Vector своими руками — конспект темы

Размещающий оператор new

Настоящий класс `vector` конструирует свои объекты в сырой памяти только при необходимости. Сырая память — область памяти, которой владеет `vector` и в которой пока не создано ни одного объекта.

Размещающий оператор `new` позволяет вручную сконструировать объект в ранее выделенной сырой области памяти. Синтаксис:

```
new (адрес) Класс(параметры конструктора)
```

Он вызывает конструктор класса `Класс` по адресу `адрес` и передаёт ему `параметры конструктора`. `Адрес` должен иметь выравнивание, нужное для типа `Класс`.

Как только созданный так объект становится не нужен, его нужно удалить, явно вызвав деструктор:

```
Obj* obj = ...;
...
obj.~Obj();
```

Когда создаёте объекты размещающим оператором `new`, всегда используйте явный вызов деструктора.

Буфер, нужный для создания объекта, необязательно должен размещаться в области стека. Его можно выделить динамически. Чтобы выделить сырой массив данных в куче, вызывают функцию `operator new`. У неё есть несколько перегрузок. В частности:

```
void* operator new(std::size_t size);
```

Функция выделяет `size` байт в динамической памяти и возвращает указатель на эту выделенную область. Указатели типа `void*` могут ссылаться на данные произвольного типа.

Чтобы вернуть сырую память обратно в кучу, используют функцию `operator delete`. Версия функции, которая используется в курсе:

```
void operator delete(void* ptr) noexcept;
```

Функции std::uninitialized_*

В стандартной библиотеке есть функции, которые создают и удаляют группы объектов в неинициализированной области памяти точь-в-точь как код вашего класса `Vector`. Только реализация вектора упрощается. Функции объявлены в файле `<memory>`:

- `uninitialized_copy` и `uninitialized_copy_n`
- `uninitialized_fill` и `uninitialized_fill_n`
- `uninitialized_move` и `uninitialized_move_n`
- `uninitialized_default_construct` и `uninitialized_default_construct_n`
- `uninitialized_value_construct` и `uninitialized_value_construct_n`
- `destroy_at`
- `destroy` и `destroy_n`
- `construct_at`

Перемещайте элементы, только если соблюдается хотя бы одно из условий:

- конструктор перемещения типа `T` не выбрасывает исключений;
- тип `T` не имеет копирующего конструктора.

Реализация методов Resize, PushBack и PopBack

Метод `Resize` изменяет количество элементов в векторе.

Возможны два сценария:

- уменьшение размера вектора,

- увеличение размера вектора.

При уменьшении размера вектора нужно удалить лишние элементы вектора, вызвав их деструкторы, а затем изменить размер вектора:

Когда метод `Resize` должен увеличить размер вектора, сначала нужно убедиться, что вектору достаточно памяти для новых элементов. Самый простой способ так сделать — вызвать метод `Reserve`. Затем новые элементы нужно проинициализировать, используя функцию `uninitialized_value_construct_n`. В конце, когда элементы сконструированы, нужно изменить размер вектора.

Метод `PushBack` добавляет новое значение в конец вектора. При нехватке памяти стандартный `vector` увеличивает вместимость в кратное число раз.

Надёжная реализация метода `PushBack` корректно вставляет в вектор объекты, находящиеся вне вектора и внутри него. Если вставка в вектор сопровождается реаллокацией его элементов, нужно сначала завершить вставку элемента в новую область памяти. И только потом удалять объекты в старой области.

Метод `PopBack` разрушает последний элемент вектора и уменьшает размер вектора на единицу. Вызов `PopBack` на пустом векторе приводит к неопределённому поведению.

Variadic templates

Variadic templates — шаблоны с переменным числом параметров или **вариативные шаблоны**. Они позволяют создать шаблон функции или класса, принимающий любое число параметров произвольных типов.

Классы `std::tuple` и `std::variant`, функции `std::tie`, `std::make_unique` — это вариативные шаблоны.

Вариативный шаблон объявляется так:

```
// Вариативный шаблон класса
template <typename... Types> // Types — пакет параметров шаблона
class Tuple{};

// Вариативный шаблон функции
template <typename... Types> // Types — пакет параметров шаблона
void Fn(Types... values) {   // values — пакет параметров функции
}
```

Выражение свёртки, или **fold expression** позволяет обработать пакет параметров шаблона, не прибегая к рекурсии.

Реализация метода EmplaceBack

Метод `EmplaceBack` возвращает ссылку на добавленный элемент вектора. Это удобно для использования объекта сразу после его конструирования:

```
using namespace std;

int main() {
    Vector<Cats> cats;
    cats.EmplaceBack("Tom"s, 5).SayMeow();
}
```

Метод `EmplaceBack` может вызвать любые конструкторы типа `T`, в том числе объявленные `explicit`. В некоторых случаях это может создать проблемы, которые были бы невозможны при использовании `PushBack`. Например, когда внутри вектора хранятся указатели `unique_ptr`:

```
vector<unique_ptr<Shape>> shapes;
Rectangle r;

// Код скомпилируется без ошибок и предупреждений, но созданный unique_ptr
// будет ссылаться на объект в области стека, что приведёт к неопределённому поведению
// при разрушения
shapes.EmplaceBack(&r);

// не скомпилируется, так как конструктор unique_ptr из указателя объявлен explicit
// shapes.PushBack(&r);
```

Реализация методов Insert, Emplace и Erase

Метод `Insert` вставляет элемент в заданную позицию вектора. `Insert` допускает вставку элемента вектора внутрь того же самого вектора:

```
template <typename T>
class Vector {
public:
    ...
    iterator Insert(const_iterator pos, const T& value);
```

```
    iterator Insert(const_iterator pos, T&& value);
};
```

Две основные ситуации:

1. Вектор имеет достаточную вместимость для вставки ещё одного элемента.
2. Вектор полностью заполнен. В результате вставка элемента приведёт к реаллокации памяти.

Основное отличие метода `Emplace` от `Insert` в том, что `Emplace` для передачи своих параметров конструктору элемента использует perfect forwarding:

```
template <typename T>
class Vector {
public:
    ...
    template <typename... Args>
    iterator Emplace(const_iterator pos, Args&&... args);
};
```

Метод `Erase` удаляет элемент, на который указывает переданный итератор:

```
template <typename T>
class Vector {
public:
    ...
    iterator Erase(const_iterator pos);
};
```