ЛЕКЦИЯ 20
БЕЗОПАСНОСТЬ ИСКЛЮЧЕНИЙ

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ЛЕКТОР ФУРМАВНИН С.А.

ИНТЕРЛЮДИЯ: NOEXCEPT

Специальное ключевое слово noexcept документирует гарантию бессбойности для кода

```
void swap(MyVector &rhs) noexcept {
  std::swap(arr_, rhs.arr_);
  std::swap(size_, rhs.size_);
  std::swap(used_, rhs.used_);
}
```

- При оптимизациях компилятор будет уверен, что исключений не будет
- Если они все-таки вылетят, то это сразу `std::terminate`
- Вы не должны употреблять `noexcept` там, где исключения возможны

ЛИНИЯ КАЛБА

```
class MyVector {
  double *arr_ = nullptr;
  size_t size_, used_ = 0;
public:
 void swap(MyVector &rhs) noexcept;
  MyVector& operator=(const MyVector &rhs) {
    if (this == &rhs) return *this;
    MyVector tmp(rhs); // тут мы можем бросить исключение
```

```
    swap(tmp); // тут мы меняем состояние класса
    return *this;
  }
```

Это дает строгую гарантию по присваиванию

ЛИНИЯ КАЛБА

При проектировании очень полезно провести в уме эту линию

```
void push(double new_elem) {
  if (used_ == size_) {
    MyVector tmp(size_ * 2 + 1);
    while(tmp.size() < used_)
      tmp.push(arr_[tmp.size()]);
    tmp.push(new_elem);
    swap(*this, tmp); // операция noexcept
    return;
  }
  // и так далее
```

Выше этой линии инварианты класса неизменны

Ниже этой линии операции не кидают исключений

УСЛОВНЫЙ NOEXCEPT

Некоторые функции непонятно аннотировать `noexcept` или нет?

```
S copy (const S &original) /* noexcept? */ {
  return original;
}
```

ОПЕРАТОР NOEXCEPT

Оценивает каждую функцию, задействованную в выражении, но не вычисляет выражение

```
struct ThrowingCtor {ThrowingCtor(){}};
void foo(ThrowingCtor) noexcept;
void foo(int) noexcept;
assert(noexcept(foo(1)) == true);
assert(noexcept(ThrowingCtor{}) == false);
```

Возвращает `false` для constant expressions

ОБСУЖДЕНИЕ

Возможна критика: что если деструктор выбросит исключение? Попробуем от этого защититься

```
void destroy(FwdIter first, FwdIter last) {
  while(first++ != last)
    try {
      destroy(&*first);
    }
    catch(...) {
      // что тут делать?
    }
}
```

ПРАВИЛО ДЛЯ ДЕСТРУКТОРОВ

- Исключения не должны покидать деструктор
- По стандарту исключение, покинувшее деструктор, если при этом остались необработанные исключения, приводит к вызову std::terminate и завершению программы.

ОБСУЖДЕНИЕ: NOEXCEPT(FALSE)

- Любой деструктор по умолчанию `noexcept`
- Одним из способов позволить исключениям покидать деструктор является его пометка `noexcept(false)`
- Вы должны быть осторожны, помечая так деструкторы, потому что деструктор сам по себе используется в процессе размотки стека
- Вы можете проверить внутри деструктора идет ли размотка стека посредством вызова `std::uncaught_exceptions()`

ИЗСЛЕЧЕНИЕ ИЗ МАССИВА

- Безопасен ли код относительно исключений?

```
class MyVector {
  S* arr_ = nullptr;
  size_t size_, used_ = 0;
public:
  S pop() {
    if (used_ <= 0) throw underflow{};
    S result = arr_[used_ - 1];
    used_ -= 1;
    return result;
}
```

ВНЕЗАПНАЯ ПРОБЛЕМА

- Кажется, что все хорошо
- Но что произойдет в точке использования?

```
MyVector v;
// тут много кода
S s = v.pop(); // исключение при копировании в s
```

- Тогда окажется, что объект уже удален, но по месту назначения не пришел и потерян навсегда

ИЗСЛЕЧЕНИЕ ИЗ МАССИВА V2

- Тут правильное проектирование страхует от проблем

```
class MyVector {
  S* arr_ = nullptr;
  size_t size_, used_ = 0;
public:
  S top() {
    if (used_ <= 0) throw outofbonds{};
    return arr_[used_ - 1];
}
  void pop {
    if (used_ <= 0) throw underflow{};
    used -= 1;
}
```

ОБСУЖДЕНИЕ

- Оказывается, безопасность исключений влияет на проектирование!!!
- Если это так, то почему бы сразу не спроектировать нечто, что нам удобно будет делать безопасным?
- Удивительно, но для этого нам надо будет посмотреть на тонкости работы с памятью

ГЛОБАЛЬНЫЕ ОПЕРАТОРЫ

- В языке C для выделения памяти служат функции malloc и free

void *p = malloc(10);
free(p);

- В языке C++ этим занимаются операторы new и delete
- При этом, в отличие от, скажем, оператора +, у них есть глобальные формы
- Когда вы пишете new-expression для встроенного типа, он будет истолкован, именно как вызов глобального оператора

```
int *n = new int(5); // выделение + конструирование
n = (int *) ::operator new(sizeof(int)); // только выделение
```

ГЛОБАЛЬНЫЕ ОПЕРАТОРЫ

- Вы можете переопределить глобальные операторы и изменить поведение всех классов, которые ими пользуются

```
void *operator new(std::size_t n) {
  void *p = malloc(n); if(!p) throw std::bad_alloc{};
  printf("Alloc: %p, size is %zu\n", p, n);
  return p;
}
```

Теперь это мы можем увидеть на экране при создании списка из одного элемента?

```
std::list<int> l;
l.push_back(42);
```

ОБСУЖДЕНИЕ

- Мы отделяем вызов конструкторов от выделения памяти, но что если конструктор выбросит исключение?

```
struct S {
  S(); // десятый конструктор кинет исключение
  ~S();
};
S *arr = new S[20];
```

Сколько тут будет конструкторов и деструкторов, если мы знаем, что `new[]` дает строгую гарантию безопасности?

ФОРМЫ ГЛОБАЛЬНЫХ ОПЕРАТОРОВ

- Основные формы все в чём-то похожи на malloc

```
void *operator new(std::size_t);
void operator delete(void*) noexcept;
void *operator new[](std::size_t);
void operator delete[](void*) noexcept;
```

- Предусмотрены также дополнительные варианты с семантикой noexcept

```
void *operator new(std::size_t, const std::nothrow_t&) noexcept;
void *operator new[](std::size_t, const std::nothrow_t&) noexcept;
```

- Пока что должно быть не слишком понятно, как их использовать

НЕБРОСАЮЩИЙ NEW

- Если для `new-expression` не передано аргументов, она раскрывается просто

```
p = new int{42};
p = (int*) ::operator new(sizeof(int)); *p = 42;
```

- Если аргументы переданы, они ставятся в конец глобального опреатора

```
p = new (nothrow) int{42};
p = (int*) ::operator new(sizeof(int), nothrow); *p = 42;
```

- Специальный аргумент `std::nothrow` типа `std::nothrow_t` показывает, что мы не хотим бросать исключение
- Тогда нам надо возвращать нулевой указатель при неудаче

РАЗМЕЩАЮЩИЙ NEW

- Поскольку аллокация/деаллокация это операторы, они могут быть переопределены
- Но есть непереопределяемый глобальный оператор

```
void* operator new(std::size_t size, void* ptr) noexcept;
void* operator new[](std::size_t size, void* ptr) noexcept;
```

- Он называется размещающим `new` и ему не соответствует никакого `delete`, потому что всё, что он делает – это размещает объект в выделенной(сырой) памяти

РАБОТА С РАЗМЕЩАЮЩИМ NEW

- Работа с памятью отделена от работы с объектом памяти

```
void *raw = ::operator new(sizeof(Widget), std::nothrow);
if (!raw) { обработка }
Widget *w = new (raw) Widget;
// тут использование и…
w->~Widget();
::operator delete(raw);
```

- Может ли это помочь проектированию безопасных контейнеров?

ПЕРЕОПРЕДЕЛЕНИЕ NEW И DELETE

- Замечательным свойством `new` и `delete` является возможность переопределить их не глобально, а на уровне своего класса

```
struct Widget {
  static void *operator new(std::size_t n);
  static void operator delete(void *mem) noexcept;
};
```

- Теперь для класса `Widget` будут использоваться его собственные операторы, а не глобальные
- При этом, в отличии от глобального, размещающий `new` тоже может быть переопределен

РАБОТА С ПОЛЬЗОВАТЕЛЬСКИМ КЛАССОМ

- new с исключениями при исчерпании памяти

```
Widget *w = new Widget; // возможно bad_alloc
```

- new с возвратом нулевого указателя

```
Widget *w = new (std::nothrow) Widget;
if (!w) { обработка }
```

- размещающий new

```
void *raw = ::operator new(sizeof(Widget)); // возможно bad_alloc
// только конструирование в готовой памяти
Widget *w = new (raw) Widget;
```

ОБСУЖДЕНИЕ

Что вы думаете о таком операторе присваивания?

```
T& T::operator=(T const& x) {
  if (this != &x) {
    this->~T();
    new (this) T(x);
  }
  return *this;
}
```

ОБСУЖДЕНИЕ (STEPANOV ASSIGNMENT)

Что вы думаете о таком операторе присваивания?

```
T& T::operator=(T const& x) {
  if (this != &x) {
    this->~T();
    new (this) T(x); // исключение тут и дальше dtor
  }
  return *this;
}
```

- Алекс Степанов написал его в одной из первых реализаций `std::vector` и эта ошибка там **была незамеченной 6 лет**!

ОТДЕЛЕННАЯ РЕАЛИЗАЦИЯ

Идея для проектирования ваших классов с учетом исключений – это разделить функциональность:

- Класс, работающий с сырой памятью
- Использующий объекты этого класса внешний класс, работающий с типизированным содержимым

Для этого часто используется управление памятью вручную через нестандартные формы `new` и `delete`

ОБСУЖДЕНИЕ

Что можно сказать о возможных исключениях в следующем коде, демонстрирующем содержимое forward-итерируемого контейнера?

```
void destroy(FwdIter first, FwdIter last) {
  while (first != last)
    destroy(&*first++);
}
```

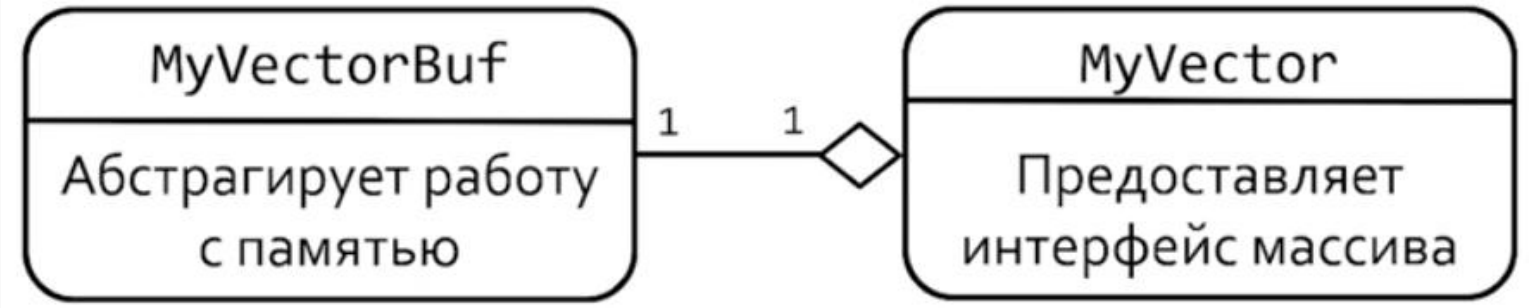
ОБСУЖДЕНИЕ

Возможна критика: что если деструктор выбросит исключение? Попробуем от этого защититься…

```
void destroy(FwdIter first, FwdIter last) {
  while (first != last)
    try {
      destroy(&*first++);
    }
    catch(...) {
      // что здесь делать?
    }
}
```

ОБЩИЙ ВЫВОД

Проектирование с использованием исключений в итоге позволяет упростить и улучшить код, структурируя его с четким распределением ответственности



В реальной libstdc++ вектор тоже будет устроен по такому принципу

ОБСУЖДЕНИЕ

Приведенный ранее метод push не очень эффективен

```
void push(double new_elem) {
  if (used_ == size_) {
    MyVector tmp(size_ * 2 + 1);
    while(tmp.size() < used_)
      tmp.push(arr_[tmp.size()]); // копирование
    tmp.push(new_elem);
    swap(*this, tmp); // операция noexcept
    return;
  }
```

Можем ли мы вместо этого использовать перемещение?

ПЕРВАЯ ПРОБЛЕМА: КОНСТАНТНОСТЬ

Нам придется немного дублировать, чтобы не снимать константность

```
void MyVector::push(const S& s) { S s2(s); push(move(s2));}

void MyVector::push(S&& new_elem) {
  if (used_ == size_) {
    MyVector tmp(size_ * 2 + 1);
    while(tmp.size() < used_)
      tmp.push(std::move(arr_[tmp.size()])); // перемещение
    tmp.push(std::move(new_elem));
    swap(*this, tmp); // операция noexcept
    return;
  }
```

Тут все хорошо?

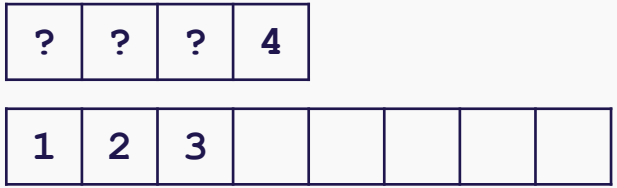
ВТОРАЯ ПРОБЛЕМА: ЛИНИЯ КАЛБА

Идея сделать его более эффективным использует move

Но это порождает проблемы: мы портим состояние arr

```
void MyVector::push(S&& new_elem) {
  if (used_ == size_) {
    MyVector tmp(size_ * 2 + 1);
    while(tmp.size() < used_)
      tmp.push(std::move(arr_[tmp.size()])); // если throw?
    tmp.push(std::move(new_elem));
    swap(*this, tmp); // операция noexcept
    return;
  }
```

Тут все хорошо?



РЕШЕНИЕ

Перемещающие конструктор и присваивание не должны бросать исключений

```
MyVector(MyVector &&rhs) noexcept = default;
MyVector& operator=(MyVector &&rhs) noexcept = default;
```

При этом если они неправильные или их нет, помещение в контейнер становится менее эффективным

```
void MyVector::push(const S& s) {
  if (std::is_nothrow_move_assignable<T>::value)
    push_move(s);
  else
    push_copy(s);
}
```

СМЕЩЕНИЕ ЛИНИИ КАЛБА

- Случай с копированием

```
MyVector tmp(size_ * 2 + 1);
while(tmp.size() < used_) tmp.push(arr_[tmp.size()]);
tmp.push(s);
```

```
swap(*this, tmp);
```

- Случай с перемещением

```
MyVector tmp(size_ * 2 + 1);
```

```
while(tmp.size() < used_) tmp.push(std::move(arr_[tmp.size()]));
tmp.push(s);
swap(*this, tmp);
```

ОБСУЖДЕНИЕ

- Исключения влияют на проектирование
- Использование перемещающих конструкторов влияет на проектирование
- Кажется, пришло время обсудить проектирование

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
2. Tom Cargill, Exception handling: a false sense of security, C++ Report 1994
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. David Abrahams, Exception-safety in generic components, 1998
5. Herb Sutter, Exceptional C++: 47 endineering puzzles, programming problems, and solutions, Addison-Wesley, 2000
6. John Calb, Exception Safe Code(3 parts), CppCon 2014
7. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.
8. Дональд Кнут, Искусство программирования. Том 2. Получисленные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 743 с.
9. Дональд Кнут, Искусство программирования. Том 3. Сортировка и поиск / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 767 с.