

Написание кода на C++

Дима Трушин

2024 — 2025

Содержание

1	Обработка ошибок	2
1.1	Виды ошибок и доступные средства	2
1.2	Доступные инструменты	4
1.3	Информация об ошибке	5
1.4	Ошибки программиста	8
1.4.1	<code>assert</code>	8
1.4.2	Кастомные <code>assert</code> -ы	11
1.4.3	Exceptions	13
1.5	Ошибки исполнения среды	13
1.5.1	Error Codes	14
1.5.2	<code>std::expected</code> и <code>std::optional</code>	15
1.5.3	Exceptions	17
1.5.4	Гарантии поведения при исключениях	19
1.5.5	Технические детали и логическая модель для исключений	20
1.5.6	Важные примеры и ожидаемое поведение для исключений	22
1.5.7	«Идеальная» имплементация	24
1.5.8	Рекомендации	27
1.5.9	Проблемы при использовании исключений	29

1 Обработка ошибок

Очень рекомендую посмотреть доклад Peter Muldoon [Exceptions in C++: Better Design Through Analysis of Real World Usage](#). Значительная часть этого раздела была написана под влиянием этого доклада.

1.1 Виды ошибок и доступные средства

Виды ошибок

1. **Ошибка программиста.** Это bug-и программы, которых в коде быть не должно. Такие ошибки надо исправлять.
2. **Ошибка при обращении к ресурсам среды исполнения.** Например вы пытаетесь выделить память, получить доступ на чтение или запись файла, получаете доступ на прослушивание клавиатуры, получаете доступ к вычислению на видеокарте и так далее. Бывают разные причины, почему такие операции могут быть не успешными. В любом случае это ошибки, которые могут встретиться в программе и которые не связаны с некорректностью кода.

Обработка ошибок Теперь давайте обсудим какие есть механизмы в языке для работы с ошибками. Существуют следующие механизмы:

1. Макрос `assert`. Технически `assert` это макрос, которые ведет себя следующим образом:
 - Если определен макрос `NDEBUG`¹, то он не дает никакого кода.
 - Если не определен макрос `NDEBUG`, он делает проверку условия внутри макроса и если оно ложно, то валит программу с сообщением об ошибке и указанием строчки кода, на которой все упало.

Вот как выглядит его использование

```
1 int f(int* x) {
2     assert(x != nullptr && "The argument must not be nullptr!");
3     return *x;
4 }
5
6 int main() {
7     int x = 1;
8     assert(x == 1 && "x must be 1!");
9     return 0;
10 }
```

2. Конструкция `static_assert`. Это по сути то же самое, что и просто `assert`, только данная конструкция выполняет проверки во время компиляции, а не во время исполнения. Все проверки условий известных на этапе компиляции должны выполняться этой функцией, а не просто `assert`-ом. Вот как выглядит его использование

```
1 template<class T>
2 class A {
3     static_assert(std::is_arithmetic_v<T>, "The type T must be arithmetic!");
4     ...
5 };
```

Строчка в начале класса будет проверяться при компиляции для каждого типа `T`. И если тип `T` не является арифметическим, то компиляция будет падать с ошибкой, где будет указана строчка, где она произошла, и сообщение.

3. Error codes или коды ошибок. Данный подход часто применяется в Си для обработки ошибок. Суть его в том, что мы используем возвращаемое значение функции для указания статуса операции, а для возвращения данных используем `out` аргументы (см. ??). Вот как это выглядит

¹Компилятор от Microsoft автоматически определяет этот макрос в релизе, а gcc и clang нет. Бывает полезно проверить условия в релизе, потому что пути оптимизации компилятора неисповедимы. Тогда в msvc вам надо убрать при компиляции в релизе объявление макроса `NDEBUG`.

```

1  enum class Status {
2      OK, Error
3  };
4
5  Status algorithm(const Data& input, Result* output);
6
7  int main() {
8      Data data = ...;
9      Result result;
10     Status status = algorithm(data, &result);
11     if (status == Status::OK) {
12         ...
13     } else {
14         ...
15     }
16 }

```

4. `std::expected`. Начиная с C++23 вам наконец-то доступна возможность возвращать из функции желаемое значение или ошибку. По сути это `std::variant<ValueType, ErrorType>` но только с приятным интерфейсом. Суть его в том, чтобы использовать коды ошибок, но при этом и ошибку и результат возвращать через `return`. В коде это выглядит как-то так

```

1  class Error {
2      ...
3  };
4
5  std::expected<Result, Error> algorithm(const Data& data);
6
7  int main() {
8      Data data = ... ;
9      auto result = algorithm(data);
10     if (result.has_value()) {
11         Result r = std::move(result).value();
12         ...
13     } else {
14         ...
15     }
16 }

```

5. Исключения – Exceptions. Это специальная фишка языка, которая позволяет генерировать исключения с помощью ключевого слова `throw`. Перехватывать их можно внутри блока `try` и реагировать на них внутри блока `catch`. Исключения являются опасным механизмом языка, потому что они могут выйти не только из текущего scope, но и в целом подниматься по стеку до тех пор, пока их не поймают или пока исключение не улетит в операционную систему. Из-за исключений вы должны всегда думать, что вы можете выйти из любого scope в любой точке. Из-за этого весь менеджмент ресурсов должен быть автоматизирован с помощью RAII (см. раздел ??). В коде это выглядит как-то так

```

1  Result algorithm(const Data& data) {
2      ...
3      if (/*some error*/)
4          throw std::runtime_error(/*error information*/);
5      ...
6      return result;
7  }
8
9  int main() {
10     Data data = ... ;
11     Result result;
12     try {
13         result = algorithm(data);
14     } catch (std::exception& e) {
15         /*handle known exceptions*/
16     } catch (...) {
17         /*handle unknown exceptions*/
18     }
19     return 0;
20 }

```

Методы выше можно сгруппировать так:

1. Задача `assert` и `static_assert` уронить программу на первой же ошибке. Программа не должна продолжать работу в некорректном состоянии. Предполагается, что вы как раз ищете ошибку в коде и это должно помочь ее найти. Эти методы для дебага.
2. Все остальные методы: коды ошибок, `std::expected` и исключения, предполагают, что вы хотите как-то отреагировать на ошибку при исполнении. Самое главное, что вы хотите и можете продолжить исполнение дальше (возможно вы все равно хотите выйти по `terminate`, но у вас есть возможность не падать). Эти методы предполагают штатное исполнение программы.

Давайте оценим методы по информативности.

1. `assert` и `static_assert` по умолчанию сообщают вам информацию о строчке кода, на которой произошла ошибка. У вас так же есть возможность сообщить любую другую информацию (хотя у `assert` добавление информации делается костылями). По умолчанию, нет информации о стеке вызовов для `assert` (для `static_assert` это не имеет смысла, так как она вызывается при компиляции, а не при исполнении). Однако начиная с C++23 появилась возможность использовать `std::stacktrace`. Он, правда, не настолько хорошо отображает информацию, как привычные для нас компиляторы.
2. Коды ошибок и `std::expected`. Так как и коды ошибок и тип ошибок внутри `std::expected` пишете вы сами, то тут нет информации, которая будет доступна по умолчанию. Получить информацию о стеке вызовов вы также можете с помощью `std::stacktrace`. Начиная с C++20 информацию о строчке кода, где произошла ошибка можно получить с помощью `std::source_location`.
3. Исключения. По умолчанию класс для исключения пишете вы сами. Потому вы можете поместить в него все ту же самую информацию, которая доступна и другими методами, а именно: `std::stacktrace` и `std::source_location`. Однако, если вы используете дебагер, то при падении на исключении, которое не было поймано, вам дебагер сообщит информацию о стеке вызовов и текущее состояние вашей программы в памяти. Это может быть удобно, но это по сути фишка дебагера, а не возможности исключений.

Ожидаемое поведение Давайте поймем, какое поведение от программы мы ожидаем при возникновении ошибок. Можно выделить следующие случаи:

1. Программа падает с сообщением об ошибке.
2. Программа сообщает об ошибке, корректно ее обрабатывает, продолжает находиться в корректном состоянии и корректно работает дальше.

Первое поведение хорошо подходит для разработки приложения, потому что вы хотите найти ошибку как можно раньше, упасть как можно ближе к ней, чтобы найти ее и исправить. Нет необходимости работать дальше. Второе же поведение желательно на машине пользователя. Какой бы ни была ошибка, пользователь не ждет, что программа упадет. В худшем случае он ждет информации, что что-то пошло не так. В связи с этим `assert` и `static_assert` подходят только для первого поведения и совсем не подходят для второго. Их суть ровно в том, чтобы уронить программу как можно раньше. В свою очередь коды ошибок, `std::expected` и исключения технически могут быть использованы для обеих ситуаций. Это может быть не очень разумно, но никто вам не мешает это сделать.

1.2 Доступные инструменты

Давайте поговорим об инструментах, которые позволяют искать ошибки программиста. Тут обычно вспоминают два:

1. Дебагер. Это специальная программа, которая позволяет следить за выполнением программы. При этом вам обычно доступна следующая информация
 - (a) Текущий стек вызовов
 - (b) Текущие переменные и присвоенные им значения. При этом доступна информация о типе переменной и ее адрес (если это класс, то какие поля у переменной и их значения).
 - (c) Текущая команда в программе.

При этом дебагер позволяет поставить точку в коде на которой вы хотите остановить исполнение. Позволяет делать пошаговое исполнение с возможностью проваливаться внутрь функций или выполнять их как единое действие.

2. Санитайзеры. Сразу оговорюсь это все инструменты доступные под Linux и MacOS и не доступные под Windows. Санитайзеры – это специальные внешние инструменты, которые следят за исполнением программы и выявляют в ней ошибки. Например для проверки корректности работы с памятью, вся выделяемая память в вашей программе имеет специальные служебные блоки памяти вокруг каждого вызова `new`. И если ваша программа обращается к служебным блокам, значит вы не верно обратились к данным.

Хочу сказать несколько важных вещей по поводу этих инструментов. Давайте посмотрим на диаграмму экосистемы C++ еще разок (см. раздел ??). Обратите внимание, что дебагер и санитайзеры находятся в самом конце пайплайна по работе с кодом. То есть дебагер – это ваша последняя линия обороны. На этом этапе вы работаете напрямую с бинарным кодом. И для того, чтобы дебагер мог делать свою работу, вам приходится компилировать программу в специальном режиме, в котором становятся доступны имена и типы объектов, расположенных в памяти. Обычно убираются разные оптимизации, чтобы бинарный код лучше сопоставлялся с исходным кодом. Как бы то ни было, если вы пользуетесь только дебагером и санитайзерами – вы совершаете большую ошибку, потому что на уровне исходного кода у вас куда больше возможностей предотвратить свои ошибки и получить помощь со стороны компилятора.

1.3 Информация об ошибке

Информация об ошибке нужна для того, чтобы выявить и исправить проблему. Это может быть как ошибка на стороне программы, так и на стороне среды исполнения. В идеале следующая информация должна быть доступна:

1. Текстовое описание ошибки. Технически эту информацию можно хранить в `std::string`. Данная информация должна быть модифицируемой, чтобы вы могли добавлять информацию об ошибке, при передаче ошибки при раскрывании стека.
2. Строчка в коде, на которой произошла ошибка. Для получения этой информации с C++20 можно использовать `std::source_location`. Эта информация должна быть константной.
3. Состояние стека вызовов во время произошедшей ошибки. Для получения этой информации с C++23 можно использовать `std::stacktrace`. Эта информация должна быть константной.
4. Какая-то специфическая для этой ошибки информация. Это произвольный тип, который настраивается под каждую ошибку. Данная информация должна быть модифицируемой, чтобы вы могли добавлять информацию об ошибке, при передаче ошибки при раскрывании стека.

Независимо от метода, который вы используете и которые описаны ниже, независимо от того ошибка это программиста или ошибка исполнения среды, вам всегда достаточно информации выше. При бывает, что можно обойтись меньшим количеством информации, например, в простых `assert`-ах.

Как собрать информацию о строке кода Давайте начнем с простого примера.

```
1 #include <source_location>
2
3 void f() {
4     std::cout << std::source_location::current() << '\n';
5 }
6
7 int main() {
8     f();
9     return 0;
10 }
```

Где для вывода информации о точке кода делается с помощью следующего оператора:

```
1 std::ostream& operator<<(std::ostream& out, const std::source_location& location) {
2     out << location.file_name() << "(" << location.line() << ":"
3     << location.column() << ")", function '" << location.function_name()
```

```
4     << " ";
5     return out;
6 }
```

Вывод будет выглядеть как-то так

```
1 main.cpp(4:38), function 'void __cdecl f(void)'
```

То есть функция `current` вызвана в строке 4 и ее имя находится в столбце 38. А что если мы теперь хотим автоматизировать печать информации и завернуть ее во вспомогательную функцию. Тогда можно было бы попытаться написать так

```
1 #include <source_location>
2
3 void print_location() {
4     std::cout << std::source_location::current() << '\n';
5 }
6
7 void f() {
8     print_location();
9 }
10
11 int main() {
12     f();
13     return 0;
14 }
```

Но в этом случае будет показана информация о 4-ой строке, а не о 8-ой. Другой вариант передавать эту информацию явно:

```
1 #include <source_location>
2
3 void print_location(const std::source_location& location) {
4     std::cout << location << '\n';
5 }
6
7 void f() {
8     print_location(std::source_location::current());
9 }
10
11 int main() {
12     f();
13     return 0;
14 }
```

Но это ничуть не лучше, потому что все равно надо писать вызов функции `current`. Завернуть этот вызов можно только в макрос, иначе она покажет точку в функции обертке, а не где мы ее вызываем. Макрос – плохая идея, потому что вы вводите глобальное имя, которое может потенциально конфликтовать с чем угодно. А для избежания конфликта, вам все равно придется давать длинное имя макросу. В этом случае есть следующий трюк – аргументы по умолчанию.

```
1 #include <source_location>
2
3 void print_location(const std::source_location& location = std::source_location::current()) {
4     std::cout << location << '\n';
5 }
6
7 void f() {
8     print_location();
9 }
10
11 int main() {
12     f();
13     return 0;
14 }
```

Теперь вывод будет как надо. В этом случае в строке 8 при вызове функции `print_location` логически аргументы по умолчанию считаются в этой строке до вызова функции. Вывод будет:

```
1 main.cpp(8:3), function 'void __cdecl f(void)'
```

Как собрать информацию о состоянии стека В этом случае работают все те же правила и трюки, что использовались в случае информации о точке исполнения в коде. Единственное – нам надо научиться печатать информацию о стеке. Начнем опять с простого примера.

```
1 #include <stacktrace>
2
3 void f() {
4     std::cout << std::stacktrace::current();
5 }
6
7 int main() {
8     f();
9     return 0;
10 }
```

В этом случае оператор вывода для состояния стека определяется так:

```
1 std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
2     for (auto iter = backtrace.begin(); iter != (backtrace.end()); ++iter) {
3         out << iter->source_file() << "(" << iter->source_line()
4             << "):" << iter->description() << '\n';
5     }
6     return out;
7 }
```

Вывод будет выглядеть следующим образом:

```
1 PathToProject\main.cpp(5599):Project!f+0x29
2 PathToProject\main.cpp(5682):Project!main+0xB
3 ..\exe_common.inl(79):Project!invoke_main+0x39
4 ..\exe_common.inl(288):Project!__scrt_common_main_seh+0x132
5 ..\exe_common.inl(331):Project!__scrt_common_main+0xE
6 ..\exe_main.cpp(17):Project!mainCRTStartup+0xE
7 (0):KERNEL32!BaseThreadInitThunk+0x14
8 (0):ntdll!RtlUserThreadStart+0x21
```

Обратите внимание, что нижние 6 строк о состоянии стека – это вызов системных функций и лишь верхние две строчки показывают стек вызова для вашей программы. Если вы хотите убрать эти последние 6 строк, то можно сделать так:

```
1 std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
2     static constexpr int bound = 6;
3     if (backtrace.size() < bound)
4         return out;
5     for (auto iter = backtrace.begin(); iter != (backtrace.end() - bound); ++iter) {
6         out << iter->source_file() << "(" << iter->source_line()
7             << "):" << iter->description() << '\n';
8     }
9     return out;
10 }
```

Что превратит вывод в

```
1 PathToProject\main.cpp(5599):Project!f+0x29
2 PathToProject\main.cpp(5682):Project!main+0xB
```

Получилось сильно лучше. Если вы хотите автоматизировать сбор информации о стеке, то надо так же воспользоваться аргументами по умолчанию.

```
1 #include <stacktrace>
2
3 void print_stack(const std::stacktrace& stack = std::stacktrace::current()) {
4     std::cout << std::stacktrace::current();
5 }
6
7 void f() {
8     print_stack();
9 }
10
11 int main() {
12     f();
13 }
```

```
13     return 0;
14 }
```

1.4 Ошибки программиста

1.4.1 assert

Как использовать К этому макросу можно относиться как к автоматически комментируемым проверкам. Работает это так

```
1 void f(int* x) {
2     assert(x != nullptr);
3     *x = 42;
4 }
```

Если вы хотите проверить preconditions для некоторой функции `f`, которая ожидает не нулевой указатель, то можно поставить в начале тела функции проверку, что указатель `x` не нулевой. При компиляции с макросом `NDEBUG` вторая строка не даст никакого кода. А без этого макроса выполняется проверка условия внутри и если оно истинно, то исполнение переходит дальше к строке три. Если же оно ложно, то программа падает с сообщением на какой строке кода она упала. Сообщение об ошибке пишется в `stderr`. Чтобы руками добавить сообщение к `assert` можно писать в поток `std::cerr`.

По какой-то странной причине в отличие от `static_assert` макрос `assert` не поддерживает второй аргумент – сообщение об ошибке. Потому приходится прибегать к следующему трюку.

```
1 void f(int* x) {
2     assert(x != nullptr && "The argument x must not be nullptr!");
3     *x = 42;
4 }
```

В этом случае мы передаем адрес статической константы (который всегда не ноль) и он конвертируется к `true` и потом делается логическое «и» с проверяемым условием.

Информация о стеке вызовов Макрос `assert` печатает строку на которой он упал. Потому если строка содержит текстовую константу, то вы увидите ее содержимое. Например

```
1 #include <cassert>
2
3 int main() {
4     int x = 0;
5     assert(x != 0 && "x must not be 0!");
6     return 0;
7 }
```

Вывод будет:

```
1 Assertion failed: x != 0 && "x must not be 0!", file PathToProject\main.cpp, line 5
```

Если же вы хотите добавить неконстантную информацию, то вам нужно писать в `std::cerr` перед вызовом `assert`. Давайте покажем, как это можно сделать на примере добавления информации о стеке вызовов. Введем вспомогательную функцию

```
1 bool stack_state(const std::stacktrace& backtrace = std::stacktrace::current()) {
2     std::cerr << "Call Stack:\n" << backtrace << '\n';
3     return true;
4 }
```

Теперь можно написать следующее:

```
1 #include <cassert>
2
3 int main() {
4     int x = 0;
5     assert(stack_state() && x != 0 && "x must not be 0!");
6     return 0;
7 }
```


Обратите внимание, что вызов функции `stack_state` должен быть самым первым из-за ленивых вычислений в логических операторах. Если первое условие ложно в конъюнкции, то второе даже не вычисляется. Так же я размещаю вызов этой функции внутри `assert`, чтобы при отключении `assert`-ов, эта функция не вызывалась. В этом случае вывод будет:

```
1 Call Stack:
2 PathToProject\main.cpp(5):Project!main+0x25
3
4 Assertion failed: stack_state() && x != 0 && "x must not be 0!",
5 file PathToProject\main.cpp, line 5
```

Аналогично можно добавлять любую информацию к `assert`. Информацию о строке кода добавлять не надо, так как она и так выводится.

Рекомендации Мои рекомендации по использованию `assert` кратко выражаются так: чем больше, тем лучше. Чем ближе к багу программа упадет на ошибке, тем быстрее вы его найдете и поправите. С `assert`-ами вам скорее всего придется забыть про использование дебагера, так как не понадобится. Вот пара мыслей об использовании:

1. Ставьте `assert` в начале каждой функции, чтобы проверить preconditions. Например у оператора деления второй аргумент должен быть не ноль. Причем это не ошибка исполнения, это программист должен был проверить этот случай перед вызовом деления. То есть программист не предусмотрел валидный случай в программе, на котором падает текущее исполнение.

```
1 class Int {
2     ...
3 };
4
5 Int operator/(Int first, Int second) {
6     assert(second != 0 && "The second argument of operator/ must be non-zero!");
7     return first.value / second.value;
8 }
```

2. Ставьте `assert` в конце каждой мутирующей функции для проверки postcondition. Вдруг вы где-то ошиблись с имплементацией.

```
1 template<class T>
2 class Tree {
3     using Node = Node<T>;
4 public:
5     ...
6     void add_value(T value) {
7         /*place the value in the tree*/
8         assert(is_correct());
9     }
10 private:
11     bool is_correct() const:
12     Node* root_ = nullptr;
13 };
```

3. Все конструкторы класса должны проверять не только preconditions для аргументов, но и инварианты класса. Выделите в классе приватные функции по проверке инвариантов и заворачивайте их внутрь `assert`.

```
1 template<class T>
2 class Tree {
3     using Node = Node<T>;
4 public:
5     ...
6     void add_value(T value) {
7         /*place the value in the tree*/
8         assert(is_correct());
9     }
10 private:
11     bool is_correct() const {
12         return is_correct(root_);
13     }
```

```

13 }
14 static bool is_correct(Node* node) {
15     if (node == nullptr)
16         return true;
17     bool is_node_good = /*do checks for node*/;
18     return is_node_good && is_correct(node->left) && is_correct(node->right);
19 }
20 Node* root_ = nullptr;
21 };

```

4. Не пишите слишком длинные и сложные выражения внутри `assert`, вы иначе никогда не поймете, что вы проверяете. Выделите отдельные вспомогательные функции с говорящими названиями. Например

Плохо

```

void f(int x, int y, int z) {
    assert(x > y && y <= z + x &&
           (x & !y) == 1 && "Ups!");
    /*function code*/
}

```

Получше

```

void f(int x, int y, int z) {
    assert(is_condition_true(x, y, z) &&
           "Ups!");
    /*function code*/
}

```

Вы конечно можете добавить комментарий и сообщение об ошибке. Но все же имя функции скажет больше, чем комментарий. Только важно в функциях, которые вызываются внутри `assert` не делать проверок с помощью `assert`, а то вы можете уйти в бесконечную рекурсию вызовов `assert`.

5. Обязательно вставляйте `assert` перед обращением через указатель, он может быть нулевой.

```

1 void f(A* a, B* b) {
2     assert(a != nullptr && "a must not be nullptr!");
3     assert(b != nullptr && "b must not be nullptr!");
4     b->consume(a->produce());
5 }

```

Обратите внимание, что две разные проверки разнесены в разные `assert`-ы, чтобы при падении было понятно на какое именно условие вы упали.

Если в классе перегружен оператор `operator->()`, то соответствующую проверку можно выполнить внутри него. Однако, для указателей нельзя этот оператор перегрузить, а потому вам придется делать эту проверку заранее. В случае же класса и перегруженного оператора `operator->()` проверка может выглядеть так.

```

1 class A {
2 public:
3     ...
4     I* operator->() const {
5         assert(ptr_ != nullptr && "The object has invalid pointer!");
6         return ptr_;
7     }
8 private:
9     I* ptr_ = nullptr;
10 };

```

В данном примере мы можем спокойно звать

```

1 int main() {
2     A a;
3     a->f();
4     return 0;
5 }

```

Если в объекте `a` хранится нулевой указатель, то в третьей строчке все упадет на проверке по `assert`.

6. НЕ делайте `assert` на ошибки, которые разрешимы на этапе компиляции (например в шаблонной магии), для этого используйте `static_assert`. Следующий пример возможно не самый лучший, тут можно было бы воспользоваться концептами или `std::enable_if` (если вы динозвар из прошлого), но он хорошо иллюстрирует проблему.

Плохо

```
template<class T>
void f(T x) {
    assert(std::is_arithmetic_v<T> &&
           "The type must be arithmetic!");
    /*function implementation*/
}
```

Получше

```
template<class T>
void f(T x) {
    static_assert(std::is_arithmetic_v<T> &&
                  "The type must be arithmetic!");
    /*function implementation*/
}
```

7. И конечно же не забывайте сообщить об ошибке текстом или добавить информацию о стеке вызовов или любую другую полезную для дебага информацию. Вам же будет проще.

Плохо

```
#include <cassert>

int main() {
    int x = 1;
    assert(x == 1);
    return 0;
}
```

Получше

```
#include <cassert>

int main() {
    int x = 1;
    assert(x == 1 && "x must be 1!");
    return 0;
}
```

1.4.2 Кастомные assert-ы

Главный недостаток `assert` заключается в том, что макрос `NDEBUG` отключает сразу все проверки во всех компилируемых файлах. Если вам нужно выборочное отключение проверок, то можно сделать свои кастомные макросы на основе `assert` и управлять включением и отключением проверок с помощью дополнительных макросов. Такое может понадобиться, если вы захотите запустить часть недорогих проверок (как например проверка указателя на `nullptr` перед разыменованием) в релизе (который идет в прод) и отключить все дорогие проверки. Вот как могут выглядеть кастомные `assert`-ы.

```
1 // file custom_asserts.h
2 #include <cassert>
3
4 #ifndef DISABLE_ASSERTS
5 #define DISABLE_ASSERTS 3
6 #endif
7
8 #if DISABLE_ASSERTS < 0
9 #define ASSERT0(expr, msg) ((void)0)
10 #else
11 #define ASSERT0(expr, msg) assert(expr && msg)
12 #endif
13
14 #if DISABLE_ASSERTS < 1
15 #define ASSERT1(expr, msg) ((void)0)
16 #else
17 #define ASSERT1(expr, msg) assert(expr && msg)
18 #endif
19
20 #if DISABLE_ASSERTS < 2
21 #define ASSERT2(expr, msg) ((void)0)
22 #else
23 #define ASSERT2(expr, msg) assert(expr && msg)
24 #endif
25
26 #if DISABLE_ASSERTS < 3
27 #define ASSERT3(expr, msg) ((void)0)
28 #else
29 #define ASSERT3(expr, msg) assert(expr && msg)
30 #endif
31
```

В этом примере мы вводим 4 вида `assert`-ов:

• ASSERT0

• ASSERT1

• ASSERT2

• ASSERT3

Думать про них надо так: чем номер меньше, тем важнее `assert`. Макрос `DISABLE_ASSERTS` отключает `assert`-ы с уровнем выше заданного. То есть `-1` отключает все сообщения об ошибках, `0` оставляет только самые важные `ASSERT0` и так далее. Используется это так

```
1 #include "custom_asserts.h"
2
3 int main() {
4     int x = 0;
5     ASSERT0(x == 0, "x must be 0!");
6     int y = 1;
7     ASSERT1(y == 1, "y must be 1!");
8     int z = 2;
9     ASSERT2(z == 2, "z must be 2!");
10    return 0;
11 }
```

При необходимости отключить `assert`-ы после уровня `0` нужно сделать так:

```
1 #define DISABLE_ASSERTS 0
2 #include "custom_asserts.h"
3
4 int main() {
5     int x = 0;
6     ASSERT0(x == 0, "x must be 0!");
7     int y = 1;
8     ASSERT1(y == 1, "y must be 1!"); // is disabled
9     int z = 2;
10    ASSERT2(z == 2, "z must be 2!"); // is disabled
11    return 0;
12 }
```

Так же можно сделать `assert`-ы не по уровням, а по управляемому макросу. Например, вы хотите сделать проверки для конкретного класса или для конкретного модуля в коде.

```
1 // file my_assert.h
2 #include <cassert>
3
4 #ifndef DISALBE_MY_ASSERT
5 #define MY_ASSERT(expr, msg) assert(expr && msg)
6 #else
7 #define MY_ASSERT() ((void)0)
8 #endif
```

Тогда используется это так

```
1 #include "my_assert"
2
3 int main() {
4     int x = 1;
5     MY_ASSERT(x == 1, "x must be 1!");
6     int y = 2;
7     ASSERT0(y == 2, "y must be 2!");
8     return 0;
9 }
```

И для отключения нужно добавить макрос отключения

```
1 #define DISALBE_MY_ASSERT
2 #include "my_assert.h"
3
4 int main() {
5     int x = 1;
6     MY_ASSERT(x == 1, "x must be 1!"); // is disabled
7     int y = 2;
8     ASSERT0(y == 2, "y must be 2!");
9     return 0;
10 }
```

Так же можно сделать свитч между тремя опциями: `assert`, исключение или ничего. Например так

```

1 // file my_check.h
2 #include <cassert>
3 #include <exception>
4
5 #ifdef MYASSERT
6 #define MyCheck(expr, msg) assert(expr && msg)
7 #else
8 #ifdef MYEXCEPT
9 #define MyCheck(expr, msg) \
10     if (!expr) \
11         throw std::runtime_error(msg)
12 #else
13 #define MyCheck(expr, msg) ((void)0)
14 #endif
15 #endif

```

Теперь этот код можно использовать в трех случаях:

Без проверок

```

#include "my_check.h"

int main() {
    int x = 1;
    MyCheck(x == 1, "Ups!");
}

```

Проверка через `assert`

```

#define MYASSERT
#include "my_check.h"

int main() {
    int x = 1;
    MyCheck(x == 1, "Ups!");
}

```

Бросаем исключение

```

#define MYEXCEPT
#include "my_check.h"

int main() {
    int x = 1;
    MyCheck(x == 1, "Ups!");
}

```

1.4.3 Exceptions

Если вы хотите проверять ошибки программиста с помощью исключений, то вам надо валить вашу программу соответствующим исключением. То есть надо кидать исключение и не перехватывать его. Тогда в режиме дебага вы сразу увидите текущее состояние стека вызова и состояние программы в памяти при исполнении. Наличие стека вызовов позволит лучше понять происхождение ошибки. Однако при таком подходе вам не доступна информация лежащая в самом исключении.

Так же не пытайтесь такие ошибки обрабатывать в блоках `catch`. Иначе вы получите управление логикой программы с помощью исключений. Лучше завести какой-то служебный класс для таких ошибок и в перехватчике исключений его выбрасывать из программы. Например так

```

// file react.h
struct Die {};
void react();

// file react.cpp
void react() {
    try {
        throw;
    } catch (std::exception& e) {
        /*react to standard exception*/
    } catch (Die& die) {
        throw; // rethrow die exception
    } catch (...) {
        /*react to unknown exception*/
    }
}

```

```

#include "react.h"

namespace {
void f() {
    throw Die{};
}

int main() {
    try {
        f();
    } catch (...) {
        react();
    }
    return 0;
}

```

Обычно дебагер позволит вам не только увидеть стек вызовов, но и прыгнуть в строчку кода, где возникло исключение. Так же из плюсов вы увидите текущее состояние всех переменных на стеке.

1.5 Ошибки исполнения среды

Ошибки исполнения среды – это ошибки, которые возникают не из-за багов в программе, а из-за того, что есть какие-то внешние проблемы. В основном это проблемы с получением доступа к ресурсу среды. Вот примеры таких ошибок:

1. Memory Corruption. Нарушение целостности памяти. Это фатальная ошибка, с которой не возможно бороться и нет смысла бороться. Тем не менее, это ошибка исполнения среды.
2. Невозможно открыть файл на чтение или запись. Это по сути отсутствие доступа к ресурсу. Такие ошибки могут быть как фатальными (без этого файла не возможно исполнение), так и не фатальными (вы знаете, что делать и можете работать дальше, без доступа к этому файлу).
3. Нет запрашиваемого устройства. Это по сути отсутствие доступа к ресурсу. Такая ошибка может быть как фатальной, так и не фатально. Все зависит от вашего приложения и устройства.
4. Процессор не поддерживает нужный вид инструкций. Это тоже по сути доступ к ресурсу. Вы не можете получить доступ к операциям на процессоре. Обычно это не фатальная проблема, вы просто откатываетесь к имплементации, которая не использует недоступные операции.
5. Ошибка при парсинге файла. Вам предоставили не корректный ресурс.

Ошибки связанные с целостностью памяти мы просто игнорируем. Если вашей системе битая память, то вам надо менять железо, а не латать программу. Конечно мы можем поговорить на тему «программируй как NASA», но все же это слишком специальный случай.

Ошибки при исполнении можно классифицировать следующим образом:

1. Фатальная ошибка, которая вызывает завершение процесса по `std::terminate`
2. Фатальная ошибка, которая вызывает перезапуск event loop на процессе, но сам процесс не умирает.
3. Локальная ошибка при исполнении, например нет доступа к файлу или некорректное содержимое файла.

Под первые два вида ошибок хорошо подходят исключения, потому что они позволяют быстро умереть или быстро подняться по стеку (при этом параллельно накапливая информацию об ошибках и/или логируя их). Последний подход плохо решается исключениями. Если их решать исключениями, то вы начнете управлять control flow с помощью исключений, а это ошибка дизайна. Ошибки третьего типа лучше решать через Error Codes или `std::expected`, в особенности если расстояние между бросающим и ловящим исключение кодом на стеке вызовов маленькое. Тем не менее, многие сторонние библиотеки по умолчанию обрабатывают локальные ошибки с помощью исключений. Один из таких примеров и что с ним делать можно глянуть ниже [2](#). Потому для использования методов из таких библиотек, к ним приходится писать обертки, которые перехватывают исключения и на выходе используют Error Codes или `std::expected`. О том, как писать такие обертки можно посмотреть ниже в примере [4](#).

1.5.1 Error Codes

Большая часть ошибок может быть обработана здесь и сейчас во время вызова функции. Например, если вы пытаетесь открыть файл на чтение, а файла нет, то вы просто принимаете решение сообщить об этом пользователю и продолжать работать. У вас нет необходимости посылать эту ошибку куда-то далеко по стеку вызовов. Или если вы опрашиваете наличие видеокарты, чтобы понять можно ли ее использовать для вычислений. Если вам приходит ответ, что ее нет, то вы просто запоминаете это и не используете видеокарту. Нет необходимости добираться до main раскручивая стек вызовов полностью. Для таких ошибок еще в Си используют Error Codes.

Давайте посмотрим как это работает. Пусть у вас есть функция

```
1 void func(int a, const B& b);
```

и вы хотите сообщить об ошибке из нее. Для этого надо понять какие могут быть все виды ошибок, которые могут произойти. Для этого заводим `enum class`, содержащий статус операции после вызова функции. Статус возвращается через `return`.

```
1 enum class Status {
2     OK, Fatal, Error, Ingored, Pending
3 };
4
5 Status func(int a, const B& b);
```

Теперь вы можете проверять наличие ошибок так

```

1 Status status = func(1, B{});
2 if (status != Status::OK) {
3     /*handle an error*/
4 }

```

Что делать, если функция возвращает данные через `return`? В этом случае надо сделать выходные аргументы (см. ??). Например, если функция имеет вид

```

1 struct Result {
2     std::string description;
3     int operations = 0;
4 };
5
6 Result func(int a, const B& b);
7
8 int main() {
9     ...
10    Result result = func(1, B{});
11    ...
12 }

```

Мы переделаем функцию в

```

1 Status func(int a, const B& b, Result* result);
2
3 int main() {
4     ...
5     Result result;
6     Status status = func(1, B{}, &result);
7     ...
8 }

```

Главный недостаток такого подхода в том, что если `Result` не является `default constructible`, то есть его нельзя построить неинициализированным, то вы не сможете просто так создать «пустой» `result` и потом заполнить его данными. В языке Си конечно же есть только структуры и там такой проблемы не бывает. В C++ же проблемы все таки есть. Многие Сишные библиотеки используют такой подход. В операционной системе Windows функции используют такой подход.

1.5.2 `std::expected` и `std::optional`

Данный подход вертится вокруг идеи про Error Codes. Мы не хотим использовать `out` аргументы у функции. В этом случае мы хотим иметь возможность сообщать об ошибке и инициализировать данные, которые не `default constructible`. В любом случае данные должны быть хотя бы `movable`. Если у вас есть хрупкий класс, то его всегда можно сделать `movable`, если завернуть его имплементацию в `unique_ptr` (стоит поглядеть в ??). Главная идея в том, что теперь по `return` мы можем вернуть и данные и ошибку.

Простой пример Самый простой вариант – ошибка не содержит никакой информации. В этом случае можно использовать `std::optional`. Пусть для примера вам нужно прочитать файл. При этом нет гарантии, что файл, который вы читаете вообще существует. Давайте предположим, что у вас есть функция, которая пытается прочесть данные из файла. Пусть функция имеет следующую сигнатуру:

```

1 Data read_from_file(const std::filesystem::path& name) ;

```

То есть она всегда по `return` возвращает данные. Что тогда должна делать эта функция, когда данных нет (например из-за отсутствия файла или ошибки чтения)? Она должна бросить исключение. В этом случае использование функции становится неудобным, приходится писать

```

1 Data data;
2 try {
3     data = read_from_file(file_name);
4 } catch (...) {
5     /*react to the error*/
6 }

```

И к сожалению это не работает, если `Data` не `default constructible`. Так как логически такая функция возвращает либо данные либо ошибку, что данных нет, то можно вернуть `std::optional<Data>`.

```

1 std::optional<Data> readFromFile(const std::filesystem::path& name) {
2     try {
3         return read_from_file(name);
4     } catch (...) {
5         return std::nullopt;
6     }
7 }

```

В этом случае проверка на стороне пользователя выглядит как-то так

```

1 auto data = readFromFile(file_name);
2 if (data.has_value()) {
3     use_data(*data);
4 } else {
5     /*react to the error*/
6 }

```

Обратите внимание, что мы не переписываем существующую функцию для чтения из файла (у нас может не быть такой опции, если это сторонняя функция).

Чуть менее простой пример В этом примере мы считаем, что ошибка может содержать какие-то данные. По сути нам нужен тип `std::variant`, который умеет хранить два типа: тип данных и тип ошибки. Только при этом мы хотим эти типы трактовать по-разному. Тип `std::expected` по сути и является частным случаем `std::variant`, с подправленным интерфейсом. Начнем с примера

```

1 struct Data { ... };
2 struct Error { ... };
3
4 using ExpectedData = std::expected<Data, Error>;

```

Теперь функция возвращающая ошибку может выглядеть так

```

1 ExpectedData parse_file(const std::filesystem::path &name);

```

В этом случае мы можем не просто узнать, что была ошибка, но и узнать детали об ошибке:

```

1 auto data = parse_file(file_name);
2 if (!data.has_value()) {
3     /* deal with data.error()*/
4     return;
5 }
6 use_data(*data);

```

В данном случае есть не симметрия. Мы ожидаем, что ошибки это редкость, потому по умолчанию по оператору `operator*` из `std::expected` вынимаются данные. В такой парадигме бывает два вида функций:

Без ошибок

```
R function1(const A&);
```

С ошибками

```
std::expected<R, E> function2(const A&);
```

Функции первого типа – это функции, которые всегда могут любой вход типа `A` переработать в выход типа `R`. Функции второго типа не всегда могут отработать и на каких-то входах дают ошибки. Потому функции второго типа могут вернуть либо тип `R` в случае успеха, либо тип `E` в случае ошибки.

Компонирующее Теперь нужно научиться компоновать функции. Если у нас только функции, которые не могут совершать ошибки, то в этом нет ничего сложного, это делается на уровне языка композицией:

```

1 B function1(const A&);
2 C function2(const B&);
3
4 C c = function2(function1(a));

```

Однако, если мы хотим работать с ошибками, то у нас периодически будут появляться типы вида `std::expected`. Давайте обсудим какие есть опции.

1. **transform**. В данном случае мы хотим выполнить операцию над содержимым `std::expected`. В этом случае сигнатура функции следующая


```
1 R function(const A&);
```

И нам дано

```
1 std::expected<A, E> data;
```

Если внутри `data` лежит объект типа `A` мы хотим применить к нему функцию `function` и запаковать результат в `std::expected<R, E>`. Если же внутри `data` лежит ошибка типа `E`, то мы хотим не трогать ее и вернуть `std::expected<R, E>` инициализированную ошибкой из `data`. Ясно что эту операцию можно сделать руками следующим образом:

```
1 std::expected<R, E> result;  
2 if (data.has_value()) {  
3     result = function(*data);  
4 } else {  
5     result = data.error();  
6 }
```

Эти действия делаются автоматически с помощью метода `transform`.

```
1 std::expected<R, E> result = data.transform(function);
```

То есть `transform` умеет применять к данным любую функцию, которая не совершает ошибок.

2. `and_then`. В этом случае мы тоже хотим применить к данным функцию, но только теперь функция может ошибаться, то есть сигнатура функции теперь такая

```
1 std::expected<R, E> function(const A&);
```

При этом данные нам даны в следующем виде

```
1 std::expected<A, E> data;
```

Мы ожидаем следующее поведение:

- (a) Если внутри `data` лежат данные, то мы применяем к ним `function` и возвращаем этот результат.
- (b) Если внутри `data` лежит ошибка, то мы возвращаем `std::expected<R, E>` инициализированный ошибкой из `data`.

Обратите внимание, что тип ошибки должен быть единый как для данных, так и для функции. Ясно, что можно реализовать такое поведение руками следующим образом

```
1 std::expected<R, E> result;  
2 if (data.has_value()) {  
3     result = function(*data);  
4 } else {  
5     result = std::unexpected(data.error());  
6 }
```

Такая операция автоматически делается с помощью `and_then` так:

```
1 std::expected<R, E> result = data.and_then(function);
```

Таким образом `and_then` умеет применять к данным с ошибкой любую функцию, которая может совершить ошибку того же типа.

1.5.3 Exceptions

Зачем? Прежде всего давайте поймем зачем нам в принципе в языке нужны исключения. Казалось бы можно использовать коды с ошибками или даже `std::expected`. Однако, есть два случая, когда вы не можете воспользоваться этими механизмами.

1. Конструкторы. Давайте начнем с примера кода

```

1  class A {
2  public:
3      A() = default;
4      A(int x) : x_(x) {}
5  private:
6      int x_ = 0;
7  };
8
9  int main() {
10     A a;
11     A b(4);
12     return 0;
13 }

```

Как мы видим в строках 10 и 11, в синтаксисе использования конструкторов у нас нет возможности вернуть код ошибки. Потому что конструктор должен вернуть сконструированный объект. Точнее, это не совсем правда. Технически устроены конструкторы объявленные в строках 3 и 4 можно представлять себе так

```

1  class A {
2  public:
3      static void ctor(A* this) = default;
4      static void ctor(A* this, int x);
5  private:
6      int x_ = 0;
7  };

```

Так вот, конструктор получает указатель на адрес, где надо создать объект и возвращает `void`. Техническую часть можно было бы изменить, но синтаксически не понятно, как это выразить. Как в строках 10 и 11 написать что-то, чтобы вернуть сообщение об ошибке? Думаю тут можно было бы придумать особый синтаксис, но что если вы используете вложенные конструкторы или напрямую используете их как аргументы функции

```

1  function(A(), B(A(), 5), C{1, 2});

```

Кто и куда тут должен возвращать ошибку? Тут вызывается 4 конструктора. Как такое должно сообщать об ошибке?

2. Операторы. Давайте посмотрим на следующий код

```

1  class A {
2  public:
3      friend operator+(const A& first, const A& second);
4  private:
5      ...
6  };
7
8  int main() {
9      A a, b, c;
10     std::cin >> a >> b;
11     c = a / b;
12     std::cout << c << '\n';
13     return 0;
14 }

```

В строчке 11 вызывается оператор деления. Что если разделить на `b` нельзя? Куда сообщить об ошибке? И опять, можно было бы придумать разные синтаксисы, но подумайте, как вы это будете применять, если у вас большое сложное арифметическое выражение. Кроме того, тут еще есть два оператора `operator>>` и `operator<<` для потокового ввода вывода. Они тоже не имеют возможности вернуть код ошибки. Тем не менее, даже потоковые операторы для `std::out` и `std::cin` имеют альтернативный механизм, в рамках которого устанавливаются биты ошибок в специальном глобальном объекте.

Как это работает Давайте посмотрим на следующий пример:

```

1 void g() {
2     throw std::runtime_error("Ups!");
3 }
4
5 void f() {
6     /*some work*/
7     g();
8     /*some work*/
9 }
10
11 int main() {
12     try {
13         f();
14     } catch(std::exception& e) {
15         std::cout << "Error: " << e.what() << '\n';
16     } catch(...) {
17         std::cout << "Unknown exception!\n";
18     }
19 }

```

Давайте опишу работу кода по строкам. Точка входа в программу – функция `main`, потому начинаем с нее.

1. Исполнение начинается в строке 13. Тут вызывается функция `f`. Так как мы ожидаем, что она кидает исключения, оборачиваем это в блок `try`, чтобы его можно было поймать.
2. Далее выполняется код в строке 6 (тут может быть много кода).
3. После этого в строке 7 вызывается функция `g`.
4. Управление переходит к функции `g` и выполняется строчка 2. В этой строчке бросается исключение. В этом месте мы вываливаемся из функции `g` в строчку 7 внутри функции `f`.
5. В строчке 7 после исключения мы должны выйти из функции `f` и будем так выходить до тех пор, пока не встретим `catch`, который поймает исключение. Но чтобы выйти мы должны почистить стек от функции `f`. Это называется `stack unwinding`. Надо удалить все локальные переменные и аргументы функции `f`. Для этого после строчки 7 вызываются деструкторы всех локальных переменных на стеке, которые создала функция `f`.
6. Почистив стек мы выходим в больший scope из которого вызвана `f`. В данном случае это строка 13. Так как мы вышли в блок `try`, то мы ловим исключение и для реакции на него нужны следующие блоки `catch`.
7. Мы идем по блокам `catch` один за одним и смотрим, есть ли хотя бы один блок, под который подходит наше исключение. В данном случае исключение типа `std::runtime_error` унаследовано от `std::exception` и так как мы принимаем исключение по ссылке, то включается полиморфизм и мы можем поймать это исключение в первом блоке. Если бы было несколько подходящих блоков, то исключение ловится первым, а остальные не реагируют. Если нет ни одного блока, который может поймать исключение, то оно улетает дальше из текущего scope.

Таким образом, если вы работаете с исключениями, то вообще говоря, вы можете выйти из любого scope в любой точке функции.

1.5.4 Гарантии поведения при исключениях

Предположим, мы выполняем следующий код

```

1 Data d = ...;
2 Result r;
3 try {
4     r = algorithm(d);
5 } catch(...) {
6 }

```

Вопрос, если во время выполнения алгоритма было брошено исключение, то в каком состоянии находятся `d` и `r`? Если `d` принимается по константной ссылке, то оно гарантировано не меняется, потому нас скорее всего интересует состояние объекта `r`. Бывает три вида гарантий:

1. Strong Exception Guaranty. В этом случае все объекты находятся в состоянии в каком они были до вызова функции бросающей исключение.
2. Basic Exception Guaranty. В этом случае все объекты находятся в корректном состоянии, но неизвестно в каком. Объектами можно пользоваться, но конкретное состояние не известно и скорее всего не представляет интереса.
3. No Exception Guaranty. Никаких гарантий. Например, у `r` из примера выше могут быть нарушены инварианты класса.

Когда вы пишете код, то важно его писать так, чтобы выполнялись Basic Exception Guaranty. Это должно быть опцией по умолчанию. Хорошо было бы иметь Strong Exception Guaranty, но это может оказаться дорогим. Последняя опция допустима только при отключенных исключениях. Однако, даже если вы не используете исключения, если вы пишете код с соблюдением Basic Exception Guaranty, то код получается лучше. Связано это с тем, что у вас появляются ограничения, которые направляют написание кода в лучшую сторону.

1.5.5 Технические детали и логическая модель для исключений

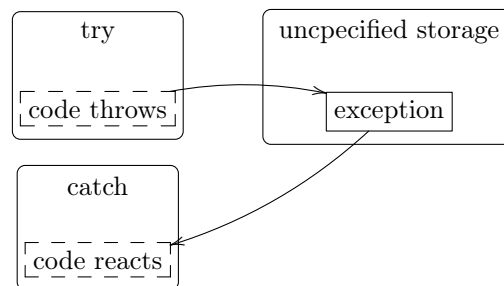
Технические детали, которые надо знать Прежде чем обсуждать какие есть сложности при использовании исключений, давайте обсудим технические детали связанные с имплементацией. Когда вы кидаете исключения надо помнить о следующем:

1. Как аллоцируется память для исключения в стандарте не определено. На практике это означает, что вы всегда аллоцируете память на куче даже для стандартных типов вроде `int`.
2. Для того, чтобы произвести stack unwinding вам нужно для каждой точки программы знать, какие деструкторы позвать. Как это должно делаться в стандарте не определено, но на практике есть два способа:
 - (a) Frame-based. Данный метод по сути поддерживает в run-time второй стек, на который складываются деструкторы объектов для текущего scope. Этот метод означает, что вы платите за исключения даже, когда их не кидаете. Этот подход использует компилятор от Microsoft.
 - (b) Table-based. В данном подходе, во время компиляции программы, вы составляете таблицы для вызова нужных деструкторов из каждой точки вашего кода. И в run-time используете эти таблицы для их вызова. Такой подход очень сильно раздувает бинарники программы, потому что нужно хранить все эти таблицы. Такой подход используют gcc и clang.

Когда мы ловим исключения надо понимать следующее:

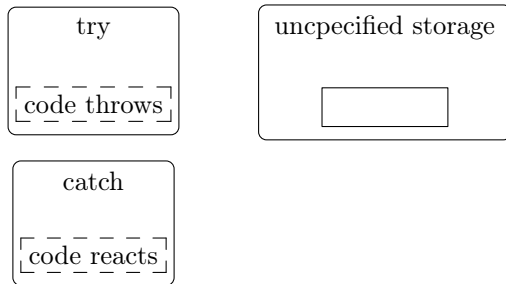
1. Язык предоставляет полиморфное поведение. То есть если вы унаследовали класс `A` от класса `B`, то по ссылке на `B` можно ловить и исключение класса `A`. В стандарте не прописано как это должно быть имплементировано, но на практике всегда используется RTTI, что дорого.
2. Деаллокация памяти для исключения не специфицирована в стандарте. На практике это значит удаление объекта с кучи и это дорого.

Возможные состояния Здесь я хочу немного поговорить про ментальную модель для исключений и в каких состояниях может находиться ваша программа логически. Давайте начнем с картинки:

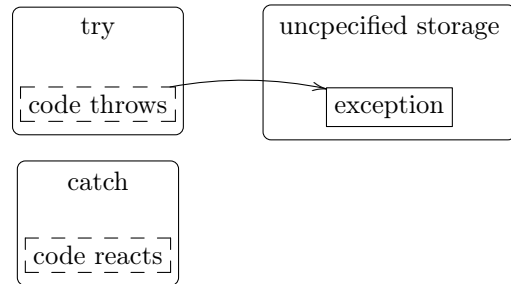


Думаю, что данная диаграмма более или менее не требует объяснений. Процесс бросания исключений в штатном режиме можно представлять себе так:

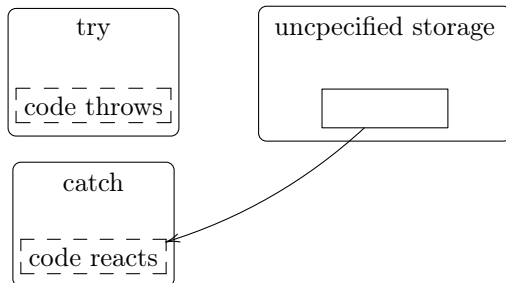
1) Исполнение в блоке `try`



2) Бросаем исключение



3) Перехватываем исключение в `catch`

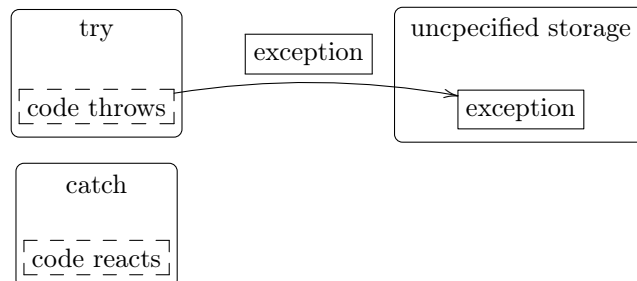


Давайте проговорим, что тут происходит:

1. Начинаем исполнение программы в блоке `try`.
2. Код в блоке `try` бросает исключение. В этот момент объект исключения аллоцируется неспецифицированным образом.
3. Как только исключение долетело до блока `try`, для него ищется нужный обработчик среди присоединенных `catch` блоков. Если нужный обработчик нашелся, то мы вынимаем объект исключения оттуда, где он хранился и отдаем его в распоряжение `catch` блока. В конце обработки `catch` блок сам удалит объект исключения, вам не надо об этом заботиться. В этом случае исключение логически не в брошенном состоянии, а в пойманном. И система помнит это исключение. Его можно перебросить по команде `throw` без аргументов.

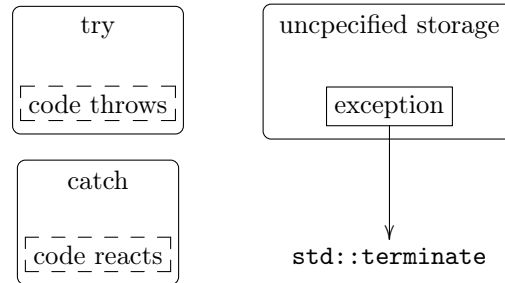
Проблемы А теперь поговорим, что в этой схеме может пойти не так.

1. Вы можете кинуть исключение, когда другое исключение все еще висит не обработанное. Такое возможно, если вы бросите исключение во время `stack unwinding`. По этой причине нельзя кидать исключения в деструкторах.



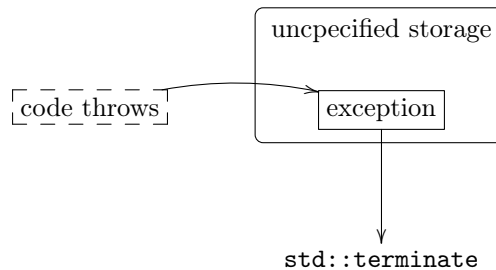
В этом случае программа падает по `std::terminate`.

2. Если не нашлось ни одного обработчика среди `catch` блоков.



При этом был ли почищен стек с помощью `stack unwinding` не специфицировано.

3. Исключение брошено не внутри блока `try`. В этом случае мы никогда не найдем соответствующий перехватчик для исключения. И мы опять падаем по `std::terminate`.



1.5.6 Важные примеры и ожидаемое поведение для исключений

1. Самый первый пример, который обычно приводя – выделение памяти. Например, вы пытаетесь выделить память под `std::vector` и ее не хватило. Это плохой пример для использования исключений по следующим причинам:
 - (a) Если вы запросили память и ее больше в системе нет, то что вы собираетесь делать? Скорее всего у вашей системы проблемы покруче, чем невозможность продолжить работу вашей программы. Смысла тут делать что-то нет.
 - (b) Память запрашивает чуть ли не каждый кусок кода в любой библиотеке. Тогда придется проверять на исключение каждое обращение к оператору выделения памяти. В таком случае программа только и будет делать, что проверять исключения и ничего больше. Это очень глупо.
2. Разумный пример, чтобы умереть. Скажем, вы хотите работать в программе с клавиатурой через какой-нибудь интерфейс операционной системы. Делаете запрос, на доступ к этому интерфейсу, а операционная система говорит, что нельзя и не дает доступ.

```
1 auto handle = access_keyboard(...);
2 if (!handle) {
3     // cannot access the keyboard
4 }
```

То есть системная функция вернула ошибку и вы не можете даже начать выполнение программы в таком случае. Вот такую ситуацию как раз надо ловить исключением. И политика тут такая, если все получилось, то хорошо, если не удалось получить доступ, то сообщаем пользователю, что у нас проблемы с доступом и завершаем работу. Обычно за получение доступа к клавиатуре у вас будет отвечать специальный объект. И доступ к клавиатуре вы запросите в его конструкторе.

```
1 class Keyboard {
2 public:
3     Keyboard() {
4         auto handle = access_keyboard(...);
5         if (!handle)
6             throw std::runtime_error("Cannot access the keyboard!");
7     }
8 private:
9     ...
10 };
```

При этом для приличия такое исключение надо поймать в `main`, чтобы не падать по `std::terminate`.

3. Разумный пример, чтобы умереть и воскреснуть. Предположим у вас в потоке крутится сервис, который может упасть в результате любой ошибки и вы хотите этот сервис перезапустить. Код для сервиса может выглядеть как-то так:

```
1 int main() {
2     Server server;
3     server.run();
4     return 0;
5 }
```

Если метод `run` бросает исключение, то можно сделать следующее:

```
1 int main() {
2     for (Status status = Running; status != Quit;) {
3         try {
4             Server server;
5             status = server.run(); // if returns normally sets status to be Quit
6         } catch (...) {
7             /*log the error*/
8         }
9     }
10    return 0;
11 }
```

В этом случае мы крутимся в бесконечном цикле, пока флаг `status` не имеет значение `Quit`. Если сервер заканчивает работу в штатном режиме, то метод `run` устанавливает `status` в состояние `Quit` и так мы понимаем, что надо выйти из цикла. Если же конструктор `server` или метод `run` умирают по исключению, то мы логируем ошибку и пытаемся еще раз инициализировать `server` и запустить его. В таком методе полезно еще установить границу максимального количества перезапусков сервера и максимальную частоту перезапусков.

4. Разумный пример для продолжения работы. Предположим, что вы парсите файл какого-то сложного формата. И в рамках этого парсинга вы вызываете много вспомогательных функций. И если возникла проблема с парсингом, то это значит, что файл прочитать нельзя, формат не верный и вы хотите быстро выйти наверх в самую старшую функцию.

```
1 Object parse(std::filesystem::path name) {
2     File file = open_file(name);
3     auto header = parse_header(file);
4     auto data = parse_data(file);
5     return Object(data);
6 }
7
8 auto parse_data(File& file) {
9     ...
10    parse_something(file);
11    ...
12    return ...;
13 }
14
15 auto parse_something(File& file) {
16     ...
17     if (error occure)
18         throw std::runtime_error("parsing error in parse_something!");
19     return ...;
20 }
```

Только теперь надо обработать эту ошибку грамотно

```
1 std::optional<Object> try_parse(std::filesystem::path name) noexcept {
2     try {
3         return parse_impl(name);
4     } catch (...) {
5         return std::nullopt
6     }
7 }
```

```

8
9 Object parse(std::filesystem::path name) {
10     File file = open_file(name);
11     auto header = parse_header(file);
12     auto data = parse_data(file);
13     return Object(data);
14 }

```

Причем мы это делаем неинтрузивно, то есть без вмешательства в имплементацию старой функции. Обратите внимание, что старая функция всегда возвращает объект нужного типа `Object`, но об ошибках она сообщает с помощью исключений, которые надо ловить снаружи. Это может быть не удобно, так как засоряет ваш код. Вы не хотите управлять логикой программы с помощью исключений. Для этого мы оборачиваем старую имплементацию в новую. Новая имплементация уже не может кидать исключения, и об ошибке она сообщает уже через возвращаемое значение. Я использовал `std::optional`, но можно и лучше использовать `std::expected`. В таком случае вы просто пытаетесь распаковать файл и если была ошибка возвращаете пустоту, а если не было ошибки, то возвращаете данные. После использования этой функции надо просто проверить удалось ли считать данные или нет. И если удалось, то надо вынуть их. Давайте сравним два обработчика:

```

try {
    Object obj = parse(file_name);
    run_program(obj);
} catch (std::exception& e) {
    print_error(e.what());
} catch (...) {
    print_error("Unknown error!");
}

```

```

std::optional<Object> obj =
    try_parse(file_name);
if (!obj.has_value()) {
    print_error("Cannot parse file_name");
    return 0;
}
run_program(*obj);

```

Обратите внимание как проще выглядит workflow во втором случае. Во втором подходе, если вы забыли, что у вас могут быть невалидные данные после парсинга и вы попытаетесь написать такое:

```

1 auto obj = try_parse(file_name);
2 run_program(obj);

```

То компилятор вам подскажет об ошибке, что вы забыли распаковать данные и передаете `std::optional` вместо `Object` в функцию `run_program`. Однако, если в первом подходе, вы забыли поставить `try/catch` блок или забыли добавить в `catch` блок нужный перехватчик, то ваша программа в этом случае упадет по `std::terminate` и никто вам в этом не поможет.

Давайте я подведу итоги написанному выше. Исключения можно использовать в двух сценариях:

1. На критическую ошибку, которая не позволяет продолжать работать, умереть и сообщить причину смерти пользователю. Это был пример 2.
2. На не критическую ошибку перехватить исключение и пойти по плану «Б» для данной ошибки. Это был пример 4. При этом обратите внимание, что на стороне пользователя лучше иметь функцию, которая не кидает исключения, а возвращает либо объект, либо ошибку. Потому что в этом случае попытка управления через исключения – это ошибка дизайна. Вы не должны управлять логикой программы с помощью исключений. Кроме того, вам придется при каждом использовании бросающей исключения функции писать `try/catch` блоки, что сделает код менее читаемым. Лучше инкапсулировать эту работу в другую не бросающую исключения функцию.

Если вы используете исключения для чего-то еще, то скорее всего вы делаете что-то не так.

1.5.7 «Идеальная» имплементация

Богатая иерархия наследования между исключениями – большая головная боль. Вы никогда не запомните (а может быть даже и не узнаете), кто от кого унаследован и унаследован ли и никогда не поставите `catch` блоки в правильном порядке. Потому система должна быть максимально простой. Нужен обязательно один базовый класс, чтобы можно было ловить сразу все исключения, а все остальные от него унаследованы и отличаются только данными. На уровне интерфейса я бы ожидал следующее:


```

1  template<class T>
2  class Exception {
3  public:
4      Exception(std::string description, T data);
5
6      const std::string& what() const;
7      std::string& what();
8      const std::source_location where() const;
9      const std::stacktrace& stack() const;
10     const T& data() const;
11     T& data();
12 private:
13     std::string description_;
14     std::source_location location_;
15     std::stacktrace backtrace_;
16     T data_;
17 };

```

Обратите внимание, что любое исключение содержит информацию, которую мы обсуждали в разделе 1.3, а именно:

1. Текстовое описание ошибки.
2. Точка в коде, где ошибка произошла.
3. Текущее состояние стека.
4. Специфические данные типа T.

В данном подходе все исключения параметризуются типом T и для одинакового типа T исключения не отличимы.

Заметим, что для T = void нужно сделать специальную имплементацию, которую можно так же использовать в качестве базового класса, а именно:

```

template<>
class Exception<void> {
public:
    Exception(std::string description);

    const std::string& what() const;
    std::string& what();
    const std::source_location where() const;
    const std::stacktrace& stack() const;
private:
    std::string description_;
    std::source_location location_;
    std::stacktrace backtrace_;
};

```

```

template<class T>
class Exception : public Exception<void> {
public:
    Exception(std::string description, T data);

    const T& data() const;
    T& data();
private:
    T data_;
};

```

Давайте обсудим некоторые важные детали.

1. Данные о точке кода, где произошла ошибка, и текущее состояние стека являются константными (это инвариант класса). Они создаются в конструкторе и потом доступны только для чтения. Поля класса не делаются константами, чтобы обеспечить хорошее поведение value semantics.
2. Строка с описанием и данные типа T доступны для модификации. Это нужно, если при перебрасывании исключения вам нужно добавить информацию. Выглядит это как-то так

```

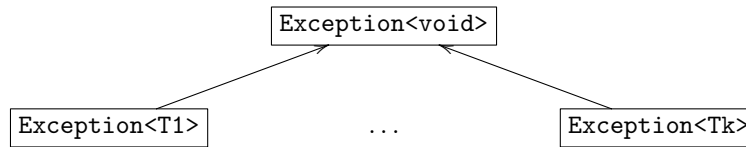
1  try {
2      function(); // throws
3  } catch(Exception<void>& e) {
4      e.what() += "info"; // adds extra information
5      throw;
6  }

```

Аналогично может понадобиться модификация кастомных данных типа T.

3. Перехват всех возможных исключений нашего типа делается с помощью перехвата базового класса `Exception<void>`, который не содержит кастомных данных.

Таким образом логическая схема наследования получается очень простой:



Теперь приведем имплементацию классов. В начале базовый класс.

```
1  template<class T>
2  class Exception;
3
4  template<>
5  class Exception<void> {
6  public:
7      Exception(std::string description,
8                std::source_location location = std::source_location::current(),
9                std::stacktrace backtrace = std::stacktrace::current())
10         : description_(std::move(description)), location_(std::move(location)),
11           backtrace_(std::move(backtrace)) {
12     }
13     std::string& what() {
14         return description_;
15     }
16     const std::string& what() const {
17         return description_;
18     }
19     const std::source_location where() const {
20         return location_;
21     }
22     const std::stacktrace& stack() const {
23         return backtrace_;
24     }
25 private:
26     std::string description_;
27     std::source_location location_;
28     std::stacktrace backtrace_;
29 };
```

Обратите внимание, что тут приходится использовать трюк из раздела 1.3, чтобы правильно перехватить строку, в которой произошла ошибка и состояние стека. Нам важно, чтобы это был не конструктор для исключения, а точка, где вызван конструктор. Это возможно с помощью аргументов по умолчанию, потому что они конструируются в точке вызова функции (в данном случае конструктора). Теперь классы с дополнительной информацией выглядят так:

```
1  template<class T>
2  class Exception : public Exception<void> {
3      using Base = Exception<void>;
4
5  public:
6      Exception(std::string description, T data,
7                std::source_location location = std::source_location::current(),
8                std::stacktrace backtrace = std::stacktrace::current())
9         : Base(std::move(description), std::move(location), std::move(backtrace)),
10           data_(std::move(data)) {
11     }
12     const T& data() const {
13         return data_;
14     }
15     T& data() {
16         return data_;
17     }
18 private:
19     T data_;
20 };
```

Обратите внимание, что тут нельзя вызвать базовый конструктор с аргументами по умолчанию, потому что они покажут точку в коде – текущий конструктор. Потому здесь придется руками еще раз прописать аргументы по умолчанию и все аргументы передаем в базовый конструктор.

Теперь для удобства ввода вывода нужно правильно перегрузить потоковые операторы. Для отображения места в коде, где произошла ошибка, и текущего состояния стека мы уже писали имплементацию в разделе 1.3, однако, для полноты картины я их еще раз тут повторю:²

```
1 std::ostream& operator<<(std::ostream& out, const std::source_location& location) {
2     out << location.file_name() << "(" << location.line() << ":"
3       << location.column() << "), function '" << location.function_name()
4       << "'";
5     return out;
6 }
7
8 std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
9     static constexpr int bound = 6;
10    if (backtrace.size() < bound)
11        return out;
12    for (auto iter = backtrace.begin(); iter != (backtrace.end() - bound); ++iter) {
13        out << iter->source_file() << "(" << iter->source_line()
14          << "):" << iter->description() << '\n';
15    }
16    return out;
17 }
```

Теперь нужны удобные операторы для отображения исключений. Придется написать оба по-отдельности, потому что в базовом классе отсутствует данное, которое есть в производных классах.

```
1 std::ostream& operator<<(std::ostream& out, const Exception<void>& e) {
2     std::cout << "Failed to process: " << e.what() << '\n'
3       << e.where() << '\n'
4       << e.stack();
5     return out;
6 }
7
8 template<class T>
9 std::ostream& operator<<(std::ostream& out, const Exception<T>& e) {
10    std::cout << "Failed to process with code (" << e.data() << ") : " << e.what()
11      << '\n'
12      << e.where() << '\n'
13      << e.stack();
14    return out;
15 }
```

Теперь этим можно пользоваться так:

```
1 void function() {
2     throw Exception<int>("Some error!", 1);
3 }
4
5 int main() {
6     try {
7         function();
8     } catch (const Exception<int>& e) {
9         std::cout << e;
10    } catch (const Exception<void>& e) {
11        std::cout << e;
12    } catch (...) {
13    }
14 }
```

1.5.8 Рекомендации

Рекомендации по использованию Тут я бы хотел сформулировать несколько соображений о том, как я вижу обработку исключений.

²Если вам интересно, что значит `bound`, то загляните в конец раздела 1.3.

1. По-хорошему у вас должна быть одна точка по обработке исключений (или несколько, но не много). Чаще всего – это event loop в вашей программе, где вы можете сообщить об ошибке и умереть. Или функция `main`, которая является последней линией обороны до выхода в операционную систему или падения по `std::terminate`.
2. Исключения никогда не должны быть в критической для производительности части. Более того, рекомендуется пометить код `noexcept` (даже если он кидает исключения, например на запрос памяти у системы), чтобы компилятору было дозволено больше оптимизаций.
3. Исключения не должны ни в каком виде контролировать workflow вашего кода. За это должна отвечать логика программы.

Технические рекомендации

1. Исключения кидаются by value.

```
1 throw A(1, "abc");
```

2. Исключения ловятся по ссылке или константной ссылке. Это нужно для обеспечения полиморфного поведения.

```
1 catch(const A& a)
```

3. Повторное кидание всякого исключения делается командой `throw` без аргументов. Этот трюк используется для того, чтобы отложить реакцию на исключение.

```
1 void react() {
2     try {
3         throw;
4     } catch (std::exception& e) {
5         /*react to std::exception*/
6     } catch (...) {
7         /*react to unknown exception*/
8     }
9 }
10
11 int main() {
12     try {
13         run();
14     } catch (...) {
15         react();
16     }
17 }
```

В этом примере, когда мы попадаем в блок `catch`, мы перехватили текущее исключение. Оно больше не висит, но система знает это последнее исключение внутри блока `catch`. И чтобы перекинуть его мы в функции `react` используем команду `throw` без аргументов в строке 3. Однако, ее надо помещать внутри `try` блока, чтобы после перекидывания, можно было поймать исключение.

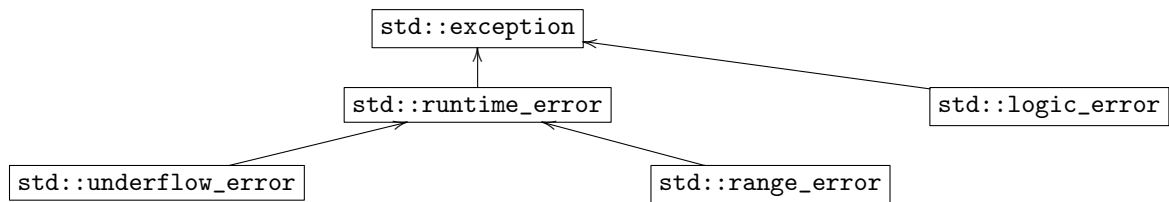
4. Производные классы исключений среди `catch` обработчиков должны идти перед базовыми классами, чтобы поймать наиболее специализированный тип.

```
1 class A {};
2 class B : public A {};
3
4 try {
5     run();
6 } catch (const B& e) {
7     /*react to B*/
8 } catch (const A& e) {
9     /*react to A*/
10 } catch (...) {
11     /*react to everything left*/
12 }
```

Напомним, что блоки `catch` проходятся сверху вниз по одному, и на первом подходящем блоке исключение обрабатывается и последующие `catch` блоки игнорируются. Проблема с этим подходом в том, что вы должны знать отношение наследования между классами, чтобы правильно добавлять обработчики. Вот отличный пример

```
1 try {
2     ...
3 } catch (const std::runtime_error& e) {
4 } catch (const std::underflow_error& e) {
5 } catch (const std::range_error& e) {
6 } catch (const std::logic_error& e) {
7 } catch (...) {
8 }
```

проблема в том, что диаграмма наследования для исключений выглядит так



Это означает, что мы никогда не поймем исключение в строках 4 и 5, потому что они унаследованы от исключения, перехватываемого в строке 3. Помнить наизусть это все невозможно. Иерархия на исключениях делает код хрупким. Куда проще было бы сразу перехватывать базовый класс.

```
1 try {
2     ...
3 } catch (const std::exception& e) {
4 } catch (...) {
5 }
```

В идеале для работы с исключениями можно пользоваться имплементацией описанной в разделе [1.5.7](#).

1.5.9 Проблемы при использовании исключений

Прежде чем вы прочитаете как жизнь с исключениями невыносима, я хочу напомнить про главную пользу исключений. Когда вы пишете код, подразумевая, что у вас могут быть исключения, вы находитесь в рамках серьезных ограничений (пусть компилятор их на вас не накладывает, но вы должны это понимать). А когда у вас есть серьезные ограничения, то вы всегда пишете код лучше.

Менеджмент ресурсов Так как исключение может возникнуть в любой строчке кода то это значит, что если исполнилась одна строчка кода, то у вас нет гарантии, что выполнится следующая. Давайте посмотрим на следующий пример:

```
1 void g() {
2     throw std::runtime_error("Ups!");
3 }
4
5 void f() {
6     int* x = new(0);
7     g();
8     delete x;
9 }
```

В этом примере при вызове `f` мы выйдем из нее в строчке 7, где функция `g` кинет исключение. Потому операция `delete` никогда не вызовется. Потому когда вы используете исключения, все ресурсы должны менеджериться автоматическими методами с использованием RAII (см. раздел ??).

Исключения в деструкторах В разделе [1.5.5](#) мы обсудили, что нельзя бросать исключения, когда есть активное не пойманное исключение в системе. Такое может произойти, когда мы освобождаем стек при выходе из текущего scope. В этот момент вызываются деструкторы всех локальных переменных из данного scope.

А это означает, что ни в коем случае нельзя кидать исключения в деструкторах. Если вы это сделаете, то программа упадет в run-time вызовом `terminate`. Потому никогда не бросайте исключения в деструкторах. Это чревато и не работает. У вас нет шансов сообщить об ошибке в деструкторе безопасно. Их надо обрабатывать внутри самого деструктора. Но лучше не кидать никогда.

Правила инициализации объектов и несколько ресурсов Теперь, что делать, если вам надо выделить память для двух указателей? Тупой вариант такой

```
1 struct IntPtrPair {
2     IntPtrPair(int x, int y) : ptr1_(new int(x)), ptr2_(new int(y)) {}
3     ~IntPtrPair() {
4         delete ptr1_;
5         delete ptr2_;
6     }
7     int* ptr1_;
8     int* ptr2_;
9 };
10
11 IntPtrPair x(1, 2);
```

Беда этого подхода вот в чем. Когда вы в строчке 11 вызываете конструктор, то сначала выделяется память под `ptr1_`, а потом под `ptr2_`. И если при первом вызове память выделится, а во втором нет и будет брошено исключение, то конструктор не завершит работу, а значит объект не будет считаться построенным, а значит и не вызовется деструктор для него во время stack unwinding. А это значит, что память, выделенная первым вызовом не освободится. Есть много костылей для этой проблемы, но правильное решение – обернуть каждый указатель в свой `wrapper`. Например так

```
1 struct IntPtrPair {
2     IntPtrPair(int x, int y) : ptr1_(std::make_unique<int>(x)),
3                               ptr2_(std::make_unique<int>(y)) {}
4
5     std::unique_ptr<int> ptr1_;
6     std::unique_ptr<int> ptr2_;
7 };
8
9 IntPtrPair x(1, 2);
```

И теперь вам вообще не надо беспокоиться о деструкторах и прочих радостях, потому что у вас везде value semantics. Все это за вас для каждого ресурса делает `wrapper` для работы с одним указателем.

Зависимые ресурсы и исключения Существует еще одна ситуация, о которой стоит рассказать. А что если у нас данные зависимы? Например, я должен сначала выделить память под `x`, а потом выделяя память под `y`, я должен туда положить информацию про `x`? То есть у вас есть естественная зависимость порядка конструирования данных. Мы уже обсуждали, что для конструирования по частям как раз и создано наследование в разделе `??`. Оно собирает объект в определенном порядке. Давайте приведу такой пример

```
1 struct A {
2     A(int x) : ptr_(std::make_unique<int>(x)) {}
3
4     std::unique_ptr<int> ptr_;
5 };
6
7 struct B : A {
8     B(int x) : A(x), ptr_(std::make_unique<int*>(A::ptr_.get())) {}
9
10    std::unique_ptr<int*> ptr_;
11 };
12
13 B x(1);
```

В таком случае у вас в строчке 13 сначала создается базовая часть объекта, то есть выделяется память под `x` и кладется адрес в `A::ptr_`. А потом вы выделяете память под `B::ptr_` и кладете туда адрес `A::ptr_` из базовой части. Понятно, что это бессмысленная деятельность, но демонстрирует зависимость данных. Тут прошу обратить внимание на несколько вещей:

1. Я везде пользуюсь `wrapper`-ами для указателей и мне вообще не надо задумываться про деструкторы.

2. Обратите внимание, что нигде не пользуюсь `new` явно, вместо этого пользуюсь `std::make_unique` функцией.
3. Если в конструкторе `B` при выделении памяти под `B::ptr_` будет кинута исключение, то так как базовая часть была построена и ее конструктор завершил работу, то вызовется деструктор базовой части `~A()`. Это позволяет безболезненно кидать исключения при построении объекта по частям.

Вот еще один пример по сборке Thread Pool через Windows Api. Для инициализации у нас есть несколько ресурсов:

1. `PTP_POOL pool`
2. `TP_CALLBACK_ENVIRON environment`
3. `PTP_CLEANUP_GROUP cleanup_group`

При этом логика инициализации следующая:

1. Создать `pool` так

```
1 pool = CreateThreadpool(nullptr);
```

Эта операция может завершиться ошибкой. В этом случае `pool` будет `nullptr`

2. Создать `environment` так

```
1 InitializeThreadpoolEnvironment(&environment);
```

3. Соединить `pool` и `environment` так:

```
1 SetThreadpoolCallbackPool(&environment, pool);
```

4. Создать `cleanup_group` так

```
1 cleanup_group = CreateThreadpoolCleanupGroup();
```

Эта операция может завершиться ошибкой. В этом случае `cleanup_group` будет `nullptr`.

5. Соединить `environment` и `cleanup_group` так

```
1 SetThreadpoolCallbackCleanupGroup(&environment, cleanup_group, nullptr);
```

Чтобы инициализировать целиком все данные можно сделать следующее. Под каждый ресурс заводим класс, который будет за него отвечать. После чего наследуем их друг от друга, чтобы инициализировать их в правильном порядке.

```
class Pool {
public:
    Pool();
    ~Pool();
    /*other methods*/
private:
    PTP_POOL pool_;
};
```

```
Pool::Pool() : pool_(CreateThreadpool(nullptr)) {
    if (pool_ == nullptr)
        throw std::runtime_error("Fail!\n");
}
```

```
Pool::~~Pool() {
    CloseThreadpool(pool_);
}
```

```
class Environment : protected Pool {
public:
    Environment();
    ~Environment();
    /*other methods*/
private:
    TP_CALLBACK_ENVIRON environment_;
};
```

```
Environment::Environment() {
    InitializeThreadpoolEnvironment(&environment_);
    SetThreadpoolCallbackPool(&environment_, pool_);
}
```

```
Environment::~~Environment() {
    DestroyThreadpoolEnvironment(&environment_);
}
```

```

class CleanupGroup : protected Environment {
public:
    CleanupGroup();
    ~CleanupGroup();
    /*other methods*/

private:
    PTP_CLEANUP_GROUP cleanup_;
};

```

```

class ThreadPool : protected CleanupGroup {
public:
    ThreadPool();
    ~ThreadPool();
    /*other methods*/
};

```

```

CleanupGroup::CleanupGroup()
: cleanup_(CreateThreadPoolCleanupGroup()) {
    if (cleanup_ == nullptr)
        throw std::exception("Fail!\n");
    SetThreadPoolCallbackCleanupGroup(
        &environment_, cleanup_, nullptr);
}
CleanupGroup::~CleanupGroup() {
    assert(cleanup_ != nullptr);
    CloseThreadPoolCleanupGroup(cleanup_);
}

```

```

ThreadPool::ThreadPool() = default;

ThreadPool::~ThreadPool() {
    CloseThreadPoolCleanupGroupMembers(
        cleanup_, false, nullptr);
}

```

В такой схеме можно спокойно кидать исключение в любом конструкторе и вся часть класса, построенная на предыдущих этапах очистится корректно.

Другие виды ресурсов Аналогично тому, что я написал про память используется для выделения других ресурсов. Например для мьютексов используются `std::lock_guard` или похожие механизмы. Любой запрос к операционной системе, который требует освобождения своих ресурсов, должен выполняться через `wtapper`, где в конструкторе вы запрашиваете ресурс, а в деструкторе освобождаете.