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

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 = 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 }
```

- El estándar de C++ garantiza que si un `struct` no tiene ningun 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++
- 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 deben tener nombres únicos. El conflicto de nombres es un problema común en proyectos grandes en C. En C++ tenemos mejores formas de resolverlos....
- Asi también es convención pasar como primer argumento un puntero al `struct`. Veremos que en C++ hay una forma más 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.

Keyword struct implícita

```
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  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.

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 };
```


- Se integran las funciones y los datos del TDA en una sola unidad.
- Los datos del TDA se lo llaman atributos y las funciones métodos.

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 };
```


- 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.

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 }
```

- 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.

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 };
```


- 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.

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 };
```

- 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 sólo los métodos internos puedan acceder a los métodos y atributos privados.
- Más sobre los permisos `public`/`protected`/`private` y su relación con la herencia en las próximas clases.

Permisos de acceso

```c
// En C
void f() {
  struct Vector v;
  vector_create(&v, 5);
  vector_get(&v, 0);

  v._data;

  vector_destroy(&v);
}
```

```cpp
// En C++
void f() {
  Vector v;
  v.create(5);
  v.get(0);

  v.data;

  v.destroy();
}
```


structs y clases en C++

Clases

Clases en C++

```cpp
struct Vector {
  int *data;  // public by default
  int size;   // public by default
};

class Vector {
  int *data;  // private by default
  int size;   // private by default
};
```


- 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.

Unidades de compilación

```cpp
class Vector {
  private:
  int *data;
  int size;

  public:
  void create(int size);
  int get(int pos);
  void destroy();

}; // en el archivo vector.h
```

```cpp
#include "vector.h"
void Vector::create(int size) {
  this->data = malloc(
  this->size = size;
}

int Vector::get(int pos) {
  return this->data[pos];
}

void Vector::destroy() {
  free(this->data);
} // en el archivo vector.cpp
```

- Hasta ahora se integró en un solo lugar el código de cada método. Es más 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

RAII: 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 };
```


- 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 sólo puede haber un único destructor.
- Un destructor es un código que se ejecuta al momento de destuirse 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.

Reduciendo la probabilidad de errores

Diferencia entre reservar memoria y construir un objeto
Memoria sin inicializar 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 }
```


- 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.

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 };
```

- 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 sólo 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.
- Hay una sutil diferencia sintáctica cuando de tipos primitivos se trata, como `int` o `char`. La expresión `new int` crea un `int` sin inicializar mientras que `new int()` lo inicializa a cero.

RAII: 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 }
```

- 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 sólo 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 con un garbage collector ni reiniciando el programa.

Aplicación del idiom RAII

```
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 };
```


- 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 con el concepto de excepciones.

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()
```


- Tomese un minuto para reflexionar. Vea el código y comparelo con otros códigos de otras personas o incluso de usted mismo. Es la difencia entre alguien que sabe C++ de alguien que escribe código que compila.
- 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.

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 };
```


- 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.

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 }
```


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

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 };
```


- 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 también `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 `int`s constantes mientras que el segundo es el puntero quien es constante.

Constantes

Initialization

Member Initialization List

```
1 struct Vector {
2   int *data;
3   int size;
4
5   Vector(int size) {
6     // atributos ya construidos;  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   }
```

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

Inicialización de atributos constantes

```
1  struct Vector {
2    int * const data;
3    const int size;
4
5    Vector(int size) {
6      // atributos ya construidos;  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    }
```


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

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  }
```


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

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        }
```

```
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        }
```

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

Appendix

Referencias

Referencias I


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