Manejo de Errores en C++

Di Paola Martín

martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

- Manejo de Errores
  - Motivación
  - Excepciones y su Mal Uso
  - RAII: Excepciones Bien Usadas
- Exception Safety
- Excepciones, y ahora qué?

Manejo de Errores

Motivación

El camino felíz: 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 }
```

Contemplando el camino menos felíz

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

Mas robusto, pero poco felíz: una mirada pesimista

```
1  int process() {
2      char *buf = (char*) malloc(sizeof(char)*20);
3      if(buf == nullptr) { return -1; }
4
5      FILE *f = fopen("data.txt", "rt");
6      if(f == nullptr) { 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 }
```

Manejo de Errores

Excepciones y su Mal Uso

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 == nullptr) { throw -1; }
5
6        FILE *f = fopen("data.txt", "rt");
7        if(f == nullptr) { 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    }
```

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 == nullptr) { throw -1; }
5
6        FILE *f = fopen("data.txt", "rt");
7        if(f == nullptr) { 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    }
```

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 == nullptr) { throw -1; }
5
6        FILE *f = fopen("data.txt", "rt");
7        if(f == nullptr) { 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    }
```

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 == nullptr) { throw -1; }
5
6        FILE *f = fopen("data.txt", "rt");
7        if(f == nullptr) { 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    }
```

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 == nullptr) { throw -1; }
5
6        FILE *f = fopen("data.txt", "rt");
7        if(f == nullptr) { 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    }
```

Manejo de Errores

RAII: Excepciones Bien Usadas

RAII: Resource Acquisition Is Initialization

```
1  class Buffer{
2     char *buf;
3
4     public:
5     Buffer(size_t count) : buf(nullptr) {
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 nullptr o no!
14    }
15 };
```

RAII y la vuelta al camino felíz

```
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
```

RAII y la vuelta al camino felíz

```
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
```

RAII y la vuelta al camino felíz

```
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
```

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(nullptr), bufB(nullptr) {
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     }
```

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    }
```

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 };
```

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  }
```

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 RAII. En general es mejor hacerlos no-copiables y movibles.

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(01);
14 } catch(...) {
15    std::cout << d;
16 }
```

El estado final del objeto `d` es . . .

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(01);
14 } catch(...) {
15    std::cout << d;
16 }
```

El estado final del objeto `d` es `...31/04`, no tiene sentido!!

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(30, 01);
12    d.set_day(31);
13    d.set_month(02);
14 } catch(...) {
15    std::cout << d;
16 }
```

Y ahora?

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(30, 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.

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.

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);
```

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 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 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 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.

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 }
```

Hay posibilidad de leak? Es exception safe weak o strong?

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 }
```

El constructor por copia nos arruina. No hay forma de poner un `try-catch` y revertir "`--count_elements`".

Por eso los containers del estándar ofrecen dos métodos, `void pop()` y `T top()`.

Exception Safety - Resumen

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

Recolectar la mayor información posible

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

Recolectar la mayor información posible

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␣
              archivo␣%s,␣linea␣%i", found, expected, filename,
              line);
5     /* ... */
6 }
```

Una excepción por dentro

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

Wrappeo de errores de C y del sistema operativo

```
1  #include <errno.h>
2  #include <cstdio>
3  #include <cstdarg>
4  OSError::OSError(const char* fmt, ...) noexcept {
5      _errno = errno;
6
7      va_list args;
8      va_start(args, fmt);
9      int s = vsnprintf(msg_error, BUF_LEN, fmt, args);
10     va_end(args);
11
12     strncpy(msg_error+s, strerror(_errno), BUF_LEN-s);
13     msg_error[BUF_LEN-1] = 0;
14 }
```

- Copiar `errno` antes de hacer cualquier cosa.

Wrappeo de errores de C y del sistema operativo

```
1  #include <errno.h>
2  #include <cstdio>
3  #include <cstdarg>
4  OSError::OSError(const char* fmt, ...) noexcept {
5      _errno = errno;
6
7      va_list args;
8      va_start(args, fmt);
9      int s = vsnprintf(msg_error, BUF_LEN, fmt, args);
10     va_end(args);
11
12     strncpy(msg_error+s, strerror(_errno), BUF_LEN-s);
13     msg_error[BUF_LEN-1] = 0;
14 }
```

- Obtener un mensaje explicativo del contexto. Aceptar un mensaje de error y argumentos variables como lo hace `printf`.

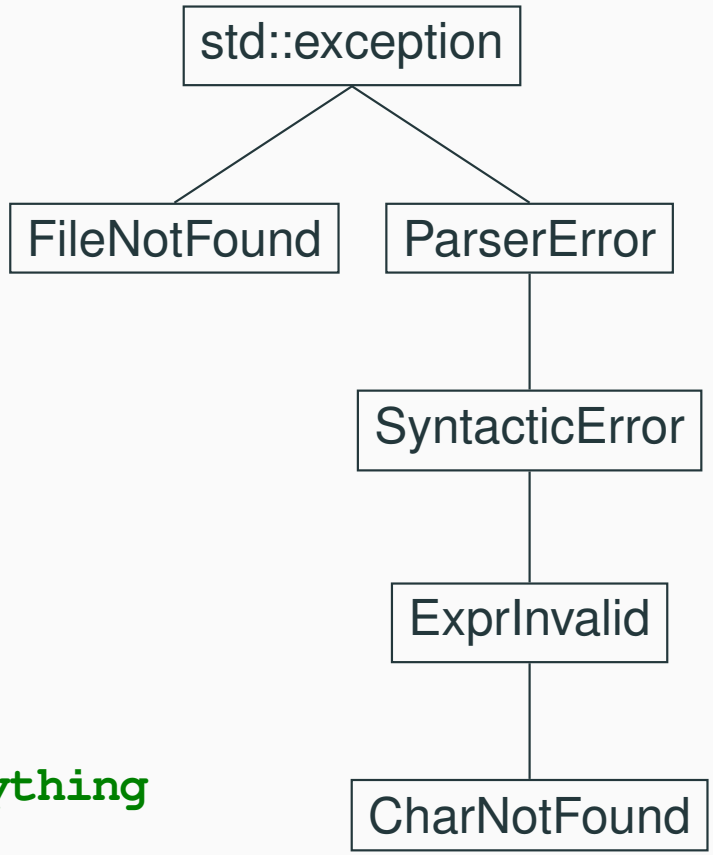
Wrappeo de errores de C y del sistema operativo

```
1  #include <errno.h>
2  #include <cstdio>
3  #include <cstdarg>
4  OSError::OSError(const char* fmt, ...) noexcept {
5      _errno = errno;
6
7      va_list args;
8      va_start(args, fmt);
9      int s = vsnprintf(msg_error, BUF_LEN, fmt, args);
10     va_end(args);
11
12     strncpy(msg_error+s, strerror(_errno), BUF_LEN-s);
13     msg_error[BUF_LEN-1] = 0;
14 }
```

- Obtener un mensaje del sistema operativo con `strerror`. Cuidado que no es not thread safe, usar `strerror_r`.

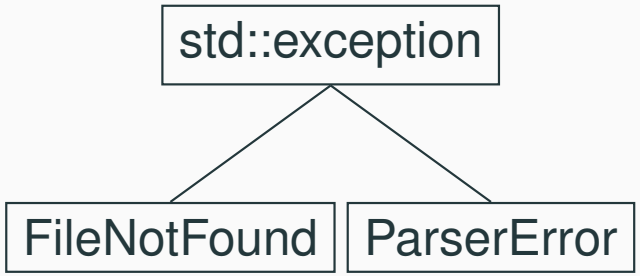
Clases de excepciones como discriminantes

```
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 }
```



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.

Basta de prints! Loguear 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());
```

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 }
```

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de `try`/`catch` para liberar recursos. Mantener a los objetos consistentes.

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de `try`/`catch` para liberar recursos. Mantener a los objetos consistentes.
- RAII + 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).

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de `try/catch` para liberar recursos. Mantener a los objetos consistentes.
- RAII + 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.

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de `try`/`catch` para liberar recursos. Mantener a los objetos consistentes.
- RAII + 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.

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de `try`/`catch` para liberar recursos. Mantener a los objetos consistentes.
- RAII + 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.

Appendix

Referencias

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