

Лекция 8. Исключения и обработка ошибок

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Причины ошибок времени выполнения.
- 2 Традиционные подходы: `exit`, `abort`, `assert`.
- 3 Функции с побочным эффектом (возврат признака ошибки).
- 4 Механизм исключений: `throw`, `try`, `catch`.
- 5 Перехват по типу, перехват всех исключений.
- 6 Вложенные `try/catch`.
- 7 Кратко о `noexcept`.
- 8 `std::expected` (C++23) — функциональная обработка ошибок.
- 9 Сравнение подходов.
- 10 Итоги.

Причины ошибок времени выполнения

Ошибки времени выполнения

Это ошибки, которые возникают во время работы программы, а не на этапе компиляции.

- **Некорректные входные данные:** деление на ноль, выход за границы массива, ввод неверного формата.
- **Некорректная работа с памятью:** разыменование нулевого указателя.
- **Некорректная работа с файлами:** попытка открыть несуществующий файл (эта тема будет позже).
- **Ошибки в логике:** непредвиденные состояния программы.

Важно

Ошибки времени выполнения могут проявляться не при каждом запуске, что затрудняет их поиск и отладку.

Последствия ошибок времени выполнения

- **Сбой программы:** аварийное завершение, потеря данных.
- **Непредсказуемое поведение:** программа продолжает работать, но выдаёт неверные результаты.
- **Уязвимости безопасности:** ошибки могут быть использованы злоумышленниками.
- **Потеря пользовательских данных:** если программа работает с файлами или базами данных.

Цель обработки ошибок

Сделать программу устойчивой к ошибкам: либо корректно завершить работу, либо восстановиться и продолжить выполнение.

Традиционные подходы к обработке ошибок в C

1 Завершение программы

```
#include <cstdlib>
if (error) exit(1);    // завершение программы с передачей кода завершения
if (error) abort();    // аварийное завершение программы
```

2 Выдача диагностики:

```
#include <cassert>
assert(условие); // если условие ложно - аварийное завершение
```

3 Возврат признака ошибки:

```
int divide(int a, int b, int* result) {
    if (b == 0) return -1; // код ошибки
    *result = a / b;
    return 0; // успех
}
```

- **exit()/abort():**

- `exit` — завершение программы с передачей кода завершения.
- `abort` — аварийное завершение программы.
- Не дают информации о причине ошибки.
- Могут привести к потере данных.

- **assert():**

- Проверка выполняется, если не определён `NDEBUG`.
- Также аварийно завершает программу.
- Предназначен для проверки предположений программиста.

- **Коды ошибок:**

- Загромождают код проверками.
- Можно случайно игнорировать возвращаемое значение.
- Сложно передавать информацию об ошибке.

Функция с побочным эффектом

- Результат записывается через параметр-ссылку, а функция возвращает признак успеха.
- Функция возвращает `bool` — признак успеха.

```
bool divide(int a, int b, int& result) {  
    if (b == 0) return false; // ошибка  
    result = a / b;  
    return true; // успех  
}  
  
int main() {  
    int res;  
    if (divide(10, 2, res)) {  
        cout << res;  
    } else {  
        cout << "Division by zero!";  
    }  
    return 0;  
}
```

Исключения (exceptions)

Это механизм, позволяющий прервать выполнение функции при возникновении ошибки и передать управление в другую часть программы вместе с информацией об ошибке.

- Код обработки ошибок отделён от основного кода.
- Можно передать подробную информацию об ошибке.
- Нельзя случайно проигнорировать ошибку.
- Автоматически раскручивает стек (уничтожаются локальные объекты).

Три ключевых оператора

Оператор `throw`

Генерирует исключение. Может «бросить» объект любого типа.

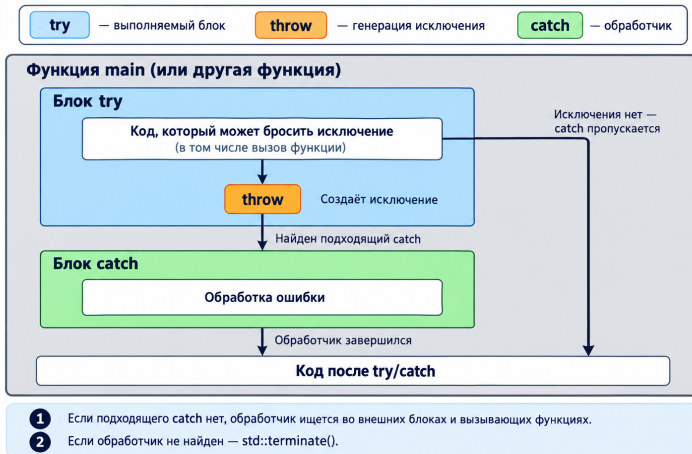
Блок `try`

Блок, к которому относятся следующие за ним обработчики исключений.

Блок `catch`

Обработчик подходящего исключения, возникшего при выполнении блока `try` или вызванных из него функций.

Как работают try, throw и catch



Генерация исключения (throw)

```
int divide(int a, int b) {  
    if (b == 0) {  
        throw "Division by zero!"; // бросаем строку (const char*)  
    }  
    return a / b;  
}
```

- throw создаёт объект исключения.
- Может завершить выполнение функции, даже если тип объекта не соответствует типу возврата функции.
- Можно бросать объекты разных типов: int, double, string, const char*, классы.

Блок try

```
try {  
    int result = divide(10, 0); // Код, в котором может возникнуть исключение  
    cout << "Result: " << result;  
}  
// catch на следующем слайде
```

- try — ключевое слово.
- Помещает код в «защищённый» блок.
- Если внутри бросается исключение, управление передаётся в блок catch. Если throw не происходит, идет по блоку try дальше.

Блок catch

```
try {  
    int result = divide(10, 0);  
    cout << "Result: " << result;  
}  
catch (const char* error) {  
    cout << "Error: " << error;  
}
```

- catch — обработчик исключения.
- Должен идти сразу после блока try.
- В скобках указывается тип перехватываемого исключения.
- Можно иметь несколько catch для разных типов.

Пример программы с исключением

```
#include <iostream>
using namespace std;

int divide(int a, int b) {
    if (b == 0) throw "Division by zero!";
    return a / b;
}

int main() {
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;
    try {
        int result = divide(a, b);
        cout << "Result: " << result << endl;
    }
    catch (const char* error) {
        cout << "Error: " << error << endl;
    }
    return 0;
}
```

Правильный способ

Перехватывайте исключения классовых типов по ссылке, чтобы избежать копирования и сохранить полиморфизм (тема весеннего семестра, любопытным — почитать).

```
catch (const std::exception& e) {  
    cout << "Error: " << e.what();  
}
```

- Избегает лишнего копирования.
- Позволяет перехватывать иерархию исключений (будет позже).
- `what()` — метод, возвращающий строку с описанием ошибки.

Перехват нескольких типов

```
try {  
    // код, который может бросить int или const char*  
}  
catch (int errorCode) {  
    cout << "Error code: " << errorCode;  
}  
catch (const char* errorMsg) {  
    cout << "Error message: " << errorMsg;  
}
```

- Порядок catch важен: сначала более конкретные типы, затем общие (при наличии иерархии классов).
- Можно перехватывать разные типы в разных обработчиках.

Перехват всех исключений

```
try {  
    // код  
}  
catch (const char* error) {  
    cout << "String error: " << error;  
}  
catch (int error) {  
    cout << "Int error: " << error;  
}  
catch (...) {  
    cout << "Unknown error!";  
}
```

- `catch (...)` перехватывает любое исключение.
- Должен быть последним в цепочке `catch`.
- Не даёт информации об ошибке, но позволяет перехватить исключение любого типа, дошедшее до этого обработчика.

Вопрос: что выведет этот код?

```
try {  
    throw 42;  
}  
catch (const char* e) {  
    cout << "char*: " << e;  
}  
catch (int e) {  
    cout << "int: " << e;  
}  
catch (...) {  
    cout << "unknown";  
}
```

Вопрос: что выведет этот код?

```
try {  
    throw 42;  
}  
catch (const char* e) {  
    cout << "char*: " << e;  
}  
catch (int e) {  
    cout << "int: " << e;  
}  
catch (...) {  
    cout << "unknown";  
}
```

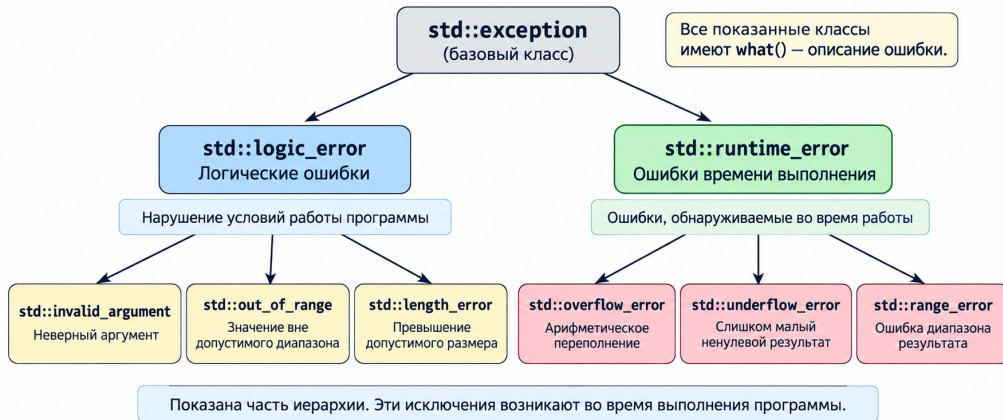
Ответ: int: 42 — исключение типа int перехвачено соответствующим catch.

Вложенные try/catch

```
try {  
    try {  
        throw "Inner error";  
    }  
    catch (const char* e) {  
        cout << "Inner catch: " << e << endl;  
        throw; // повторная генерация  
    }  
}  
catch (const char* e) {  
    cout << "Outer catch: " << e << endl;  
}
```

- Исключение может быть обработано на разных уровнях.
- `throw;` без аргумента повторно генерирует текущее исключение.

Иерархия стандартных исключений



Класс	Назначение
<code>std::exception</code>	Базовый класс стандартных классов исключений
<code>std::logic_error</code>	Логические ошибки в программе
<code>std::invalid_argument</code>	Некорректный аргумент функции
<code>std::out_of_range</code>	Значение вне допустимого диапазона
<code>std::runtime_error</code>	Ошибки времени выполнения
<code>std::overflow_error</code>	Арифметическое переполнение

- Все эти классы имеют метод `what()`, возвращающий строку с описанием ошибки.
- Покажу на примерах, но без углубления в ООП (будет дальше).

Пример с `std::out_of_range`

```
#include <iostream>
#include <stdexcept>
int getElement(int arr[], int size, int index) {
    if (index < 0 || index >= size) {
        throw std::out_of_range("Index out of range");
    }
    return arr[index];
}
int main() {
    int arr[] = {1, 2, 3};
    try {
        std::cout << getElement(arr, 3, 5);
    }
    catch (const std::out_of_range& e) {
        std::cout << "Error: " << e.what();
    }
    return 0;
}
```

Почему свои исключения?

Чтобы различать ошибки разных типов и передавать дополнительную информацию.

```
#include <stdexcept>
#include <string>

class DivisionByZeroError {
public:
    DivisionByZeroError(const std::string& msg) : message(msg) {}
    const char* what() const { return message.c_str(); }
private:
    std::string message;
};
```

- Свой класс может хранить любую информацию: код ошибки, имя файла, строку с описанием.
- Не обязательно наследовать от `std::exception` (хорошая практика, но мы пока не изучали классы и наследование).

Что такое noexcept?

Ключевое слово, которое означает, что исключение не должно выходить за пределы функции. Если это происходит, вызывается `std::terminate()`.

```
void safeFunction() noexcept {  
    // исключения не должны выходить за пределы функции  
}
```

- Используется для оптимизации и как обещание разработчика.
- Внутри такой функции можно бросить исключение и **перехватить его там же**. Это допустимо.
- В стандартной библиотеке многие функции помечены `noexcept`.

Зачем нужен noexcept?

- **Оптимизация:** компилятор может не генерировать код для раскрутки стека.
- **Безопасность контейнеров:** помогает разработчику избегать ошибок.
- **Деструкторы:** a few moments later...
- **Правило:** если функция не должна бросать исключения, помечайте её noexcept.

Важно

noexcept — это обещание. Если исключение может выйти из функции, не объявляйте её noexcept.

- **Производительность:** исключения могут быть дорогими (раскрутка стека).
- **Управление потоком:** исключения сложно использовать в некоторых парадигмах.
- **Не всегда нужны:** для ожидаемых неуспешных результатов часто удобнее возвращать признак ошибки.
- **Совместимость:** не все библиотеки используют исключения.

Альтернатива

В C++23 появился `std::expected` — тип, который явно хранит либо значение, либо ошибку.

Определение

`std::expected<T, E>` — тип, который может содержать либо значение типа `T`, либо ошибку типа `E`.

- Похож на `std::optional`, но хранит информацию об ошибке.
- Позволяет сообщать об ошибке через возвращаемый результат, без `throw`.
- Явно показывает, что функция может завершиться ошибкой.
- Перед обращением к значению или ошибке нужно проверить состояние результата.

Это альтернатива исключениям для функционального стиля.

Синтаксис `std::expected`

```
#include <expected>

// Для ошибки используем const char*
std::expected<int, const char*> divide(int a, int b) {
    if (b == 0) {
        return std::unexpected("Division by zero!");
    }
    return a / b;
}
```

- `std::expected<T, E>` — шаблон с двумя параметрами.
- `std::unexpected` — обёртка для ошибки.
- Можно возвращать значение или ошибку.

Использование `std::expected`

```
int main() {  
    auto result = divide(10, 0);  
    if (result) {  
        cout << "Result: " << *result;  
    } else {  
        cout << "Error: " << result.error();  
    }  
    return 0;  
}
```

- `result` преобразуется к `bool` — `true`, если есть значение.
- `*result` — доступ к значению (если есть).
- `result.error()` — доступ к ошибке (если есть).
- Можно использовать `result.value()` и `result.value_or(default)`.
- `value()` возвращает значение, а если сохранена ошибка — бросает `std::bad_expected_access`.
- `value_or(...)` возвращает значение либо указанную замену.

Сравнение подходов к обработке ошибок

Характеристика	Коды ошибок	Исключения	<code>std::expected</code>
Передача ошибки	Возвращаемый код	Брошенный объект	Значение ошибки внутри результата
Передача выше по вызовам	Явные проверки и возвраты	Поиск подходящего обработчика	Явные проверки и возвраты
Если ошибку не обработали	Может остаться незамеченной	При отсутствии обработчика — <code>std::terminate</code>	Результат можно проигнорировать

- **Коды ошибок:** просто, но легко пропустить проверку.
- **Исключения:** удобно, но дорого и сложно в некоторых случаях.
- `std::expected`: баланс между явностью и удобством.

- Изучили причины ошибок времени выполнения и способы их обработки.
- Рассмотрели традиционные подходы: `exit`, `abort`, `assert`, возврат кодов ошибок.
- Рассмотрели механизм исключений: `throw`, `try`, `catch`.
- Познакомились с иерархией стандартных исключений (обзорно).
- Увидели потенциал в создании своих исключений.
- Научились перехватывать исключения разных типов и все исключения.
- Узнали о `noexcept` как обещание не выпускать исключения за пределы функции.
- Познакомились с современным подходом `std::expected` (C++23).
- Научились выбирать подход в зависимости от ситуации.

Что дальше?

- Следующая лекция: Динамическая память (массивы, матрицы).
- На практике: Обработка исключений. Тестирование перехвата и генерации исключений.

Вопросы?