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 Ошибки программиста . . . 5
  - 1.3.1 Информация об ошибке . . . 5
  - 1.3.2 `assert` . . . 5
  - 1.3.3 Кастомные `assert`-ы . . . 8
  - 1.3.4 Exceptions . . . 10
- 1.4 Ошибки исполнения среды . . . 10
  - 1.4.1 Error Codes . . . 10
  - 1.4.2 `std::expected` . . . 10
  - 1.4.3 Exceptions . . . 10
- 1.5 Exceptions, Stack Unwinding и RAII . . . 11
- 1.6 Исключения и конструирование объектов . . . 12

1 Обработка ошибок

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

```
1  enum class Status {
2    OK, Error
3  };
4
5  Status algorithm(const Data& input, Result* output);
```

[1] Компилятор 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`.

```

int main() {
  Data data = ...;
  Result result;
  Status status = algorithm(data, &result);
  if (status == Status::OK) {
    ...
  } else {
    ...
  }
}
```

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

```
class Error {
  ...
};

std::expected<Result, Error> algorithm(const Data& data);

int main() {
  Data data = ... ;
  auto result = algorithm(data);
  if (result.has_value()) {
    Result r = std::move(result).value();
    ...
  } else {
    ...
  }
}
```

5. Исключения – Exceptions. Это специальная фича 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 доложен быть автоматизирован с помощью RAII (см. раздел **??**). В коде это выглядит как-то так

```
Result algorithm(const Data& data) {
  ...
  if (/*some error*/)
    throw std::runtime_error(/*error information*/);
  ...
  return result;
}

int main() {
  Data data = ... ;
  Result result;
  try {
    result = algorithm(data);
  } catch (std::exception& e) {
    /*handle known exceptions*/
  } catch (...) {
    /*handle unknown exceptions*/
  }
  return 0;
}
```

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 на столько 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` пишите 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 пишите 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 фича 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 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 Ошибки программиста

1.3.1 Информация об ошибке

TO DO

1.3.2 `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 строчка 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 строчке кода она упала.

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.

Рекомендации Мои рекомендации по 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   }
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.

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

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.

```
class A {
public:
  ...
  I* operator->() const {
    assert(ptr_ != nullptr && "The object has invalid pointer!");
    return ptr_;
  }
private:
  I* ptr_ = nullptr;
};
```

В данном примере мы можем спокойно звать

```
int main() {
  A a;
  a->f();
  return 0;
}
```

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 inplementation*/
}
```

Получше

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

7. И конечно же не забывайте сообщить об ошибке текстом. 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;
}
```

1.3.3 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. Вот как могут выглядеть кастомные `assert`-ы.

```
// file custom_asserts.h
#include <cassert>

#ifndef DISABLE_ASSERTS
#define DISABLE_ASSERTS 3
#endif

#if DISABLE_ASSERTS < 0
#define ASSERT0(expr, msg) ((void)0)
#else
#define ASSERT0(expr, msg) assert(expr && msg)
#endif

#if DISABLE_ASSERTS < 1
#define ASSERT1(expr, msg) ((void)0)
#else
#define ASSERT1(expr, msg) assert(expr && msg)
#endif

#if DISABLE_ASSERTS < 2
#define ASSERT2(expr, msg) ((void)0)
#else
#define ASSERT2(expr, msg) assert(expr && msg)

#endif

#if DISABLE_ASSERTS < 3
#define ASSERT3(expr, msg) ((void)0)
#else
#define ASSERT3(expr, msg) assert(expr && msg)
#endif
```

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

```
#include "custom_asserts.h"

int main() {
  int x = 0;
  ASSERT0(x == 0, "x must be 0!");
  int y = 1;
  ASSERT1(y == 1, "y must be 1!");
  int z = 2;
  ASSERT2(z == 2, "z must be 2!");
  return 0;
}
```

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

```
#define DISABLE_ASSERTS 0
#include "custom_asserts.h"

int main() {
  int x = 0;
  ASSERT0(x == 0, "x must be 0!");
  int y = 1;
  ASSERT1(y == 1, "y must be 1!"); // is disabled
```

```
  int z = 2;
  ASSERT2(z == 2, "z must be 2!"); // is disabled
  return 0;
}
```

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.

```
// file my_assert.h
#include <cassert>

#ifndef DISALBE_MY_ASSERT
#define MY_ASSERT(expr, msg) assert(expr && msg)
#else
#define MY_ASSERT() ((void)0)
#endif
```

Тогда используется это так

```
#include "my_assert"

int main() {
  int x = 1;
  MY_ASSERT(x == 1, "x must be 1!");
  int y = 2;
  ASSERT0(y == 2, "y must be 2!");
  return 0;
}
```

И для отключения нужно добавить макрос отключения

```
#define DISALBE_MY_ASSERT
#include "my_assert.h"

int main() {
  int x = 1;
  MY_ASSERT(x == 1, "x must be 1!"); // is disabled
  int y = 2;
  ASSERT0(y == 2, "y must be 2!");
  return 0;
}
```

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

```
// file my_check.h
#include <cassert>
#include <exception>

#ifdef MYASSERT
#define MyCheck(expr, msg) assert(expr && msg)
#else
#ifdef MYEXCEPT
#define MyCheck(expr, msg)       \
  if (!expr)                     \
  throw std::runtime_error(msg)
#else
#define MyCheck(expr, msg) ((void)0)
#endif
#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!");
}
```

1.3.4 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 die; // 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.

1.4 Ошибки исполнения среды

1.4.1 Error Codes

1.4.2 `std::expected`

1.4.3 Exceptions

Зачем?

Как использовать Теперь поговорим о ситуации, когда нужны исключения. Причина такой ошибки – программа хотела, система не смогла. Например, вы пытаетесь запросить у системы ресурс, а она вам его не дает. Обычно приводят такой пример: вы пытаетесь запросить память под вектор и система отказывает, потому что нет памяти. Это не лучший пример вот почему

1. 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 работать вашей программы. Смысла тут делать что-то нет.

2. Память запрашивает чуть ли не каждый кусок кода в любой библиотеке. И что же, тогда придется проверять на исключение каждое обращение к операционной системе? В таком случае программа только и будет делать, что проверять исключения и ничего больше. Это очень глупо.

Давайте я постараюсь привести другой, более удачный пример обработки исключения. 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. То есть у системной функции, которую вы зовете есть возможность отказать вам в доступе 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 умираем по `terminate`.

Не злоупотреблять исключениями Тут я бы хотел сформулировать несколько соображений о том, как я вижу обработку исключений.

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.

2. Исключений никогда не должно быть в критической для производительности части. Более того, рекомендуется помечать 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.

1.5 Exceptions, Stack Unwinding и RAII

Простой пример Мы уже с вами говорили, что такое RAII в разделе **??**. Данный подход позволяет автоматизировать менеджмент ресурсов. Пока мы не говорим об исключениях, этот RAII кажется лишь удобной автоматизацией рутинной работы. Однако, как только вы включаете исключения в языке, то программа может выйти из любого scope в совершенно любой точке, а не только по `return`, потому что именно так работают исключения. Внутри любой функции исключение может возникнуть в любой строчке кода, а значит 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 строчке 6 вы явно выделяете память, а в строчке 7 бросается исключение и 8-я строчка никогда не вызовется. Как вообще работает код в функции `f`. Давайте опишу его по строчкам:

1. В строчке 6 происходит сразу несколько вещей.

   (a) Выделение памяти на стеке для указателя `x`.

   (b) Вызов оператора `new` для выделения памяти под переменную типа `int`, и запись по этому адресу значения `0`.

   (c) Пишем адрес памяти в переменную `x`.

2. В строчке 7 вызываем функцию `g`.

3. Функция `g` бросает исключение. В этом месте мы вываливаемся из функции `g` в строчку 7 внутри функции `f`.

4. 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 вызывается деструктор `x`, а так как это встроенный тип, то никакого деструктора нет и мы просто уменьшаем стек и выкидываем переменную `x` со стека.

5. Почистив стек мы выходим в больший scope из которого вызвана `f` и продолжаем вываливаться дальше.

Проблемы с исключениями У этого процесса есть две вещи о которых надо обязательно поговорить.

1. Безопасное выделение ресурсов. Чтобы безопасно выделить и удалить память, или любой другой ресурс мы вынуждены пользоваться специальными обертками, которые делают всю работу автоматически при выходе из scope. Для памяти такая обертка называется `std::unique_ptr`. Вот тут простейший пример обертки

```
1  class IntPtr {
2    IntPtr(int value) : ptr_(new int(value)) {}
3    IntPtr(const IntPtr) = delete;
4    IntPtr(IntPtr&& other) : ptr_(std::exchange(other.ptr_, nullptr)) {}
5    IntPtr& operator=(const IntPtr&) = delete;
6    IntPtr& operator=(IntPtr&& other) {
7      IntPtr tmp = std::move(other);
8      std::swap(ptr_, tmp.ptr_);
9      return *this;
10   }
11   ~IntPtr() {
12     delete ptr_;
13   }
14   int* ptr() const {
15     return ptr_;
16   }
17 private:
18   int* ptr_;
19 };
20
21 void f() {
22   IntPtr x(0);
23   throw std::runtime_error("Exception");
24 }
```

   Теперь в строчке 22 происходит выделение памяти в конструкторе `IntPtr`. А при выходе из `f` по исключению, во время stack unwinding вызовется деструктор для `x` и удалится выделенная память. Такой механизм позволяет гарантированно выполнять действия при выходе из любого scope.

2. Обратите внимание на тонкий момент в предыдущей схеме. А что если во время stack unwinding вы бросите исключение? По стандарту – это недопустимо. 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.

1.6 Исключения и конструирование объектов

Несколько ресурсов и исключения 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 потечет круче вашей подружки. Есть много костылей для этой проблемы, но правильное решение – обернуть каждый указатель в свой wrapper. Например так

```
struct IntPtrPair {
  IntPtrPair(int x, int y) : ptr1_(x), ptr2_(y) {}

  IntPtr ptr1_;
  IntPtr ptr2_;
};

IntPtrPair x(1, 2);
```

И теперь вам вообще не надо париться о деструкторах и прочих радостях. Все это за вас для каждого ресурса делает wrapper для работы с одним указателем. На практике, конечно, надо пользоваться библиотечными опциями, тут подойдет `std::unique_ptr`.

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. А для конструирования по частям как раз и создано наследование. Оно собирает объект в определенном порядке. Давайте приведу такой пример

```
1  struct A {
2    A(int x) : ptr_(std::make_unique<int>(x)) {}
3    std::unique_ptr<int> ptr_;
4  };
5
6  struct B : A {
7    A(int x)
8     : A(x),
9       ptr_(std::make_unique<int*>(A::ptr_.get()) {}
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 честно, то `std::unique_ptr` не самая высоко оптимизированная вещь.[2] Но для большинства ситуаций этой обертки хватает на ура. Правда деревья я бы на ее основе не собирал, как минимум убьете стэк в деструкторе или получите неописуемые тормоза.

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 wrapper, где в конструкторе вы запрашиваете ресурс, а в деструкторе освобождаете. Вот тут я могу привести пример того, как я оборачивал Windows нативный thread pool в exception safe обертку. Это уже не искусственный пример, где много разных взаимозависимых ресурсов требуют своего последовательного выделения или освобождения. Наследование позволяет это сделать удобным и контролируемым. Но надо пописать какой-то код. Обратите внимание, что в имплементации методов я даже позволяю себе бросать исключения в конструкторах, при наличии ошибки. Тогда все выделенные ресурсы в базовой части автоматом почистятся и не надо переживать на эту тему.

[2] Кто-то, читая это, сейчас должен был брызнуть слезами смеха и отчаяния от моей аккуратности в формулировках.

Зависимые ресурсы еще раз Напоследок хочу сказать еще одно замечание про реализацию выделения памяти под зависимые данные. Есть другая альтернатива

```
1  struct A {
2    A(int x)
3     : ptr1_(std::make_unique<int>(x)),
4       ptr2_(std::make_unique<int*>(ptr1_.get()) {}
5
6    std::unique_ptr<int> ptr1_;
7    std::unique_ptr<int*> ptr2_;
8  };
9
10 A x(1);
```

Плюс такого подхода – не вызываются лишние конструкторы, все делается в одном. При сложной цепочке вложенных конструкторов гипотетически можно потратить много на накладные расходы. Опять же, это все нужно измерять и я никогда не испытывал с этим проблемы, но быть они могут. Минус в том, что этот код хрупкий. Вы опираетесь на порядок расположения данных в коде. А именно, вариант ниже

```
1  struct A {
2    A(int x)
3     : ptr1_(std::make_unique<int>(x)),
4       ptr2_(std::make_unique<int*>(ptr1_.get())) {}
5
6    std::unique_ptr<int*> ptr2_;
7    std::unique_ptr<int> ptr1_;
8  };
9
10 A x(1); // incorrect object state
```

порадует вас ошибкой в run-time. Ваш объект будет находиться в некорректном состоянии. Дело в том, что данные конструируются не в том порядке, в каком они идут в списке инициализации (строки 3 и 4), а в каком они объявлены в классе (строки 6 и 7). Так что у вас сначала выполнится строка для инициализации `ptr2_` и в ней вы обратитесь к функции `get` неинициализированной `ptr1_` и получите `nullptr`. И только потом инициализируете `ptr1_`. Как вы видите, после отработы конструктора `ptr2_` не будет содержать адрес данных из `ptr1_`. И такие ошибки хрен найдешь. Причем совершить их очень легко – достаточно переставить данные. А со временем вы попросту забудете, что ваши данные были зависимы, ваш код обрастет дополнительными костылями и вы просто не заметите, что это именно эта ситуация. Причем, код скомпилируется, запустится и даже отработает, пока вы не упадете на каком-нибудь тонком тесте.