]` no puede ser la variable `a1` del `main`, porque al formar parte de un array, `a[1]` tiene que estar inmediatamente precedido de otro elemento `a[0]` de tipo `A`.