

Estructuras de Datos

Ownership y templates

RAII, copiar y mover, templates, y la pila como tipo abstracto

Gerson Garrido

Semana 2

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Del owner por convención al owner en el tipo
2. RAII
3. La copia superficial, vista con ASan
4. La regla de cinco
5. Copiar de verdad: copy and swap
6. Mover: qué hace `std::move`, y qué no
7. Templates
8. Tipo abstracto, implementación e invariantes: la pila

> **Lectura**
>
> Apuntes de la semana 2. Open Data Structures, secs. 1.2 y 2.1. Código en `code/week-02`.

SECCIÓN 01

El owner en el tipo

01



En C, el owner era una convención

owner_in_c.c

```c
struct int_array numbers = int_array_new(4);
if (numbers.data == NULL) {
    return 1;
}
numbers.data[0] = 7;
printf("%d\n", numbers.data[0]);
int_array_free(&numbers);
```

En C, el owner era una convención

owner_in_c.c

```c
struct int_array numbers = int_array_new(4);
if (numbers.data == NULL) {
    return 1;
}
numbers.data[0] = 7;
printf("%d\n", numbers.data[0]);
int_array_free(&numbers);
```

- Si olvidan `int_array_free`, hay una fuga.

En C, el owner era una convención

owner_in_c.c

```c
struct int_array numbers = int_array_new(4);
if (numbers.data == NULL) {
    return 1;
}
numbers.data[0] = 7;
printf("%d\n", numbers.data[0]);
int_array_free(&numbers);
```

- Si olvidan `int_array_free`, hay una fuga.
- Si hay tres `return`, hay que liberar en los tres.

En C, el owner era una convención

owner_in_c.c

```c
struct int_array numbers = int_array_new(4);
if (numbers.data == NULL) {
    return 1;
}
numbers.data[0] = 7;
printf("%d\n", numbers.data[0]);
int_array_free(&numbers);
```

- Si olvidan `int_array_free`, hay una fuga.
- Si hay tres `return`, hay que liberar en los tres.
- Si dos variables guardan el mismo `struct`, nada dice cuál libera.

En C, el owner era una convención

owner_in_c.c

```c
struct int_array numbers = int_array_new(4);
if (numbers.data == NULL) {
    return 1;
}
numbers.data[0] = 7;
printf("%d\n", numbers.data[0]);
int_array_free(&numbers);
```

- Si olvidan `int_array_free`, hay una fuga.
- Si hay tres `return`, hay que liberar en los tres.
- Si dos variables guardan el mismo `struct`, nada dice cuál libera.

En C++, liberar puede ser parte del tipo. Hacen falta dos piezas: `new` y `delete`, y el destructor.

`new` reserva y construye; `delete` destruye y libera

new_basics.cpp

```cpp
std::string* name = new std::string("pila");
delete name;

std::size_t n = 4;
int* zeros = new int[n]();
std::string* words = new std::string[n];
delete[] zeros;
delete[] words;
```

	`new`
Recibe	Un tipo y argumentos
Devuelve	Un puntero del tipo correcto
Constructor	Corre, para cada elemento
Sin memoria	Lanza `std::bad_alloc`

`new` reserva y construye; `delete` destruye y libera

new_basics.cpp

```cpp
std::string* name = new std::string("pila");
delete name;

std::size_t n = 4;
int* zeros = new int[n]();
std::string* words = new std::string[n];
delete[] zeros;
delete[] words;
```

	`new`
Recibe	Un tipo y argumentos
Devuelve	Un puntero del tipo correcto
Constructor	Corre, para cada elemento
Sin memoria	Lanza `std::bad_alloc`

- `new int[n]` deja basura; `new int[n]()` deja ceros.

`new` reserva y construye; `delete` destruye y libera

new_basics.cpp

```cpp
std::string* name = new std::string("pila");
delete name;

std::size_t n = 4;
int* zeros = new int[n]();
std::string* words = new std::string[n];
delete[] zeros;
delete[] words;
```

	`new`
Recibe	Un tipo y argumentos
Devuelve	Un puntero del tipo correcto
Constructor	Corre, para cada elemento
Sin memoria	Lanza `std::bad_alloc`

- `new int[n]` deja basura; `new int[n]()` deja ceros.
- `new std::string[n]` construye *n* cadenas vacías.

`new` reserva y construye; `delete` destruye y libera

new_basics.cpp

```cpp
std::string* name = new std::string("pila");
delete name;

std::size_t n = 4;
int* zeros = new int[n]();
std::string* words = new std::string[n];
delete[] zeros;
delete[] words;
```

	`new`
Recibe	Un tipo y argumentos
Devuelve	Un puntero del tipo correcto
Constructor	Corre, para cada elemento
Sin memoria	Lanza `std::bad_alloc`

- `new int[n]` deja basura; `new int[n]()` deja ceros.
- `new std::string[n]` construye *n* cadenas vacías.
- `new[]` va con `delete[]`. Mezclarlos es comportamiento indefinido, y ASan reporta `alloc-dealloc-mismatch`.

El destructor *(destructor)*

Destructor

Método que corre cuando un objeto muere: `~Noisy()`. Sin argumentos, sin retorno, y uno solo por clase.

Cuándo corre lo decide el lenguaje:

El destructor *(destructor)*

Destructor

Método que corre cuando un objeto muere: `~Noisy()`. Sin argumentos, sin retorno, y uno solo por clase.

Cuándo corre lo decide el lenguaje:

- Una local muere al salir de su **alcance** *(scope)*: por el final, por un `return` o por una excepción que se atrape.

El destructor *(destructor)*

Destructor

Método que corre cuando un objeto muere: `~Noisy()`. Sin argumentos, sin retorno, y uno solo por clase.

Cuándo corre lo decide el lenguaje:

- Una local muere al salir de su **alcance** *(scope)*: por el final, por un `return` o por una excepción que se atrape.
- Las locales de un bloque mueren en **orden inverso** al de su creación.

El destructor *(destructor)*

Destructor

Método que corre cuando un objeto muere: `~Noisy()`. Sin argumentos, sin retorno, y uno solo por clase.

Cuándo corre lo decide el lenguaje:

- Una local muere al salir de su **alcance** *(scope)*: por el final, por un `return` o por una excepción que se atrape.
- Las locales de un bloque mueren en **orden inverso** al de su creación.
- Un objeto creado con `new` muere en el `delete`, y solo entonces.

Veámoslo correr

scope_trace.cpp

```cpp
int first_positive(int a, int b) {
    Noisy guard("guard");
    if (a > 0) {
        return a;
    }
    return b;
}
...
{
    Noisy x("x");
    Noisy y("y");
    std::cout << "  fin del bloque\n";
}
```

Por cualquier camino de salida del bloque. Lo que se escribe en un destructor **no se puede olvidar**.

terminal

```
$ build/scope_trace
bloque:
  crea x
  crea y
  fin del bloque
  destruye y
  destruye x
retorno temprano:
  crea guard
  destruye guard
  resultado 5
new y delete:
  crea z
  z sigue vivo
  destruye z
fin de main
```

SECCIÓN 02

RAII

02

RAII

> **RAII** *(resource acquisition is initialization)*
> Adquirir un recurso en el constructor y liberarlo en el destructor.

errors/shallow_copy.cpp

```cpp
class IntBuffer {
public:
    explicit IntBuffer(std::size_t size) : data_(new int[size]()), size_(size) {}
    ~IntBuffer() {
        delete[] data_;
    }
private:
    int* data_;
    std::size_t size_;
};
```

El mismo owner de CS201, ahora automático

	C, en CS201	C++, con RAII
Quién es el owner	Lo dice un comentario	Lo dice el tipo
Cuándo se libera	Cuando alguien se acuerde	Al salir del alcance
Tres `return`	Tres `free`, o un `goto`	Nada que agregar
Quién toca el puntero	Cualquiera	Solo los métodos

En la biblioteca

`std::string`, `std::vector` y `std::ifstream` son clases con RAII. Por eso la semana pasada no hizo falta liberar nada. Regla del curso: `new` y `delete` solo dentro de una clase con RAII.

SECCIÓN 03

La copia superficial

03

Copia superficial: dos objetos, un mismo arreglo

errors/shallow_copy.cpp

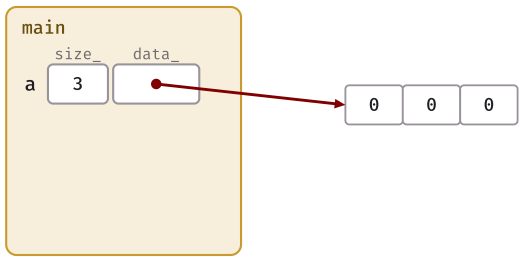
```cpp
int main() {
    IntBuffer a(3);
    a.at(0) = 10;
    IntBuffer b = a;
    b.at(0) = 99;
    std::cout << "a.at(0) = " << a.at(0) << "\n";
    return 0;
}
```

La **copia por defecto** copia cada campo. Para un puntero, copia la dirección, no el bloque.

Un solo bloque, dos owners

CÓDIGO

```
IntBuffer a(3);
a.at(0) = 10;
IntBuffer b = a;
b.at(0) = 99;
}
```

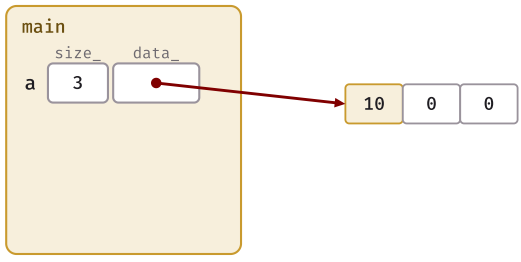


El constructor de a reserva un bloque de 3 enteros, en cero

Un solo bloque, dos owners

CÓDIGO

```
IntBuffer a(3);
a.at(0) = 10;
IntBuffer b = a;
b.at(0) = 99;
}
```

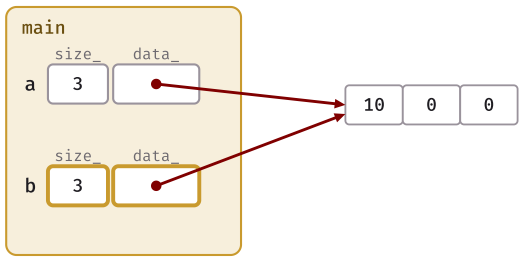


a escribe en su bloque

Un solo bloque, dos owners

CÓDIGO

```
IntBuffer a(3);
a.at(0) = 10;
IntBuffer b = a;
b.at(0) = 99;
}
```

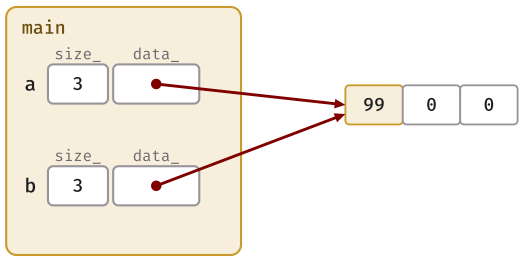


La copia por defecto copia los campos. data_ es una dirección: el bloque no se copió

Un solo bloque, dos owners

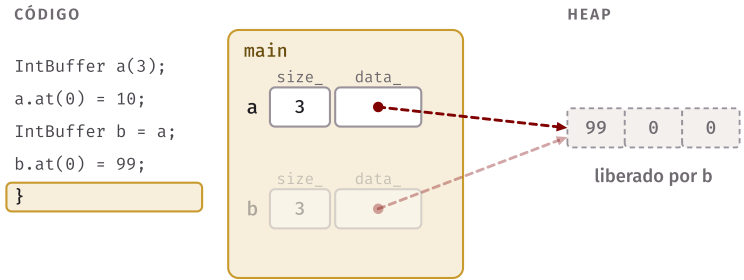
CÓDIGO

```
IntBuffer a(3);
a.at(0) = 10;
IntBuffer b = a;
b.at(0) = 99;
}
```



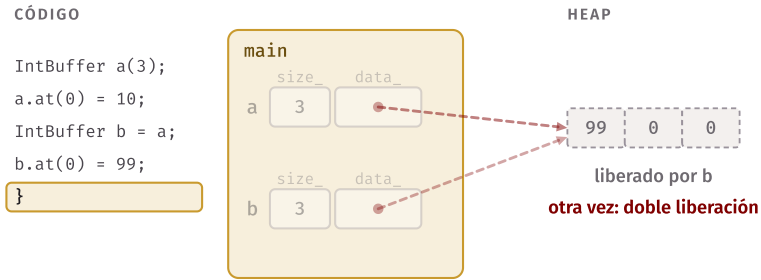
b escribe en el bloque de a. Por eso el programa imprime a.at(0) = 99

Un solo bloque, dos owners



Al cerrar main, b muere primero: su destructor hace delete[]. a queda colgando

Un solo bloque, dos owners



Después muere a, y hace delete[] del mismo bloque: attempting double-free

El compilador no dice nada. ASan, sí

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/shallow_copy.cpp -o build/shallow_copy
$ build/shallow_copy
a.at(0) = 99
=================================================================
==1341==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #0 0xffff9a79b9c0 in operator delete[](void*) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:155
    #1 0xaaaaca672634 in IntBuffer::~IntBuffer() errors/shallow_copy.cpp:9
    #2 0xaaaaca672294 in main errors/shallow_copy.cpp:30
0x502000000010 is located 0 bytes inside of 12-byte region [0x502000000010,0x50200000001c)
freed by thread T0 here:
    #0 0xffff9a79b9c0 in operator delete[](void*) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:155
    #1 0xaaaaca672634 in IntBuffer::~IntBuffer() errors/shallow_copy.cpp:9
    #2 0xaaaaca67228c in main errors/shallow_copy.cpp:30
previously allocated by thread T0 here:
    #0 0xffff9a79b090 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:98
    #1 0xaaaaca67239c in IntBuffer::IntBuffer(unsigned long) errors/shallow_copy.cpp:7
    #2 0xaaaaca671f04 in main errors/shallow_copy.cpp:24
...
SUMMARY: AddressSanitizer: double-free ../../../../src/libsanitizer/asan/asan_new_delete.cpp:155 in operator delete[](void*)
```

Cómo leer ese informe

1. **Qué pasó:** `attempting double-free`, en el destructor, llamado desde la línea 30. Esa línea es la llave que cierra `main`: ahí corren los destructores.

Cómo leer ese informe

1. **Qué pasó:** `attempting double-free`, en el destructor, llamado desde la línea 30. Esa línea es la llave que cierra `main`: ahí corren los destructores.
2. **Quién liberó antes** (`freed by`): el mismo destructor, desde la misma línea. Es el otro objeto.

Cómo leer ese informe

1. **Qué pasó:** `attempting double-free`, en el destructor, llamado desde la línea 30. Esa línea es la llave que cierra `main`: ahí corren los destructores.
2. **Quién liberó antes** (`freed by`): el mismo destructor, desde la misma línea. Es el otro objeto.
3. **Quién reservó** (`previously allocated by`): solo la línea 24, `IntBuffer a(3)`. La línea 26, la copia, no aparece, porque no reservó nada. Esa ausencia es la pista.

Cómo leer ese informe

1. **Qué pasó:** `attempting double-free`, en el destructor, llamado desde la línea 30. Esa línea es la llave que cierra `main`: ahí corren los destructores.
2. **Quién liberó antes** (`freed by`): el mismo destructor, desde la misma línea. Es el otro objeto.
3. **Quién reservó** (`previously allocated by`): solo la línea 24, `IntBuffer a(3)`. La línea 26, la copia, no aparece, porque no reservó nada. Esa ausencia es la pista.

La región es de 12 bytes

Tres enteros de 4 bytes. Un solo bloque, y dos objetos que creen ser sus owners.

La asignación por defecto: además, una fuga

errors/shallow_assign.cpp

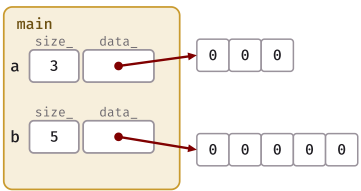
```
1  IntBuffer a(3);
2  IntBuffer b(5);
3  b = a;
```

ASan se detiene en la doble liberación, antes de buscar fugas. Valgrind no se detiene, y muestra las dos:

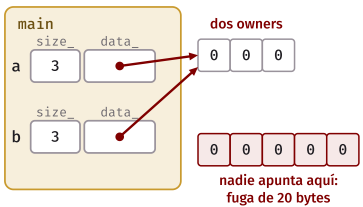
terminal

```
$ valgrind --leak-check=full build/shallow_assign_vg
==1891== Invalid free() / delete / delete[] / realloc()
==1891== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1891==    by 0x1089CB: main (shallow_assign.cpp:18)
```

Antes de b = a



Después de b = a



El error de esta semana

Error común

Escribir el destructor y olvidar la copia. El programa compila sin avisos, con `-Wall -Wextra`, y falla al final, en una llave de cierre, lejos de la línea que causó el problema.

- La causa no está en lo que escribieron, sino en lo que el compilador escribió por ustedes.

El error de esta semana

Error común

Escribir el destructor y olvidar la copia. El programa compila sin avisos, con `-Wall -Wextra`, y falla al final, en una llave de cierre, lejos de la línea que causó el problema.

- La causa no está en lo que escribieron, sino en lo que el compilador escribió por ustedes.
- `-Wdeprecated-copy-dtor` sí avisa, pero no viene en `-Wall -Wextra`. No contamos con ella.

El error de esta semana

Error común

Escribir el destructor y olvidar la copia. El programa compila sin avisos, con `-Wall -Wextra`, y falla al final, en una llave de cierre, lejos de la línea que causó el problema.

- La causa no está en lo que escribieron, sino en lo que el compilador escribió por ustedes.
- `-Wdeprecated-copy-dtor` sí avisa, pero no viene en `-Wall -Wextra`. No contamos con ella.
- Una prueba con ASan que copie el objeto lo encuentra en un segundo.

SECCIÓN 04

La regla de cinco

04

Las cinco funciones

Función	Firma	Cuándo corre
Destructor	`~IntBuffer()`	El objeto muere
Constructor de copia	`IntBuffer(const IntBuffer&)`	Nace copiando a otro
Asignación de copia	`operator=(const IntBuffer&)`	Ya existe y recibe una copia
Constructor de movimiento	`IntBuffer(IntBuffer&&)`	Nace tomando lo de otro
Asignación de movimiento	`operator=(IntBuffer&&)`	Ya existe y toma lo de otro

Regla de cinco ***(rule of five)***

Si una clase necesita escribir una, casi seguro necesita las cinco: todas hablan de quién es owner del recurso.

Las cinco funciones

Función	Firma	Cuándo corre
Destructor	`~IntBuffer()`	El objeto muere
Constructor de copia	`IntBuffer(const IntBuffer&)`	Nace copiando a otro
Asignación de copia	`operator=(const IntBuffer&)`	Ya existe y recibe una copia
Constructor de movimiento	`IntBuffer(IntBuffer&&)`	Nace tomando lo de otro
Asignación de movimiento	`operator=(IntBuffer&&)`	Ya existe y toma lo de otro

Regla de cinco *(rule of five)*

Si una clase necesita escribir una, casi seguro necesita las cinco: todas hablan de quién es owner del recurso.

El signo = engaña. `IntBuffer b = a;` es el **constructor** de copia. Solo `b = a;`, con `b` ya creado, es asignación.

Qué función elige el compilador

terminal

```
$ build/special_members
Traced a("a");
  constructor a
Traced b = a;
  constructor de copia, desde a
Traced c = std::move(a);
  constructor de movimiento, desde a
b = c;
  asignación de copia, a+c = a+m
b = std::move(c);
  asignación de movimiento, a+c = a+m
...
  destructor a+m
  destructor a+c
  destructor a
```

`Traced` escribe el nombre de cada función especial cuando corre.

- `+c`: nació de una copia.
- `+m`: nació de un movimiento.

Qué función elige el compilador

terminal

```
$ build/special_members
Traced a("a");
  constructor a
Traced b = a;
  constructor de copia, desde a
Traced c = std::move(a);
  constructor de movimiento, desde a
b = c;
  asignación de copia, a+c = a+m
b = std::move(c);
  asignación de movimiento, a+c = a+m
...
  destructor a+m
  destructor a+c
  destructor a
```

`Traced` escribe el nombre de cada función especial cuando corre.

- `+c`: nació de una copia.
- `+m`: nació de un movimiento.
- Las dos asignaciones no crean ningún objeto.

Qué función elige el compilador

terminal

```
$ build/special_members
Traced a("a");
  constructor a
Traced b = a;
  constructor de copia, desde a
Traced c = std::move(a);
  constructor de movimiento, desde a
b = c;
  asignación de copia, a+c = a+m
b = std::move(c);
  asignación de movimiento, a+c = a+m
...
  destructor a+m
  destructor a+c
  destructor a
```

`Traced` escribe el nombre de cada función especial cuando corre.

- `+c`: nació de una copia.
- `+m`: nació de un movimiento.
- Las dos asignaciones no crean ningún objeto.
- Los destructores corren en orden inverso.

Lo que escribe el compilador por su cuenta

Si la clase declara...	Copia implícita	Movimiento implícito
Ninguna de las cinco	Sí, campo por campo	Sí, campo por campo
Destructor	Sí, pero obsoleta	**No existe**
Constructor o asignación de copia	Sí la otra, pero obsoleta	**No existe**
Constructor o asignación de movimiento	**Borrada**	No se genera la otra

La trampa es la fila marcada

Con un destructor declarado, no hay movimiento implícito, pero la copia sigue ahí. Entonces `std::move` termina usando la copia.

Un destructor apaga el movimiento

implicit_moves.cpp

```cpp
class WithDestructor {
public:
    ~WithDestructor() {}
    Traced member{"m"};
};
```

terminal

```
$ build/implicit_moves
sin destructor declarado:
  constructor m
  constructor de movimiento, desde m
con destructor declarado:
  constructor m
  constructor de copia, desde m
```

`OnlyMember` es igual, pero sin el destructor.

Por eso, una clase con solo destructor falla igual aunque intenten moverla:

terminal

```
$ build/shallow_move
==1353==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
```

La regla de cero *(rule of zero)*

Regla de cero

Una clase que no administra un recurso directamente no declara ninguna de las cinco. Las que genera el compilador son correctas, porque copiar cada campo copia de verdad.

La regla práctica para este curso:

- ¿Su clase usa un `new`? Es owner de un recurso: escriban las cinco.
- ¿No? No escriban ninguna. `VectorStack` solo tiene un `std::vector`, y funciona.

La regla de cero *(rule of zero)*

Regla de cero

Una clase que no administra un recurso directamente no declara ninguna de las cinco. Las que genera el compilador son correctas, porque copiar cada campo copia de verdad.

La regla práctica para este curso:

- ¿Su clase usa un `new`? Es owner de un recurso: escriban las cinco.
- ¿No? No escriban ninguna. `VectorStack` solo tiene un `std::vector`, y funciona.
- ¿No debería copiarse? `MyClass(const MyClass&) = delete;` y el error sale al compilar.

SECCIÓN 05

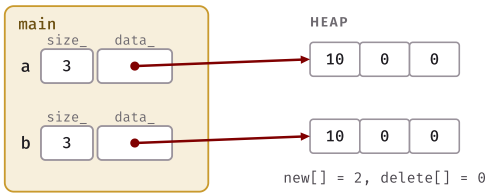
Copiar de verdad

05

El constructor de copia

int_buffer.cpp

```cpp
IntBuffer::IntBuffer(const IntBuffer& other) :
      IntBuffer(other.size_) {
    for (std::size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }
}
```



- **Copia profunda** *(deep copy)*: un bloque nuevo, con los mismos valores.
- **Constructor delegado**: primero corre el otro constructor, que reserva.
- `private` es por clase: puede leer `other.data_`.

La asignación ingenua, y `a = a`

errors/self_assign.cpp

```cpp
IntBuffer& operator=(const IntBuffer& other
    ) {
    delete[] data_;
    data_ = new int[other.size_];
    size_ = other.size_;
    for (std::size_t i = 0; i < size_; i++)
      {
        data_[i] = other.data_[i];
    }
    return *this;
}
```

Autoasignación: `items[i] = items[j]` con `i` igual a `j`. Aquí `other` es el propio objeto.

Línea	`other.data_`	**Datos**
Antes	bloque viejo	`10 0 0`

La asignación ingenua, y `a = a`

errors/self_assign.cpp

```cpp
IntBuffer& operator=(const IntBuffer& other
    ) {
    delete[] data_;
    data_ = new int[other.size_];
    size_ = other.size_;
    for (std::size_t i = 0; i < size_; i++)
      {
        data_[i] = other.data_[i];
    }
    return *this;
}
```

Autoasignación: `items[i] = items[j]` con `i` igual a `j`. Aquí `other` es el propio objeto.

Línea	`other.data_`	**Datos**
Antes `delete[]`	bloque viejo colgante	`10 0 0` perdidos

La asignación ingenua, y `a = a`

errors/self_assign.cpp

```cpp
IntBuffer& operator=(const IntBuffer& other
    ) {
    delete[] data_;
    data_ = new int[other.size_];
    size_ = other.size_;
    for (std::size_t i = 0; i < size_; i++)
      {
        data_[i] = other.data_[i];
    }
    return *this;
}
```

Autoasignación: `items[i] = items[j]` con `i` igual a `j`. Aquí `other` es el propio objeto.

Línea	`other.data_`	**Datos**
Antes	bloque viejo	`10 0 0`
`delete[]`	colgante	perdidos
`new int[]`	bloque nuevo	basura

La asignación ingenua, y `a = a`

errors/self_assign.cpp

```cpp
IntBuffer& operator=(const IntBuffer& other
    ) {
    delete[] data_;
    data_ = new int[other.size_];
    size_ = other.size_;
    for (std::size_t i = 0; i < size_; i++)
      {
        data_[i] = other.data_[i];
    }
    return *this;
}
```

Autoasignación: `items[i] = items[j]` con `i` igual a `j`. Aquí `other` es el propio objeto.

Línea	`other.data_`	**Datos**
Antes	bloque viejo	`10 0 0`
`delete[]`	colgante	perdidos
`new int[]`	bloque nuevo	basura
El ciclo	bloque nuevo	basura

La asignación ingenua, y `a = a`

errors/self_assign.cpp

```cpp
IntBuffer& operator=(const IntBuffer& other
    ) {
    delete[] data_;
    data_ = new int[other.size_];
    size_ = other.size_;
    for (std::size_t i = 0; i < size_; i++)
      {
        data_[i] = other.data_[i];
    }
    return *this;
}
```

Autoasignación: `items[i] = items[j]` con `i` igual a `j`. Aquí `other` es el propio objeto.

Línea	`other.data_`	**Datos**
Antes	bloque viejo	`10 0 0`
`delete[]`	colgante	perdidos
`new int[]`	bloque nuevo	basura
El ciclo	bloque nuevo	basura

El ciclo copia basura sobre sí misma.

ASan no reporta nada

terminal

```
$ build/self_assign
antes: 10
después: -1094795586
```

- El último valor es basura y depende del sistema. El programa termina bien, sin un informe.

ASan no reporta nada

terminal

```
$ build/self_assign
antes: 10
después: -1094795586
```

- El último valor es basura y depende del sistema. El programa termina bien, sin un informe.
- **Un sanitizer encuentra accesos inválidos, no valores equivocados.** `other.data_` ya apunta al bloque nuevo: memoria válida, con basura. Solo lo encuentra una prueba que haga `a = a`.

ASan no reporta nada

terminal

```
$ build/self_assign
antes: 10
después: -1094795586
```

- El último valor es basura y depende del sistema. El programa termina bien, sin un informe.
- **Un sanitizer encuentra accesos inválidos, no valores equivocados.** `other.data_` ya apunta al bloque nuevo: memoria válida, con basura. Solo lo encuentra una prueba que haga `a = a`.
- Un `if (this == &other)` lo parcha. Pero si `new` lanza una excepción después del `delete[]`, queda un puntero colgante.

ASan no reporta nada

terminal

```
$ build/self_assign
antes: 10
después: -1094795586
```

- El último valor es basura y depende del sistema. El programa termina bien, sin un informe.
- **Un sanitizer encuentra accesos inválidos, no valores equivocados.** `other.data_` ya apunta al bloque nuevo: memoria válida, con basura. Solo lo encuentra una prueba que haga `a = a`.
- Un `if (this == &other)` lo parcha. Pero si `new` lanza una excepción después del `delete[]`, queda un puntero colgante.
- **El orden correcto:** primero conseguir lo nuevo, y solo después soltar lo viejo.

Copy and swap

int_buffer.cpp

```cpp
IntBuffer& IntBuffer::operator=(const IntBuffer& other) {
    IntBuffer copy(other);
    swap(copy);
    return *this;
}

void IntBuffer::swap(IntBuffer& other) noexcept {
    std::swap(data_, other.data_);
    std::swap(size_, other.size_);
}
```

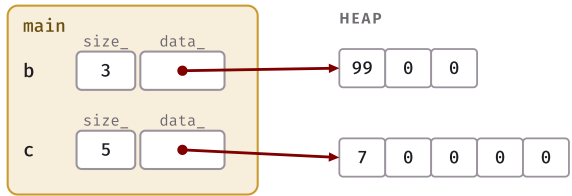
Sin ningún `delete`, y sin copiar a mano.

Copy and swap, paso a paso

```
b = c

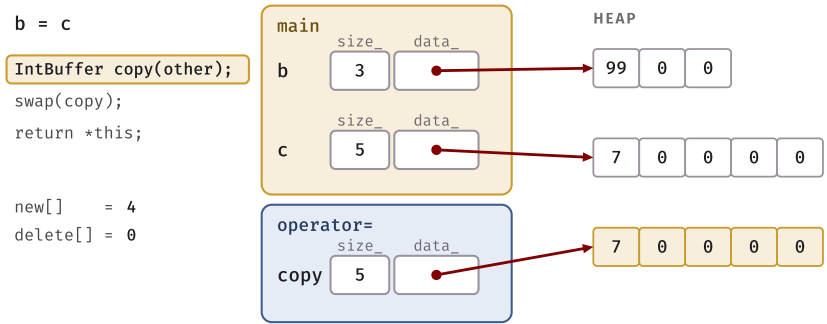
IntBuffer copy(other);
swap(copy);
return *this;


new[]    = 3
delete[] = 0
```



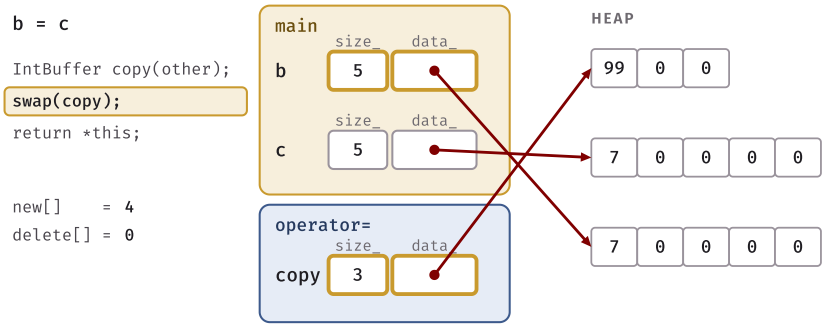
Antes de b = c. b tiene 3 enteros y c tiene 5. copy_demo lleva 3 new[]: a, b y c

Copy and swap, paso a paso



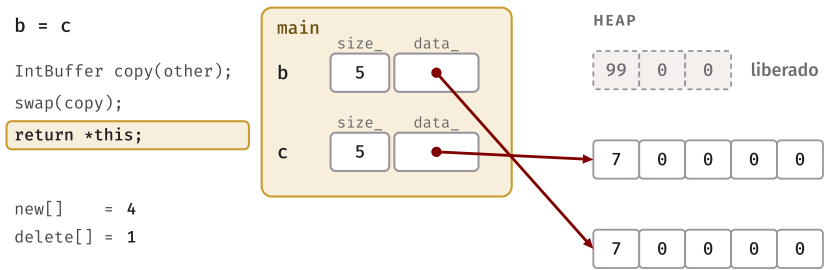
Primero lo nuevo: copy reserva otro bloque. Si aquí faltara memoria, b seguiría intacto

Copy and swap, paso a paso



swap intercambia data_ y size_. Ningún bloque se mueve: solo cambian de owner

Copy and swap, paso a paso



copy muere al salir de la función, y su destructor libera el bloque viejo de b

Copy and swap, comprobado

terminal

```
$ build/copy_demo
copia: a.at(0)=10 b.at(0)=10 | new[]=2 delete[]=0
cambio b: a.at(0)=10 b.at(0)=99 | new[]=2 delete[]=0
b = c: a.at(0)=10 b.at(0)=7 | new[]=4 delete[]=1
b.size()=5
b = b: a.at(0)=10 b.at(0)=7 | new[]=5 delete[]=2
b.size()=5
```

Copy and swap, comprobado

terminal

```
$ build/copy_demo
copia: a.at(0)=10 b.at(0)=10 | new[]=2 delete[]=0
cambio b: a.at(0)=10 b.at(0)=99 | new[]=2 delete[]=0
b = c: a.at(0)=10 b.at(0)=7 | new[]=4 delete[]=1
b.size()=5
b = b: a.at(0)=10 b.at(0)=7 | new[]=5 delete[]=2
b.size()=5
```

- **`b = b`** conserva los datos, sin ningún `if`, a cambio de un `new[]` y un `delete[]`.

Copy and swap, comprobado

terminal

```
$ build/copy_demo
copia: a.at(0)=10 b.at(0)=10 | new[]=2 delete[]=0
cambio b: a.at(0)=10 b.at(0)=99 | new[]=2 delete[]=0
b = c: a.at(0)=10 b.at(0)=7 | new[]=4 delete[]=1
b.size()=5
b = b: a.at(0)=10 b.at(0)=7 | new[]=5 delete[]=2
b.size()=5
```

- **`b = b`** conserva los datos, sin ningún `if`, a cambio de un `new[]` y un `delete[]`.
- **Garantía fuerte** *(strong guarantee)*: si falla, falla en la primera línea, sin haber tocado nada. Termina bien, o es como si no hubiera empezado. `buffer_test.cpp` lo prueba con un tipo cuya copia lanza una excepción.

Copy and swap, comprobado

terminal

```
$ build/copy_demo
copia: a.at(0)=10 b.at(0)=10 | new[]=2 delete[]=0
cambio b: a.at(0)=10 b.at(0)=99 | new[]=2 delete[]=0
b = c: a.at(0)=10 b.at(0)=7 | new[]=4 delete[]=1
b.size()=5
b = b: a.at(0)=10 b.at(0)=7 | new[]=5 delete[]=2
b.size()=5
```

- **`b = b`** conserva los datos, sin ningún `if`, a cambio de un `new[]` y un `delete[]`.
- **Garantía fuerte** *(strong guarantee)*: si falla, falla en la primera línea, sin haber tocado nada. Termina bien, o es como si no hubiera empezado. `buffer_test.cpp` lo prueba con un tipo cuya copia lanza una excepción.
- **Por qué delegar:** si un constructor falla, su destructor no corre. Pero si ya terminó el constructor delegado, sí. Por eso la copia no pierde el bloque.

SECCIÓN 06

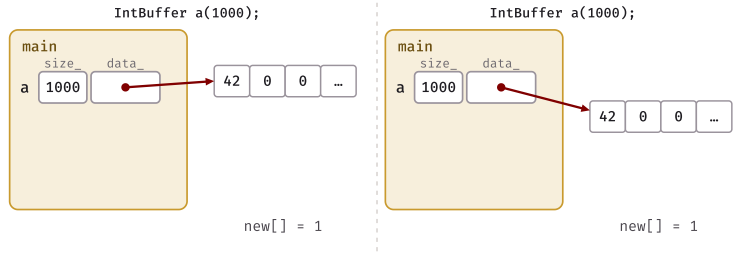
Mover

06

Mover *(move)*

Mover

Pasar los recursos de un objeto a otro, sin copiarlos, dejando al original en un estado válido.

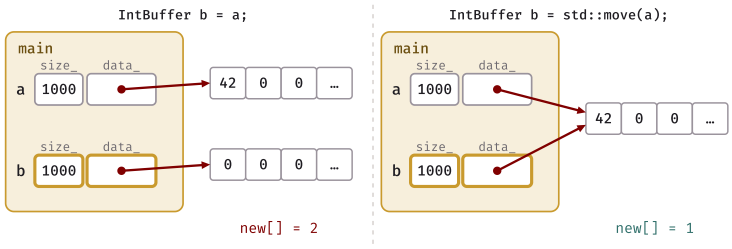


En los dos casos, a tiene un bloque de 1000 enteros, con 42 en el primero

Mover *(move)*

Mover

Pasar los recursos de un objeto a otro, sin copiarlos, dejando al original en un estado válido.

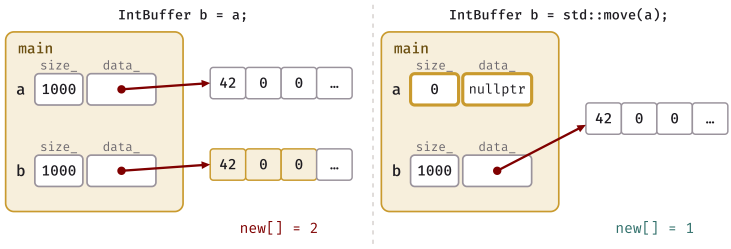


Copiar reserva otro bloque de 1000. Mover copia dos campos: por un momento, dos apuntan al mismo

Mover *(move)*

Mover

Pasar los recursos de un objeto a otro, sin copiarlos, dejando al original en un estado válido.

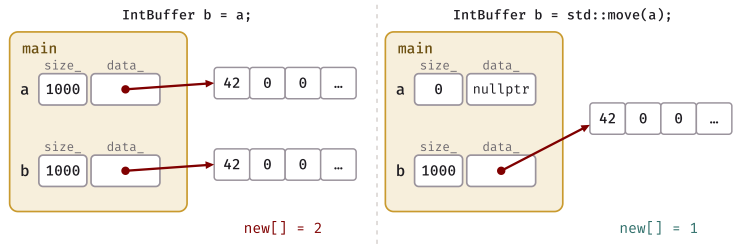


Copiar recorre los 1000 enteros. Mover deja a a en nullptr y 0, y así su destructor no libera nada

Mover *(move)*

Mover

Pasar los recursos de un objeto a otro, sin copiarlos, dejando al original en un estado válido.



Copiar: dos bloques, y a sigue igual. Mover: new[] no subió, sin importar el tamaño del bloque

Cómo sabe el compilador que puede mover

- Mover cuesta copiar dos campos.

Cómo sabe el compilador que puede mover

- Mover cuesta copiar dos campos.
- **lvalue**: algo que sigue existiendo (`a`, `items[i]`). **rvalue**: un valor a punto de desaparecer (`IntBuffer(4)`).

Cómo sabe el compilador que puede mover

- Mover cuesta copiar dos campos.
- **lvalue**: algo que sigue existiendo (`a`, `items[i]`). **rvalue**: un valor a punto de desaparecer (`IntBuffer(4)`).
- `T&&` es una **referencia a rvalue**: solo se une a algo que nadie más va a usar. Así se distinguen las dos sobrecargas.

El constructor y la asignación de movimiento

int_buffer.cpp

```
1  IntBuffer::IntBuffer(IntBuffer&& other)
       noexcept
2      : data_(other.data_), size_(other.
       size_) {
3      other.data_ = nullptr;
4      other.size_ = 0;
5  }
```

Sin las dos últimas líneas, los dos apuntarían al mismo bloque, y volvería la doble liberación.

int_buffer.cpp

```
1  IntBuffer& IntBuffer::operator=(
       IntBuffer&& other) noexcept {
2      IntBuffer taken(std::move(other));
3      swap(taken);
4      return *this;
5  }
```

`taken` se lleva el bloque viejo y lo libera al morir. Con `a = std::move(a)`, el `swap` le devuelve todo a `a`.

`std::move` no mueve nada

terminal

```
$ build/move_demo
a creado: new[]=1 delete[]=0
solo std::move(a): new[]=1 delete[]=0
same.size()=1000 a.size()=1000 a.at(0)=42
constructor de movimiento: new[]=1 delete[]=0
a.size()=0 b.size()=1000 b.at(0)=42
c creado: new[]=2 delete[]=0
asignación de movimiento: new[]=2 delete[]=1
b.size()=0 c.size()=1000 c.at(0)=42
se reutiliza b, con una copia: new[]=3 delete[]=1
b.size()=1000 b.at(0)=42
d creado, const: new[]=4 delete[]=1
std::move de un const: new[]=5 delete[]=1
d.size()=20 e.size()=20
```

- Es un **cast**: `static_cast<T&&>(x)`. No hace nada al ejecutar.

`std::move` no mueve nada

terminal

```
$ build/move_demo
a creado: new[]=1 delete[]=0
solo std::move(a): new[]=1 delete[]=0
same.size()=1000 a.size()=1000 a.at(0)=42
constructor de movimiento: new[]=1 delete[]=0
a.size()=0 b.size()=1000 b.at(0)=42
c creado: new[]=2 delete[]=0
asignación de movimiento: new[]=2 delete[]=1
b.size()=0 c.size()=1000 c.at(0)=42
se reutiliza b, con una copia: new[]=3 delete[]=1
b.size()=1000 b.at(0)=42
d creado, const: new[]=4 delete[]=1
std::move de un const: new[]=5 delete[]=1
d.size()=20 e.size()=20
```

- Es un **cast**: `static_cast<T&&>(x)`. No hace nada al ejecutar.
- Solo `std::move(a)`: `a` sigue intacto.

`std::move` no mueve nada

terminal

```
$ build/move_demo
a creado: new[]=1 delete[]=0
solo std::move(a): new[]=1 delete[]=0
same.size()=1000 a.size()=1000 a.at(0)=42
constructor de movimiento: new[]=1 delete[]=0
a.size()=0 b.size()=1000 b.at(0)=42
c creado: new[]=2 delete[]=0
asignación de movimiento: new[]=2 delete[]=1
b.size()=0 c.size()=1000 c.at(0)=42
se reutiliza b, con una copia: new[]=3 delete[]=1
b.size()=1000 b.at(0)=42
d creado, const: new[]=4 delete[]=1
std::move de un const: new[]=5 delete[]=1
d.size()=20 e.size()=20
```

- Es un **cast**: `static_cast<T&&>(x)`. No hace nada al ejecutar.
- Solo `std::move(a)`: `a` sigue intacto.
- Quien mueve es el constructor. `new[]` sigue en 1.

`std::move` no mueve nada

terminal

```
$ build/move_demo
a creado: new[]=1 delete[]=0
solo std::move(a): new[]=1 delete[]=0
same.size()=1000 a.size()=1000 a.at(0)=42
constructor de movimiento: new[]=1 delete[]=0
a.size()=0 b.size()=1000 b.at(0)=42
c creado: new[]=2 delete[]=0
asignación de movimiento: new[]=2 delete[]=1
b.size()=0 c.size()=1000 c.at(0)=42
se reutiliza b, con una copia: new[]=3 delete[]=1
b.size()=1000 b.at(0)=42
d creado, const: new[]=4 delete[]=1
std::move de un const: new[]=5 delete[]=1
d.size()=20 e.size()=20
```

- Es un **cast**: `static_cast<T&&>(x)`. No hace nada al ejecutar.
- Solo `std::move(a)`: `a` sigue intacto.
- Quien mueve es el constructor. `new[]` sigue en 1.
- Un objeto movido se puede volver a usar, si se le asigna algo.

`std::move` no mueve nada

terminal

```
$ build/move_demo
a creado: new[]=1 delete[]=0
solo std::move(a): new[]=1 delete[]=0
same.size()=1000 a.size()=1000 a.at(0)=42
constructor de movimiento: new[]=1 delete[]=0
a.size()=0 b.size()=1000 b.at(0)=42
c creado: new[]=2 delete[]=0
asignación de movimiento: new[]=2 delete[]=1
b.size()=0 c.size()=1000 c.at(0)=42
se reutiliza b, con una copia: new[]=3 delete[]=1
b.size()=1000 b.at(0)=42
d creado, const: new[]=4 delete[]=1
std::move de un const: new[]=5 delete[]=1
d.size()=20 e.size()=20
```

- Es un **cast**: `static_cast<T&&>(x)`. No hace nada al ejecutar.
- Solo `std::move(a)`: `a` sigue intacto.
- Quien mueve es el constructor. `new[]` sigue en 1.
- Un objeto movido se puede volver a usar, si se le asigna algo.
- Con un `const`, copia, y sin avisar: `new[]` sube a 5.

Qué queda en el objeto movido

Sigue existiendo, y su destructor correrá. Debe quedar en un **estado válido**.

Qué queda en el objeto movido

Sigue existiendo, y su destructor correrá. Debe quedar en un **estado válido**.

- **Sus clases:** ustedes lo deciden. `IntBuffer` promete quedar vacío, y cumple sus invariantes.

Qué queda en el objeto movido

Sigue existiendo, y su destructor correrá. Debe quedar en un **estado válido**.

- **Sus clases:** ustedes lo deciden. `IntBuffer` promete quedar vacío, y cumple sus invariantes.
- **La biblioteca: válido pero no especificado**. Funcionan `size()`, `clear()` o asignarle. No se puede suponer su contenido.

terminal

```
$ build/library_moved_from
text.size()=0 other.size()=50
```

Aquí `text` quedó vacía, pero eso no es una promesa: lo hace esta versión de la biblioteca.

Qué queda en el objeto movido

Sigue existiendo, y su destructor correrá. Debe quedar en un **estado válido**.

- **Sus clases:** ustedes lo deciden. `IntBuffer` promete quedar vacío, y cumple sus invariantes.
- **La biblioteca: válido pero no especificado**. Funcionan `size()`, `clear()` o asignarle. No se puede suponer su contenido.

terminal

```
$ build/library_moved_from
text.size()=0 other.size()=50
```

Aquí `text` quedó vacía, pero eso no es una promesa: lo hace esta versión de la biblioteca.

terminal

```
$ build/use_after_move
use_after_move: int_buffer.cpp:42: int& IntBuffer::at(std::size_t): Assertion 'index < size_' failed.
```

Al devolver una local, no escriban `std::move`

special_members.cpp

```cpp
Traced make_traced() {
    Traced local("local");
    return local;
}
```

Un solo constructor, y ningún movimiento: **elisión de copia** *(copy elision)*. Para un objeto con nombre se permite, no se exige. Si no se aplica, el `return` mueve solo.

Con `return std::move(local);` ya no se puede aplicar, y `-Wall` avisa:

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/pessimizing_move.cpp -o build/pessimizing_move
errors/pessimizing_move.cpp:8:21: warning: moving a local object in a return statement prevents copy elision [-Wpessimizing-move]
$ build/pessimizing_move
  constructor local
  constructor de movimiento, desde local
```

`noexcept` cambia lo que hace `std::vector`

Al crecer, `std::vector` pasa sus elementos a otro bloque. Si un movimiento fallara a la mitad, no habría vuelta atrás. Por eso usa `std::move_if_noexcept`:

- mueve si el movimiento es `noexcept`, y por eso lo escribimos siempre que se pueda cumplir;
- si no, **copia**, porque el original sigue intacto. Solo mueve de todos modos si el tipo no se puede copiar.

vector_growth.cpp

```cpp
SafeMove(SafeMove&& other) noexcept : counts_(other.counts_) {
RiskyMove(RiskyMove&& other) : counts_(other.counts_) {
```

terminal

```
$ build/vector_growth
SafeMove después de crecer: copias=0 movimientos=1000
RiskyMove después de crecer: copias=1000 movimientos=0
```

SECCIÓN 07

Templates

07



En CS201: `void *` y punteros a función

max_void.c

```c
const void *max_of(const void *items, size_t count
     , size_t size, compare_fn compare) {
    const char *bytes = items;
    const void *best = items;
    for (size_t i = 1; i < count; i++) {
        const void *item = bytes + i * size;
        if (compare(item, best) > 0) {
            best = item;
        }
    }
    return best;
}
```

El compilador borró los tipos. Con `compare_int` sobre `double`:

errors/void_wrong_compare.c

```c
double prices[] = {1.5, 0.25,
      7.75};
```

terminal

```
$ build/void_wrong_compare
1.50
```

Sin avisos. Es comportamiento indefinido.

Un template: código con un hueco para un tipo

max_of.cpp

```cpp
template <typename T>
const T& max_of(const std::vector<T>& items) {
    assert(!items.empty());
    const T* best = &items[0];
    for (std::size_t i = 1; i < items.size();
     i++) {
        if (items[i] > *best) {
            best = &items[i];
        }
    }
    return *best;
}
```

max_of.cpp

```cpp
std::cout << max_of(numbers) << "\n";
std::cout << max_of(names) << "\n";
```

terminal

```
$ build/max_of
9
Zoila
```

Se genera una versión por tipo

El compilador **deduce** `T` del argumento e **instancia** el template: genera una función por cada tipo usado. Se puede ver en el archivo objeto, con `nm`:

terminal

```
$ g++ -std=c++20 -O0 -c max_of.cpp -o build/max_of.o
$ nm -C build/max_of.o | /usr/bin/grep 'max_of<int>'
0000000000000000 W int const& max_of<int>(std::vector<int, std::allocator<int> > const&)
$ nm -C build/max_of.o | /usr/bin/grep -c 'max_of<'
2
```

- Son dos funciones distintas, compiladas por separado.
- `W` es un símbolo débil *(weak)*: si varios archivos generan la misma versión, el enlazador se queda con una.

Templates contra `void *`

	`void *` (CS201)	Templates
Quién comprueba los tipos	Nadie	El compilador
Un error de tipo	Se ve al ejecutar, si se ve	No compila
Copias del código	Una, para todos los tipos	Una por cada tipo usado
Llamar a la comparación	Por puntero	Directa, se puede optimizar
Dónde va el código	Se compila una vez, en un `.c`	En el encabezado

Los requisitos de `T` no están escritos: son lo que el cuerpo use. Busquen `required from here` y `with T =`.

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/no_operator.cpp -o build/no_operator
errors/no_operator.cpp: In instantiation of 'const T& max_of(const std::vector<T>&) [with T = Point]':
errors/no_operator.cpp:25:11:   required from here
```

Por qué un template va en el encabezado

Cada `.cpp` se compila solo. Para generar una versión, hacen falta **el cuerpo y el tipo al mismo tiempo**:

Por qué un template va en el encabezado

Cada `.cpp` se compila solo. Para generar una versión, hacen falta **el cuerpo y el tipo al mismo tiempo**:

- `main.cpp` pide `Box<int>`, pero solo ve las declaraciones de `box.h`.

Por qué un template va en el encabezado

Cada `.cpp` se compila solo. Para generar una versión, hacen falta **el cuerpo y el tipo al mismo tiempo**:

- `main.cpp` pide `Box<int>`, pero solo ve las declaraciones de `box.h`.
- `box.cpp` ve los cuerpos, pero nadie le pidió `Box<int>`: no genera nada.

Por qué un template va en el encabezado

Cada `.cpp` se compila solo. Para generar una versión, hacen falta **el cuerpo y el tipo al mismo tiempo**:

- `main.cpp` pide `Box<int>`, pero solo ve las declaraciones de `box.h`.
- `box.cpp` ve los cuerpos, pero nadie le pidió `Box<int>`: no genera nada.
- El enlazador no encuentra la función.

terminal

```
$ g++ -std=c++20 -g errors/split_template/main.cpp errors/split_template/box.cpp -o build/split_template
/work/code/week-02/errors/split_template/main.cpp:7:(.text+0x94): undefined reference to `Box<int>::Box(int)'
/usr/bin/ld: /work/code/week-02/errors/split_template/main.cpp:8:(.text+0x180): undefined reference to `Box<int>::value() const'
```

Por qué un template va en el encabezado

Cada `.cpp` se compila solo. Para generar una versión, hacen falta **el cuerpo y el tipo al mismo tiempo**:

- `main.cpp` pide `Box<int>`, pero solo ve las declaraciones de `box.h`.
- `box.cpp` ve los cuerpos, pero nadie le pidió `Box<int>`: no genera nada.
- El enlazador no encuentra la función.

terminal

```
$ g++ -std=c++20 -g errors/split_template/main.cpp errors/split_template/box.cpp -o build/split_template
/work/code/week-02/errors/split_template/main.cpp:7:(.text+0x94): undefined reference to `Box<int>::Box(int)'
/usr/bin/ld: /work/code/week-02/errors/split_template/main.cpp:8:(.text+0x180): undefined reference to `Box<int>::value() const'
```

Por eso

`IntBuffer` tiene `int_buffer.h` e `int_buffer.cpp`. `Buffer<T>` solo tiene `buffer.h`.

Una clase template

buffer.h

```
1  template <typename T>
2  class Buffer {
3  public:
4      explicit Buffer(std::size_t size) : data_(size == 0 ? nullptr : new T[size]()), size_(size) {}
```

- Es `IntBuffer` con un hueco. Las cinco funciones cambian poco.
- `Buffer<int>` y `Buffer<std::string>` son **dos tipos sin relación**.
- Dentro de la clase, `Buffer` a secas significa `Buffer<T>`.
- Los métodos solo se instancian si alguien los usa.

Lo que le exigimos a `T`

Por ahora, `T` debe tener un constructor sin argumentos

`new T[n]` construye todo al reservar. También en la pila del laboratorio: `new T[capacity]`.

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/no_default.cpp -o build/no_default
errors/no_default.cpp:15:31:   required from here
errors/../buffer.h:16:69: error: no matching function for call to 'Student::Student()'
```

terminal

```
$ build/array_constructs
después de new[]: construidos=1000 destruidos=0
```

`std::vector` no tiene esta limitación: separa reservar de construir.

SECCIÓN 08

Tipo abstracto e implementación

08

Qué promete, y cómo lo cumple

Tipo abstracto de datos ***(abstract data type)***

Un conjunto de operaciones, junto con lo que cada una promete, sin decir cómo se logra. La **implementación** es una forma concreta de cumplirlo.

Contrato ***(contract)***

La promesa escrita con precisión: **precondición** (qué debe cumplirse antes) y **poscondición** (qué será cierto después).

Idea clave: un programa escrito contra el qué no se entera si cambia el cómo.

La pila *(stack)*

El último en entrar es el primero en salir.

Operación	Precondición	Qué hace
`push(x)`	Hay espacio, si la capacidad es fija	Agrega `x` arriba
`pop()`	No está vacía	Saca el elemento de arriba
`top()`	No está vacía	Devuelve el de arriba, sin sacarlo
`empty()`	Ninguna	Dice si no tiene elementos

La pila *(stack)*

El último en entrar es el primero en salir.

Operación	Precondición	Qué hace
`push(x)`	Hay espacio, si la capacidad es fija	Agrega `x` arriba
`pop()`	No está vacía	Saca el elemento de arriba
`top()`	No está vacía	Devuelve el de arriba, sin sacarlo
`empty()`	Ninguna	Dice si no tiene elementos

¿Tipo abstracto, o implementación?

- `push`
- “El último en entrar es el primero en salir”
- El arreglo
- El tope

La pila *(stack)*

El último en entrar es el primero en salir.

Operación	Precondición	Qué hace
`push(x)`	Hay espacio, si la capacidad es fija	Agrega `x` arriba
`pop()`	No está vacía	Saca el elemento de arriba
`top()`	No está vacía	Devuelve el de arriba, sin sacarlo
`empty()`	Ninguna	Dice si no tiene elementos

¿Tipo abstracto, o implementación?

- `push`: **tipo abstracto**
- "El último en entrar es el primero en salir": **tipo abstracto**
- El arreglo: **implementación**
- El tope: **implementación**

Un uso: la notación postfija

postfix.cpp

```
1  int eval_postfix(const std::vector<std::string
       >& tokens) {
2      VectorStack<int> values;
3      for (const std::string& token : tokens) {
4          if (token == "+" || token == "-" ||
       token == "*") {
5              assert(values.size() >= 2);
6              int right = values.top();
7              values.pop();
8              int left = values.top();
9              values.pop();
```

`3 4 + 2 *` significa `(3 + 4) * 2`.

Token	Pila (tope a la derecha)
3	3
4	3 4

Un uso: la notación postfija

postfix.cpp

```
1  int eval_postfix(const std::vector<std::string
       >& tokens) {
2      VectorStack<int> values;
3      for (const std::string& token : tokens) {
4          if (token == "+" || token == "-" ||
       token == "*") {
5              assert(values.size() >= 2);
6              int right = values.top();
7              values.pop();
8              int left = values.top();
9              values.pop();
```

`3 4 + 2 *` significa `(3 + 4) * 2`.

Token	Pila (tope a la derecha)
3	3
4	3 4
+	7

Un uso: la notación postfija

postfix.cpp

```cpp
int eval_postfix(const std::vector<std::string
    >& tokens) {
    VectorStack<int> values;
    for (const std::string& token : tokens) {
        if (token == "+" || token == "-" ||
    token == "*") {
            assert(values.size() >= 2);
            int right = values.top();
            values.pop();
            int left = values.top();
            values.pop();
```

`3 4 + 2 *` significa `(3 + 4) * 2`.

Token	Pila (tope a la derecha)
3	3
4	3 4
+	7
2	7 2

Un uso: la notación postfija

postfix.cpp

```cpp
int eval_postfix(const std::vector<std::string
    >& tokens) {
    VectorStack<int> values;
    for (const std::string& token : tokens) {
        if (token == "+" || token == "-" ||
    token == "*") {
            assert(values.size() >= 2);
            int right = values.top();
            values.pop();
            int left = values.top();
            values.pop();
```

`3 4 + 2 *` significa `(3 + 4) * 2`.

Token	Pila (tope a la derecha)
3	3
4	3 4
+	7
2	7 2
*	14

Un uso: la notación postfija

postfix.cpp

```
1  int eval_postfix(const std::vector<std::string
       >& tokens) {
2      VectorStack<int> values;
3      for (const std::string& token : tokens) {
4          if (token == "+" || token == "-" ||
       token == "*") {
5              assert(values.size() >= 2);
6              int right = values.top();
7              values.pop();
8              int left = values.top();
9              values.pop();
```

`3 4 + 2 *` significa `(3 + 4) * 2`.

Token	Pila (tope a la derecha)
3	3
4	3 4
+	7
2	7 2
*	14

La función nunca menciona un arreglo.

Una implementación: un arreglo y un tope

pseudocódigo

```
algoritmo push(datos, tope, valor)
    datos[tope] ← valor
    devolver tope + 1
fin algoritmo

algoritmo pop(tope)
    devolver tope - 1
fin algoritmo

algoritmo top(datos, tope)
    devolver datos[tope - 1]
fin algoritmo
```

Una implementación: un arreglo y un tope

pseudocódigo

```
algoritmo push(datos, tope, valor)
    datos[tope] ← valor
    devolver tope + 1
fin algoritmo

algoritmo pop(tope)
    devolver tope - 1
fin algoritmo

algoritmo top(datos, tope)
    devolver datos[tope - 1]
fin algoritmo
```

- `pop` no borra. En `Stack<T>`, el elemento sacado sigue vivo hasta que otro `push` lo reemplace.

Una implementación: un arreglo y un tope

pseudocódigo

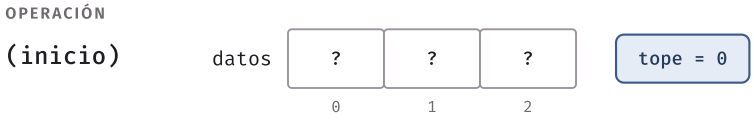
```
algoritmo push(datos, tope, valor)
    datos[tope] ← valor
    devolver tope + 1
fin algoritmo

algoritmo pop(tope)
    devolver tope - 1
fin algoritmo

algoritmo top(datos, tope)
    devolver datos[tope - 1]
fin algoritmo
```

- `pop` no borra. En `Stack<T>`, el elemento sacado sigue vivo hasta que otro `push` lo reemplace.
- No crece: `push` requiere espacio. El crecimiento, que sí tiene el `ArrayStack` de la lectura, llega en la semana 4.
- Otra implementación: `VectorStack`, sobre `std::vector`.

La pila, paso a paso

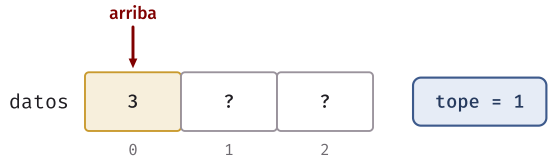


Capacidad 3 y tope = 0: el tope dice cuántas casillas se están usando

La pila, paso a paso

OPERACIÓN

`push(3)`

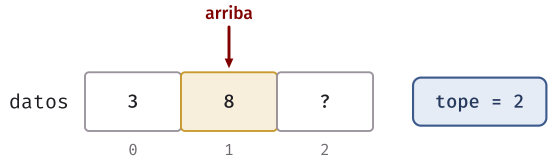


push escribe en datos[tope], y después sube el tope

La pila, paso a paso

OPERACIÓN

`push(8)`



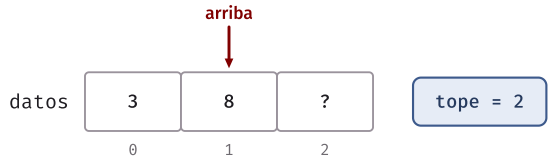
El elemento de arriba siempre está en datos[tope - 1]

La pila, paso a paso

OPERACIÓN

`top()`

devuelve 8

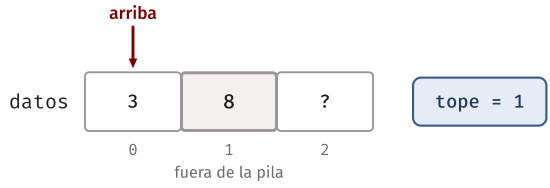


top lee datos[tope - 1], y no cambia nada

La pila, paso a paso

OPERACIÓN

pop()



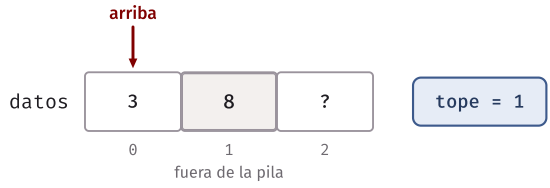
pop solo baja el tope. El 8 sigue en el arreglo, pero ya no cuenta como parte de la pila

La pila, paso a paso

OPERACIÓN

`top()`

devuelve 3

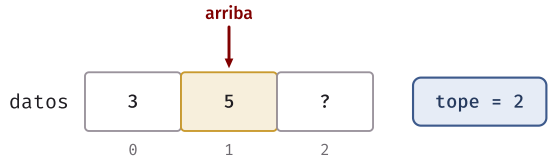


Ahora arriba está el 3

La pila, paso a paso

OPERACIÓN

`push(5)`



push(5) escribe sobre el 8. Recién ahí desaparece

SECCIÓN 09

Invariantes

09

Invariantes

Invariante *(invariant)*

Condición sobre los campos de una estructura que se cumple siempre **entre dos operaciones**: al terminar el constructor, y al terminar cada método.

lab/stack.h

```
1  //   1. size_ <= capacity_.
2  //   2. data_ == nullptr si y solo si capacity_ == 0.
3  //   3. Los elementos están en data_[0], ..., data_[size_ - 1]. El tope es data_[size_ - 1].
```

Invariantes

Invariante *(invariant)*

Condición sobre los campos de una estructura que se cumple siempre **entre dos operaciones**: al terminar el constructor, y al terminar cada método.

lab/stack.h

```
1  //   1. size_ <= capacity_.
2  //   2. data_ == nullptr si y solo si capacity_ == 0.
3  //   3. Los elementos están en data_[0], ..., data_[size_ - 1]. El tope es data_[size_ - 1].
```

- Dentro de un método se puede romper por un momento. Por eso los campos son privados.

Invariantes

Invariante *(invariant)*

Condición sobre los campos de una estructura que se cumple siempre **entre dos operaciones**: al terminar el constructor, y al terminar cada método.

lab/stack.h

```
1  //   1. size_ <= capacity_.
2  //   2. data_ == nullptr si y solo si capacity_ == 0.
3  //   3. Los elementos están en data_[0], ..., data_[size_ - 1]. El tope es data_[size_ - 1].
```

- Dentro de un método se puede romper por un momento. Por eso los campos son privados.
- Cada método lo supone al entrar, y lo deja cierto al salir.

Invariantes

Invariante *(invariant)*

Condición sobre los campos de una estructura que se cumple siempre **entre dos operaciones**: al terminar el constructor, y al terminar cada método.

lab/stack.h

```
1  //   1. size_ <= capacity_.
2  //   2. data_ == nullptr si y solo si capacity_ == 0.
3  //   3. Los elementos están en data_[0], ..., data_[size_ - 1]. El tope es data_[size_ - 1].
```

- Dentro de un método se puede romper por un momento. Por eso los campos son privados.
- Cada método lo supone al entrar, y lo deja cierto al salir.
- También vale para un objeto movido, que tiene capacidad 0: `push` sobre él viola `!full()`.

`check_invariants` encuentra el bug justo donde ocurre

sorted_ints.h

```
1  void check_invariants() const {
2      for (std::size_t i = 1; i < items_.
         size(); i++) {
3          assert(items_[i - 1] <= items_[i
         ]);
4      }
5  }
```

errors/broken_sorted.cpp

```
1  while (i > 1 && items_[i - 1] > value) {
```

Se detiene una casilla antes. Insertar 5 y luego 3 deja 5 3.

terminal

```
$ build/broken_sorted
broken_sorted: errors/broken_sorted.cpp:25: void SortedInts::check_invariants() const: Assertion ‘items_[i - 1] <= items_[i]’ failed.
```

`check_invariants` encuentra el bug justo donde ocurre

sorted_ints.h

```
1  void check_invariants() const {
2      for (std::size_t i = 1; i < items_.
         size(); i++) {
3          assert(items_[i - 1] <= items_[i
         ]);
4      }
5  }
```

errors/broken_sorted.cpp

```
1  while (i > 1 && items_[i - 1] > value) {
```

Se detiene una casilla antes. Insertar 5 y luego 3 deja 5 3.

terminal

```
$ build/broken_sorted
broken_sorted: errors/broken_sorted.cpp:25: void SortedInts::check_invariants() const: Assertion 'items_[i - 1] <= items_[i]' failed.
```

- **No reemplaza a las pruebas:** con un valor equivocado, el invariante se cumple igual.

SECCIÓN 10

Laboratorio

10

Laboratorio: una pila genérica sin fugas

Sesión 1

1. `IntStack` con `new[]` y destructor.
2. `make copy-demo`: lean el informe de ASan y dibújenlo.
3. Copia y asignación. Prueben `a = a`.

Sesión 2

4. Movimiento. La prueba cuenta los `new[]`.
5. `Stack<T>` con `check_invariants`, con `int` y `std::string`.
6. Verificador de paréntesis, corchetes y llaves.

`make test-lab` falla al principio, y tiene una clase de prueba por paso.

Recuerden

Ahora hay dos números, `size_` y `capacity_`. Decidan qué copiar de cada uno. Y `T` debe tener un constructor sin argumentos.

Para la próxima clase

- Leer los apuntes de la semana 3: cómo medir el costo de un programa.
- Laboratorio: terminar la pila, con `make test-lab` sin errores.
- Ejercicios de los apuntes de esta semana.

Pregunta

¿En qué estado debe quedar un objeto después de que se mueve desde él?
¿Cuánto cuesta `push` en su pila, si tiene 10 elementos, y si tiene un millón?