Лекция 5. Исключения

Способы обработки ошибок

- Ошибка повод для std::terminate
- Вернуть признак ошибки и выставить глобальный код ошибки
- Вернуть код ошибки
- Бросить исключение:
  - его невозможно проигнорировать;
  - распространяется автоматически;
  - выносит обработку из основного потока выполнения
  - незаменимо в конструкторах или операторах.

try...catch

```
1.  try
2.  {
3.      // throws explicit or implicit
4.      throw exception_type(/*...*/);
5.      //...
6.  }
7.  catch(std::exception const& err)
8.  {/*...*/}
9.  catch(...)
10. {
11.     /*...*/
12.     throw;
13. }
14.
15. //...
16. Type::Type(/*...*/)
17. try
18.     : field1_(/*...*/)
19.     , field2_(/*...*/)
20. {
21.     /*...*/
22. }
23. catch(std::exception const&)
24. {
25.     /*some logging*/
26.     throw;
27. }
```

Типы исключений

- Исключение может быть любым типом (хоть int или const char*)
- Разумно делать класс исключения, наследуя его от std::exception (и реализовать what)
- Есть стандартные исключения, например, std::runtime_error, logic_error (<stdexcept>)
- Перехват по
  - точному соответствию типа;
  - ссылке на базовый класс;
  - по приводимому указателю.

Использование RAII

```
1.  double sqrt(double value)
2.  {
3.      if (value < 0) throw runtime_error("negative");
4.      /*...*/
5.  }
6.
7.  double* make_roots(double* values, size_t size)
8.  {
9.      double* roots = new double[size];
10.     transform(values, values + size, roots, sqrt);
11.     return roots;
12. }
```

- Есть ли проблемы в этом коде?

Использование RAII (2)

```
1.  double* make_roots(double* values, size_t size) /*case 1*/
2.  {
3.      double* roots = new double[size];
4.      try
5.      {
6.          transform(values, values + size, roots, sqrt);
7.      }
8.      catch(std::exception const& err)
9.      {
10.         delete [] roots;
11.         cerr << err.what() << endl;
12.         throw;
13.     }
14.     return roots;
15. }
16.
17. double* make_roots(double* values, size_t size) /*case 2*/
18. {
19.     unique_ptr<double[]> roots(new double[size]);
20.     transform(values, values + size, roots.get(), sqrt);
21.     return roots.release();
22. }
```

Передача параметров

```
1.  void foo(T* a, U* b);
2.  /*...*/
3.  foo(new T(/*...*/), new U(/*...*/));
4.
5.  typedef shared_ptr<T> T_ptr;
6.  typedef shared_ptr<U> U_ptr;
7.
8.  void foo(T_ptr a, U_ptr b);
9.  /*...*/
10. void foo(T_ptr(new T(/*...*/)), U_ptr(new U(/*...*/)));
```

- Исправляет ли вызов в (10) ошибочный вызов в (3)?

Передача параметров(2)

```
1.  void foo(T_ptr a, U_ptr b);
2.
3.  //(1)
4.  T_ptr a(new T(/*...*/));
5.  U_ptr b(new U(/*...*/));
6.  foo(a, b);
7.
8.  // (2)
9.  foo(make_shared<T>(/*...*/), make_shared<U>(/*...*/));
```

- Если для соответствующего RAII класса нет фабрики, используйте способ (1).

Класс stack

- Попробуем разработать безопасную реализацию класса stack

```
1.  template<class T> struct stack
2.  {
3.      explicit stack(size_t def_size = 10);
4.     ~stack();
5.
6.     T    pop();
7.     void push(T const&);
8.     /*...*/
9.  private:
10.     size_t  size_; // buffer size
11.     size_t  used_; // number of objects
12.     T*      buf_ ; // main buffer
13. };
```

Класс stack, конструктор

```
1.  template<class T>
2.  stack<T>::stack(size_t def_size)
3.      : size_(def_size)
4.      , used_(0)
5.      , buf_ (new T[def_size])
6.  {}
```

- Такой код нейтрален – любое исключение передается дальше.
- Не вызывает утечек памяти.
- Независимо от наличия исключений стек остается согласованным (либо несозданным).

Гарантии безопасности исключений

- **Базовая гарантия:** даже при наличии генерируемых объектом класса T (или иных) исключений утечки в классе stack отсутствуют и остается согласованность состояния объекта.
- **Строгая гарантия:** если операция прекращается из-за генерации исключения, состояние программы остается неизменным (rollback).
- **Гарантия отсутствия исключений:** функция не генерирует исключений ни при каких обстоятельствах.

Гарантия отсутствия исключений

- Деструкторы:

```
1.    Type array [42];
```

Что будет, если 13-й объект при создании бросит исключение, а затем 12-й при удалении?

- Функция swap

Копирование стека

- Воспользуемся вспомогательной функцией:

```
1.  template<class T>
2.  T* new_copy(const T* src, size_t src_size, size_t
3.  dst_size)
4.  {
5.      // could be made via unique_ptr, but no need here
6.      T* dst = new T[dst_size];
7.      try
8.      {
9.          copy(src, src + src_size, dst);
10.     }
11.     catch(...)
12.     {
13.         delete[] dst;
14.         throw;
15.     }
16.
18.     return dst;
19. }
```

Копирование стека (2)

```
1.  template<class T>
2.  stack<T>::stack(stack const& o)
3.      : size_(o.size_)
4.      , used_(o.used_)
5.      , buf_ (new_copy(o.buf_, o.src_size, o.dst_size))
6.  {
7.  }
8.
9.  template<class T>
10. stack<T>& stack<T>::operator=(stack<T> const& o)
11. {
12.     if (this != &o)
13.     {
14.         T* cp = new_copy(o.buf_, o.src_size, o.dst_size);
15.         delete[] buf_; // no more exceptions
16.         buf_  = cp;
18.         size_ = o.size_;
19.         used_ = o.used_;
20.     }
21.     return *this;
22. }
```

push / pop

```
1.  template<class T>
2.  void stack<T>::push(T const& t)
3.  {
4.      if (used_ == size_)
5.      {
6.          size_t sz = 2 * size_ + 1;
7.          unique_ptr<T[]> cp(new_copy(buf_, size_, sz));
8.          cp[used_] = t;
9.
10.         delete [] buf_; // now ok
11.         buf_  = cp.release();
12.         size_ = sz;
13.     }
14.     else
15.         buf_[used_] = t;
16.     ++used_;
17. }
18.
19. template<class T>
20. T stack<T>::pop()
21. {
22.     if (empty()) throw /*...*/;
23.     T res = buf[used_ - 1];
24.     --used_;
25.     return res; // what to do if no move-semantics?
26. }
```

pop / top

- Разделим pop на pop и top:

```
1.  template<class T>
2.  T const& stack<T>::top() const
3.  {
4.      if (empty()) throw /*...*/;
5.      return buf[used_ - 1];
6.  }
7.
8.  template<class T>
9.  void stack<T>::pop()
10. {
11.     if (empty()) throw /*...*/;
12.     --used_; // is it enough?
13. }
```

Гарантии и требования stack

- Требования:
  - наличие конструктора по умолчанию;
  - наличие копирующего конструктора;
  - деструктор без исключений;
  - оператор присваивания.
- Гарантии:
  - строгая гарантия при удовлетворении требований.

Уменьшение требований stack

- Для начала рассмотрим функции:

```
 1. template<class T1, class T2>
 2. void construct(T1* p, T2 const& value)
 3. { new (p) T1(value); }
 4.
 5. template<class T>
 6. void destroy(T* p)
 7. { p->~T();}
 8.
 9. template<class fwd_it>
10. void destroy(fwd_it beg, fwd_it end)
11. {
12.     while (beg != end)
13.     {
14.         destroy(&*beg);
15.         ++beg;
16.     }
17. }
```

stack_impl

- Сделаем вспомогательный класс:

```
1.  stack_impl::stack_impl(size_t size)
2.      : buf_ (static_cast<T*>(size == 0
3.          ? nullptr
4.          : aligned_alloc(alignof(T), size * sizeof(T))
5.      , size_(size)
6.      , used_(0)
7.  {}
8.
9.  stack_impl::~stack_impl()
10. {
11.     destroy (buf_, buf_ + used_);
12.     std::free(buf_);
    }
```

Новый вариант stack

```
1.  struct stack
2.  {
3.      /**/
4.      stack_impl impl_;
5.  }
6.
7.  stack::stack(size_t def_size)
8.      : impl_(def_size)
9.  {}
10.
11. stack::stack(stack const& o)
12.     : impl_(o.impl_.used_)
13. {
14.     while (impl_.used_ < o.impl_.used_)
15.     {
16.         construct(impl_. buf_ + impl_.used_,
17.                   o.impl_.buf_ + impl_.used_);
18.         ++impl_.used_;
19.     }
20. }
21.
22. stack& stack::operator=(stack o)
23. {
24.     swap(*this, o);
25.     return *this;
26. }
```

Вспомним гарантии безопасности исключений

- **Базовая гарантия:** даже при наличии генерируемых объектом класса T (или иных) исключений утечки в классе stack отсутствуют + согласованность.
- **Строгая гарантия:** если операция прекращается из-за генерации исключения, состояние программы остается неизменным (rollback).
- **Гарантия отсутствия исключений:** функция не генерирует исключений ни при каких обстоятельствах.

Ключевое правило работы с исключениями

- В каждой функции следует **собрать весь код**, который может привести к генерации исключений, и **выполнить его отдельно** безопасным способом.
- **После** этого зная, что все «тяжелая» работа уже проделана, **можно изменять состояние** программы способом, не генерирующим исключения.

Спецификация исключений

```
1.  void translate(std::string const& sentence)
2.      throw(unknown_word, bad_grammar);
3.
4.  void translate() throw ();
5.
6.  // usual way
7.  void translate(); // throw
```

- Если будет брошено исключение, не соответствующее спецификации, вызовется `std::unexpected`.
- Требуется при перегрузке виртуальной функции со спецификацией.
- Реализует лишь runtime проверку, дает overhead ко времени работы.
- Вывод: не используйте ее. Не даст ни проверки в compile time, ни оптимизации. А даже перехваченный вызов вызов `std::unexpected` мало чем поможет.
- Deprecated начиная с C++11.

noexcept operator (C++11)

```
1.  template<class T>
2.  struct has_no_except_copy
3.  {
4.      enum {value = noexcept(T(*(T*)0))};
5.  };
```

- Обеспечивает проверку в compile time
- Как и `sizeof` не вычисляет выражение, а проверяет отсутсвие:
  - вызовов функций, не имеющих объявления `noexcept`
  - явных `throw`
  - преобразований `dynamic_cast` для ссылок
  - вывозов `typeid` для выражений полиморфного типа.
- Чаще всего используется в паре со спецификацией `noexcept`

noexcept спецификация (C++11)

```
1.  struct my_type
2.  {
3.      my_type (my_type&&) noexcept;
4.
5.      void self_assign(my_type& rhs)
6.          noexcept(noexcept(rhs = rhs));
7.  }
```

- Не выполняет статическую проверку, а лишь декларирует отсутствие исключений.
- Позволяет выполнить оптимизацию при компиляции.
- Требуется для `move` конструкторов, чтобы STL контейнеры не копировали объекты (в GCC, но пока не в MSVS).
- Любое непойманное исключение приводит к моментальному `std::terminate`
- Заменяет пустой спецификатор `throw` ()

STL контейнеры

- Все итераторы безопасны и могут копироваться без исключений
- Все стандартные контейнеры реализуют как минимум базовую гарантию. Почти все – строгую. Исключения: `vector` и `deque`, а также множественный `insert`. Они строгобезопасны, если копирование и move-копирование не генерируют исключений.
- Требование от типов: бессбойные деструкторы.

Вопросы?