

# 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

# **Pasaje de objetos**

---

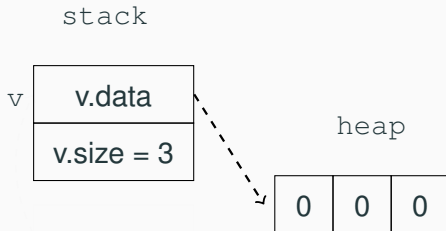
## **Pasaje por referencia**

## 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

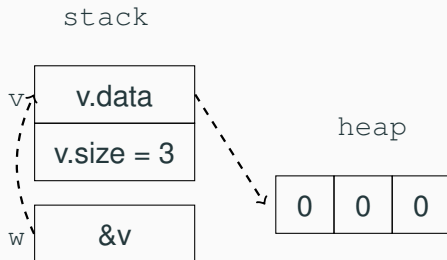
```
1 // con punteros
2 int foo() {
3     Vector v(3);
4     bar(&v);
5
6     v.get(0);
7 }
8
9 void bar(Vector* w) {
10     for (int i = 0; /*...*/)
11         w->set(i, 1);
12 }
```



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

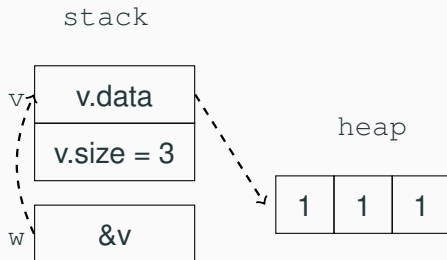
## Pasaje por referencia usando punteros

```
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 }
```



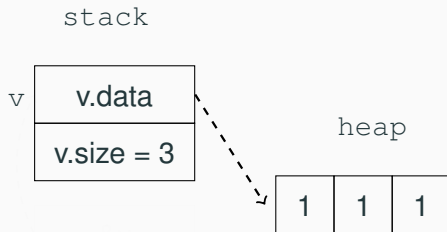
# Pasaje por referencia usando punteros

```
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 }
```



## Pasaje por referencia usando punteros

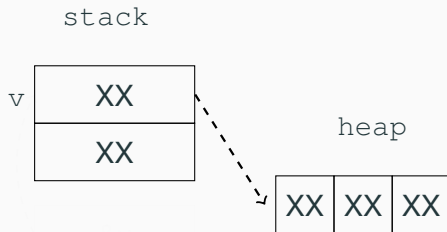
```
1 // con punteros
2 int foo() {
3     Vector v(3);
4     bar(&v);
5
6     v.get(0);
7 }
8
9 void bar(Vector* w) {
10     for (int i = 0; /*...*/)
11         w->set(i, 1);
12 }
```





# Pasaje por referencia usando punteros

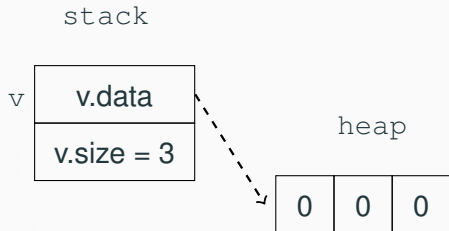
```
1 // con punteros
2 int foo() {
3     Vector v(3);
4     bar(&v);
5
6     v.get(0);
7 }
8
9 void bar(Vector* w) {
10     for (int i = 0; /*...*/)
11         w->set(i, 1);
12 }
```



```
10 ~Vector() { // destroy
11     free(data);
12 }
```

## Pasaje por referencia usando referencias

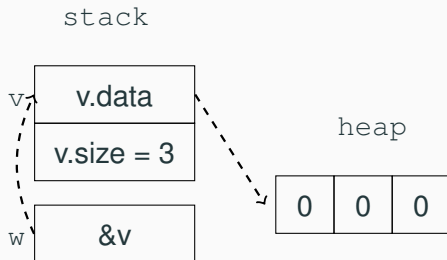
```
1 // con referencias
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector& w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



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

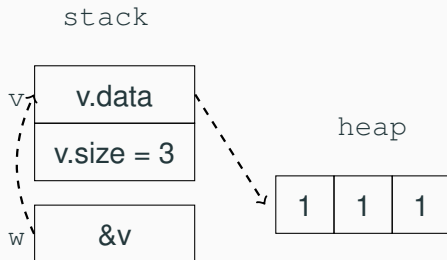
## Pasaje por referencia usando referencias

```
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 }
```



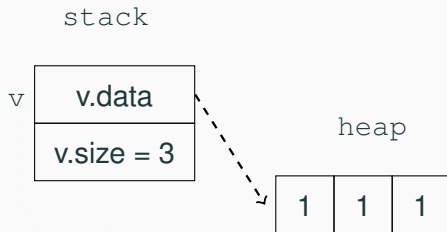
## Pasaje por referencia usando referencias

```
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 }
```



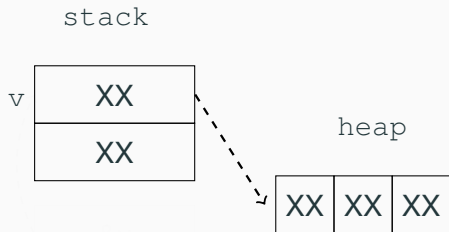
# Pasaje por referencia usando referencias

```
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 }
```



# 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 }
```



```
10 ~Vector() { // destroy
11     free(data);
12 }
```

# 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;
```

## 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;
```



## 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;
```

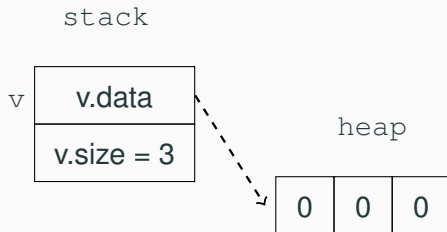
# **Pasaje de objetos**

---

**Pasaje por copia**

## Pasaje por copia naive: bit a bit

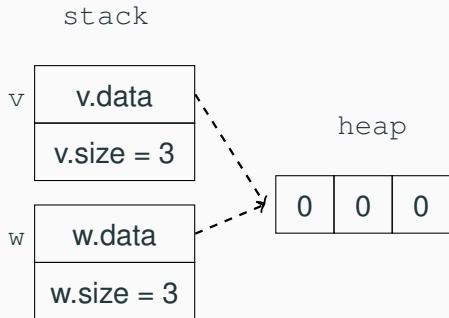
```
1 // por copia
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



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

## Pasaje por copia naive: bit a bit

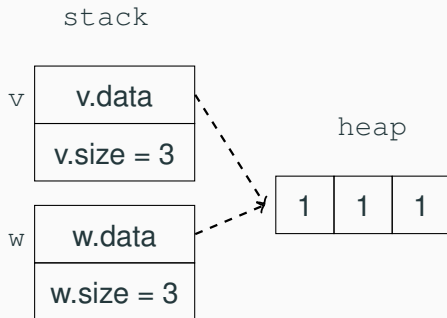
```
1 // por copia
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



Constructor por copia por default.

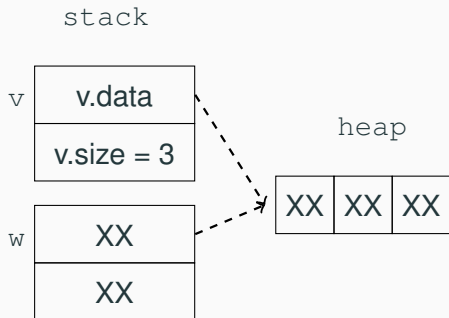
## Pasaje por copia naive: bit a bit

```
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 }
```



## Pasaje por copia naive: bit a bit

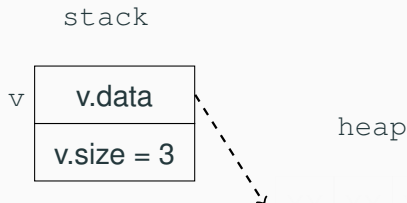
```
1  // por copia
2  int foo() {
3      Vector v(3);
4      bar(v);
5
6      v.get(0);
7  }
8
9  void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
10 ~Vector() { // destroy
11     free(data);
12 }
```

## 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 }
```



Use after free!!

## 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 }
```



Double free!!

```
10 ~Vector() { // destroy
11     free(data);
12 }
```

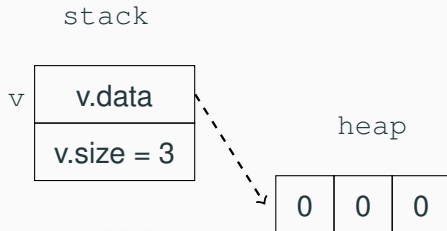


# Constructor por copia

```
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 };
```

## Pasaje por copia: constructor por copia

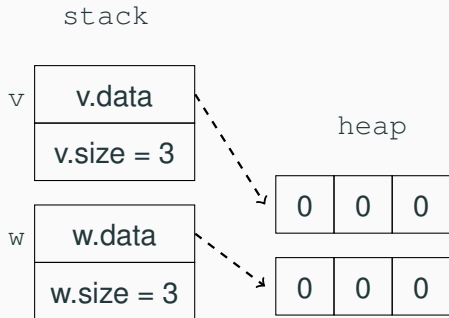
```
1 // por copia
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



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

# Pasaje por copia: constructor por copia

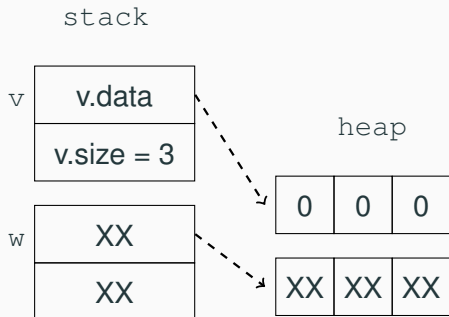
```
1 // por copia
2 int foo() {
3     Vector v(3);
4     bar(v);
5
6     v.get(0);
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
5     Vector(const Vector &other) {
6         data = malloc(..);
7         size = other.size;
8
9         memcpy(data, other.data, ..);
10    }
```

## Pasaje por copia: constructor por copia

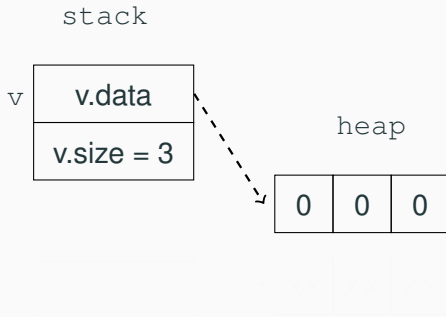
```
1  // por copia
2  int foo() {
3      Vector v(3);
4      bar(v);
5
6      v.get(0);
7  }
8
9  void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
10 ~Vector() { // destroy
11     free(data);
12 }
```

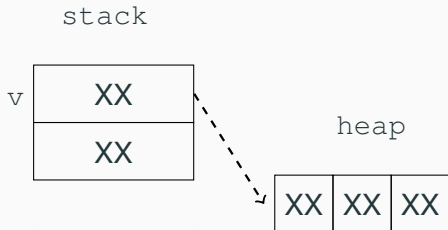
## Pasaje por copia: constructor por copia

```
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 }
```



## 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 }
```



```
10 ~Vector() { // destroy
11     free(data);
12 }
```

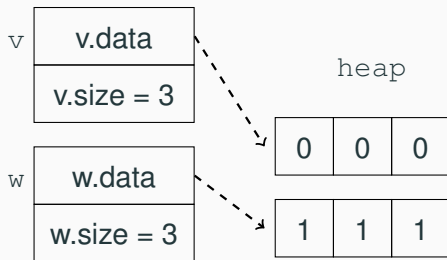
## **Pasaje de objetos**

---

**Pasaje por movimiento: Move semantics**

# Ownership

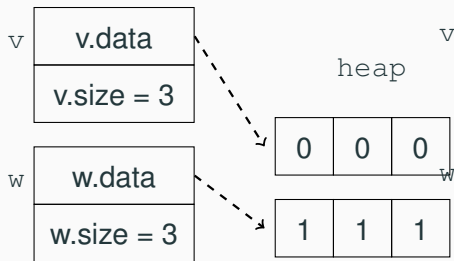
Cada objeto se hace cargo de sus recursos. Tienen el ownership de ellos.



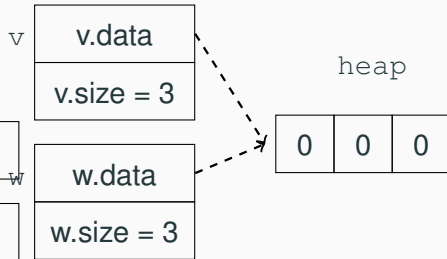


# Ownership

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 };
```

## 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 };
```

## Constructor por movimiento: transferencia del ownership

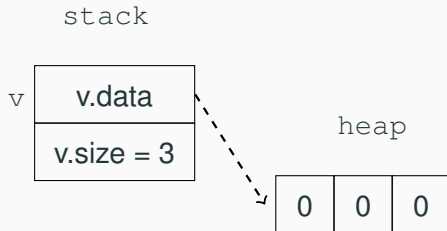
```
1 struct Vector {  
2     int *data;  
3     int size;  
4  
5     Vector(Vector&& other) {  
6         this->data = other.data;  
7         this->size = other.size;  
8  
9         other.data = nullptr;  
10        other.size = 0;  
11    }  
12  
13    ~Vector() {  
14        if (data)  
15   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

## 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 };
```

# Pasaje por movimiento

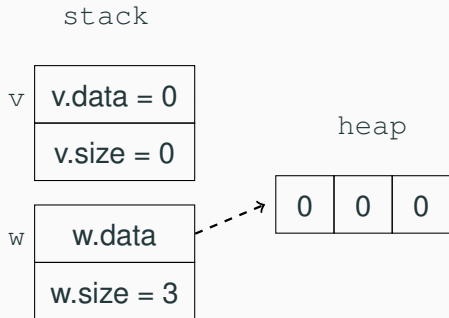
```
1 // por movimiento
2 int foo() {
3     Vector v(3);
4     bar(std::move(v));
5
6     v.get(0); // ??
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



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

# Pasaje por movimiento

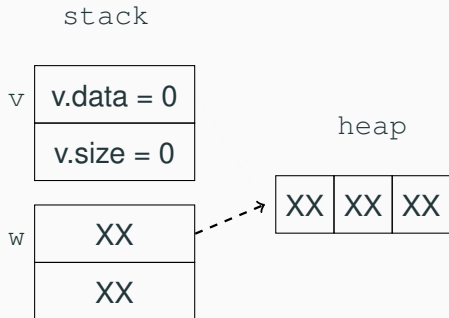
```
1 // por movimiento
2 int foo() {
3     Vector v(3);
4     bar(std::move(v));
5
6     v.get(0); // ??
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



```
5 Vector(Vector&& other) {
6     this->data = other.data;
7     this->size = other.size;
8
9     other.data = nullptr;
10    other.size = 0;
```

## 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 }
```

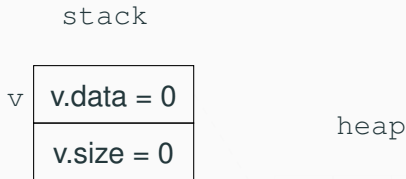


```
10 ~Vector() { // destroy
11     if (data)
12         free(data);
13 }
```



## Pasaje por movimiento

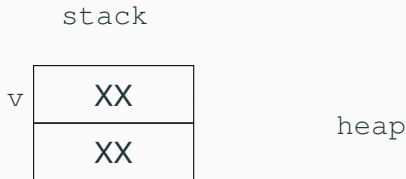
```
1 // por movimiento
2 int foo() {
3     Vector v(3);
4     bar(std::move(v));
5
6     v.get(0); // ??
7 }
8
9 void bar(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12 }
```



Use after free!! Pero fue mi culpa. Cuando un objeto es movido solo se le puede invocar el operador asignación o el destructor, cualquier otra cosa esta mal.

## 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 }
```



Pero no hay un double free.

```
10 ~Vector() { // destroy
11     if (data)
12         free(data);
13 }
```

## Motivación: retorno de un Socket

Por referencia?

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

## Motivación: retorno de un Socket

Por referencia? Ineficiente o viola RAI

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

# 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?

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

## Motivación: retorno de un Socket

Por referencia? Ineficiente o viola RAI

```
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 | }
```

## Motivación: retorno de un Socket

Por referencia? Ineficiente o viola RAI

```
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?

## 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.



## 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.

## 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!)
```

# Asignación

---

## Asignación por copia

# 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 }
```

# 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 }
```

## 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 };
```

## 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 };
```

# 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 };
```



## 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 };
```

# **Asignación**

---

## **Asignación por movimiento**

# 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;
```

## Ej Asignación por movimiento: swap de objetos

```
10 void swap(Vector& a, Vector& b) {  
11     Vector t = a; // copia (constructor)  
12     a = b; // copia (asignacion)  
13     b = t; // copia (asignacion)  
14 }
```

```
10 void swap(Vector& a, Vector& b) {  
11     Vector t = std::move(a); // a se mueve a t (constructor)  
12     a = std::move(b); // b se mueve a a (asignacion)  
13     b = std::move(t); // t se mueve a b (asignacion)  
14 }
```

# Asignación

---

**Objetos no copiables**

# 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 };
```

# **Appendix**

---

## **Referencias**



Bjarne Stroustrup.

***The C++ Programming Language.***

Addison Wesley, Fourth Edition.