Pasaje de objetos en C++ - Pasajes

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

Facultad de Ingeniería
Universidad de Buenos Aires

Pasaje de objetos

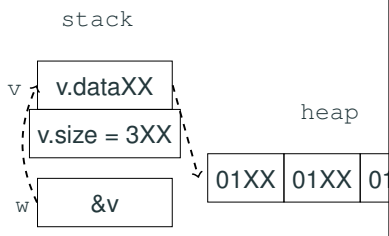
Pasaje por referencia

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 };
```


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 }
```

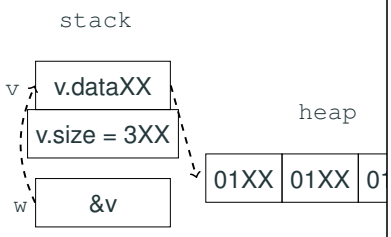



```
5  Vector(int size) {  // create
6    data = malloc(..);
7    memset(data, 0 ..);
8    this->size = size;
9  }

10 ~Vector() { // destroy
11   free(data);
```


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);
```


- 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;
```

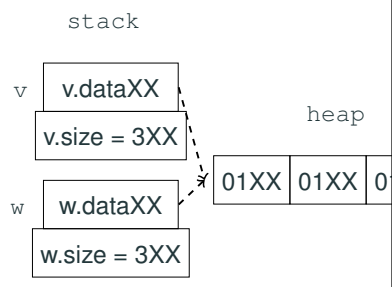

- Las referencias en C++ deben ser inicializadas al construirse y una vez que referencian a algun objeto no pueden referenciar a otro.
- Las referencias funcionan como un alias y el compilador en algunos casos ni siquiera reservara memoria para una referencia.
- En cambio, los punteros pueden crearse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

Pasaje de objetos

Pasaje por copia

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  }
```


- 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.

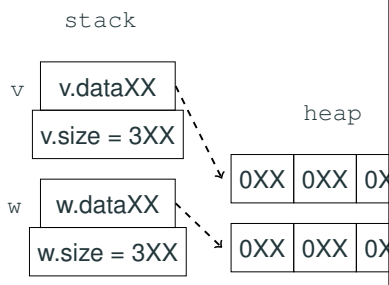
Constructor por copia

```
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 razon 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  }
```

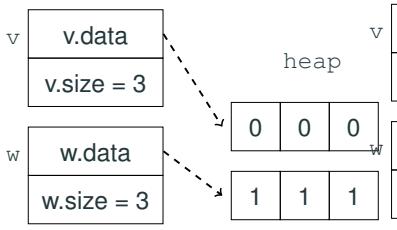

- En C y en C++ el pasaje por default es por copia: cuidado de hacer una copia sin intención, puede traer un comportamiento inesperado (como en el ejemplo) y ser ineficiente.
- Si no se implementa un constructor por copia se corre el riesgo de caer en un use after free o double free o similar.
- Evitar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

Pasaje de objetos

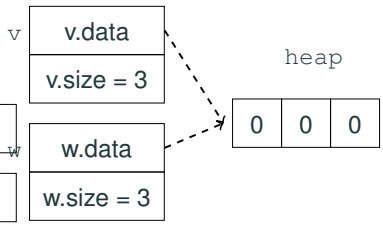
Pasaje por movimiento: Move semantics

Ownership

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



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



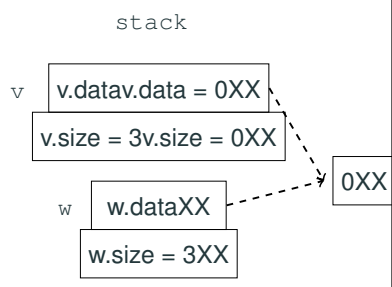
Constructor por movimiento: transferencia del ownership

```
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 };
```

- 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  }
```


Motivación: retorno de un Socket

Por referencia? Ineficiente o viola RAII

```
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.

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

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.

- El método `accept` retorna un nuevo objeto `Socket`.
- El método no quiere tener el ownership del nuevo socket creado, quiere moverlo y darselo 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!)
```


- 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`**.