

Clases en C++

¹Facultad de Ingeniería
Universidad de Buenos Aires



7542

Clases en C++

TDAs - Clases en C

```
1 struct Vector {
2     int *_data; /*private*/
3     int _size; /*private*/
4 };

5 void vector_create(struct Vector *v, int size) {
6     v->_data = (int*)malloc(size*sizeof(int));
7     v->_size = size;
8 }
9 int vector_get(struct Vector *v, int pos) {
10     return v->_data[pos];
11 }
12 void vector_destroy(struct Vector *v) {
13     free(v->_data);
14 }
```

- La forma de hacer TDAs en C aplica a C++
- El estándar de C++ garantiza que si un **struct** no tiene ningún feature de C++ (o sea, se parece a un struct de C) se lo llama "plain struct" y puede ser usado por libs de C desde C++ (el estándar garantiza la compatibilidad)
- Por convención los nombres de las funciones del TDA deben tener como prefijo de su nombre el nombre del TDA: esto es por que en C todas las funciones terminan en el mismo espacio global y para no tener conflictos cada función debe tener un nombre único. El conflicto de nombres es un problema común en proyectos grandes en C. En C++ tenemos mejores formas de resolverlos...
- Así también es convención pasar como primer argumento un puntero al **struct**. Veremos que en C++ hay una forma mas conveniente de hacer esto...
- Y otra convención mas: los atributos que no deberían ser ni leídos ni modificados por el usuario son marcados como privados. Dependiendo de la convención hay gente que le pone un guion bajo al principio de la variable, otros al final y otros ponen solamente un comentario. C++ nos dara herramientas para forzar esto en tiempo de compilación.

De qué va esto?

- 1 **structs y clases en C++**
 - Introducción
 - Constructor y destructor
 - RAI: Resource Acquisition Is Initialization
- 2 **Moviendo objetos**
 - Pasaje por referencia
 - Pasaje por copia
 - Move semantics
- 3 **More C++**
 - Constantes
 - Initialization
 - Otros



7542

Clases en C++

TDAs - Clases en C

```
1 struct Vector {
2     int *_data; /*private*/
3     int _size; /*private*/
4 };

5 void vector_create(struct Vector *v, int size) {
6     v->_data = (int*)malloc(size*sizeof(int));
7     v->_size = size;
8 }
9 int vector_get(struct Vector *v, int pos) {
10     return v->_data[pos];
11 }
12 void vector_destroy(struct Vector *v) {
13     free(v->_data);
14 }

15 void f() {
16     struct Vector v;
17     vector_create(&v, 5);
18     vector_get(&v, 0);
19     vector_destroy(&v);
20 }
```

Keyword struct implícita y adios a los typedefs de los structs

```
1 struct Vector {
2     int *_data; /*private*/
3     int _size; /*private*/
4 };

5 void vector_create(Vector *v, int size) {
6     v->_data = (int*)malloc(size*sizeof(int));
7     v->_size = size;
8 }
9 int vector_get(Vector *v, int pos) {
10     return v->_data[pos];
11 }
12 void vector_destroy(Vector *v) {
13     free(v->_data);
14 }

15 void f() {
16     Vector v;
17     vector_create(&v, 5);
18     vector_get(&v, 0);
19     vector_destroy(&v);
20 }
```

- En C++ no es necesario usar la keyword `struct` en todos lados.

- Se integran las funciones y los datos del TDA en una sola unidad.
- Los datos del TDA se lo llaman atributos
- Las funciones del TDA pasan a ser métodos del TDA y reciben como parámetro implícito un puntero a la instancia
- El puntero es un puntero constante a la instancia (`Vector *const`) y se lo nombra con la keyword `this`. En otras palabras `this` es un puntero constante que apunta al objeto sobre el cual se esta invocando el método.

- Se accede a los atributos y/o métodos como en C
- En C, la instancia sobre la que se quiere invocar un método es pasada como parámetro de forma explícita mientras que en C++ es implícita e invisible.

Bundle: atributos + métodos

Keyword `this` como un parámetro implícito: un puntero a la instancia

```
1 struct Vector {
2     int *_data; /*private*/
3     int _size; /*private*/
4
5     void vector_create(int size) {
6         this->_data = (int*)malloc(size*sizeof(int));
7         this->_size = size;
8     }
9
10    int vector_get(int pos) {
11        return this->_data[pos];
12    }
13
14    void vector_destroy() {
15        free(this->_data);
16    }
17 };
```

7542

Invocación de métodos

El objeto instancia es pasado como parámetro de forma implícita.

<pre>14 // En C 15 void f() { 16 struct Vector v; 17 vector_create(&v, 5); 18 vector_get(&v, 0); 19 20 v._data; 21 22 vector_destroy(&v); 23 }</pre>	<pre>14 // En C++ 15 void f() { 16 Vector v; 17 v.vector_create(5); 18 v.vector_get(0); 19 20 v._data; 21 22 v.vector_destroy(); 23 }</pre>
--	---

7542

Reducción de colisiones

Las clases crean sus propios namespaces

```
1 struct Vector {
2     int *_data; /*private*/
3     int _size; /*private*/
4
5     void create(int size) { // Vector::create
6         this->_data = (int*)malloc(size*sizeof(int));
7         this->_size = size;
8     }
9
10    int get(int pos) { // Vector::get
11        return this->_data[pos];
12    }
13
14    void destroy() { // Vector::destroy
15        free(this->_data);
16    }
17 };
```

7542

- Los métodos de un TDA no entran en conflicto con otros aunque se llamen iguales. El método `get` de `Vector` no entra en conflicto con el método `get` de `Matrix`, por ejemplo
- En rigor un método de un TDA se lo llama `NombreTDA::NombreMetodo`, por eso `Vector::get` es distinto de `Matrix::get`.
- Veremos con mas detalle el concepto de namespace en las próximas clases.

- Por default, un `struct` tiene sus atributos y métodos públicos. Esto significa que pueden accederse desde cualquier lado.
- Se puede cambiar el default forzando distintos permisos.
- `private` hace que solo los métodos internos puedan acceder a los métodos y atributos privados.
- Mas sobre los permisos `public/protected/private` y su relación con la herencia en las próximas clases.

Permisos de acceso

Controlan quien puede acceder a los atributos y métodos de un objeto.

```
1 struct Vector {
2     private:
3     int *data;
4     int size;
5
6     public:
7     void create(int size) {
8         this->data = (int*)malloc(size*sizeof(int));
9         this->size = size;
10    }
11
12    int get(int pos) {
13        return this->data[pos];
14    }
15
16    void destroy() {
17        free(this->data);
18    }
19 };
```

structs y clases en C++
Moviendo objetos
More C++

Introducción
Constructor y destructor
RAII: Resource Acquisition Is Initialization

Permisos de acceso

<pre>14 // En C 15 void f() { 16 struct Vector v; 17 vector_create(&v, 5); 18 vector_get(&v, 0); 19 20 v._data; 21 22 vector_destroy(&v); 23 }</pre>	<pre>14 // En C++ 15 void f() { 16 Vector v; 17 v.create(5); 18 v.get(0); 19 20 v.data; 21 22 v.destroy(); 23 }</pre>
--	---

7542

Clases en C++

structs y clases en C++
Moviendo objetos
More C++

Introducción
Constructor y destructor
RAII: Resource Acquisition Is Initialization

Clases en C++

Son iguales que los structs solo que el acceso es privado por default.

```
1 struct Vector {
2     int *data; // public by default
3     int size; // public by default
4 };
5
6 class Vector {
7     int *data; // private by default
8     int size; // private by default
9 };
```

- Absolutamente todo lo visto con structs en C++ aplica a las clases de C++. La única diferencia es que las clases tienen sus atributos y métodos privados por default.

7542

Clases en C++

Clases en C++

Son iguales que los structs solo que el acceso es privado por default.

```
1 class Vector {
2     private:
3         int *data;
4         int size;
5
6     public:
7         void create(int size) {
8             this->data = (int*)malloc(size*sizeof(int));
9             this->size = size;
10        }
11
12        int get(int pos) {
13            return this->data[pos];
14        }
15
16        void destroy() {
17            free(this->data);
18        }
19    };
```

- El constructor es un código que se ejecuta al momento de crear un nuevo objeto. C++ siempre llama a algun constructor al crear un nuevo objeto.
- Todos los objetos son creados por un constructor. Si un TDA no tiene un constructor, C++ crea un constructor por default
- Un TDA puede tener múltiples constructores (que los veremos a continuación). Sin embargo solo puede haber un único destructor.
- Un destructor es un código que se ejecuta al momento de destruirse un objeto (cuando este se va de scope o es eliminado del heap con `delete`).
- Todos los objetos tienen un destructor. Si un TDA no tiene un destructor, C++ crea un destructor por default

- Con los constructores (si estan bien escritos) no se puede usar un objeto sin inicializar
- Con los destructores (si estan bien escritos, se usa RAII y usamos el stack) no vamos a tener leaks.
- Los destructores se llaman automáticamente cuando el objeto se va de scope.

structs y clases en C++
Moviendo objetos
More C++

Introducción
Constructor y destructor
RAII: Resource Acquisition Is Initialization

Constructor y Destructor

Código a ejecutar de forma automática cuando se construye/destruye un objeto.

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(int size) { // create
6         this->data = (int*)malloc(size*sizeof(int));
7         this->size = size;
8     }
9
10    int get(int pos) {
11        return this->data[pos];
12    }
13
14    ~Vector() { // destroy
15        free(this->data);
16    }
17 };
```

7542

structs y clases en C++
Moviendo objetos
More C++

Clases en C++

Introducción
Constructor y destructor
RAII: Resource Acquisition Is Initialization

Reduciendo las probabilidades de errores

Constructores y destructores

```
29 // En C
30 void g() {
31     struct Vector v;
32
33     v.data;
34
35     vector_create(&v, 5);
36     //...
37
38 }
```

```
29 // En C ++
30 void g() {
31     Vector v(5);
32
33     v.data;
34
35     //...
36
37
38 }
```

7542

Clases en C++

Manejo de errores en C (madness)

Ejemplo motivador para el buen uso de objetos en C++ con constructores y destructores bien diseñados.

```
1 int process() {
2     char *buf = (char*) malloc(sizeof(char)*20);
3
4     FILE *f = fopen("data.txt", "rt");
5     if (!f) { free(buf); return -1; }
6
7     fread(buf, sizeof(char), 20, f);
8
9     if (errno != 0) {
10        fclose(f);
11        free(buf);
12        return -1;
13    }
14
15    fclose(f);
16    free(buf);
17    return 0;
18 }
```

- En C hay que chequear los valores de retorno para ver si hubo un error o no.
- En caso de error se suele abortar la ejecución de la función actual requiriendo previamente liberar los recursos adquiridos
- El problema esta en que es muy fácil equivocarse y liberar un recurso aun no adquirido u olvidarse de liberar un recurso que si lo fue.
- No solo es una cuestión de leaks de memoria. Datos corruptos por archivos o sockets mal cerrados o leaks en el sistema operativo son otros factores que no se solucionan simplemente reiniciando el programa.

- La idea es simple, si hay un recurso (memoria en el heap, un archivo, un socket) hay que encapsular el recurso en un objeto de C++ cuyo constructor lo adquiera e inicialize y cuyo destructor lo libere.
- Nótese como la clave esta en el diseño simétrico del par constructor-destructor.
- Vamos a refinar el concepto RAII en las próximas clases

- Al instanciarse los objetos RAII en el stack, sus constructores adquieren los recursos automáticamente
- Al irse de scope cada objeto se les invoca su destructor automáticamente y por ende liberan sus recursos sin necesidad de hacerlo explícitamente
- El código C++ se simplifica y se hace más robusto a errores de programación: RAII + Stack es uno de los conceptos claves en C++
- Veremos mas sobre RAII, manejo de errores y excepciones en C++ en las próximas clases

RAII - Resource Acquisition Is Initialization

El constructor adquiere el recurso mientras que el destructor lo libera.

```
1 struct Buffer {
2     Buffer(int size) {
3         this->data = (char*)malloc(size*sizeof(char));
4     }
5     ~Buffer() {
6         free(this->data);
7     }
8 };
9
10 struct File {
11     File(const char *name, const char *flags) {
12         this->f = fopen(name, flags);
13     }
14     ~File() {
15         fclose(this->f);
16     }
17 };
```

7542

RAII + Stack: No leaks

La instanciación de objetos RAII en el stack simplifica muchísimo el trabajo en C++.

```
1 int process() {
2     Buffer buf(20);
3
4     File f("data.txt", "rt");
5     if (f.failed()) { return -1; }
6
7     f.read(buf, sizeof(char), 20, f);
8
9     if (error) {
10        return -1;
11    }
12
13    return 0;
14 }
```

7542

Pasaje por referencia

y nos olvidamos de usar punteros

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

7542

- 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.

- 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.

- Al igual que en C, C++ pasa todos los objetos por copia por default
- Aunque util en algunas situaciones, el pasaje por copia es uno de los cuellos de botella mas grandes en terminos de performance que tienen los programas en C/C++. En general, hay que evitar las copias a toda costa!

Diferencias entre referencias y punteros

<pre> 1 int* p = nullptr; 2 int* q; 3 4 int i = 1, j = 2; 5 6 int* r = &i; 7 r = &j; </pre>	<pre> 1 int& p = nullptr; 2 int& q; 3 4 int i = 1, j = 2; 5 6 int& r = i; 7 r = j; // i = j </pre>
---	--

Pasaje por copia

Es el default y trae serios problemas de performance

```

1 int h() {
2     Vector v(5);
3     Vector x = ones(v);
4
5     v.get(0); // ??
6     x.get(0); // 1
7 }
8
9 Vector ones(Vector w) {
10     for (int i = 0; /*...*/)
11         w.set(i, 1);
12
13     return w;
14 }

```

Copia bit a bit naive

La implementación por default

- La copia tanto en C como en C++ es bit a bit y funciona bien para objetos simples.
- Pero cuando un objeto (A) hace referencia a otro (B) a través de un puntero, la copia bit a bit genera una copia (C) que termina apuntando a B: la copia fue superficial y no en profundidad (deep copy).
- Aunque en algunos casos es deseable, la mayoría de las veces el hecho de que dos objetos (A y C) apunten ambos a otro (B) termina con problemas.
- No solo hay que ver con detenimiento las relaciones A→B a través de punteros sino tambien de otros tipos de referencias como los file descriptors.

Constructor por copia

Crea un nuevo objeto copiando a otro.

```
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 };
```



7542

Clases en C++

(DRAFT) Copia de objetos

Crear un objeto copiando otro o reasignar los datos de un objeto ya creado copiando a otro.

```
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 }
```



7542

Clases en C++

(DRAFT) Copia por asignación

Un objeto ya creado copia el contenido 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        free(this->data);  
11        this->data = (int*)malloc(other.size*sizeof(int));  
12        this->size = other.size;  
13        memcpy(this->data, other.data, this->size);  
14  
15        return *this;  
16    }  
17 };
```



7542

Clases en C++

- 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 crea 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.

- 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.

- 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.
- Todos los objetos en C++ son copiables por asignación así que si un objeto no implementa la sobrecarga del operador asignación, C++ le crea una implementación por default que haga 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.

(DRAFT) Objetos no copiables

Hay objetos que no tiene sentido que se puedan copiar. Forzar esta prohibición.

```
1 struct Vector {
2     int *data;
3     int size;
4
5     private:
6     Vector(const Vector &other) = delete;
7     Vector& operator=(const Vector &other) = delete;
8
9 };
```



- 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.

7542

Clases en C++

Constructor por movimiento

Crea un nuevo objeto robándole a otro.

```
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    }
18 };
```



- A diferencia de una copia, el constructor por movimiento le roba o mueve los atributos del objeto fuente.
- Como los atributos se movieron en general es necesario modificar al objeto fuente (**other**) para que deje de apuntar a los recursos ahora apropiados por **this**, de otro modo tendríamos 2 objetos apuntando a un mismo recurso y un bug de memoria a la vuelta de la esquina. Es por esta razón que el parámetro del constructor no es constante a diferencia de lo que sucedía en el constructor por copia.
- Es importante aclarar, el objeto que fue movido (**other**) debe seguir siendo válido de tal manera que se le puede ejecutar sobre **other** el operador asignación y el destructor.

7542

Clases en C++

Asignación por movimiento

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

Ejemplo: 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 }
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 (asignacion)
19     b = std::move(t); // t se mueve a b (asignacion)
20 }
```



7542

Clases en C++

Ejemplo: Accept de un Socket

```
1 struct Socket {
2     // no es copiable
3     Socket(const Socket&) = delete;
4     Socket& operator=(const Socket&) = delete;
5
6     // pero si movable
7     Socket(Socket&& s) { /* ... */ }
8     Socket& operator=(Socket&& s) { /* ... */ }
9
10    Socket accept() {
11        int skt_fd = ::accept(/*...*/); // accept de C
12        Socket accepted(skt_fd);
13
14        return accepted; // retorno por movimiento: super!
15    }
16};
```



Ejemplo: Moviendo objetos de un hilo a otro

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



7542

Clases en C++

Métodos constantes

Nosotros garantizamos que no modifican el estado interno del objeto.

```
1 struct Vector {
2     int *data;
3     int size;
4
5     void set(int pos, val) {
6         this->data[pos] = val;
7     }
8
9     int get(int pos) const {
10        return this->data[pos];
11    }
12
13    /* ... */
14};
```



- Un método constante es un método que no modifica el estado interno del objeto. Esto es, no cambia ningún atributo ni llama a ningún método salvo que este sea también constante.
- Sirve para detectar errores en el código en tiempo de compilación: si un método no modifica el estado debería poderse ponerle la keyword `const`; si el compilador falla es por que hay un bug en el código y nuestra hipótesis de que el método no cambiaba el estado interno del objeto es errónea.

7542

Clases en C++

Objetos constantes

Sólo se pueden invocar métodos que sean constantes.

```
17 void f() {
18     Vector v(5);
19
20     v.set(0, 1); // no const
21     v.get(0); // const
22 }
23
24 void f() {
25     const Vector v(5); // objeto constante
26
27     v.set(0, 1); // no const
28     v.get(0); // const
29 }
```



7542

Clases en C++

Const como promesa

Una función que promete no modificar el objeto pasado como parámetro

```
17 void f() {
18     Vector v(5);
19
20     g(v);
21 }
22
23 void g(const Vector &v) {
24     v.set(0, 1); // no const
25     v.get(0); // const
26 }
```



7542

Clases en C++

- Es comun recibir parámetros constantes. La función promete que no va a cambiar al objeto recibido como parámetro.

- También podemos tener atributos constantes. Estos toman un valor cuando se crean y lo mantienen durante toda la vida del objeto.
- Pequeña aclaración: `const int` y `int const` son equivalentes asi como tambien `const int *` y `int const *`. Sin embargo es distinto `int *` `const`. Confuso?
- `const int *` `p` se lee como "`p` es un puntero; a `int`; constante" mientras que `int *` `const p` se lee como "`p` es constante; puntero; a `int`". El primero apunta a `ints` constantes mientras que el segundo es el puntero quien es constante.

- Al ejecutarse el cuerpo del constructor todos sus atributos ya estan creados.
- Si se necesita construir alguno o todos sus atributos con parámetros especiales hay que usar la member initialization list.
- Esto es útil no solo para crear objetos que no pueden cambiar una vez contruidos (como los atributos `const` y las referencias) sino que también es necesario si queremos construir otros objetos con parámetros custom, sean nuestros atributos o nuestros ancestros (herencia).

Atributos constantes

Atributos que no cambian su valor una vez instanciados.

```
1 struct Vector {
2     int * const data; // no confundir con int const * data;
3     const int size; // equivalente a int const size;
4
5     void set(int pos, val) {
6         this->data[pos] = val;
7     }
8
9     int get(int pos) const {
10        return this->data[pos];
11    }
12
13    /* ... */
14};
```



Member Initialization list

Pasaje de argumentos a los constructores de los atributos.

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(int size) {
6         // atributos ya contruidos; aca solo los re-asigno
7         this->data = (int*)malloc(size*sizeof(int));
8         this->size = size;
9     }
10
11    Vector(int size) : data((int*)malloc(size*sizeof(int))),
12                     size(size) {
13    }
14};
```



Inicialización de atributos

Member Initialization list para inicializar atributos constantes y referencias.

```
1 struct Vector {
2     int * const data;
3     const int size;
4
5     Vector(int size) {
6         // atributos ya contruidos; aca solo los re-asigno
7         this->data = (int*)malloc(size*sizeof(int));
8         this->size = size;
9     }
10
11    Vector(int size) : data((int*)malloc(size*sizeof(int))),
12                     size(size) {
13    }
14};
```



- La member initialization list es el único lugar para inicializar atributos constantes y referencias.

- La member initialization list es el único lugar para inicializar atributos que son objetos que no tienen un constructor por default o sin parámetros.

- La member initialization list permite llamar a otro constructor para delegarle parte de la construcción del objeto. Esto permite reutilizar código entre los constructores.

Inicialización de atributos

Member Initialization list para inicializar atributos que no tienen un constructor por defaults.

```
1 struct DoubleVector {
2     Vector fg;
3     Vector bg;
4
5     DoubleVector(int size) {
6         // fg, bg??
7     }
8 }

1 struct DoubleVector {
2     Vector fg;
3     Vector bg;
4
5     DoubleVector(int size) : fg(size), bg(size) {
6     }
7 }
```

Delegating constructors

```
1 struct DoubleVector {
2     Vector fg;
3     Vector bg;
4
5     DoubleVector(int size) : fg(size), bg(size) {
6     }
7
8     DoubleVector(int size) : fg(size), bg(size) {
9         for (int i = 0; i < size; ++i) {
10             fg.set(i, 0);
11             bg.set(i, 0);
12         }
13     }
14 }

1 struct DoubleVector {
2     Vector fg;
3     Vector bg;
4 }
```

Unidades de compilación

```
1 struct Vector {
2     int *data;
3     const int size;
4
5     Vector(int size);
6     int get(int pos) const;
7     ~Vector();
8 }; // en el archivo vector.h

1 #include "vector.h"
2 Vector::Vector(int size) : size(size) {
3     this->data = (int*)malloc(size*sizeof(int));
4 }
5
6 int Vector::get(int pos) const {
7     return this->data[pos];
8 }
9
10 Vector::~~Vector() {
11     free(this->data);
12 } // en el archivo vector.cpp
```

- Hasta ahora se integró en un solo lugar el código de cada método. Es mas simple pero trae problemas de performance del proceso de compilación.
- Para evitar recompilar una y otra vez el código de los métodos se le define en un archivo .cpp separado de las declaraciones del .h

- Las funciones `malloc` y `free` reservan y liberan memoria pero no crean objetos (no llaman a los constructores ni los destruyen)
- El operador `new` y su contraparte `delete` no solo manejan la memoria del heap sino que también llaman al respectivo constructor y destructor.
- Para crear un array de objetos hay que usar los operadores `new[]` y `delete[]` y la clase a instanciar debe tener un constructor sin parámetros.

Operadores new y delete: uso del heap en C++

```
1 | Vector *vec = (Vector*) malloc(sizeof(Vector));
2 | vector_create(vec, 5);
3 | vector_destroy(vec);
4 | free(vec);

1 | Vector *vec = new Vector(5);
2 | delete vec;

1 | Vector *vecs = new Vector[10];
2 | delete[] vecs;
```



Referencias I

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

