

# Clases en C++

---

Di Paola Martín

`martinp.dipaola <at> gmail.com`

Facultad de Ingeniería  
Universidad de Buenos Aires

# De qué va esto?

- structs y clases en C++

  - Bundle

  - Permisos de acceso

  - Clases

- RAll: Resource Acquisition Is Initialization

  - Constructor y destructor

  - Manejo de errores

- Constantes

  - Constantes

  - Initialization

# **structs y clases en C++**

---

## **Bundle**

## TDAs - Clases en C

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

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

5  void vector_create(struct Vector *v, int size) {
6      v->_data = 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 }
```

# Keyword struct implícita

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

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

```
5 void vector_create(Vector *v, int size) {
6     v->_data = 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 }
```

## Bundle: atributos + métodos

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

## this: 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 = 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 };
```

# Invocación de métodos

```
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 }
```

```
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 }
```



# Reducción de colisiones de nombres

```
1 struct Vector {  
2     int *_data; /*private*/  
3     int _size; /*private*/  
4  
5     void create(int size) { // Vector::create  
6         this->_data = 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};
```

# **structs y clases en C++**

---

## **Permisos de acceso**

# Permisos de acceso

```
1 struct Vector {  
2     private:  
3     int *data;  
4     int size;  
5  
6     public:  
7     void create(int size) {  
8         this->data = 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 };
```

# Permisos de acceso

```
1 struct Vector {  
2     private:  
3     int *data;  
4     int size;  
5  
6     public:  
7     void create(int size) {  
8         this->data = 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 };
```

# Permisos de acceso

```
1 struct Vector {
2     private:
3     int *data;
4     int size;
5
6     public:
7     void create(int size) {
8         this->data = 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 };
```

# Permisos de acceso

```
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 }
```

```
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 }
```

# **structs y clases en C++**

---

## **Clases**

# Clases en C++

```
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 };
```



# Unidades de compilación

```
1 class Vector {  
2     private:  
3         int *data;  
4         int size;  
5  
6     public:  
7         void create(int size);  
8         int get(int pos);  
9         void destroy();  
10  
11 }; // en el archivo vector.h
```

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

# **RAll: Resource Acquisition Is Initialization**

---

**Constructor y destructor**

## Constructor/destructor: manejo de recursos automático

```
1 struct Vector {  
2     int *data;  
3     int size;  
4  
5     Vector(int size) { // create  
6         this->data = 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 };
```

## Constructor/destructor: manejo de recursos automático

```
1 struct Vector {
2     int *data;
3     int size;
4
5     Vector(int size) { // create
6         this->data = 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};
```

## Reduciendo la probabilidad de errores

Diferencia entre reservar memoria y construir un objeto

```
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 }
```

# Reduciendo la probabilidad de errores

## Memoria sin inicializar

```
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 }
```

# Reduciendo la probabilidad de errores

## Destrucción automática

```
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 }
```

## Operadores new y delete

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



# **RAll: Resource Acquisition Is Initialization**

---

**Manejo de errores**

## Manejo de errores en C (madness)

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

# Aplicación del idiom RAI

```
1 struct Buffer {
2     Buffer(int size) {
3         this->data = new char[size];
4     }
5     ~Buffer() {
6         delete[] this->data;
7     }
8 };
9
10 struct File {
11     File(const char *name, const char *flags) {
12         this->f = fopen(name, flags);
13         if (!this->f) throw std::exception("fopen_failed");
14     }
15     ~File() {
16         fclose(this->f);
17     }
18 };
```

## RAII + Stack: No leaks

```
1  int process() {  
2      Buffer buf(20);  
3  
4      File f("data.txt", "rt");  
5      int s = f.read(buf, sizeof(char), 20, f);  
6  
7      if (s != 20)  
8          return -1;  
9  
10     return 0;  
11 } // <-- ~File()  
12 //      ~Buffer()
```

# Constantes

---

## Constantes

## Métodos constantes: no modifican al objeto

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

# Objetos constantes

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

# Objetos constantes

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

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



## Const como promesa

```
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 }
```

# Atributos constantes

```
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, int val) {  
6         this->data[pos] = val;  
7     }  
8  
9     int get(int pos) const {  
10        return this->data[pos];  
11    }  
12  
13    /* ... */  
14 };
```

# Atributos constantes

```
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, int val) {  
6         this->data[pos] = val;  
7     }  
8  
9     int get(int pos) const {  
10        return this->data[pos];  
11    }  
12  
13    /* ... */  
14 };
```

# Atributos constantes

```
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, int val) {  
6         this->data[pos] = val;  
7     }  
8  
9     int get(int pos) const {  
10         return this->data[pos];  
11     }  
12  
13     /* ... */  
14 };
```

# Constantes

---

## Initialization

## Member Initialization List

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

# Member Initialization List

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

# Member Initialization List

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

```
1 struct Vector {  
2     int *data;  
3     int size;  
4  
5     Vector(int size) : data(malloc(size*sizeof(int))),  
6                       size(size) {  
7  
8     }
```



## Inicialización de atributos constantes

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

# Inicialización de atributos constantes

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

# Inicialización de atributos constantes

```
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 = malloc(size*sizeof(int));
8         this->size = size;
9     }
```

```
1 struct Vector {
2     int * const data;
3     const int size;
4
5     Vector(int size) : data(malloc(size*sizeof(int))),
6                       size(size) {
7
8     }
```

# Inicialización de atributos constantes

```
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 = malloc(size*sizeof(int));  
8         this->size = size;  
9     }
```

```
1 struct Vector {  
2     int * const data;  
3     const int size;  
4  
5     Vector(int size) : data(malloc(size*sizeof(int))),  
6                         size(size) {  
7  
8     }
```

## Inicialización de atributos no-default

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

# Inicialización de atributos no-default

```
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     DoubleVector(int size) : fg(size), bg(size) { }  
3  
4     DoubleVector(int size, int val) : fg(size), bg(size) {  
5         for (int i = 0; i < size; ++i) {  
6             fg.set(i, val);  
7             bg.set(i, val);  
8         }  
    }
```

# Delegating constructors

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

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



# **Appendix**

---

## **Referencias**



Bjarne Stroustrup.

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

Addison Wesley, Fourth Edition.