

Manejo de Errores en C++

Di Paola Martín

`martinp.dipaola <at> gmail.com`

7542 - Taller de Programación
Facultad de Ingeniería
Universidad de Buenos Aires



1

De qué va esto?

Manejo de Errores

Motivación

Excepciones y su Mal Uso

RAII: Excepciones Bien Usadas

Exception Safety

Excepciones, y ahora qué?



2

Manejo de Errores

El camino feliz: mirada optimista pero ingenua

```
1 void process() {
2     char *buf = (char*) malloc(sizeof(char)*20);
3
4     FILE *f = fopen("data.txt", "rt");
5
6     fread(buf, sizeof(char), 20, f);
7
8     /* ... */
9
10    fclose(f);
11    free(buf);
12 }
```



3

- Código simple pero sin chequeos. Un peligro!
- Ignorar un error puede hacer que el programa crashee o se comporte de forma indefinida. Y lo que es peor, el crash se va a producir en un lugar distanciado de donde realmente hubo un error lo que lo hace mucho mas difícil de debuggear.

Contemplando el camino menos feliz

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



4

- Para evitar que los errores pasen desapercibidos, hay que chequear y hay que hacerlo lo antes posible. Jamás se debe dejar que un error se propague ya que al final tendremos un programa errático muy difícil de debuggear.
- Por cada chequeo hay que manualmente liberar los recursos anteriores.
- Para que el caller sepa que sucedió, hay que retornar un código de error.

Mas robusto, pero poco feliz: una mirada pesimista

```

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

```



5

- Al final, tantos chequeos hacen engorroso un código que era sencillo
- En C se usan otras estrategias. En C++ se usan Excepciones

Las excepciones no implican un buen manejo de errores

```

1 void process() {
2     try {
3         char *buf = (char*) malloc(sizeof(char)*20);
4         if(buf == NULL) { throw -1; }
5
6         FILE *f = fopen("data.txt", "rt");
7         if(f == NULL) { throw -2; }
8
9         int n = fread(buf, sizeof(char), 20, f);
10        if(n < 0) { throw -3; }
11
12        int s = fclose(f);
13        if(s != 0) { throw -4; }
14    } catch(...) {
15        free(buf);
16        fclose(f);
17        throw; // re lanza la excepcion
18    }

```



6

- Lanzamos una excepción con la instrucción **throw**
- En un primer approach, las excepciones nos evitan tener que retornar códigos de error
- Usando **try-catch** atrapamos las excepciones para centralizar la liberación de los recursos
- Una vez hecho esto, la instrucción **throw** dentro de un **catch** relanza la excepción atrapada así quien nos llamó (el caller de la función **process**) sabe que algo salió mal.
- **Y el código queda cada vez peor!!** Este es un ejemplo de un **mal manejo de los errores**: el uso de excepciones no garantiza un buen manejo de los errores.
- El código es tan complejo que estoy liberando un recurso sin antes haberlo adquirido y encima tengo un leak!!

RAII: Resource Acquisition Is Initialization

```

1 class Buffer{
2     char *buf;
3
4     public:
5     Buffer(size_t count) : buf(NULL) {
6         buf = (char*) malloc(sizeof(char)*count);
7         if (!buf) { throw -1; }
8     }
9
10    /* ... */
11
12    ~Buffer() {
13        free(buf); //No pregunto si es NULL o no!
14    }
15 };

```



7

- Todo recurso debe ser encapsulado en una clase
- El constructor se encarga de adquirirlo y el destructor de liberarlo
- Si durante la construcción del objeto algo sale mal, lanzar un excepción.
- Aplicable a Sockets, Buffers, Files, Mutexs, Locks, etc...

RAII y la vuelta al camino feliz

```

1 void process() {
2     Buffer buf(20);
3
4     File f("data.txt", "rt");
5
6     f.read(buf->raw_ptr(), sizeof(char), 20);
7
8     /* ... */
9
10    f.close();
11 } // Destruyo los objetos creados

```



8

- Código simple pero con chequeos ocultos en cada objeto RAII
- Si hay un error, se lanza una excepción. Los errores no se silencian. Se hacen todos los chequeos en un solo lugar: la clase que encapsula al recurso.
- Al salir del scope, por proceso normal o por excepción, los objetos construidos son destruidos mientras que los objetos que no fueron construidos no se destruyen: no hay leaks ni tampoco free(s) de objetos sin aloar.
- Es importante resaltar esto: los objetos RAII deben adquirir los recursos en sus constructores y destruirlos en sus destructores y cada objeto RAII debe hacerse cargo del recurso que encapsula, si algo sale mal, lanzar una excepción.

Si el constructor falla, el destructor no se invoca.

```

1 class DoubleBuffer {
2     char *bufA;
3     char *bufB;
4     /* ... */
5
6     DoubleBuffer(size_t count) : bufA(NULL), bufB(NULL) {
7         bufA = (char*) malloc(sizeof(char)*count);
8         bufB = (char*) malloc(sizeof(char)*count);
9
10        if(!bufA || !bufB) throw -1; // Leak!
11    }
12
13    ~DoubleBuffer() {
14        free(bufA);
15        free(bufB);
16    }

```



9

- El destructor se llama sobre objetos bien construidos. Si hay una excepción en el constructor de DoubleBuffer, C++ va a considerar que el objeto no fue construido y por ende no debe destruirse así que no se le llamara a su destructor. Es importante escribir los constructores con sumo cuidado.
- Para evitar problemas, revisar dos veces la implementación de los constructores

RAII over RAII

```

1 class DoubleBuffer {
2     Buffer bufA;    // No son punteros, ese es el truco!
3     Buffer bufB;
4     /* ... */
5
6     DoubleBuffer(size_t count) : bufA(count), bufB(count) {
7     } //Constructor simple, sin try-catch
8
9     ~DoubleBuffer() {
10    }

```



10

- Al usar objetos RAII como atributos de otros objetos (y no punteros a...), los destructores se llaman automáticamente sobre los objetos creados
- Si el primer `Buffer` (`bufA`) se creo pero el segundo falló, solo el destructor de `bufA` se va a llamar.

RAII - Ejemplos

```

1 class Socket {
2     public:
3         Socket(/*...*/) {
4             this->fd = socket(AF_INET, SOCK_STREAM, 0);
5             if (this->fd == -1)
6                 throw OSError("The_socket_cannot_be_created.");
7         }
8
9         ~Socket() {
10             close(this->fd);
11         }
12 };

```



11

RAII - Ejemplos

```

1 class Lock {
2     Mutex &mutex;
3
4     public:
5         Lock(Mutex &mutex) : mutex(mutex) {
6             mutex.lock();
7         }
8
9         ~Lock() {
10             mutex.unlock();
11         }

```

```

1 void change_shared_data() {
2     this->mutex.lock();
3     /* ... */
4     this->mutex.unlock();
5 }

```

```

1 void change_shared_data() {
2     Lock lock(this->mutex);
3     /* ... */
4 }
5 }

```

12

RAII - Resumen

- Los recursos deben ser encapsulados en objetos, adquiriendolos en el constructor y liberandolos en el destructor.
- Hacer uso del stack. Los objetos del stack son siempre destruidos al final del scope llamando a su destructor.
- Los objetos que encapsulan recursos deben detectar condiciones anómalas y lanzar una excepción.
- Pero **jamás** lanzar una excepción en un destructor. (Condición `noexcept`)
- Una excepción en un constructor hace que el objeto no se cree (y su destructor no se llamara). Liberar sus recursos **a mano** antes de salir del constructor.
- Cuidado con copiar objetos RAI. En general es mejor hacerlos no-copiables y movibles.



13

Exception Safety

No sólo es una cuestión de leaks

```

1 struct Date {
2     void set_day(int day) {
3         this._day = day; if (/* invalid */) throw -1;
4     }
5
6     void set_month(int month) {
7         this._month = month; if (/* invalid */) throw -1;
8     }
9 }
10 try {
11     Date d(30, 04);
12     d.set_day(31);
13     d.set_month(02);
14 } catch(...) {
15     std::cout << d;
16 }

```

El estado final del objeto `d` es ... **31/02**, no tiene sentido!!



14

- Los errores pueden suceder en cualquier momento. No solo hay que evitar leaks (y otros) sino que también hay que tratar de dejar a los objetos en un estado consistente y sin corromper (donde siguen manteniendo sus invariantes).
- Claramente la estrategia "modifico el objeto y luego chequeo" no trae mas que problemas.

Exception safe weak (o basic): objetos consistentes.

```

1 struct Date {
2     void set_day(int day) {
3         if (/* invalid */) throw -1; this._day = day;
4     }
5
6     void set_month(int month) {
7         if (/* invalid */) throw -1; this._month = month;
8     }
9 }
10 try {
11     Date d(28, 01);
12     d.set_day(31);
13     d.set_month(02);
14 } catch(...) {
15     std::cout << d;
16 }
```

Y ahora? imprime 31/01, fecha válida pero no es la original.

15

- Con una simple modificación de código se puede mejorar mucho la situación.
- Ante un error el objeto no queda corrupto (sigue manteniendo sus invariante) aunque puede no quedar en un estado definido.
- Cuando ante una excepción un objeto no se corrompe pero no queda en el estado anterior se dice que el método en cuestión es exception safe weak.

Exception safe strong: objetos inalterados.

```

1 struct Date {
2     void load_date(int day, int month) {
3         if (/* invalid */) throw -1;
4         this.set_day(day);
5         this.set_month(month);
6     }
7
8     void set_day(int day) { /* ... */ }
9     void set_month(int month) { /* ... */ }
10 }
11 try {
12     Date d(28, 01);
13     d.load_date(31, 02);
14 } catch(...) {
15     std::cout << d;
16 }
```

Ahora imprime 28/01, el objeto no cambió.

16

- Cuando ante una excepción un objeto no se corrompe y además queda en el estado anterior se dice que el método en cuestión es exception safe strong.
- Los objetos deberían en lo posible implementar sus métodos como exception safe strong. No siempre es posible, pero en la medida que se pueda debería intentarse.

Encapsulación de objetos y Exception safety

Los setters son un peligro, no es posible garantizar una interfaz strong exception safe:

```

1 public:
2     void load_date(int day, int month);
3     void set_day(int day);
4     void set_month(int month);
```

Pero si la interfaz esta bien diseñada, es más fácil hacer garantías:

```

1 public:
2     void load_date(int day, int month);
3
4 private:
5     void set_day(int day);
6     void set_month(int month);
```

17

- El manejo de errores y las excepciones no se pueden agregar al final de un proyecto, deben diseñarse en conjunto con el resto del objeto pues para poder garantizar los exception safe weak/strong y se requiere de un diseño cuidadoso de la interfaz.
- A grandes razgos, los métodos set son los causantes de muchos problemas por que pueden dejar inválidos a los objetos, sobre todo si algo falla en el medio. Evitar a toda costa los set.
- La encapsulación de un objeto no significa "poner los atributos privados y los getters y setters públicos" sino que significa "poner todo privado salvo los métodos que no pueden dejar inconsistente al objeto".
- Por ejemplo, sea la fecha `Date d(30, 04)` y se quiere cambiar a `28/02` (ambas fechas válidas). Usando solamente los métodos `set_day` y `set_month`, en que orden hay que invocar a esos métodos para obtener la fecha requerida? Fácil no? Y si ahora de la fecha `30/04` se quiere ir a `31/01`? Vale el mismo orden o lo tuviste que cambiar? :) Los setters son malos.

- Los containers de C++ son exception safe strong.
- Este ejemplo explica por que el `std::stack` de C++ tiene un `void pop()` a diferencia de otros lenguajes donde el `pop()` retorna el objeto sacado.

Excepciones, y ahora qué?

El diseño de la interfaz es afectada

```

1  template<class T>
2  T Stack::pop() {
3      if (count_elements == 0) {
4          throw "Stack_empty";
5      }
6      else {
7          T temp;
8          temp = elements[count_elements-1];
9          --count_elements;
10         return temp;
11     }
12 }

```

Qué puede salir mal y lanzar una excepción?

- El constructor por default.
- El operador asignación (=).
- El constructor por copia.

Exception Safety - Resumen

- Tratar de dejar los objetos inalterados (Exception safe strong)
- No poner setters. Es muy fácil equivocarse y dejar objetos inconsistentes.

Explicar el por qué

Pobre

```

1  void parser(/* ... */) {
2      /* ... */
3      if (/* error */)
4          throw ParserError();
5      /* ... */
6  }

```

Mucho mejor

```

1  void parser(/* ... */) {
2      /* ... */
3      if (/* error */)
4          throw ParserError("Encontre_%s_pero_esperaba_%s_en_el_
5                           archivo_%s,_linea_%i", found, expected, filename,
6                           line);
7      /* ... */
8  }

```



- Lo más importante es explicar el error. No usar decenas de clases de errores, solo usar una o dos y que estas reciban un mensaje como parámetro
- Los mensajes deben ser lo mas descriptivos posibles. Piensen en ustedes mismos tratando de entender un error, no sería mejor que el mensaje de la excepción les diga qué paso y en dónde? Hasta sería mejor si les dice las posibles causas y soluciones!
- No hacer `throw new Error()` (usa el heap), usar directamente `throw Error()`. Eviten usar el heap innecesariamente.

- En C++ cualquier cosa puede ser una excepción: un `int`, un puntero, un objeto. Pero es preferible un objeto cuya clase herede de `std::exception`.
- Implementar un método `what()` para retornar el mensaje de error.
- Los métodos con la keyword `noexcept` en sus firmas dicen "este método no lanzará ninguna excepción". Si el método no cumple su promesa, todo el programa crashea (se invoca a la función `terminate`).
- En C++ se puede decir que excepciones un método puede llegar a lanzar escribiendo en su firma por ejemplo `int metodo() throw(OutOfRange)` pero eviten usar este feature como documentación sobre que excepciones se podrían lanzar o no (al estilo Java). Hace que sus clases sean **mas difíciles** de extender.

- Muchas funciones del sistema operativo y de C en general dicen "retorna -1 en caso de error". Esas funciones guardan en la variable global `errno` un código de error más descriptivo que un simple `-1`
- Como `errno` es una variable global cualquier función puede modificarla. Ante la detección de un error, se debe copiar `errno` **inmediatamente** para tener una copia del código sin miedo a que otra función posterior la sobrescriba
- El valor de `errno` puede ser traducido a un mensaje de error con `strerror` o `strerror_r`, este último thread safe. De esa manera se traslada un simple valor numérico en un texto mas explicativo.

Una excepción por dentro

```

1 #include <typeinfo>
2
3 #define OSErrror_LEN_BUFF_ERROR 256
4
5 class OSErrror : public std::exception {
6     private:
7         char msg_error[OSErrror_LEN_BUFF_ERROR];
8
9     public:
10        explicit OSErrror(/* ... */) noexcept;
11        virtual const char *what() const noexcept;
12        virtual ~OSErrror() noexcept {}
13 };

```



21

Wrappeo de errores de C y del sistema operativo

```

1 #include <errno.h>
2 OSErrror::OSErrror(/* ... */) throw() {
3     _errno = errno;
4
5     const char *_m = strerror(_errno);
6     strncpy(msg_error, _m, OSErrror_LEN_BUFF_ERROR);
7
8     msg_error[OSErrror_LEN_BUFF_ERROR-1] = 0;
9 }

```

- Copiar `errno` antes de hacer cualquier cosa.
- Obtener un mensaje explicativo. `strerror` es **not thread safe**, usar `strerror_r`.
- Cuidado de no copiar de más ni olvidarse un `null` al final para evitar **buffer overflows**.



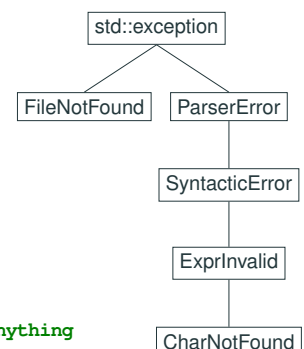
22

Clases de excepciones como discriminantes

```

1 try {
2     parser();
3 } catch(const CharNotFound &e) {
4     printf("%s", e.what());
5 } catch(const ExprInvalid &e) {
6     printf("%s", e.what());
7 } catch(const SyntacticError &e) {
8     printf("%s", e.what());
9 } catch(const ParserError &e) {
10    printf("%s", e.what());
11 } catch(const std::exception &e) {
12    printf("%s", e.what());
13 } catch(...) { // ellipsis: catch anything
14    printf("Unknow_error!");
15 }

```



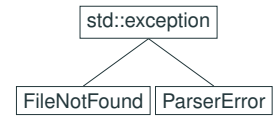
23

- Los `catch` funcionan de forma polimórfica. La clase de una excepción no necesariamente debe coincidir con la firma del `catch`: un `catch` más genérico puede atrapar la excepción igualmente.
- De esto se deduce que los `catch` mas genéricos deben estar al final
- Si una excepción no es atrapada, continua su viaje por el stack
- Preferir la expresión `const Exception&` para atrapar excepciones. Al usar referencias se evitan copias.

Simplificar!

```

1 try {
2     parser();
3 } catch(const std::exception &e) {
4     printf("%s", e.what());
5 } catch(...) {
6     printf("Unknow_error!");
7 }
```



- Pocas clases de errores: no necesitamos tanto poder de discriminación. Menos clases y mejor hechas con buenos mensajes de error.
- Pocos `catch`, solo poner aquellos que van a hacer algo distinto con la excepción.



24

- Como las clases de excepciones se usan principalmente como discriminantes en los `catch`, la razón de tener varias clases es por que hay códigos distintos a ejecutar en cada `catch`.
- En la práctica el código que se encuentra en los `catch` es de simple loggueo que aplica a todos los errores en general. Por lo tanto no deberían haber muchas clases de errores sino unas pocas pero con buenos mensajes de error.

Basta de prints! Loggear a un archivo con syslog

```

1 syslog(LOG_DEBUG, "Mensaje_de_debug.");
2
3 syslog(LOG_INFO, "Un_mensaje_informativo:"
4         "escuchando_en_el_puerto_%i", port);
5
6 syslog(LOG_CRIT, "Un_error:_%s", e.what());
```



25

- En vez de loggear a la consola con un `printf` se puede loggear a traves de una librería llamada `syslog` (para linux) que logguea directamente a un archivo (típicamente `/var/log/messages` o `/var/log/syslog`)
- `syslog` permite loggear poniendo data extra en los mensajes: el process id, el timestamp. Data muy útil si se trabaja con múltiples procesos.
- Hay otras libs útiles similares a `syslog` como `log4j` o `log4cpp` entre otras, todas con las mismas capacidades.
- Vean la página de manual de `syslog`.

No dejar escapar a ninguna excepción

```

1 int main(int argc, char *argv[]) try {
2     /* ... */
3     return 0;
4
5 } catch(const std::exception &e) {
6     syslog(LOG_CRIT, "[Crit]_Error!:_%s", e.what());
7     return 1;
8
9 } catch(...) {
10    syslog(LOG_CRIT, "[Crit]_Unknow_error!");
11    return 1;
12 }
```



26

- Nunca dejar escapar una excepción. En C++ causan un crash.
- En todo el código deberían haber apenas unos pocos `try-catch`. Uno de los lugares en donde deberían estar es en el `main` para atrapar y loggear cualquier excepción antes de que escape del `main` y hagan crashear el programa.

Resumen




- RAI + Objetos en el Stack == (casi) **ningún leak** y no hay necesidad de `try/catch` para liberar recursos. Mantener a los objetos consistentes.
- RAI + Buena Interfaz == Chequeos (al estilo pesimista) en un **solo lugar** (no hay que repetirlos). Quien use esos objetos puede asumir que todo va a salir bien (mirada optimista).
- Chequeos + Excepciones con info + Loggueo a un archivo (Los errores no se silencian, sino que se detectan, propagan y registran) == El debuggeo es **más fácil**.
- Clases de Errores: Usar las que tiene el estándar C++. Crear las propias pero sólo si hacen falta.
- `try/catch`: Deberían haber pocos. En el `main` y tal vez en algún constructor en particular.



27

Appendix

Referencias I

-  Herb Sutter.
Exceptional C++: 47 Engineering Puzzles.
Addison Wesley, 1999.
-  Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, Fourth Edition.
-  man page: syslog

