

Pasaje de objetos en C++

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

- Pasaje de objetos
 - Pasaje por referencia
 - Pasaje por copia
 - Pasaje por movimiento: Move semantics
- Asignación
 - Asignación por copia
 - Asignación por movimiento

1

Pasaje de objetos

Pasaje por referencia

Código base

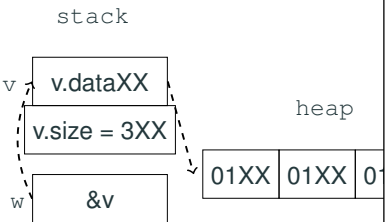
```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(int size) { // create
6         this->data = (int*)malloc(size*sizeof(int));
7         memset(this->data, 0, size*sizeof(int));
8         this->size = size;
9     }
10
11     ~Vector() { // destroy
12         free(this->data);
13     }
14 };
```

2

3

Pasaje por referencia usando punteros

```
1 // con punteros
2 int foo() {
3     Vector v(3);
4     bar(&v);
5
6     v.get(0);
7 }
8
9 void bar(Vector* w) {
10     for (int i = 0; /*...*/)
11         w->set(i, 1);
12 }
```

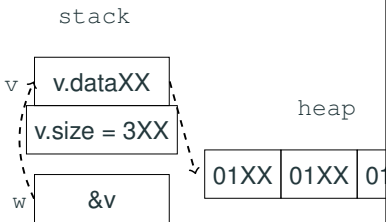


```
5 Vector(int size) { // create
6     data = malloc(..);
7     memset(data, 0 ..);
8     this->size = size;
9 }
10 ~Vector() { // destroy
11     free(data);
```

4

Pasaje por referencia usando referencias

```
1 // con referencias
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector& w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
5 Vector(int size) { // create
6     data = malloc(..);
7     memset(data, 0 ..);
8     this->size = size;
9 }
10 ~Vector() { // destroy
11     free(data);
```

5

- En C todo se pasa por copia. Si queremos pasar por referencia en realidad se pasa por copia un puntero.
- En C++ podemos usar el pasaje por referencia. Una referencia es como un alias del objeto referenciado.

Diferencias entre referencias y punteros

```
1 | int* p = nullptr;
2 | int* q;
3 |
4 | int i = 1, j = 2;
5 |
6 | int* r = &i;
7 | *r = j;
```

```
1 | int& p = nullptr;
2 | int& q;
3 |
4 | int i = 1, j = 2;
5 |
6 | int& r = i;
7 | r = j;
```

6

- Las referencias en C++ deben ser inicializadas al construirse y una vez que referencian a algun objeto no pueden referenciar a otro.
- Las referencias funcionan como un alias y el compilador en algunos casos ni siquiera reservara memoria para una referencia.
- En cambio, los punteros pueden crearse sin inicializar, cambiar de objeto al que apuntan y siempre consumen memoria.
- Como colorario, las referencias no pueden referenciar a nulls. Una referencia nunca puede ser null! Es muy útil y reduce la posibilidad de crashes.

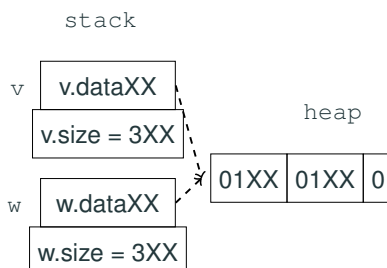
Pasaje de objetos

Pasaje por copia

7

Pasaje por copia naive: bit a bit

```
1 | // por copia
2 | int foo() {
3 |     Vector v(3);
4 |     bar(v);
5 |
6 |     v.get(0);
7 | }
8 |
9 | void bar(Vector w) {
10 |     for (int i = 0; /*...*/)
11 |         w.set(i, 1);
12 | }
```



```
5 | Vector(int size) { // create
6 |     data = malloc(..);
7 |     memset(data, 0 ..);
8 |     this->size = size;
9 | }
```

8

- La copia tanto en C como en C++ es bit a bit y funciona bien para objetos simples.
- Pero cuando hay punteros, la copia es del puntero y no del valor apuntado: la copia es superficial y no en profundidad (deep copy).
- Con 2 objetos apuntando al mismo heap, al destruirse uno libera el heap dejando al segundo objeto apuntando a la nada (use after free).
- Y peor, cuando el segundo objeto se destruya también liberará el heap, otra vez (double free).
- No sólo hay problemas con los punteros y el heap, sino también con otros tipos de indirecciones como los file descriptors, sockets, threads entre otros.

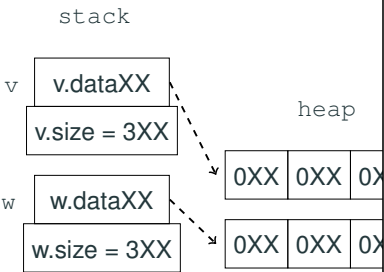
Constructor por copia

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(const Vector &other) {
6         this->data = (int*)malloc(other.size*sizeof(int));
7         this->size = other.size;
8
9         memcpy(this->data, other.data, this->size);
10    }
11
12};
```

- Para crear un objeto nuevo a partir de otro se invoca al constructor por copia.
- Como cualquier otro constructor, el constructor por copia tiene una member initialization list para pasarle argumentos a los constructores de sus atributos.
- Todos los objetos en C++ son copiables por default. Si un objeto no tiene un constructor por copia, C++ le creará un constructor por copia por default que implementa una copia bit a bit naive. Por esta razón es muy fácil que un objeto se copie sin querer, algo que es difícil de debuggear.

Pasaje por copia: constructor por copia

```
1 // por copia
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
5 Vector(int size) { // create
6     data = malloc(..);
7     memset(data, 0 ..);
8     this->size = size;
9 }
```

- En C y en C++ el pasaje por default es por copia: cuidado de hacer una copia sin intención, puede traer un comportamiento inesperado (como en el ejemplo) y ser ineficiente.
- Si no se implementa un constructor por copia se corre el riesgo de caer en un use after free o double free o similar.
- Evitar a toda costa las copias, son la principal causa de ineficiencias en código C y C++.

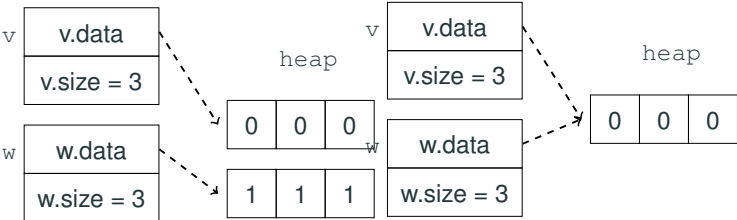
Pasaje de objetos

Pasaje por movimiento: Move semantics

Ownership

Cada objeto se hace cargo de sus recursos. Tienen el ownership de ellos.

Ambos objetos comparten los recursos: no hay un ownership claro.



Constructor por movimiento: transferencia del ownership

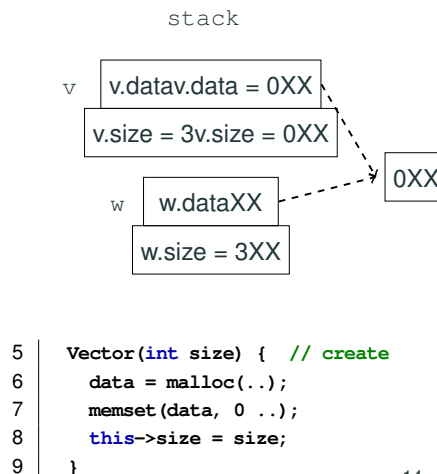
```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(Vector&& other) {
6         this->data = other.data;
7         this->size = other.size;
8
9         other.data = nullptr;
10        other.size = 0;
11    }
12
13    ~Vector() {
14        if (data)
15            free(data);
16    }
17 };
```

13

- A diferencia de una copia, el constructor por movimiento le roba o mueve los atributos del objeto fuente.
- Para marcar el cambio de ownership es necesario modificar al objeto fuente (`other`) (por eso no debe ser una constante). Debe dejar de apuntar a los recursos ahora apropiados, de otro modo tendríamos 2 objetos apuntando a un mismo recurso y un bug de memoria a la vuelta de la esquina.
- Es importante aclarar que luego que el objeto fue movido (`other`) debe seguir siendo válido de tal manera que se le puede ejecutar sobre `other` el operador asignación y el destructor. La implementación de estos dos métodos deben ser acordes como en el ejemplo en donde el destructor pregunta si `data == nullptr`
- Cómo se implementa la transferencia del ownership dependerá de cada objeto. En este caso, al poner el puntero `data = nullptr` indicamos que no tiene más el ownership del recurso y por lo tanto no tiene que destruirlo.

Pasaje por movimiento

```
1 // por movimiento
2 int foo() {
3     Vector v(3);
4     bar(std::move(v));
5
6     v.get(0); // ??
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
5 Vector(int size) { // create
6     data = malloc(..);
7     memset(data, 0 ..);
8     this->size = size;
9 }
```

14

Motivación: retorno de un Socket

Por referencia? Ineficiente o viola RAIL

```
1 Socket s;
2 acep.accept(s);
10 void accept(Socket &s) {
11     close(s.fd);
12     s.fd = ::accept(/*...*/); // accept de C
13 }
```

Usando el heap? Y si nos olvidamos del `delete`?

```
1 Socket *s = acep.accept();
10 Socket* accept() {
11     int fd = ::accept(/*...*/); // accept de C
12     return new Socket(fd);
13 }
```

Retornar una copia? Tiene sentido? Simplemente No.

15

Solución?: Mover el Socket!!

Por movimiento!

```
1 Socket s = acep.accept();
10 Socket accept() {
11     int fd = ::accept(/*...*/); // accept de C
12     return std::move(Socket(fd));
13 }
```

- El constructor `Socket(int)` debe ser privado.
- El socket creado dentro del método `accept` es movido hacia afuera.
- Todos los objetos involucrados viven en el stack y por lo tanto se destruyen automáticamente.

16

- Por referencia: Ineficiente, creamos un `fd` para cerrarlo y asignar otro.
- Por referencia: O bien, el constructor de `socket` no abren ningún `fd` (pero entonces no serían RAIL)
- Por heap: El constructor `socket(int)` debe ser privado.
- Por heap: Perdemos la ventaja de usar el stack.

- El método `accept` retorna un nuevo objeto `Socket`.
- El método no quiere tener el ownership del nuevo socket creado, quiere moverlo y dárselo a quien lo llamó.
- En C y en C++ antes del estándar C++11 no había otra forma que o pasaje por referencia (lo que implicaba que había que construir previamente un `Socket` dummy para luego inicializarlo correctamente dentro de `accept`) o bien retornarlo usando el heap (perdiendo el beneficio de ser RAII).
- El compilador puede deducir que el objeto `accepted` se lo desea mover. Usar `std::move` para hacerlo explícito.

Otro ejemplo: pasando objetos a un hilo

```

10 std::thread aceptar_un_cliente(Socket &aceptador) {
11     Socket skt_cliente = aceptador.accept();
12
13     // movimiento de un socket, todo ok
14     std::thread t {manejador_del_cliente,
15                  std::move(skt_cliente)};
16
17     return std::move(t);
18 } // <--el socket skt_cliente se destruye, pero como se movio
19 // no deberia pasar nada (siempre que se implemente el
20 // constructor por movimiento y el destructor acorde!)

```

17

- La función crea un nuevo socket `skt_cliente` y se lo pasa a un hilo para que lo procese en paralelo.
- El fin de `aceptar_un_cliente` no implica que el socket `skt_cliente` se deba cerrar: el lifetime del objeto debería estar atado al del hilo.
- No podemos pasar una copia ya que no tiene sentido copiar un socket.
- Tampoco una referencia ya que el objeto `skt_cliente` vive en el stack frame de `aceptar_un_cliente` y se destruirá al finalizar esta.
- En C y en C++ antes del estándar C++11 no hay otra alternativa que poner el socket `skt_cliente` en el heap perdiendo los beneficios RAII. En C++11 se lo mueve directamente.
- Lo mismo ocurre con el retorno del objeto thread `t`.

Asignación

Asignación por copia

18

Asignación

```

1 Vector f(Vector v) {
2     Vector a(v);
3     Vector b = v;
4
5     Vector c(5);
6
7     c = v;
8
9     return v;
10 }

```

- En la línea 1 se recibe por copia un vector al que llamaremos `v`.
- En la línea 2 y 3 se crean 2 vectores más copiándose de `v`, ambos llaman al constructor por copia.
- En la línea 9 se retorna un vector por copia también salvo que `Vector` implemente el constructor por movimiento en cuyo caso `v` se mueve y no se copia.
- En la línea 7 sucede algo distinto. El vector `c` copia el contenido del vector `v`. Pero el objeto `c` ya estaba creado así que en vez de llamar al constructor por copia llama al operador asignación por copia.

19

Asignación por copia: un objeto creado copiando de otro

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector& operator=(const Vector &other) {
6         if (this == &other) {
7             return *this; // other is myself!
8         }
9
10        if (this->data)
11            free(this->data);
12
13        this->data = (int*)malloc(other.size*sizeof(int));
14        this->size = other.size;
15        memcpy(this->data, other.data, this->size);
16
17        return *this;
18    }
19};
```

20

- Para copiar el contenido de un objeto en otro ya creado se usa el operador asignación.
- Como el objeto `this` ya está creado, debemos recordar que todos sus atributos están ya creados: no podemos cambiar ninguno de sus atributos constantes.
- Validar que no nos hayan quitado el ownership de nuestros recursos.
- Todos los objetos en C++ son copiables por asignación así que si un objeto no implementa la sobrecarga del operador asignación, C++ le creará una implementación por default que hará una copia bit a bit naive.
- También es posible que nos asignemos a nosotros mismos (haciendo `vec = vec;`). Debemos programar el operador asignación de tal forma que evite copiarse a sí mismo.
- El operador asignación no es el único operador que se puede sobrecargar. Ya veremos otros y en más detalle en las próximas clases.

Asignación

Asignación por movimiento

21

Asignación por movimiento

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector& operator=(Vector&& other) {
6         if (this == &other) {
7             return *this; // other is myself!
8         }
9
10        if (this->data)
11            free(this->data);
12
13        this->data = other.data;
14        this->size = other.size;
15
16        other.data = nullptr;
17        other.size = 0;
18
19        return *this;
20    }
21};
```

Ej Asignación por movimiento: swap de objetos

```
10 void swap(Vector& a, Vector& b) {
11     Vector t = a; // copia (constructor)
12     a = b; // copia (asignación)
13     b = t; // copia (asignación)
14 }
15
16 void swap(Vector& a, Vector& b) {
17     Vector t = std::move(a); // a se mueve a t (constructor)
18     a = std::move(b); // b se mueve a a (asignación)
19     b = std::move(t); // t se mueve a b (asignación)
20 }
21
```

23

Asignación

Objetos no copiables

24

Objetos no copiables

```
1 struct File {  
2     public:  
3     File copy(const char *to_where) { ... }  
4  
5     private:  
6     File(const File &other) = delete;  
7     File& operator=(const File &other) = delete;  
8  
9 };
```

25

- En C++11 podemos decir que tanto el constructor por copia como el de asignación están borrados (**delete**). Si en algún momento intentamos hacer una copia el compilador dará un error.
- Pero si trabajamos en C++98, debemos usar algún workaround: declarar y definir el constructor por copia y el operador asignación y que su implementación sea fallar (lanzar una excepción). El intento fallido de copia se detecta en runtime.
- Otra forma sería declarar pero no definir ni el constructor por copia ni el operador asignación y hacerlos privados. El intento fallido de copia se detecta en tiempo de compilación y linkeo.

Take home message

- En el 99% de los casos, no quieres que tus clases sean copiables (sea por que no tiene sentido o por performance). Deshabilitar el constructor y operador asignación por copia.
- Implementar siempre move semantics (constructor y operador asignación).
- Usar const en donde sea posible.

26

Appendix

Referencias

Referencias i

 Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, Fourth Edition.