Обработка ошибок

Постановка задачи

Дана функция f. Необходимо выбрать способ сообщения клиентскому коду об успешности выполнения операции.

Конкретные примеры функции f будем выбирать по ситуации.

Возвращаемое значение

1. Специальное значение (вне области определения величины)
2. Флаг или код ошибки
3. Прогресс операции

Специальное значение

```
enum class Weekday {
  Monday, Tuesday, ..., Unknown = -1 /*or default 7*/
};

Weekday str_to_weekday(std::string_view weekday_str);

//! \return empty string_view for Weekday::Unknown
std::string_view weekday_to_str(Weekday weekday);
```

Специальное значение

✓ Примеримо, если область допустимых значений величины ограничена

✗ Теряется причина ошибки

Флаг или код ошибки

Рассмотрим функцию `str_to_int`:

- на входе строка
- на выходе число или ошибка

Предложите варианты сигнатуры такой функции

Без диагностик

```
int str_to_int(std::string_view str);
```

✓ Можно, если гарантируется корректность входных данных

✗ Непригодно для парсинга данных из внешних источников

Пример: `std::atoi`

Неразличимые ситуации

```
for (const auto str :
     {"", "0", "a", "0x1", "0a", " "}) {
  std::cout << std::atoi(str) << ' ';
}

// Output: 0 0 0 0 0 0
```


https://godbolt.org/z/sbohfv8Pz

Вернуть текущее состояние

```
int str_to_int(const char* str, char** str_end);
```

- ✓ Сохранение состояния парсера, делегирование решения клиенту
- ✗ Сложная сигнатура
- ✗ Специальное значение при переполнении и установка глобальной **errno**

Пример: `std::strtol`

Пример

```
char* end;
const auto value = std::strtol(str, &end, 10);

if (str != end) {
  // Ok? Распарсили префикс
}

if (str + strlen(str) != end) {
  // Распарсили не всю строку
}
```

У вас есть все данные

Use it wisely

https://godbolt.org/z/4x6f81Tqd

В чем проблема errno?

В чем проблема errno?

```
print_parse_result("99999999999999999");
// Out of range
print_parse_result("10");
// Out of range
```


https://godbolt.org/z/WoTj3Taaf

Статус в out-параметр

```
int str_to_int(std::string_view str,
               bool* ok = nullptr);
```

✗ Теряется точка возникновения ошибки («10000»)

✗ Легко проигнорировать out-параметр

Пример:

```
int QString::toInt(bool *ok = nullptr, int base = 10)
```

Результат в out-параметр

```
bool str_to_int(std::string_view str, int& result);
```

- ✓ Легко строить последовательность вызовов
- ✗ Теряется точка возникновения ошибки
- ✗ Легко проигнорировать возвращаемое значение
- ✗ Результат в out-параметре — может показаться несемантичным

Результат в out-параметр

```
bool str_to_int(std::string_view str, int& result);
```

- ✓ Легко строить последовательность вызовов
- ✗ Теряется точка возникновения ошибки
- ✗ Легко проигнорировать возвращаемое значение
- ✗ Результат в out-параметре — может показаться несемантичным

Результат в out-параметр

```
bool str_to_int(Parser& parser, int& result);
```

✓ Легко строить последовательность вызовов

~~✗ Теряется точка возникновения ошибки~~

✗ Легко проигнорировать возвращаемое значение

✗ Результат в out-параметре — может показаться несемантичным

Результат в out-параметр

```
[[nodiscard]]
bool str_to_int(Parser& parser, int& result);
```

✓ Легко строить последовательность вызовов

~~✗ Теряется точка возникновения ошибки~~

~~✗ Легко проигнорировать возвращаемое значение~~

✗ Результат в out-параметре — может показаться несемантичным

Результат в out-параметр

```
[[nodiscard]]
bool str_to_int(Parser& parser, int& result);
```

✓ Легко строить последовательность вызовов

~~✗ Теряется точка возникновения ошибки~~

~~✗ Легко проигнорировать возвращаемое значение~~

✗ Результат в out-параметре — может показаться несемантичным

Но это частный случай для парсеров, а нам интересен более общий

Часто встречается в linux/posix api

```
//! \return 1 on success,
//!         0 if the string is invalid
int inet_aton(const char *cp, struct in_addr *inp);

//! \return 0 on success
//!         on error, it returns an error number
int pthread_create(pthread_t *thread, ...);
```

std::optional<T>

```
//! \return value on success
//!         std::nullopt on error
std::optional<int> str_to_int(std::string_view str);
```

✗ Все еще нет сообщения об ошибке

❗ `std::optional` — это отсутствие значения.
Отнесистесь к семантике внимательно.

Проблемы out- и возвращаемых значений

1. С out-параметрами объект создается в неконсистентном состоянии (инвариант устанавливает фабричная функция)
2. В ряде случаев возвращаемых значений и out-параметров нет
3. Код логики приложения смешивается с кодом обработки ошибок
4. Сложности с освобождением ресурсов

Поздняя установка инварианта

```
pthread_t thread;

int err = pthread_create(&thread, ...);

if (err == 0) {
  // handle error
}

// use thread
```

Только сейчас можно использовать thread

Конструкторы и операторы

```
class Rational {
 public:
  // У конструкторов нет возвращаемого значения
  Rational(int n, int d) { … }
};

// Сигнатура операторов фиксирована,
// нет out-параметров
Rational operator/(Rational lhs, Rational rhs);
```

Если без исключений…

```
class MyVector {
 public:
  ...
  void push_back(...);
  bool is_valid() const;
};
```

Проблемы такого подхода?

Если без исключений…

```
MyVector v;
if (v.is_valid()) { ... }
v.push_back(...);
if (v.is_valid()) { ... }
v.push_back(...);
if (v.is_valid()) { ... }
v.push_back(...);
```

Тип сложно использовать

Структура кода

```
if (fd = open(...) < 0) {
  // handle errors
}

if (mmap(...) < 0) {
  // handle errors
}
```

Освобождение ресурсов

```
foo = malloc(sizeof(Foo));
foo->bar = malloc(sizeof(Bar));
if (foo->bar == NULL) free(foo);

baz = malloc(sizeof(Baz));
if (baz == NULL) {
  free(foo->bar);
  free(foo);
}

free(baz)
```

Освобождение ресурсов

```
foo = malloc(sizeof(Foo));
foo->bar = malloc(sizeof(Bar));
if (foo->bar == NULL) goto err_free_foo;

baz = malloc(sizeof(Baz));
if (baz == NULL) goto err_free_bar;

err_free_baz:
  free(baz);
err_free_bar:
  free(foo->bar);
err_free_foo:
  free(foo);
```

Linux kernel coding style

Проблемы out- и возвращаемых значений

1. С out-параметрами объект создается в неконсистентном состоянии (инвариант устанавливает фабричная функция)
2. В ряде случаев возвращаемых значений и out-параметров нет
3. Код логики приложения смешивается с кодом обработки ошибок

Исключения решают все эти проблемы
(и создают новые, только тссс...)

Исключения: простейший пример

```
double divide(double a, double b) {
  if (b == 0) throw std::domain_error("...");
  return a / b;
}

try {
  const auto result = divide(1, 0);
} catch (const std::domain_error& e) {
  std::cerr << e.what() << '\n';
}
```


https://godbolt.org/z/9fW3W9Kco

Исключения: простейший пример

```
void f() {
  const auto result = divide(1, 0);
  // ...
}
```

f() — нейтральна к исключениям

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


https://godbolt.org/z/rjE5f9jbf

Предварительные выводы

Коды возврата требуют локальной обработки.

Исключения — нелокальный механизм обработки ошибок.

Некоторые правила

```
try { throw 42; } catch (long) { }        // Не поймали
try { throw 42; } catch (int) { }         // Ok
try { throw 42; } catch (const int&) { }  // Ok
try { throw 42; } catch (int&) { }        // Ok
```

Выбор обработчика достаточно строг к типу

Некоторые правила

```
enum ErrCode { Overflow = 0 };

try {
  throw Overflow;
} catch (int) {
  std::cout << "catch (int)\n";      // Не поймали
} catch (ErrCode) {
  std::cout << "catch (ErrCode)\n"; // Ok
}
```

Некоторые правила

```
struct BaseError {};

struct DerivedError : public BaseError {};


try {
  throw DerivedError();
} catch (const BaseError&) {
  std::cout << "catch (const BaseError&)\n"; // Ok
}
```

Резюме

1. При выборе обработчика исключений множество допустимых преобразований сильно ограничено.
2. Наиболее часто используемое разрешенное преобразование — от дочернего к базовому типу.

Механика выброса исключения

1. Поиск обработчика по стеку
   Выбирается первый подходящий обработчик.
2. Если обработчик не найден — std::terminate
3. Если обработчик найден — раскрутка стека
   В каждом стекфрейме вызов деструкторов локальных объектов
4. Если во время раскрутки стека генерируется новое исключение — std::terminate.

Раскрутка стека: happy case

```
void g() { throw std::runtime_error("error"); }

void f() { Tracer t; g(); }

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

Что будет выведено?

https://godbolt.org/z/4354oxMc5

Кабы не было RAII…

```
void f() {
  auto* arr = new int[10];
  g();                          // May throw
  delete[] arr;                 // Oops…
}
```

Раскрутка стека: bad case

```
void g() { throw std::runtime_error("error"); }

void f() {
  Tracer t;
  g();
}

int main() {
  f();
}
```

Что будет выведено?

https://godbolt.org/z/1Kc5Y8zhn

Перехват всех исключений

```
// Есть соблазн написать так:
int main() {
  try {
    f();
  } catch (...) {
    // ???
  }
}
```

Не рекомендуется

CppCoreGuidelines: E.17

Исключения в конструкторе

```
class Widget {
 public:
  Widget() {
    std::cout << "Widget()\n";
    throw std::runtime_error("Error text");
  }
  ~Widget() { std::cout << "~Widget()\n"; }
 private:
  Tracer t_;
};
```


https://godbolt.org/z/cs5axTYj1

Function try block

```
void g() {
  throw std::runtime_error("runtime error");
}

void f() try {
  g();
} catch (const std::exception& e) {
  std::cout << e.what() << '\n';
}
```


https://godbolt.org/z/Wsxonv811

Исключения в списке инициализации

```
class Widget {
 public:
  Widget() try /*: text_()*/ {
    std::cout << "Widget()\n";
  } catch (const std::exception& ex) {
    std::cout << ex.what() << '\n';
  }
 private:
  Text text_;
};
```


https://godbolt.org/z/19j7Mf7va

Исключения в деструкторе

```
struct Widget {
  Widget() { std::cout << "Widget()\n"; }

  ~Widget() {
    std::cout << "~Widget()\n";
    throw std::runtime_error("Error");
  }
};
```

Исключения в деструкторе

```
try {
  Widget w1, w2;
} catch (const std::exception& ex) {
  std::cout << ex.what() << "\n";
}
```

Вывод?

Исключения в деструкторе

```
try {
  Widget w1, w2;
} catch (const std::exception& ex) {
  std::cout << ex.what() << "\n";
}
```

Terminate

https://godbolt.org/z/GczjEKnT9

Исключения в деструкторе

```
try {
  Widget w1;
} catch (const std::exception& ex) {
  std::cout << ex.what() << "\n";
}
```

Вывод?

https://godbolt.org/z/GczjEKnT9

Исключения в деструкторе

C++03:

Widget()
~Widget()
Error

https://godbolt.org/z/j7dbE18sz

C++11:

Terminate

https://godbolt.org/z/YThnTKexj

Деструкторы и noexcept

```
struct Widget {
  Widget() { ... }

  ~Widget() noexcept { ... }
};
```

Деструкторы по умолчанию noexcept.
Явно можно не писать.

Исключения стандартной библиотеки

exception

- **logic_error**
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error
- **runtime_error**
  - range_error
  - …
- bad_typeid
- bad_cast
- …

Базовые классы для ваших исключений

Что будет выведено?

```
try {
  throw std::runtime_error("runtime error");
} catch (const std::exception&) {
  std::cout << "std::exception\n";
} catch (const std::runtime_error&) {
  std::cout << "std::runtime_error\n";
}
```


https://godbolt.org/z/4fnz9Y5Ko

Пользовательские исключения

```
class DivisionByZeroError : public std::runtime_error
{
 public:
  DivisionByZeroError(const std::string& message)
    : std::runtime_error(message) {}

  DivisionByZeroError(const char* message)
    : std::runtime_error(message) {}
};
```

Пользовательские исключения

```
class DivisionByZeroError : public std::runtime_error
{
 public:
  using std::runtime_error::runtime_error;
};
```

См. Inheriting Constructors в Using Declaration

Рекомендация

Use purpose-designed user-defined types as exceptions
(not built-in types)

```
throw 7;                    // bad
throw "something bad";      // bad
throw std::exception{};     // bad - no info
```


CppCoreGuidelines: E.14

Summary

Плюсы исключений:

- Код логики приложения отделен от кода обработки ошибок
- Исключения невозможно игнорировать

Минусы

- Появляются неожиданные нелокальные переходы
- Накладные расходы на исключения

std::expected<T, E>

std::expected — сумма типов, где

T — тип результата

E — тип ошибки

Инвариант: экземпляр std::expected не пустой.

Пример использования

```
std::expected<double, std::string>
divide(double a, double b) {
  if (b == 0) {
    return std::unexpected("Division by zero");
  }
  return a / b;
}
```

Пример использования

```
const auto result = divide(1, 0);
if (result) { // if (result.has_value())
  std::cout << *result << '\n';
} else {
  std::cout << result.error() << '\n';
}
```

Пример использования

```
const auto result = divide(1, 0);
const auto x = result.value_or(0);
std::cout << x << '\n';
```

Пример использования

```
try {
  const auto result = divide(1, 0);
  std::cout << result.value() << '\n';
} catch (
    const std::bad_expected_access<std::string>& e) {
  std::cout << e.error() << '\n';
}
```


https://godbolt.org/z/TKczE7f4f

Summary

`result.has_value()` — проверяет наличие значения

`*result` — возвращает значение или поведение неопределено

`result.value()` — возвращает значение или бросает исключение

Что вы думаете о таком подходе?

Summary

✓ Разработчик библиотеки делегирует клиенту способ обработки ошибок

✗ При использовании метода `.value()` точка возникновения ошибки и генерации исключения могут быть далеко друг от друга

Q&A