Manejo de Errores en C++

[1] Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

El camino felíz

Una mirada optimista pero muy 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 }
```

El camino felíz
Una mirada optimista pero muy 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 }
```

- Código simple pero sin chequeos. Un peligro!
- Ignorar un error puede hacer que el programa crashee o se comporte de forma indefinida.

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 == 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 }
```

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 == 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 }
```

- Para evitar que los errores pasen desapercibidos, hay que chequear.
- Por cada chequeo hay que manualmente liberar los recursos anteriores.
- Para que el caller sepa que sucedio, hay que retornar un código de error.

Mas robusto, pero poco felíz

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 }
```

Mas robusto, pero poco felíz
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 }
```

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

Excepciones, primer approach

Excepciones no implica Manejo de Errores

```
1  void 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          throw -1;
9      }
10
11     fread(buf, sizeof(char), 20, f);
12
13     /* ... */
14     fclose(f);
15     free(buf);
16 }
```

Excepciones, primer approach
Excepciones no implica Manejo de Errores

```
1  void 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        throw -1;
9     }
10
11    fread(buf, sizeof(char), 20, f);
12
13    /* ... */
14    fclose(f);
15    free(buf);
16 }
```

- Lanzamos una excepción con la instrucción throw
- En un primer approach, las excepciones nos evitan tener que retornar códigos de error

Las excepciones no son mágicas

```
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;
18     }
```

Las excepciones no son mágicas

```
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;
18    }
```

- La instrucción throw dentro de un catch relanza la excepcion atrapada
- Usando try-catch atrapamos las excepciones para centralizar la liberación de los recursos
- Y el código queda cada vez peor!!
- El código es tan complejo que me olvide de un free!

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 };
```

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 };
```

- Todo recurso debe ser encapsulado en una clase
- El constructor se encarga de adquirirlo y el destructor de liberarlo
- Aplicable a Sockets, Buffers, Files, Mutexs, Locks, etc...

RAII y la vuelta al camino felíz

Una mirada optimista pero para nada ingenua!

```
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

Una mirada optimista pero para nada ingenua!

```
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
```

- 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 alocar.
- Si los objetos son RAII, la destrucciń de ellos libera los recursos. No hay leaks!

Excepciones en un constructor

Si el constructor falla, el destructor no se invoca. Cuidado con los leaks!

```
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     }
```

Excepciones en un constructor

Si el constructor falla, el destructor no se invoca. Cuidado con los leaks!

```
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    }
```

- El destructor se llama sobre objetos bien construidos. Si hay una excepción en el constructor de DoubleBuffer, no se le llamara a su destructor. Por lo que es importante escribir los constructores con sumo cuidado.
- Para evitar problemas, doble chequear 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     }
```

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    }
```

- 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 };
```

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 anomalas y lanzar una excepción.
- Pero jamás lanzar una excepción en un destructor. (Condición no-throw)
- 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.

No sólo es una cuestión de leaks

Los objetos deben quedar en un estado consistente.

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

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

No sólo es una cuestión de leaks
Los objetos deben quedar en un estado consistente.

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

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

- 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 sin corromper (donde siguen manteniendo sus invariantes).
- Claramente la estrategia "modifico el objeto y luego chequeo" no trae mas que problemas.

No sólo es una cuestión de leaks

Exception safe weak, objetos consistentes luego de una excepción.

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

Y ahora? La fecha final es 31/01, válida pero no es ni la fecha pedida ni la original.

No sólo es una cuestión de leaks

Exception safe weak, objetos consistentes luego de una excepción.

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

Y ahora? La fecha final es 31/01, válida pero no es ni la fecha pedida ni la original.

- Con una simple modifición 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.

No sólo es una cuestión de leaks

Exception safe strong, objetos inalterados luego de una excepción.

```
1  class Date {
2     public:
3     void load_date(int day, int month) {
4        if (/* invalid */) throw -1;
5        this.set_day(day);
6        this.set_month(month);
7     }
8
9     void set_day(int day) { /* ... */ }
10    void set_month(int month) { /* ... */ }
11 }
12 /* ... */
13    Date d(28, 01);
14    d.load_day(31, 02);
```

La fecha final es 28/01, la misma que se tenía antes de la excepción.

No sólo es una cuestión de leaks

Exception safe strong, objetos inalterados luego de una excepción.

```
1  class Date {
2     public:
3     void load_date(int day, int month) {
4        if (/* invalid */) throw -1;
5        this.set_day(day);
6        this.set_month(month);
7     }
8
9     void set_day(int day) { /* ... */ }
10    void set_month(int month) { /* ... */ }
11 }
12 /* ... */
13    Date d(28, 01);
14    d.load_day(31, 02);
```

La fecha final es 28/01, la misma que se tenía antes de la excepción.

- 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.

El diseño de la interfaz es afectada

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

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 manejo de errores y las excepciones no se pueden agregar, deben diseñarse en conjunto con el resto del objeto pues para poder garantizar los exception safe weak/strong 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.

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.

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

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.

Hay posibilidad de leak? Es exception safe weak o strong? 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()`.

- Los containers de C++ son exception safe strong.
- Por este ejemplo se explica por que el stack de C++ tiene un void pop() a diferencia de otros lenguages que el pop() retorna el objeto sacado.

Exception Safety - Resumen

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

Explicar el por qué

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 }
```

Explicar el por qué
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 }
```

- 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.
- No hacer throw new Error() (usa el heap), usar directamente throw Error(). Eviten usar el heap innecesariamente.

Excepciones, y ahora qué?
Explicar el por qué

- Cualquier cosa puede ser una excepción: un `int`, un puntero, un objeto. Pero es preferible un objeto cuya clase herede de `std::exception`.
- Mantener una implementación sencilla que arme un buen mensaje que explique el por qué del error. Y si puede decir como arreglarlo, mejor!

Una excepción por dentro

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

Una excepción por dentro

```cpp
#include <typeinfo>

#define OSError_LEN_BUFF_ERROR  256

class OSError : public std::exception {
   private:
    char msg_error[OSError_LEN_BUFF_ERROR];

   public:
    explicit OSError(/* ... */) throw();
    virtual const char *what() const throw();
    virtual ~OSError() throw() {}
};
```

- En C++ cualquier cosa es una excepción, pero si se implementa un clase de error, que herede de std::exception
- Implementar un método what() para retornar el mensaje de error.
- Los métodos con la keyword throw en sus firmas dicen "este método no lanzará ninguna excepción". Si el método no cumple su promeza, todo el programa crashea.
- No usen la keyword throw en la firma de los metodos como documentación sobre que excepciónes se podrían lanzar o no (al estilo Java). Hace que sus clases sean mas difíciles de extender.

Una excepción por dentro

Ejemplo de wrapper de errno

```
1 #include <errno.h>
2 OSError::OSError(/* ... */) throw() {
3     _errno = errno;
4
5     const char *_m = strerror(_errno);
6     strncpy(msg_error, _m, OSError_LEN_BUFF_ERROR);
7
8     msg_error[OSError_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.

Una excepción por dentro
Ejemplo de wrapper de errno

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

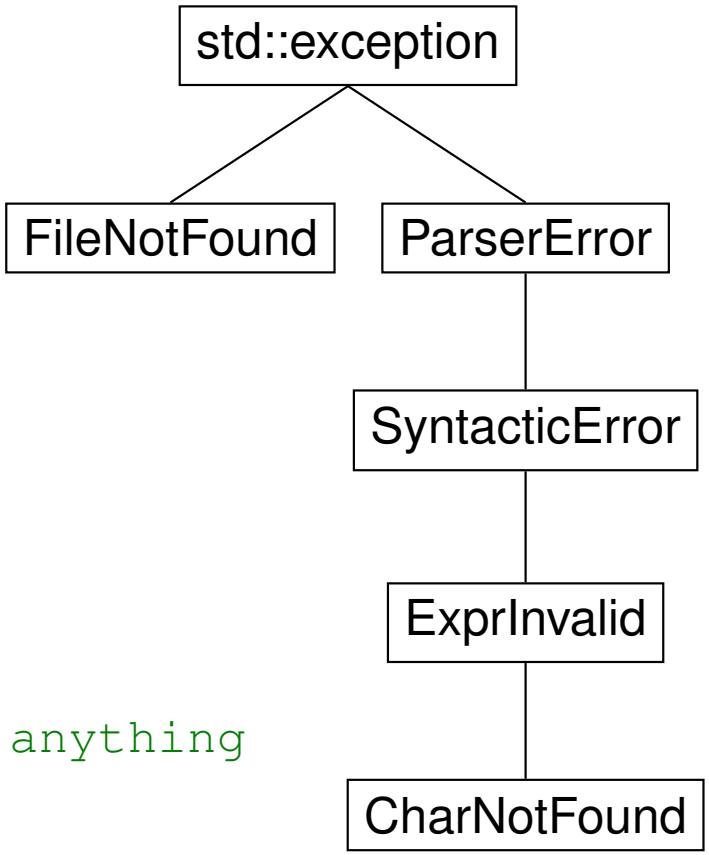
- Copiar `errno` antes de hacer cualquier cosa.
- Obtener un mensaje explicativo. `strerror` es not thread safe, usar `strerror_r`.
- Cuidado de no copiar de más ni olvidarse un null al final para evitar buffer overflows.

- 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 para tener una copia del código sin miedo a que otra función posterior la sobreescriba
- 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 numerico en un string mas explicativo.

Clases de excepciones como discriminantes

De lo más específico a lo más genérico.

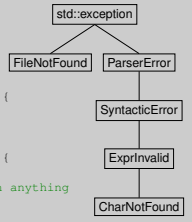
```
1  try {
2      parser();
3  } catch(const CharNotFound &e) {
4      printf("%s", e.what());
5  } catch(const ExprInvalid &e) {
6      printf("%s", e.what());
7  } catch(const SyntacticError &e) {
8      printf("%s", e.what());
9  } catch(const ParserError &e) {
10     printf("%s", e.what());
11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



Clases de excepciones como discriminantes

De lo más específico a lo más genérico.

```
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 }
```

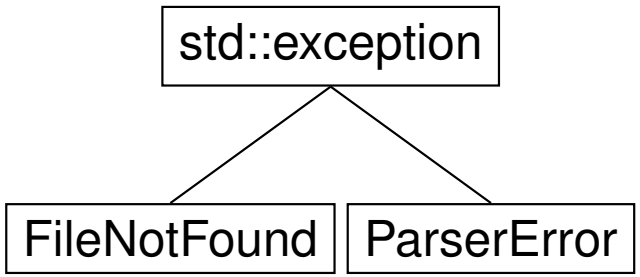


- Los catch funcionan de forma polimórfica. La clase de una excepción no necesariamente debe coincidir con la firma de catch. Un catch más genérico puede atrapar la excepción igualmente.
- De esto se deduce que los catch mas genéricos deben estar al final
- Si una excepción no es atrapada, continua su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

Jerarquías simples

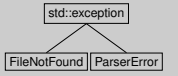
Clases distintas sólo si hago con ellas cosas distintas

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



Jerarquías simples
Clases distintas sólo si hago con ellas cosas distintas

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



- Como las clases de excepciones se usan principalmente como discriminantes en los catch, la razón de tener varias clases es por que hay código distinto 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! Loguear a un archivo

syslog ofrece timestamps, niveles de log y cierre automático del archivo.

```
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());
```

Basta de prints! Loguear a un archivo

syslog ofrece timestamps, niveles de log y cierre automático del archivo.

```
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());
```

- En vez de logguear a la consola con un printf se puede logguear a traves de una librería llamada syslog (para linux)
- syslog permite logguear poniendo data extra en los mensajes: el process id, el timestamp. Data muy útil si se trabaja con multiples procesos.
- Hay otras libs útiles similares a syslog como log4j o log4cpp entre otras, todas con las mismas capacidades.

Basta de prints! Loguear a un archivo

syslog ofrece timestamps, niveles de log y cierre automático del archivo.

```
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 }
```

Basta de prints! Loguear a un archivo

syslog ofrece timestamps, niveles de log y cierre automático del archivo.

```
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 }
```

- Nunca dejar escapar una excepción. En C++ causa 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 logguear cualquier excepción antes de que escapen del main y crasheen el programa.

Resumen

- RAII + Objetos en el Stack == (casi) ningún leak y no hay necesidad de try/catch para liberar recursos.
- RAII + Buena Interfaz == Chequeos (al estilo pesimista) en un solo lugar (no hay que repetirlos). Objetos consistentes.
- 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.

Referencias I


Herb Sutter.
Exceptional C++: 47 Engineering Puzzles.
Addison Wesley, 1999.


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


man page: syslog