Clases en C++ - RAII

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

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.