Написание кода на C++

Дима Трушин

2024 — 2025

Содержание

1 Обработка ошибок

1.1 Виды ошибок и доступные средства языка

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.

Механизмы языка 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.

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.

3. 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 **??**).

Главное отличие между этими двумя методами заключается в том, что `assert` – это безальтернативный механизм уронить вашу программу на первой ошибке. Исключения же позволяют как-то на них реагировать в штатном порядке, не завершая работу программы по `terminate`. В частности отсюда следует, что `assert` уж точно не подходит для реагирования на ошибки при обращении к ресурсам системы, потому что программа не должна завершаться по аварийно. А значит `assert` – это прежде всего механизм для борьбы с bug-ами. С исключениями не все так просто. Их можно использовать для обоих целей. Долгое время главным преимуществом исключений являлось возможность показать текущий стек вызовов. Однако начиная с C++23 появилась возможность использовать `std::stacktrace`. Он, правда, не на столько хорошо отображает информацию, как привычные для нас компиляторы. Тем не менее, в плане информативности оба метода сейчас практически одинаковые.

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).

[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`.

(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.

1.2 Ошибки программиста

assert 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.

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.

2. Ставьте `assert` в конце каждой мутирующей функции для проверки postcondition.

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`.

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.

5. Обязательно вставляйте `assert` перед обращением через указатель, он может быть нулевой. Другие методы могут содержать `assert` внутри себя, но если `operator->()` не перегружен (а для указателей он не может быть не перегружен), то вы не можете вставить `assert` внутрь, потому проверки надо делать до вызова функции.

6. НЕ делайте `assert` на ошибки, которые разрешимы на этапе компиляции (например в шаблонной магии), для этого используйте `static_assert`.

7. И конечно же не забывайте сообщить об ошибке текстом. Вам же будет проще.

Кастомные `assert` 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.

Exceptions TO DO

1.3 Ошибки исполнения среды

Exceptions 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.

1.4 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.

1.5 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

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(x), ptr2_(y) {}
3
4    IntPtr ptr1_;
5    IntPtr ptr2_;
6  };
7
8  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)) {}
```

```
  std::unique_ptr<int> ptr_;
};

struct B : A {
  A(int x)
   : A(x),
     ptr_(std::make_unique<int*>(A::ptr_.get()) {}
   std::unique_ptr<int*> ptr_;
};

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.

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

```
struct A {
  A(int x)
   : ptr1_(std::make_unique<int>(x)),
     ptr2_(std::make_unique<int*>(ptr1_.get()) {}

  std::unique_ptr<int> ptr1_;
  std::unique_ptr<int*> ptr2_;
};

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

```
struct A {
  A(int x)
   : ptr1_(std::make_unique<int>(x)),
     ptr2_(std::make_unique<int*>(ptr1_.get())) {}

  std::unique_ptr<int*> ptr2_;
  std::unique_ptr<int> ptr1_;
```

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

```
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.

2 Оптимизация и вокруг