Обработка ошибок

Александр Смаль

CS центр
15 марта 2017
Санкт-Петербург

Способы обработки ошибок

- Отсутствие обработки ошибок.

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

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);
```

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);
```

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 переменной для кода ошибки:

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

Концепция исключений

Исключение — объект, содержащий описание ошибки, который передаётся от места её возникновения к месту обработки.

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

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") ...
    } catch (int a) { ...
    } catch (double d) { ...
    } catch (...) { ...
    }
}
```

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

```
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;
    }
}
```

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;
    }
}
```

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

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;
    }
};
```

Спецификация исключений

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 программа аварийно завершится.

```
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 их можно обработать.
5. Если ошибку тут не обработать — пересылать её выше.
6. Отлавливать все ошибки в точке входа.

Правила использования исключений:

1. Отлавливать исключения в деструкторах, если нужно.
2. Не использовать спецификацию исключений.
3. Передавать исключения по значению, а принимать — по ссылке.
4. Осторожно использовать исключения в библиотеках.

Гарантии безопасности исключений

- **Гарантия отсутствия исключений**
  “Ни при каких обстоятельствах функция не будет генерировать исключения”.

Гарантии безопасности исключений

- **Гарантия отсутствия исключений**
  “Ни при каких обстоятельствах функция не будет генерировать исключения”.
- **Базовая гарантия**
  “При возникновении любого исключения в некотором методе, состояние программы должно оставаться согласованным”.

Гарантии безопасности исключений

- **Гарантия отсутствия исключений**
  “Ни при каких обстоятельствах функция не будет генерировать исключения”.
- **Базовая гарантия**
  “При возникновении любого исключения в некотором методе, состояние программы должно оставаться согласованным”.
- **Строгая гарантия**
  “Если при выполнении операции возникает исключение, то программа должна остаться в состоянии, которое было до начала выполнения операции”.

Строгая гарантия исключений

- В каком случае мы не можем обеспечить строгую гарантию исключений?
- Как обеспечить строгую гарантию в остальных случаях?
- Когда можно обеспечить строгую гарантию *эффективно*?

В чём сложность?

```
template<class T>
struct Array {
    void resize(size_t n) {
        T * ndata = new T[n];
        for (size_t i = 0; i != n && i != size_; ++i)
            ndata[i] = data_[i];

        delete [] data_;
        data = ndata;
        size_ = n;
    }

T *     data_;
size_t  size_;
};
```

В чём сложность?

```
template<class T> struct Array {
    void resize(size_t n) {
        T * ndata = 0;
        try {
            ndata = new T[n];
            for (size_t i = 0; i != n && i != size_; ++i)
                ndata[i] = data_[i];
        } catch (...) {
            delete [] ndata;
            throw;
        }
        delete [] data_;
        data = ndata;
        size_ = n;
    }
T *     data_;
size_t  size_;
};
```

Использование RAII

```
template<class T>
struct Array {
    void resize(size_t n) {
        shared_array<T> ndata(new T[n]);

        for (size_t i = 0; i != n && i != size_; ++i)
            ndata.get()[i] = data_.get()[i];

        data_ = ndata;
        size_ = n;
    }

shared_array<T> data_;
size_t          size_;
};
```

Использование swap

```
template<class T>
struct Array {
    void resize(size_t n) {
        Array t(n);
        for (size_t i = 0; i != n && i != size_; ++i)
            t[i] = data_[i];

        t.swap(*this);
    }

T       * data_;
size_t    size_;
};
```

Проектирование с учётом исключений

Рассмотрим традиционный интерфейс стека:

```
template<class T>
struct Stack {
    void push(T const& t)
    {
        data.push_back(t);
    }

    T pop() {
        T tmp = data_.back();
        data_.pop_back();
        return tmp;
    }

    std::vector<T> data_;
};
```

Проектирование с учётом исключений

Рассмотрим традиционный интерфейс стека:

```
template<class T>
struct Stack {
    void push(T const& t)
    {
        data.push_back(t);
    }

    void pop(T & res) {
        res = data_.back(); // *
        data_.pop_back();
    }

    std::vector<T> data_;
};
```

Использование auto_ptr

```
template<class T>
struct Stack {
    void push(T const& t)
    {
        data.push_back(t);
    }

    auto_ptr<T> pop() {
        auto_ptr<T> tmp = new T(data_.back());
        data_.pop_back();
        return tmp;
    }

    std::vector<T> data_;
};
```

Проблемы с RAII

Следите за порядком операций:

```
void f(auto_ptr<T> p, auto_ptr<V> q);

// incorrect
f(auto_ptr<T>(new T()), auto_ptr<V>(new V()));

// correct
auto_ptr<T> p(new T());
f(p, auto_ptr<V>(new V()));
```

Исключения в стандартной библиотеке

- `vector, deque, string, bitset` кидают `std::out_of_range` (фукция `at`).
- Оператор `new T` кидает `std::bad_alloc`.
  Оператор `new (std::nothrow) T` в возвращает 0.
- Потоки ввода-вывода.

```
std::ifstream file;
file.exceptions (std::ifstream::failbit | std::ifstream::badbit);
try {
    file.open ("test.txt");
    while (!file.eof()) file.get();
    file.close();
}
catch (std::ifstream::failure const& e) {
    std::cerr << "Exception opening/reading/closing file\n";
}
```