Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2020

Содержание

Исключения

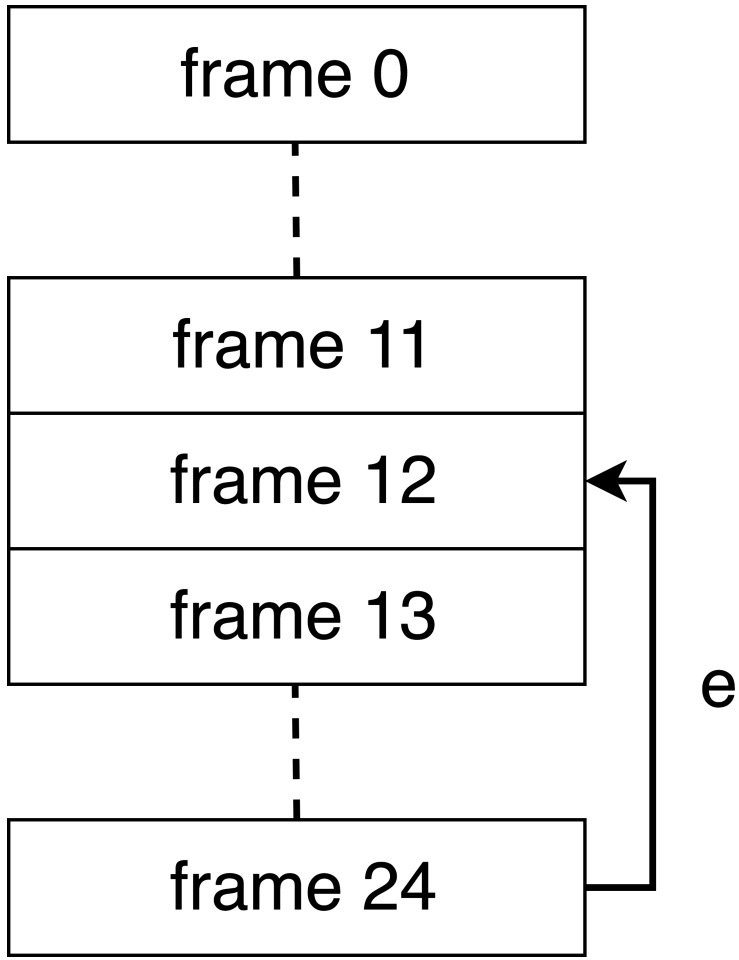
Обработка исключений и развёртывание стека

Спецификация исключений и гарантии безопасности

RAII

Некоторые вопросы безопасности в контексте комплексных объектов

Исключения



Для чего используются исключения

Функция не может:

- удовлетворить предусловие вызова другой функции, являющейся очередным шагом логики данной
- удовлетворить постусловие своей работы
- обеспечить сохранение инварианта, за который она ответственна

Доступные механизмы уведомления вызывающий код об ошибках

- коды возврата:
  - функция имеет возвращаемое значение и резервирует специальное значение(я) для индикации ошибок
  - функция имеет среди аргументов те, через которые может передать в вызывающий код индикацию ошибок
  - функция использует глобальные переменные для индикации ошибок
- исключения

Пример исключения

```
 1    template <class T>
 2    class PoolAllocator
 3    {
 4    public:
 5        T * allocate()
 6        {
 7            if (/* no space in this pool */) {
 8                throw std::bad_alloc{};
 9            }
10        }
11    };
12
13    void append(const X & x, std::vector<X, PoolAllocator<X>> & v)
14    {
15        try {
16            v.push_back(x);
17        }
18        catch (const std::bad_alloc & e) {
19            // report?
20        }
21    }
```

`throw`

1. `throw` *выражение*
2. `throw`

1. Из *выражения* конструируется объект исключения
   - если выражение задаёт локальный объект с автоматическим размещением, возможен вызов перемещающего вместо копирующего конструктора
   - возможна NRVO-подобная оптимизация

   затем управление передаётся в ближайший подходящий по типу обработчик исключения
2. Пере-выбрасывает исключение: если какое-то исключение уже обрабатывается, `throw` вызовет выход из текущего обработчика и передачу управления ближайшему следующему, уровнем выше, подходящему по типу. Объект исключения заново не создаётся.

try-catch

`try` *блок последовательность обработчиков*

Обработчик:

- `catch ( expr )` *блок*
- `catch (...)` *блок*

expr – не может быть rvalue ссылкой или неполным типом.

Выбор подходящего обработчика

- в одной `try-catch` инструкции обработчики проверяются последовательно, от первого к последнему
- если параметр обработчика объявлен ссылкой на тип **T**, то объект исключения должен иметь тот же тип, либо тип класса-потомка
- если параметр обработчика объявлен указателем на тип **T**, то объект исключения должен быть указателем на тот же тип или быть указателем, неявно приводимым к типу параметра обработчика
- иначе тип объекта исключения должен совпадать с типом параметра обработчика или быть его однозначным доступным потомком
- `catch (...)` перехватывает исключение любого типа
- если подходящего обработчика не найдено, то поиск продолжается уровнем выше
- если подходящего обработчика не найдено вообще, программа аварийно завершается

Пример выбора обработчика

```
1   class MyException : public std::exception
2   {
3   public:
4       const char * what() const
5       { return "MyException"; }
6   };
7
8   void f(int n)
9   {
10      if (n < 0) {
11          throw std::invalid_argument("Argument must be non-negative");
12      }
13      else {
14          throw MyException;
15      }
16  }
17
18  void g(int n)
19  {
20      try {
21          f(n);
22      }
23      catch (const std::logic_error & e) {
24          std::cerr << e.what() << std::endl;
25      }
26  }
27
28  int main()
29  {
30      try {
31          g(-1);
32          g(1);
33      }
34      catch (std::runtime_error e) {
35          std::cerr << "Runtime!" << std::endl;
36      }
37      catch (MyException e) {
38          std::cerr << e.what() << std::endl;
39      }
40  }
```

Содержание

Исключения

Обработка исключений и развёртывание стека

Спецификация исключений и гарантии безопасности

RAII

Некоторые вопросы безопасности в контексте комплексных объектов

Объект исключения

- не имеет определённого типа размещения
- тип = типу выражения в `throw`, cv-квалификаторы верхнего уровня отбрасываются
- является временным объектом, но связывается с lvalue ссылками в параметрах обработчиков
- существует до выхода из последнего вызванного обработчика, который не перебросил это исключение

Обработка исключения

В точке выброса исключения нормальное исполнение прерывается и

- производится поиск первого подходящего обработчика – пока он не найден производится развёртывание стека
- в `try` блоке инструкции, где найден подходящий обработчик, разрушаются все локальные автоматические переменные в порядке, обратном порядку их создания
- управление передаётся в найденный обработчик
- если выбрасывается другое исключение (до момента вызова обработчика или в процессе его работы), программа аварийно завершается
- после завершения блока обработчика, исполнение переходит к следующей за `try-catch` инструкции и восстанавливается нормальное исполнение программы

Развёртывание стека

Пока развёртывание не дошло до уровня с подходящим обработчиком:

- все автоматические переменные в данной функции разрушаются в порядке, обратном порядку их создания
- если текущая функция – конструктор или деструктор класса, то также разрушаются все полностью сконструированные поля этого объекта
- если текущая функция – делегирующий конструктор класса и делегируемый конструктор успешно завершился, то вызывается деструктор этого класса
- если текущая функция – конструктор, вызванный в рамках `new`, то также вызывается освобождение выделенной памяти
- исполнение текущей функции полностью завершается и процесс повторяется уровнем выше

Содержание

Исключения

Обработка исключений и развёртывание стека

Спецификация исключений и гарантии безопасности

RAII

Некоторые вопросы безопасности в контексте комплексных объектов

Гарантии безопасности

- никаких гарантий: после исключения программа оказывается в некорректном состоянии
- базовая гарантия: программа находится в валидном, но неопределённом состоянии
- строгая гарантия: состояние программы корректно и возвращено к состоянию до вызова функции, в которой произошло исключение
- гарантия отсутствия исключений: функция не выбрасывает наружу никаких исключений ни при каких условиях

Спецификация исключений

Тип функции включает в себя спецификацию исключений:

- либо функция не может приводить к выбросу исключений
- либо функция может приводить к выбросу исключений

Спецификацию можно задать явно с помощью `noexcept` в конце объявления функции.

Как и в случае с типом возвращаемого значения, спецификация исключений не участвует в разрешении перегрузки.

Функция, имеющая спецификацию “без выброса исключений” может вызывать код, выбрасывающий исключения, но если такое исключение выйдет за рамки этой функции, программа аварийно завершится.

noexcept

Спецификатор `noexcept`:

- `noexcept( expr )` – если *expr* вычисляется в `true`, то функция не может выбрасывать исключения
- `noexcept` – то же, что `noexcept(true)`

Оператор `noexcept( expr )` – вычисляется на этапе компиляции, результат `true`, если выражение не имеет потенциальных исключений.

```
1   void f() noexcept;
2   void g() noexcept(sizeof(int) < 8)
3   {
4   }
5   template <class T>
6   void h(T && x) noexcept(noexcept(T(std::declval<T>())));
```

Определение спецификации исключений для функции

- функция может приводить к выбросу исключений, если
  - объявлена с `noexcept(expr)` и `expr` вычисляется в `false`
  - объявлена без `noexcept` и не является
    - деструктором, кроме деструкторов класса чьи подобъекты имеют потенциально выбрасывающие деструкторы
    - методом класса, сгенерированным автоматически, если только вызов такого метода не может выбросить исключение
- в противном случае функция не может приводить к выбросу исключений

Определение спецификации исключений для выражения

Выражение имеет потенциальные исключения, если

- вызывает функцию, которая может выбрасывать исключения
- неявно вызывает такую фукнцию (например, в конце выражения вызывается деструктор временного объекта)
- является `throw` выражением
- является `dynamic_cast` выражением для полиморфной ссылки
- является `typeid` выражением
- содержит в себе подвыражение, имеющее потенциальные исключения

Содержание

Исключения

Обработка исключений и развёртывание стека

Спецификация исключений и гарантии безопасности

RAII

Некоторые вопросы безопасности в контексте комплексных объектов

RAII

- время жизни объектов
- автоматическое размещение
- порядок инициализации
- развёртывание стека

- ресурс связывается с объектом
- инициализация объекта $\equiv$ захват ресурса
- время жизни объекта $\equiv$ время доступности ресурса
- разрушение объекта $\equiv$ освобождение ресурса

Умные указатели

```
 1    void bad()
 2    {
 3        X * x = new X();
 4        ...
 5        delete x;
 6    }
 7
 8    void good()
 9    {
10        const auto px = std::make_unique<X>();
11        ...
12    } // x is implicitly deleted
```

Scope guards

```
 1    std::mutex m;
 2
 3    void bad()
 4    {
 5        m.lock();
 6        ...
 7        m.unlock();
 8    }
 9
10    void good()
11    {
12        std::lock_guard<std::mutex> g(m); // implicit lock
13        ...
14    } // implicit unlock
```

Реализация класса, инкапсулирующего ресурс

- ресурс захватывается в конструкторе, подготавливается к использованию, выброс исключения, если что-то пошло не так
- ресурс освобождается в деструкторе, освобождение не должно бросать исключений
- обычно такой класс не имеет операций копирования
- можно реализовать передачу владения ресурсом из одного контекста в другой с помощью конструктора перемещения / оператора перемещающего присваивания класса

Содержание

Исключения

Обработка исключений и развёртывание стека

Спецификация исключений и гарантии безопасности

RAII

Некоторые вопросы безопасности в контексте комплексных объектов

Инициализация

```
 1    class X
 2    {
 3        A * m_a = nullptr;
 4        B * m_b = nullptr;
 5        std::vector<C> m_c;
 6    public:
 7        X(std::size_t n, const C & c)
 8        {
 9            m_a = new A();
10            m_b = new B();
11            m_c.resize(n, c);
12        }
13    };
```

Более безопасная инициализация

```
1   class X
2   {
3       std::unique_ptr<A> m_a;
4       std::unique_ptr<B> m_b;
5       std::vector<C> m_c;
6   public:
7       X(std::size_t n, const C & c)
8           : m_a(std::make_unique<A>())
9           , m_b(std::make_unique<B>())
10          , m_c(n, c)
11      {
12      }
13  };
```

Копирование

```
1    X::X(const X & other)
2        : m_a(std::make_unique<A>(*other.m_a))
3        , m_b(std::make_unique<B>(*other.m_b))
4        , m_c(other.m_c)
5    {
6    }
7
8    X & X::operator = (const X & other)
9    {
10       m_a = std::make_unique<A>(*other.m_a);
11       m_b = std::make_unique<B>(*other.m_b);
12       m_c = other.m_c;
13   }
```

Более безопасное копирование

```
1    X & X::operator = (const X & other)
2    {
3        using std::swap;
4        X tmp(other);
5        swap(m_a, tmp.m_a);
6        swap(m_b, tmp.m_b);
7        swap(m_c, tmp.m_c);
8    }
```

Объединение операторов копирующего и перемещающего присваивания

```
1    X & X::operator = (X tmp)
2    {
3        using std::swap;
4        swap(m_a, tmp.m_a);
5        swap(m_b, tmp.m_b);
6        swap(m_c, tmp.m_c);
7    }
```

Извлечение элемента из коллекции

```
 1    template <class T>
 2    class Queue
 3    {
 4        std::vector<T> m_data;
 5    public:
 6        T dequeue()
 7        {
 8            using std::swap;
 9            swap(m_data.front(), m_data.back());
10            T res = std::move(m_data.back());
11            m_data.pop_back();
12            return res;
13        }
14    };
```

Более безопасное извлечение элемента из коллекции

```
 1    template <class T>
 2    T & Queue<T>::top()
 3    {
 4        return m_data.front();
 5    }
 6
 7    template <class T>
 8    void Queue<T>::pop()
 9    {
10        using std::swap;
11        swap(m_data.front(), m_data.back());
12        m_data.pop_back();
13    }
```