Логические ошибки и исключительные ситуации

- **Логические ошибки**.

  Ошибки в логике работы программы, которые происходят из-за неправильно написанного кода, т.е. это ошибки программиста:

  - выход за границу массива,
  - попытка деления на ноль,
  - обращение по нулевому указателю,
  - …

Логические ошибки и исключительные ситуации

- **Логические ошибки**.

  Ошибки в логике работы программы, которые происходят из-за неправильно написанного кода, т.е. это ошибки программиста:

  - выход за границу массива,
  - попытка деления на ноль,
  - обращение по нулевому указателю,
  - …

- **Исключительные ситуации**.

  Ситуации, которые требуют особой обработки.

  Возникновение таких ситуаций — это „нормальное“ поведение программы.

  - ошибка записи на диск,
  - недоступность сервера,
  - неправильный формат файла,
  - …

Выявление логических ошибок на этапе разработки

- Оператор `static_assert`.

```
#include<type_traits>



template<class T>
void countdown(T start) {
    static_assert(std::is_integral<T>::value
                  && std::is_signed<T>::value,
                     "Requires signed integral type");

    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Выявление логических ошибок на этапе разработки

- Оператор `static_assert`.
- Макрос `assert`.

```
#include<type_traits>
//#define NDEBUG
#include <cassert>

template<class T>
void countdown(T start) {
    static_assert(std::is_integral<T>::value
                  && std::is_signed<T>::value,
                     "Requires signed integral type");
    assert(start >= 0);
    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

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

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

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

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

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

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

- Глобальная переменная для кода ошибки:

```
size_t write(string file, string data);
```

```
size_t bytes = write(f, data);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

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

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

- Глобальная переменная для кода ошибки:

```
size_t write(string file, string data);
```

```
size_t bytes = write(f, data);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

- Исключения.

Исключения

```
size_t write(string file, string data) {
    if (!open(file)) throw FileOpenError(file);
    //...
}
double safediv(int x, int y) {
    if (y == 0) throw MathError("Division by zero");
    return double(x) / y;
}
void write_x_div_y(string file, int x, int y) {
    try {
        write(file, to_string(safediv(x, y)));
    } catch (MathError & s) {
        // обработка ошибки в safediv
    } catch (FileError & e) {
        // обработка ошибки в write
    } catch (...) {
        // все остальные ошибки
    }
}
```

Stack unwinding

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

```
void foo() {
    D d;
    E e(d);
    if (!e) throw F();
    G g(e);
}
void bar() {
    A a;
    try {
        B b;
        foo();
        C c;
    } catch (F & f) {
        // обработка и пересылка
        throw f;
    }
}
```

Почему не стоит бросать встроенные типы

```
int foo() {
    if (...) throw -1;
    if (...) throw 3.1415;
}
void bar(int a) {
    if (a == 0) throw string("Not my fault!");
}
int main () {
    try { bar(foo());
    } catch (string & s) {
        // только текст
    } catch (int a) {
        // мало информации
    } catch (double d) {
        // мало информации
    } catch (...) {
        // нет информации
    }
}
```

Стандартные классы исключений

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

```
struct 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';
    }
}
```

Исключения в стандартной библиотеке

- Метод at контейнеров array, vector, deque, basic_string, bitset, map, unordered_map бросает out_of_range.
- Оператор new T бросает bad_alloc.
  Оператор new (std::nothrow) T в возвращает 0.
- Оператор typeid от разыменованного нулевого указателя бросает bad_typeid.
- Потоки ввода-вывода.

```
std::ifstream file;
file.exceptions( std::ifstream::failbit
                 | std::ifstream::badbit );
try {
    file.open ("test.txt");
    cout << file.get() << endl;
    file.close();
}
catch (std::ifstream::failure const& e) {
    cerr << e.what() << endl;
}
```

Как обрабатывать ошибки?

Есть несколько „правил хорошего тона“.

- Разделяйте „ошибки программиста“ и „исключительные ситуации“.
- Используйте `assert` и `static_assert` для выявления ошибок на этапе разработки.
- В пределах одной логической части кода обрабатывайте ошибки централизованно и однообразно.
- Обрабатывайте ошибки там, где их можно обработать.
- Если в данном месте ошибку не обработать, то пересылайте её выше при помощи исключения.
- Бросайте только стандартные классы исключений или производные от них.
- Бросайте исключения по значению, а отлавливайте по ссылке.
- Отлавливайте все исключения в точке входа.

Исключения в деструкторах

Исключения не должны покидать деструкторы.

- Двойное исключение:

```
void foo() {
    try {
        Bad b; // исключение в деструкторе
        bar(); // исключение
    } catch (std::exception & e) {
        // ...
    }
}
```

- Неопределённое поведение:

```
void bar() {
    Bad * bad = new Bad[100];
    // ислючение в деструкторе №20
    delete [] bad;
}
```

Исключения в конструкторе

Исключения — это единственный способ прервать конструирование объекта и сообщить об ошибке.

```
struct Database {
    explicit Database(string const& uri) {
        if (!connect(uri))
            throw ConnectionError(uri);
    }
    ~Database() { disconnect(); }
    // ...
};
int main() {
    try {
        Database * db = new Database("db.local");
        db->dump("db-local-dump.sql");
        delete db;
    } catch (std::exception const& e) {
        std::cerr << e.what() << '\n';
    }
}
```

Исключения в списке инициализации

Позволяет отловить исключения при создании полей класса.

```
struct System
{
    System(string const& uri, string const& data)
    try : db_(uri), dh_(data)
    {
        // тело конструктора
    }
    catch (std::exception & e) {
        log("System constructor: ", e);
        throw;
    }

    Database    db_;
    DataHolder  dh_;
};
```

Спецификация исключений

Устаревшая возможность C++, позволяющая указать список исключений, которые могут быть выброшены из функции.

```
void foo() throw(std::logic_error) {
    if (...) throw std::logic_error();
    if (...) throw std::runtime_error();
}
```

Если сработает второй `if`, то программа аварийно завершится.

```
void foo() {
    try {
        if (...) throw std::logic_error();
        if (...) throw std::runtime_error();
    } catch (std::logic_error & e) {
        throw e;
    } catch (...) {
        terminate();
    }
}
```

Ключевое слово `noexcept`

- Используется в двух значениях:
  - Спецификатор функции, которая не бросает исключение.
  - Оператор, проверяющий во время компиляции, что выражение специфицированно как небросающее исключение.
- Если функцию со спецификацией `noexcept` покинет исключение, то стек не обязательно будет свёрнут, перед тем как программа завершится.
  В отличие от аналогичной ситуации с `throw()`.
- Использование спецификации `noexcept` позволяет компилятору лучше оптимизировать код, т.к. не нужно заботиться о сворачивании стека.

Использование noexcept

```
void no_throw() noexcept;
void may_throw();

// копирующий конструктор noexcept
struct NoThrow { int m[100] = {}; };

// копирующий конструктор noexcept(false)
struct MayThrow { std::vector<int> v; };
```

```
MayThrow mt;
NoThrow  nt;

bool a = noexcept(may_throw());  // false
bool b = noexcept(no_throw());   // true

bool c = noexcept(MayThrow(mt)); // false
bool d = noexcept(NoThrow(nt));  // true
```

Условный noexcept

В спецификации noexcept можно использовать условные выражения времени компиляции.

```
template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N])
        noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
    void swap(pair & p)
        noexcept(noexcept(swap(first, p.first)) &&
                 noexcept(swap(second, p.second)))
    {
        swap(first, p.first);
        swap(second, p.second);
    }

    T1 first;
    T2 second;
};
```

Зависимость от noexcept

Проверка `noexcept` используется в стандартной библиотеке для обеспечения строгой гарантии безопасности исключений с помощью `std::move_if_noexcept` (например, `vector::push_back`).

```
struct Bad {
    Bad() {}
    Bad(Bad&&);       // может бросить
    Bad(const Bad&); // не важно
};
struct Good {
    Good() {}
    Good(Good&&) noexcept; // не бросает
    Good(const Good&) ;    // не важно
};
```

```
Good g1;
Bad  b1;
Good g2 = std::move_if_noexcept(g1); // move
Bad  b2 = std::move_if_noexcept(b1); // copy
```

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
состояние программы останется согласованным”.

Гарантии безопасности исключений

Гарантия отсутствия исключений

“Ни при каких обстоятельствах функция
не будет генерировать исключения”.

Базовая гарантия

“При возникновении любого исключения
состояние программы останется согласованным”.

Строгая гарантия

“Если при выполнении операции возникнет исключение,
то программа останется том же в состоянии,
которое было до начала выполнения операции”.

Строгая гарантия безопасности исключений

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

- Когда можно обеспечить строгую гарантию *эффективно*?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

- Когда можно обеспечить строгую гарантию *эффективно*?

  Это вопрос архитектуры приложения.

Как добиться строгой гарантии?

```
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_;
};
```

Как добиться строгой гарантии: вручную

```
template<class T>
struct Array
{
    void resize(size_t n) {
        T * ndata = new T[n];
        try {
            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_;
};
```

Как добиться строгой гарантии: RAII

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

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

        data_ = std::move(ndata);
        size_ = n;
    }

    unique_ptr<T[]> data_;
    size_t          size_;
};
```

Как добиться строгой гарантии: 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 unique_ptr

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

    unique_ptr<T> pop()
    {
%        unique_ptr<T> tmp(new T(data_.back()));
        data_.pop_back();
        return std::move(tmp);
    }

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

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.
- Там, где это возможно, старайтесь обеспечить строгую гарантию безопасности исключений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.
- Там, где это возможно, старайтесь обеспечить строгую гарантию безопасности исключений.
- Используйте `swap`, умные указатели и другие RAII объекты для обеспечения строгой безопасности исключений.