Обработка ошибок: исключения

Александр Смаль

Академический университет
13 марта 2014
Санкт-Петербург

Способы обработки ошибок

- Отсутствие обработки ошибок.

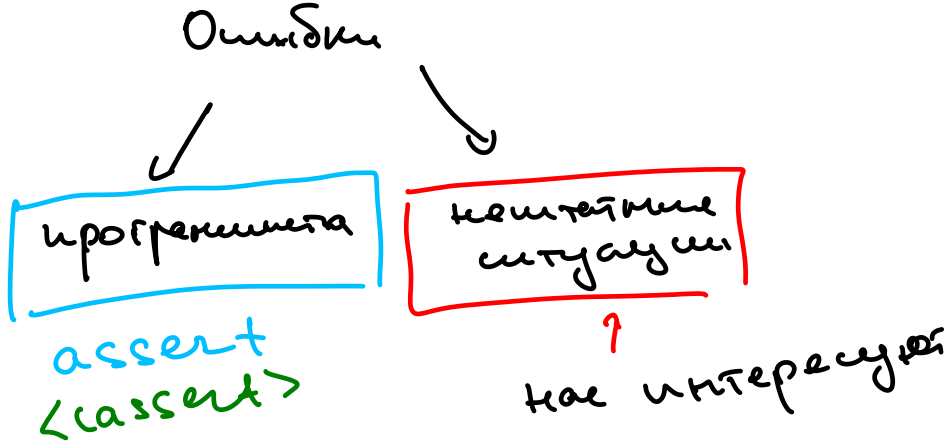
```
size_t write(string file, DB const& data);
```

\+

- меньше кода
- код проще
- быстрее
- оптимистично
- круто

—

- не надежно
- бесит пользователя
- можно потерять данные



Способы обработки ошибок

- Отсутствие обработки ошибок.

```
size_t write(string file, DB const& data);
```

- Возврат статуса операции:

```
bool write(string file, DB const& data, size_t & bytes);
```

+

- можно понять, что произошла ошибка
- быстро
- просто

−

- можно проигнорировать
- усложняет сигнатуру ф-ий
- нельзя в конструкторах
- нет информации об ошибке

Способы обработки ошибок

- Отсутствие обработки ошибок.

```
size_t write(string file, DB const& data);
```

- Возврат статуса операции:

```
bool write(string file, DB const& data, size_t & bytes);
```

- Возврат кода ошибки:

```
enum Err { OK, IO_FAIL, NET_FAIL };

Err write(string file, DB const& data, size_t & bytes);
```

HRESULT →

+

- можно понять, что произошла ошибка
- быстро
- просто
- есть дифференциация ошибок

—

- можно проигнорировать
- изменяет сигнатуру ф-ий
- нельзя в конструкторах
- мало информации

COM

Способы обработки ошибок

- Отсутствие обработки ошибок.

```
size_t write(string file, DB const& data);
```

- Возврат статуса операции:

```
bool write(string file, DB const& data, size_t & bytes);
```

- Возврат кода ошибки:

```
enum Err { OK, IO_FAIL, NET_FAIL };

Err write(string file, DB const& data, size_t & bytes);
```

- Использование глобальной кода ошибки:

```
size_t write(string file, DB const& data);
...
size_t bytes = write(f, db);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

глобальная перем.

\+
- есть информация об ошибках
- не изменяется сигнатура
...

−
- глобальность
- можно испортить
- надо обнулять
...

Концепция исключений

Исключение — это объект, содержащий информацию об ошибке, который передаётся от места возникновения ошибки к месту её обработки.

```
double div( int x, int y ) {
    if ( y == 0 )
        throw string("Division by zero");
    return double(x) / y;
}

struct FileError { string name; ... };
void dump(string file, double x) {
    if (!exist(file))
        throw FileError(file);
    write(file, x);
}

void foo(string file, int x, int y) {
    try {
        dump(file, div(x, y))
    } catch (string & s) {
        // log
    } catch (FileError & e) {
        // log
    } catch (...) { // any other
        throw; // have no idea what to do
    }
}
```

+
- сложно проектировать.
- не нужно менять сигнатуру
- можно передать всю информацию
- побуждает писать обработку ошибок

—
- медленней
- сложный механизм

не происходит приведения типов

Почему не стоит бросать встроенные типы

```
int foo() {
    if (...) throw 1;
    ...
    if (...) throw 3.14;
}

void bar(int a) {
    if (a == 0) throw string("Division by zero");
    else if (a % 2 != 0) throw string("Invalid data");
    else throw string("Not my fault!");
}

int main () {
    try {
        bar(foo());
    } catch (string & s) {
        if (s == "Invalid data")
        else ...
    } catch (int a) {
        ...
    } catch (double d) {
        ...
    } catch (...) {
        ...
    }
}
```

можно ловить производный класс, а кидать по ссылке на базовый

Стандартные классы исключений

Базовый класс для всех исключений (в <exception>):

```
class exception {
  virtual ~exception();
  virtual const char* what() const;
};
```

Стандартные классы ошибок (в <stdexcept>):

- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - range_error
  - overflow_error
  - underflow_error

vector::at(i)

```
int main() {
    try { ... }
    catch (std::exception const& e) {
       std::cerr << e.what() << '\n';
    }
}
```

Stack unwinding

При возникновении исключения, объекты на стеке удаляются в естественном (обратном) порядке.

```
void foo() {
    D d;
    E e;
    throw 42;
    F f;
}

void bar() {
    A a;
    try {
        B b;
        foo();
        C c;
    } catch (int i) {
        throw i;
    }
}
```

~E(), ~D(), ~B(), ~A()

Исключения в конструкторе

Исключения в конструкторе — единственный способ сообщить об ошибке в процессе конструирования объекта.

```
struct Database {
    Database(string const& uri) {
        if (!connect(uri))
            throw NetworkError();
    }
    ...
};

int main() {
    string uri = ...;
    try {
        Database * db = new Database(uri);
        db->dump(file);
        delete db;
    } catch (std::exception const& e) {
        std::cerr << e.what() << '\n';
    }
}
```

не вызывается деструктор

Исключения в списке инициализации

```
struct System {
    Database    db_;
    DataHolder  dh_;

    System(string const& db_uri, string const& data)
    try : db_(db_uri), dh_(data)
    {
        ... // constructor
    }
    catch (std::exception const& e) {
        log("Problem with system creation");
        throw;
    }
};
```

Недопустимость исключений в деструкторах

```
void foo() {
    D d;
    E e; // exception in destructor
    throw 42;
    F f;
}

void bar() {
    A a;
    try {
        B b;
        foo();
        C c;
    } catch (int i) {
        throw i;
    }
}
```

Нельзя бросать исключения в деструкторе!

Спецификация исключений

Редкоиспользуемая и устаревшая возможность C++, позволяющая указать у функции список бросаемых исключений.

```
int foo() throw(int) {
    if (...) throw 1;
    ...
    if (...) throw 3.14;
}
```

Если сработает второй if, то вызовется программа аварийно завершится. Эквивалентно.

```
int foo() {
    try {
        if (...) throw 1;
        ...
        if (...) throw 3.14;
    } catch (int i) {
        throw i;
    } catch (...) {
        terminate(); // set_unexpected
    }
}
```

Стратегии обработки исключений

Есть несколько правил хорошего тона:

1. Обрабатывать ошибки.
2. Обрабатывать ошибки единообразно.
3. Централизованно обрабатывать ошибки в пределах одной логической части кода.
4. Обрабатывать ошибки там, где на них можно адекватно отреагировать.
5. Если ошибку на этом уровне не обработать — пересылать её выше.
6. Отлавливать все ошибки в точке входа.

Правила использования исключений:

1. Отлавливать исключения в деструкторах, если они могут там быть.
2. Не использовать спецификацию исключений.
3. Передавать исключения по значению, а принимать — по ссылке.
4. Аккуратно работать с исключениями в динамических библиотеках.