Основы разработки на C++. Константность и `unique_ptr`

Оглавление

Константность как элемент проектирования программ **2**
1.1 Введение . . . 2
1.2 const защищает от случайного изменения . . . 5
1.3 Использование const для поддержания инвариантов в классах и объектах . . . 7
1.4 Идиома immediately invoked lambda expression (IILE) . . . 10
1.5 Константные объекты в многопоточных программах . . . 12
1.6 Логическая константность и mutable . . . 13
1.7 Ещё раз о константности в многопоточной среде . . . 14
1.8 Рекомендации по использованию const . . . 17

Умные указатели. Часть 1 **20**
2.1 Введение . . . 20
2.2 Обнаружение утечки памяти в ObjectPool . . . 20
2.3 Откуда берётся утечка памяти? . . . 24
2.4 Умный указатель unique_ptr . . . 27
2.5 unique_ptr для исправления утечки . . . 31

Разбор задачи «Дерево выражения» **36**
3.1 Разбор задачи «Дерево выражения» . . . 36

Константность как элемент проектирования программ

1.1 Введение

Мы посмотрим, как **константность** позволяет существенно повысить качество вашего кода, сделать его проще для понимания и безопасней. Давайте рассмотрим класс `Database`, который вы писали в задаче «Вторичный индекс» в модуле про ассоциативные контейнеры. Давайте представим, что этот класс `Database` используется в большом проекте. И у нас есть много-много файлов, которые каким-то образом этот класс используют. Тот факт, что класс используется в большом количестве файлов — моделируем тем, что у нас есть три функции: `FindMinimalKarma`, `FindUsersWithGivenKarma` и `WorstFiveUsers`. Все эти три функции принимают объект класса `Database` по константной ссылке. Мы просто представляем, что эти функции находятся в отдельных файлах и их на самом деле больше, чем три.

```
int FindMinimalKarma(const Database& db);

vector<string> FindUsersWithGivenKarma(const Database& db, int value);

vector<string> WorstFiveUsers(const Database& db)
```

Эти функции принимают базу данных и как-то ее исследуют. Давайте рассмотрим `WorstFiveUsers`. Она принимает базу данных, и вызывает метод `RangeByKarma` с какими-то параметрами, и находит пять пользователей с самой худшей минимальной кармой. Метод `RangeByKarma`, который вызывает функцию `WorstFiveUsers` — константный. Потому что мы просто перебираем записи в нашей базе данных, и для каждой записи вызываем переданный коллбэк.

```
vector<string> WorstFiveUsers(const Database& db) {
    vector<string> result;
    db.RangeByKarma(numeric_limits<int>::min(), numeric_limits<int>::max(),
        [&](const Record& r) {
            result.push_back(r.id);
            return result.size() < 5;
        });
    return result;
}
```

Теперь давайте представим, что мы в нашем проекте занялись оптимизацией скорости работы и провели исследования, профилирования и узнали такой интересный шаблон использования метода `RangeByKarma`. Мы узнали, что очень часто он вызывается с одними и теми же параметрами `low` и `high`, но с разным коллбэком. То есть мы фиксируем какие-то значения `low` и `high` и вызываем этот метод все время с этими значениями, передавая разные коллбэки, чтобы какие-то разные исследования проводить. Кроме того, мы узнали, что внутри метода `RangeByKarma` мы находим и перебираем не более трех элементов. Вот такой шаблон использования класса `Database` в нашем проекте.

```
template <typename Callback>
void RangeByKarma(int low, int high, Callback callback) const;
```

И мы решили, воспользовавшись знаниями о том, как мы используем метод, как-то его ускорить. Давайте оценим асимптотику метода `RangeByKarma`. Что мы делаем? Мы сначала вызываем методы `lower_bound` и `upper_bound` на поле `karma_index`. И, как мы знаем, эти методы имеют сложность $\log(N)$, где N — это количество записей в базе данных.

```
template <typename Callback>
void RangeByKarma(int low, int high, Callback callback) const {
    auto it_begin = karma_index.lower_bound(low);
    auto it_end = karma_index.upper_bound(high);
```

Таким образом, если в нашей базе данных хранится N элементов, то вот эти две команды выполняются за логарифм.

Дальше мы запускаем цикл от `begin` до `end` и для каждого найденного элемента вызываем коллбэк. То есть эта часть имеет асимптотику $O(K)$, где K — это количество найденных элементов.

```
    for (auto it = it_begin; it != it_end; ++it) {
        if (!callback(*it->second)) {
            break;
        }
    }
}
```

Метод `RangeByKarma` имеет асимптотику $O(K + \log(N))$. При этом мы знаем, что K не превосходит 3 в большинстве сценариев использования нашего метода. Следовательно, так как записей в базе данных очень много, то K несущественна в данной оценке и в этой асимптотической оценке превалирует логарифм.

Мы хотим от этого логарифма избавиться и получить оценку метода $O(K)$. Так как мы знаем, что наш метод вызывается много раз подряд с одними и теми же параметрами `low` и `high`, мы решили, что вот **эти вызовы** `lower_bound` и `upper_bound` **можно закэшировать**, то есть выполнить поиск итераторов один раз, а дальше, если нас вызвали с теми же параметрами, что и в предыдущем вызове, мы этот логарифмический поиск не осуществляем, а уже достаем значения из кэша. И давайте это кэширование мы с вами реализуем и добавим в класс `Database`.

Мы объявим структуру `Cache`. У нее будут поля `low` и `high` — это значения параметров нашего метода `RangeByKarma`. И два итератора константных: `begin` и `end`. Это, собственно, те значения, которые вернут вызовы `lower_bound` и `upper_bound`. И заведем поле `cache`, которое будет иметь

тип `optional` от `Cache`. Почему `optional`? Потому что наш кэш может быть пустым, может быть непроинициализирован, поэтому мы это моделируем с помощью применения класса `optional`.

```
struct Cache {
    int low, high;
    Index<int>::const_iterator begin, end;
};
std::optional<Cache> cache;
```

И давайте пойдем в наш метод `RangeByKarma` и воспользуемся нашим кэшом.

```
    if (!cache || cahce->low != low || cache->high != high) {
        cache = Cache{low, high, karma_index.lower_bound(low),
            karma_index.upper_bound(high)};
    }

    for (auto it = cache->begin; it != cache->end; ++it) {
        if(!callback(*it->second)) {
            break;
        }
    }
```

Кроме того, нам наш кэш надо иногда чистить. Чистить его надо, когда база данных меняется, то есть в методе `Put` мы должны наш кэш почистить. И то же самое нужно сделать в методе `Erase`, потому что когда база данных поменялась, то не важно, что мы вызовем метод `RangeByKarma` с теми же самыми аргументами — он может уже вернуть другие значения.

```
cashe.reset();
```

Вот таким образом мы добавили кэширование в наш класс `Database`. Давайте запустим компиляцию — и у нас не компилируется. У нас не компилируется, потому что компилятор пишет: «Нет оператора `=` для операндов `const std::optional<Database::Cache>` и просто `Database::Cache`». В чем тут дело? Дело в том, что наш метод `RangeByKarma` — константный, поэтому он не может менять поле `cache`.

Но нам надо поменять поле `cache` — мы ничего не можем сделать, мы делаем оптимизацию, нам надо кэш обновлять. Поэтому нам ничего не остается, кроме как убрать константность у метода `RangeByKarma`.

Запускаем компиляцию — и снова не компилируется, но уже по другой причине. Нам компилятор пишет: «passing '`const Database`' as '`this`' discards qualifiers». В чем тут дело? Давайте смотреть, где это происходит. Это происходит в функции `FindMinimalKarma`, которая принимает константную ссылку на класс `Database` и вызывает метод `RangeByKarma`, но метод `RangeByKarma` больше не константный, а мы **не можем вызывать неконстантные методы у константных объектов**. Что делать? Придется в функцию `FindMinimalKarma` передавать неконстатную ссылку. И то же самое нам придется делать с другими нашими функциями, которые тоже вызывают `RangeByKarma` — `FindUsersWithGivenKarma` и `WorstFiveUsers`. Кроме того, у нас для этих функций есть еще отдельные объявления, и в них нам тоже нужно убрать константность.

Запускаем компиляцию. И здесь у нас наконец-то компилируется. Все работает. Смотрите,

что произошло: мы с вами добавили кэширование в класс `Database`, и из-за того, что мы стали в констаном методе менять поле, которое хранит кэш, нам этот метод пришлось сделать неконстантным. Из-за того что он стал неконстантным, нам пришлось осуществить такое «веерное» удаление константности — нам пришлось пройтись по всем функциям, которые принимали нашу базу данных по константной ссылке и вызывали метод `RangeByKarma` — и у них тоже убрать константность.

А я напомню, что мы говорили в начале, что мы представляем, что у нас есть большой проект и класс `Database` используется в большом количестве различных файлов. И может возникнуть логичный вопрос: а есть ли польза от константности в C++? Потому что сейчас мы поменяли класс внутри, и, из-за того что метод перестал быть константным, нам пришлось перелопатить весь свой проект и поудалять константности из большого количества методов, классов, функций и методов, в которых использовался класс `Database`. Это большая ненужная работа. Может быть, вообще нет смысла использовать константные методы, чтобы застраховать себя от таких «веерных» изменений кода.

Чтобы ответить на этот вопрос, мы с вами изучим, в каких ситуациях в C++ константность бывает полезна. Дальше мы с вами обсудим семантику константных методов, и что на самом деле означает констатность у метода. Разобрав эти вещи, мы с вами узнаем, как правильно нужно было добавлять кэширование в класс `Database` и походу мы разберем еще немало интересных вещей о константности в C++.

1.2 const защищает от случайного изменения

Поговорим и покажем, как константность защищает нас от случайного изменения объекта в программе. Это на самом деле должно быть довольно знакомо вам, потому что мы еще в первом курсе, в «белом поясе по C++» говорили о том, что константность защищает нас от случайных изменений. И давайте рассмотрим вот такой пример.

Допустим, нам надо реализовать шаблон `CopyIfNotEqual`, который берет вектор `src` — входной вектор — и копирует из него все элементы в вектор `dst`, которые не равны параметру `value`. Кроме того, у нас даже есть юнит-тест один, который проверяет работоспособность нашей реализации. И вот он вызывается в функции `main`. Как можно эту функцию, `CopyIfNotEqual`, реализовать?

На самом деле для этого есть стандартный алгоритм, который реализует именно эту функциональность. Давайте мы им и воспользуемся. Этот алгоритм называется `remove_copy`, и воспользуемся мы им вот так:

```
#include "test_runner.h"

#include <algorithm>
#include <iostream>
#include <map>
#include <string>
#include <unordered_map>
#include <optional>

using namespace std;
```

```
template <typename T>
void CopyIfNotEqual(vector<T>& src, vector<T>& dst, T value) {
    std::remove_copy(begin(src), end(src), back_inserter(src), value);
}

void TestCopyIfNotEqual() {
    vector<int> values = {1, 3, 8, 3, 2, 4, 8, 0, 9, 8, 6};
    vector<int> dest;
    CopyIfNotEqual(values, dest, 8);

    const vector<int> expected = {1, 3, 3, 2, 4, 0, 9, 6};
    ASSERT_EQUAL(dest, expected);
}

int main() {
    TestRunner tr;
    RUN_TEST(tr, TestCopyIfNotEqual());
    return 0;
}
```

Написали, наш код компилируется, и давайте запустим юнит-тест, убедимся, что наша реализация правильная. Запустили юнит-тест и видим, что что-то у нас не так. Сработал `assert`, и мы вернули пустой вектор, хотя должны были вернуть вот такой вот набор элементов.

Только запустив программу, мы поняли, что в ней есть ошибка. И в нашем примере все было достаточно просто — мы запустили программу, сразу увидели, что юнит-тест не сработал. На практике же эта программа может раскатиться на большое количество серверов, работать там несколько часов, дней, а может быть, и недель, и только после такого вот долгого процесса работы она, вдруг, может начать работать неправильно.

Это следствие того, что в параметрах шаблона `CopyIfNotEqual` мы не используем константность. Потому что здесь намеренно допущена опечатка. В `back_inserter` передан не `dst`, не выходной вектор, а входной — `src`. Если же мы входной вектор объявим, как константную ссылку, потому что это входной вектор, из которого мы только читаем и копируем элементы, то есть мы не собираемся его изменять. Если мы объявляем его константным, запускаем компиляцию, наша программа не компилируется. Таким образом **компилятор нас защищает от случайного изменения объекта, который по своему смыслу не должен изменяться**.

Итак, мы замечаем, что мы опечатались и вместо `dst` написали `src`. Меняем `src` на `dst`, компилируем, компилируется, и юнит-тест проходит. Это было наглядной демонстрацией того, как константность защищает нас от случайного изменения тех объектов, которые по своему смыслу изменены быть не должны.

Давайте рассмотрим другой, менее очевидный пример. Пойдем в функцию `main`, уберем отсюда вызов юнит-теста, он нам больше не нужен. И сделаем вот что: мы объявим переменную `value`. Допустим, она равна 4. И создадим лямбда-функцию, назовем ее `increase`, которая будет захватывать `value` по значению, принимать целочисленный аргумент и возвращать `value + x`.

```
int main() {
    int value = 4;
```

```
    auto increase = [value](int x) {
        return value + x;
    };

    cout << increase(5) << endl;
    cout << increase(5) << endl;
}
```

Что мы можем ожидать от функции `increase`? То, что если мы дважды вызовем ее с одним и тем же аргументом, то она нам вернет одно и то же значение. Потому что это функция, и мы ожидаем, что если ей даешь одни и те же значения на вход, она будет возвращать одни и те же значения. Сейчас это так и происходит.

Но допустим, мы с вами при реализации тела этой функции допустили опечатку и написали не `value + x`, а `value += x`. Соответственно, в таком случае последовательные вызовы функции `increase`, должны приводить к изменению переменной `value`, и тогда она от одних и тех же аргументов будет возвращать разные результаты.

Запустим компиляцию, и видим, что наша программа не компилируется. При этом компилятор пишет нам: «assignment of read-only variable '`value`'», то есть он говорит, что переменная `value` внутри тела нашей лямбды, она доступна только для чтения, ее нельзя изменять.

Как это происходит? Дело в том, что **когда компилятор встречает лямбда-функцию, то в процессе компилирования он создает класс, у которого имя не определено, однако у него есть публичный оператор вызова**. Когда мы захватываем какие-то переменные по значению в лямбда-функциях, то этим переменным соответствуют поля этого класса. И что очень важно, что **оператор вызова для этого класса константный**. То есть семантика лямбда-функции такова, что когда мы захватываем переменные по значению, они превращаются в поля этого класса, а так как оператор вызова у него константный, эти поля менять нельзя. И именно благодаря этой самой константности в операторе вызова, мы гарантируем, что вызов лямбда-функции с одними и теми же аргументами всегда возвращает одно и то же значение. И это обеспечивается неявной константностью оператора вызова лямбда-функции.

1.3 Использование const для поддержания инвариантов в классах и объектах

Продолжим изучать, в чем польза от применения константности в C++. И давайте рассмотрим вот такой пример.

```
#include "test_runner.h"

#include <algorithm>
#include <iostream>
#include <map>
#include <string>
#include <unordered_map>
#include <optional>
```

```
using namespace std;

int main() {
    vector<int> numbers = {3, 7, 5, 6, 20, 4, 22, 17, 9};
    auto it = std::find(begin(numbers), end(numbers), 4);
    *(it) = 21;

    for (auto x : numbers) {
        cout << x << ' ';
    }
    return 0;
}
```

Запустим эту программу и увидим, что она отработала, как и ожидалось. У нас была 4, мы ее нашли и через итератор поменяли — сделали 21. Смотрите: когда мы через итератор поменяли 4 на 21, то вектор остался вектором в том смысле, что он остался последовательностью целых чисел.

А теперь давайте заменим вектор на `set`. Запустим компиляцию и увидим, что наша программа не компилируется — она не компилируется в том месте, где мы по итератору пытаемся поменять теперь уже элемент множества. И сообщение компилятора такое: «assignment of read-only location». То есть мы не можем поменять по итератору элемент множества. Почему так происходит?

Давайте вспомним, как устроено **стандартное множество** внутри. Внутри оно **представляет из себя двоичное дерево поиска**, и, как мы помним, основным свойством двоичного дерева поиска является то, что все узлы в поддереве слева, они меньше, чем значение в текущем узле, а все значения в правом поддереве больше, чем значения текущего узла. И вот теперь давайте представим, что у нас есть итератор, который указывает на узел со значением 4. И мы через вот этот итератор меняем значение в узле на 21. Что происходит? Мы полностью разрушаем наше свойство, свойство нашего двоичного дерева поиска, и получившееся двоичное дерево перестает быть деревом поиска, потому что для него больше не выполняется основное свойство. Значит, мы нарушаем инвариант, который хранится внутри класса `set`, и дальнейшие операции с этим множеством могут работать некорректно.

Каким же образом реализация стандартного множества гарантирует нам, что мы не можем поменять элемент через итератор? Очень просто: **разыменование итератора множества возвращает константную ссылку на элемент**. Но если же мы попробуем присвоить неконстантные ссылки, то у нас компиляция не удастся. Таким образом, константность позволяет нам поддерживать инвариант внутри класса `set`.

Давайте рассмотрим другой пример, в котором константность позволяет нам сохранить инвариант.

```
vector<int> Sorted(vector<int> data) {
    sort(begin(data), end(data));
    return data;
}

int main() {
    vector<int> sorted_numbers = Sorted({5, 4, 2, 1, 5, 1, 3, 4, 5, 6, 8});
```

```
    for (auto x : numbers) {
        cout << x << ' ';
    }

    // some code

    ProcessNumbersOne(sorted_numbers);
    ProcessNumbersTwo(sorted_numbers);
    ProcessNumbersThree(sorted_numbers);

    // some code

    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        cout << x << ' ';
        cout << std::binary_search(begin(sorted_numbers), end(sorted_numbers), x) << '\n';
    }
}
```

При этом, так как мы `sorted_numbers` создали с помощью вызова функции `Sorted`, то в цикле мы для проверки — есть элемент в векторе или нет его, — используем двоичный поиск, потому что мы знаем, что вектор отсортирован, поэтому мы можем выполнять не линейный поиск, а более быстрый — двоичный.

Однако возникает вопрос. Смотрите, вот мы вектор создали, затем у нас много кода. Дальше он передается в три какие-то функции. И по вызову этих функций совершенно неочевидно, что они этот вектор не меняют. А нам нужно быть уверенными, что к моменту, когда мы придем в цикл, вектор всё так же останется отсортированным, чтобы мы были уверены, что мы можем искать в нем двоичным поиском.

В текущей ситуации, чтобы понять, действительно ли вектор остается отсортированным, нам нужно заходить внутрь этих функций `ProcessNumbersOne`, `Two` and `Three`, смотреть, как они принимают этот вектор. Если они принимают его по неконстантной ссылке, то нам нужно смотреть их реализацию и понимать, меняется этот вектор или нет. Это довольно сложно и, соответственно, наш код, он со временем еще будет меняться, и нам нужно, получается, каждый раз его весь перечитывать, чтобы убеждаться, что вектор остается отсортированным. Это неудобно и непрактично.

Вместо этого мы можем наш вектор `sorted_numbers` сделать константным, и тогда сам язык гарантирует нам, что вот к этому моменту, когда мы будем выполнять двоичный поиск по вектору, он останется отсортированным, потому что мы объявили его константным и уже не важно, как он передается в функции `ProcessNumbersOne`, `Two` и `Three` — мы знаем, что он не будет изменен, потому что он константный. И тогда мы уверены, что мы можем искать по нему с помощью двоичного поиска.

Смотрите какая важная здесь возникает особенность использования константности в C++. У

нас вектор `sorted_numbers` — это не просто вектор, это не просто упорядоченная последовательность целых чисел. Это отсортированный вектор. **Сортированность — это дополнительное свойство этого конкретного объекта класса** `vector<int>`. И **это дополнительное свойство**, которое присуще только одному объекту, **мы поддерживаем и гарантируем с помощью константности**.

1.4 Идиома immediately invoked lambda expression (IILE)

Давайте продолжим работать с примером, в котором мы создаем сортированный вектор, и давайте представим, что нам нужно сделать так, чтобы он был не только отсортирован, но еще чтобы в нем отсутствовали дубликаты, то есть чтобы он был уникализирован. Как мы можем это сделать? Первый вариант такой. Мы можем написать функцию `Unique` по аналогии с функцией `Sorted`.

```
vector<int> Sorted(vector<int> data) {
    sort(begin(data), end(data));
    return data;
}

vector<int> Unique(vector<int> data) {
    auto it = unique(begin(data), end(data));
    data.erase(it, end(data));
    return data;
}

int main() {
    vector<int> sorted_numbers = Unique(Sorted({5, 4, 2, 1, 5, 1, 3, 4, 5, 6, 8}));
    ...
```

Вполне себе подходящий рабочий хороший вариант. Мы вызываем одну функцию и передаем объект в другую, и потом инициализируем константный вектор. Но у такого подхода может быть ряд недостатков. Например, если эта дополнительная инициализация какая-то уникальная и нужна только в одном месте, то делать для нее отдельную функцию, которую мы помещаем куда-то в глобальное пространство имен, может быть не самым подходящим решением, потому что мы и название для этой функции должны придумать, и мы тем самым засоряем глобальное пространство имен. Поэтому иногда это создание такой дополнительной функции может быть неудобным.

Кроме того, когда мы читаем вот этот вот код, нам может захотеться посмотреть, а что же делает, собственно, функция `Unique`. И если это какая-то опять же нетривиальная и единоразовая инициализация, то иногда лучше иметь ее код рядом с тем местом, где мы используем, а не выносить куда-то в другое место.

Таким образом, мы можем захотеть попробовать каким-то другим образом выполнить инициализацию нашего вектора. Как мы можем это сделать? Мы можем, например, снять с него константность, но мы уже подробно разобрали, насколько она полезна и насколько она необходима в этом самом месте. Поэтому такой вариант инициализации нам особо не подходит. Нам нужен какой-то другой способ оставить вектор константным, но при этом выполнить какую-то нетриви-

альную инициализацию рядом с его объявлением. И именно для этого в C++ есть специальная **идиома**, которая называется `immediately invoked lambda expression`. Суть ее в том, что для инициализации константного объекта мы создаем lambda-функцию и тут же, рядом с тем местом, где мы ее создали, мы ее вызываем.

Таким образом, вызывая вот эту вот лямбду в месте ее создания, мы можем не терять константность.

Давайте применим вот эту идиому `IILE` в нашем примере. Выглядеть это будет таким образом.

```
int main() {
    const vector<int> sorted_numbers = [] {
        vector<int> data = Sorted({5, 4, 2, 1, 5, 1, 3, 4, 5, 6, 8});
        auto it = unique(begin(data), end(data));
        data.erase(it, end(data));
        return data;
    }();
    ...
```

И таким образом мы выполняем инициализацию нашего вектора, сначала сортируя входные данные, затем гарантируя, что сам вектор будет уникализирован, и сохраняя его константность.

На самом деле идиома `IILE` бывает очень полезна в случае, когда нам нужно замерить время конструирования объекта. Смотрите, как это делается. Давайте подключим заголовочный файл `profile.h`, который мы разработали в «Красном поясе по C++», и допустим, мы вернемся к ситуации, когда у нас вектор `sorted_numbers` инициализировался только с сортированными числами, не уникализированными.

Мы хотим замерить, а сколько же времени у нас уходит на конструирование этого вектора

```
int main() {
    vector<int> sorted_numbers = Sorted({5, 4, 2, 1, 5, 1, 3, 4, 5, 6, 8});
    ...
```

И мы это можем сделать опять же с помощью нашей идиомы.

```
const vector<int> sorted_numbers = [] {
    LOG_DURATION("Sorted numbers build");
    return Sorted({5, 4, 2, 1, 5, 1, 3, 4, 5, 6, 8});
}();
```

Конечно, вы можете подумать, что мы могли бы `LOG_DURATION` поместить внутрь функции `Sorted`, но тогда будет замеряться время всех вызовов этой функции, у нас будет засоряться поток ошибок, а здесь мы конкретный вызов, который нас интересует, замеряем. И это делается очень удобно, с помощью идиомы `immediately invoked lambda expression`.

У вас может возникнуть вопрос: а не приводит ли использование вот этой вот лямбды, ее конструирование, вызов, к замедлению кода? На самом деле идиома `IILE` никак не сказывается негативно на скорости работы вашей программы.

И последний момент, который стоит отметить, говоря об этой идиоме, заключается вот в чем:

при объявлении и инициализации переменной с помощью идиомы `IILE` крайне не рекомендуется использовать `auto` для указания типа этой переменной. Когда же у нас тип указан явно, то при чтении кода мы видим, что мы объявляем переменную такого-то типа. Дальше мы видим

```
const vector<int> sorted_numbers = []{
```

И мы понимаем, что это идиома `immediately invoked lambda expression`. Мы создаем лямбду, и сейчас в конце мы ее вызовем, чтобы проинициализировать этот вектор. И таким образом мы не вводим читателя в заблуждение, и он сразу понимает, что здесь не лямбда создается, а здесь создается вектор целых чисел.

1.5 Константные объекты в многопоточных программах

Когда мы в «Красном поясе по C++» изучали с вами многопоточность, мы говорили, что в многопоточных программах возможны ситуации «гонки». Я напомню, что **гонка — это ситуация, когда несколько потоков обращаются к одной и той же переменной, и минимум один из этих потоков эту переменную изменяет**. Опасность ситуации гонки заключается в том, что она может привести к нарушению целостности данных и некорректной работе программы. Чтобы защититься от возникновения гонки, нужно выполнять синхронизацию. Об этом мы с вами также говорили в «Красном поясе по C++». Синхронизация в C++ выполняется с помощью специального класса `mutex`, который гарантирует так называемое взаимное исключение и обеспечивает ситуацию, в которой не более одного потока обращается к разделяемой переменной. **Код, защищаемый `mutex`, называется «критической секцией»**. И в «Красном поясе» мы с вами посмотрели, что чем больше размер критической секции, тем медленнее работает наша многопоточная программа.

Давайте с вами рассмотрим решение задачи исследования блогов, которая у была нас в «Красном поясе по C++» как раз в блоке про многопоточность. Здесь, в этой задаче нужно было написать `ExploreKeyWords`, которая исследовала текст из входного потока и проверяла, какие слова из него входят в множество ключевых слов.

```
Stats ExploreKeyWords(const set<string>& key_words, istream& input) {
    const size_t max_batch_size = 5000;

    vector<string> batch;
    batch.reserve(max_batch_size);

    vector<future<Stats>> futures;

    for (string line; getline(input, line); ) {
        batch.push_back(move(line));
        if (batch.size() >= max_batch_size) {
            futures.push_back(async(ExploreBatch), ref(key_words), move(batch));
            batch.reserve(max_batch_size);
        }
    }
}
```

Сейчас нам надо обратить внимание на то, что функция `ExploreKeyWords` создает потоки с помощью вызова функции `async`, и в каждом из этих потоков она вызывает функцию `ExploreBatch`, передавая в эту функцию по ссылке наш словарь `key_words`, словарь ключевых слов. Давайте посмотрим, как устроена функция `ExploreBatch`.

```
Stats ExploreBatch(const set<string>& key_words, vector<string> lines) {
    Stats result;
    for (const string& line : lines) {
        result += ExploreLine(key_words, line);
    }
    return result;
}
```

Что здесь важно? Что мы вот этот наш словарь ключевых слов передаем по ссылке в различные потоки, каждый из которых к этому словарю обращается, и не выполняем никакой синхронизации. У нас в коде нигде не используются `mutex`. Почему так? Потому что наш словарь ключевых слов, наша переменная `key_words`, представляет собой константный объект. Мы говорили, что чтобы возникла ситуация гонки, нужно чтобы был хотя бы один поток, который изменяет разделяемый объект. Но так как `key_words` — это константная ссылка, то ни один поток в принципе не может этот словарь изменить. И поэтому нам нет необходимости выполнять синхронизацию доступа к этой переменной. И это очень важное свойство константности в C++. **Константность гарантирует потокобезопасность**. Константному объекту нет необходимости осуществлять синхронизацию доступа.

1.6 Логическая константность и mutable

Итак, мы с вами увидели, что константность C++ крайне полезна. Она защищает объекты от случайного изменения, она гарантирует им варианты в классах и объектах, она упрощает понимание кода и упрощает многопоточный код, делая его проще для понимания и иногда быстрее. И давайте теперь вернемся к примеру, с которого мы начинали рассматривать константность.

У нас был класс `Database`, в котором мы решили добавить кэширование результатов в метод `RangeByKarma`. При этом, из за того, что мы стали изменять поле внутри метода `RangeByKarma`, нам пришлось убрать у этого метода константность, что привело к «веерному» удалению константности из кода, который использовал наш класс `Database`, и мы задались вопросом: а есть ли смысл вообще использовать константность C++, когда она приводит к таким «веерным» изменениям. Теперь мы с вами знаем, что константность крайне важна и крайне полезна, и поэтому, если у вас есть константный метод, и вы вдруг понимаете, что вы хотите убрать у него константность, этого ни в коем случае нельзя делать, потому что это может фатальным образом отразиться на корректности вашей программы. Поэтому, нам сейчас нужно разобраться, каким образом нам и кэширование встроить в наш метод, и константность сохранить.

И для этого давайте поглубже разберемся с тем, а какие же гарантии дают нам константные методы? Что вообще это с точки зрения языка означает, константный метод? В «белом поясе по C++?» мы говорили, что константный метод не имеет право менять текущий объект. На самом деле это формулировка не совсем правильная, и с точки зрения C++ константность означает

несколько другое. На самом деле константные методы не меняют наблюдаемое состояние объекта, то есть константный метод дает гарантию того, что если у объекта есть какое-то наблюдаемое состояние, то он его менять не будет. Давайте посмотрим какое наблюдаемое состояние есть у нашего класса.

- Результат метода `Put`
- Результат метода `GetByld`
- Результат метода `Erase`
- Записи, переданные в `callback` в методах `RangeByKarma`, `RangeByTimestamp`, `AllByUser`

При этом вот этот самый кэш, который мы с вами добавили, не является наблюдаемым состоянием, потому что снаружи класса пользователя никак не может узнать в каком состоянии сейчас находится кэш для метода `RangeByKarma`. Да и вообще, этот самый кэш не является частью состояния базы данных, это исключительно детали реализации, которые мы добавили с целью ускорения работы нашего класса, и с точки зрения этой классификации на наблюдаемое и ненаблюдаемое состояние, мы можем сказать, что константный метод имеет право изменять состояние поля кэш, потому что оно не относится к наблюдаемому состоянию, и возникает вопрос: а как же нам сказать компилятору, что поле кэш не относится к наблюдаемому состоянию нашего объекта?

Очень просто, для этого есть специальное ключевое слово `mutable`, поля, которые помечены этим ключевым словам можно изменять в контактных методов.

```
mutable std::optional<Cache> cache;
```

И теперь мы можем вернуть константность методу `RangeByKarma`. Кроме того, мы в начале этого модуля удаляли константность из функции, которые работают с нашей базой данных. Теперь у нас есть функции, которые принимают базу данных по константной ссылке, вызывают метод `RangeByKarma`. Сам метод `RangeByKarma` является константным, но при этом он изменяет поле кэш.

Запускаем компиляцию — и наша программа компилируется, то есть за счет применения ключевого слова `mutable` к полю кэш, мы смогли сделать, сохранить наш метод константным, но при этом добавить, реализовать кэширование.

Итак, мы с вами познакомились с ключевым словом `mutable`. Оно **позволяет изменять внутреннее состояние объекта, оставляя его методы константными**. Кроме того вы теперь более точно знаете гарантии, которые дают **константные методы в C++**. Они **обеспечивают так называемую логическую константность**. Есть понятие физической константности, это когда ни один бит внутри объекта не изменяется. Так вот, теперь вы знаете, что константные методы не гарантируют физической константности, а гарантирует логическую константность, то есть они гарантируют, что наблюдаемое состояние объекта не будет изменено.

1.7 Ещё раз о константности в многопоточной среде

Мы с вами говорили, что в многопоточных программах нет необходимости выполнять синхронизацию доступа к константным объектам. Потому что мы можем вызывать только константные

методы и не можем изменить объект. Поэтому нет необходимости синхронизировать доступ. Однако мы с вами узнали о константности кое-что новое. А именно, мы познакомились с `mutable` полями. И поэтому нам нужно еще раз посмотреть на константность многопоточных программ. Давайте воспользуемся шаблоном `LazyValue`, который вы разработали в задаче ранее.

```
template <typename T>
class LazyValue {
public:
    explicit LazyValue(std::function<T()> init) : init_(std::move(init)) {}

    const T& Get() const {
        if (!value) {
            value = init_();
        }
        return *value;
    }
private:
    std::function<T()> init_;
    mutable std::optional<T> value;
};
```

И воспользуемся мы им таким образом: мы создадим ленивый `map` из строки в число, который в случае обращения будет инициализироваться списком населения городов России.

```
int main() {
    LazyValue<map<string, int>> city_population([&] {
        return map<string, int> {
            {"Moscow", 11514330},
            ...
            {"Omsk", 1154000},
            ...
            {"Saratov", 836900},
            ...
            {"Tula", 501129},
        };
    };
```

Более того, мы к нашему объекту `city_population` будем обращаться из разных потоков.

```
    auto kernel = [&] {
        return city_population.Get().at("Tula");
    };

    vector<std::future<int>> ts;
    for (size_t i = 0; i < 25; ++i) {
        ts.push_back(async(kernel));
    }
```

И дальше мы дожидаемся, когда все потоки отработают, и в конце программы еще выводим

население города Саратов.

```
    for (auto& t : ts) {
        t.get();
    }
    const string saratov = "Saratov";
    cout << saratov << ' ' << city_population.Get().at("Saratov");
```

Давайте посмотрим внутрь функции `kernel`. Она вызывает метод `Get` у объекта `city_population`. Метод `Get` — это метод нашего шаблона `LazyValue`, этот метод константный, и поэтому, как мы говорили, нам нет необходимости выполнять какую-либо синхронизацию доступа к объекту `city_population`, потому что мы у него вызываем только константные методы. Вроде программа написана правильно. Давайте мы ее скомпилируем и запустим. Мы ее запускаем, она у нас корректно отработала и вывела, что в Саратове живет 836900 человек.

Однако давайте мы позапускаем нашу программу несколько раз. Вот она снова корректно отработала. Продолжаем запускать. И она пока что продолжает корректно работать... А вот мы сейчас ее запустили, и она что-то подвисла. Она подвисла и что-то думает, думает... и она завершилась, но в консоли ничего не выведено. Кроме того, если мы посмотрим, нам Eclipse пишет, что код возврата нашего процесса — минус 1 миллиард 73 миллиона и так далее. То есть наша программа упала. Она отработала некорректно и вернула ненулевой код возврата. Значит, с ней что-то не то. И так как мы сделали достаточно много успешных запусков, то мы можем сразу предположить, что дело у нас явно в многопоточной коммуникации, и явно у нас наши потоки обращаются к объекту `city_population`, и иногда это не получается. Давайте заглянем внутрь метода `Get`, и, думаю, тут вам станет очевидна причина падения нашей программы.

Смотрите: мы проверяем, что, если поле `value` типа `optional` непроинициализировано, не хранит никакого значения, то мы заходим внутрь условного оператора и инициализируем поле `value`. А теперь давайте представим: у нас несколько потоков параллельно приходят в метод `Get`, одновременно смотрят на поле `value`, видят, что оно пустое, оно не хранит никакого значения, заходят внутрь, параллельно выполняют функцию `init` и потом параллельно начинают записывать в поле `value` целый `map`. А так как `map` — это сложный объект, это дерево поиска, при параллельной записи что-то пошло не так. Что-то иногда идет не так. Что нам нужно сделать, чтобы таких ситуаций не возникало? Мы имеем дело с классической ситуацией гонки, нам нужно выполнить синхронизацию доступа к полю `value`. Как это делается? С помощью `mutex`. Добавим `mutex` в наш объект.

```
template <typename T>
class LazyValue {
public:
    explicit LazyValue(std::function<T()> init) : init_(std::move(init)) {}

    const T& Get() const {
        if (lock_guard g(m); !value) {
            value = init_();
        }
        return *value;
    }
```

```
private:
    std::function<T()> init_;
    mutable std::optional<T> value;
    mutex m;
};
```

Давайте скомпилируем наш код, и он не компилируется. Почему? Давайте посмотрим на сообщение компилятора. Компилятор пишет: «passing std::lock_guard mutex type (aka const std::mutex) as this argument discards qualifiers». Очень знакомое нам сообщение, которое говорит о том, что у нас что-то не то с константностью. При этом мы тут видим, что мы передаем объект `const mutex`. Ну, логично. У нас метод `Get` константный, а мы здесь, создавая `lock_guard`, пытаемся `mutex` захватить, то есть изменить его. А `mutex` у нас не объявлен со словом `mutable`, поэтому изменять его нельзя. Что надо сделать?

Нужно объявить `mutex` с ключевым словом `mutable`. Тогда наш код будет компилироваться.

```
    mutable mutex m;
```

Он компилируется. Мы его запустим, и он корректно работает. Если его позапускать много раз, он все равно будет работать корректно, потому что мы сделали здесь синхронизацию.

На самом деле, в реальных программах, вот в такой ситуации, когда у нас есть какой-то многопоточный кэш, доступ к нему реализуют немного по-другому, потому что даже когда наш кэш, вот это `value`, оно проинициализировано, мы все равно при каждом обращении захватываем `mutex`. Это, безусловно, не очень эффективно. Поэтому в реальных программах это делают чуть иначе, но для нашего учебного примера мы взяли простой вариант.

Что мы хотели показать этим примером? То, что, если вы разрабатываете класс, который предназначен для использования в многопоточных программах, то его `mutable` поля должны быть потокобезопасными. Потому что вы изменяете эти поля внутри константных методов. И поэтому, если вы не гарантируете в `mutable` полях потокобезопасность, программа может вести себя некорректно.

Еще одна важная вещь, которую мы с вами узнали, состоит в том, что поля типа `mutex`, можно сказать, хотят быть `mutable`. Потому что сами по себе они потокобезопасны — это же примитив синхронизации, и он точно не является частью наблюдаемого состояния. И поэтому нет смысла делать мьютексы не `mutable`, потому что они чаще всего захватываются внутри константных методов только для того, чтобы гарантировать потокобезопасность. И ещё раз напомним, что **в C++ предполагается, что все константные методы являются потокобезопасными**. Собственно, именно поэтому и нужно делать `mutable` поля потокобезопасными, чтобы гарантировать, что в многопоточной программе при обращении к константным объектам нет необходимости выполнять внешнюю синхронизацию доступа.

1.8 Рекомендации по использованию const

Мы говорили, что константность защищает от случайного изменения объектов в программах. И из этого свойства часто делают такую рекомендацию. Говорят, что делайте константными все переменные, какие только можете. Но на самом деле, это не самая лучшая рекомендация. Дело в том, что слово `const` — это пять символов, после которых еще идет пробел. Это шесть символов

всего, и если чрезмерно использовать `const` при объявлении переменных, то программа становится слишком громоздкой. Поэтому наши рекомендации заключаются в том, чтобы искать золотую середину между двумя на самом деле немного противоречащими друг другу советами.

- если переменная не изменяется, то объявляйте ее константной
- не загромождайте ваш код

Вот давайте рассмотрим простой пример. У нас есть функция `Area`, которая считает площадь треугольника по трем сторонам с помощью формулы Герона. И мы в ней объявляем переменную `p`, которая хранит в себе полупериметр, и эта переменная только читается. Мы ее объявили, проинициализировали и только читаем в этой функции. И поэтому мы объявляем ее константной — потому что мы не собираемся ее изменять.

```
double Area(double a, double b, double c) {
    const double p = (a + b + c) / 2.0;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}
```

Ну, вроде бы вполне нормально, код легко читается и не захламлен.

Но можно эту функцию написать и вот так. Можно сказать, что наши входные параметры `a`, `b` и `c` тоже не изменяются, мы из них тоже читаем. И поэтому давайте их сделаем тоже константными.

```
double Area(const double a, const double b, const double c) {
    const double p = (a + b + c) / 2.0;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}
```

Но на субъективный взгляд, это уже перебор, потому что у нас маленькая функция, всего из двух строчек. И из нее очевидно, что мы вот эти параметры a, b и c не меняем. Но вместо этого ее объявление стало довольно большим, оно распухло на целых 18 символов, и поэтому, это уже перебор.

Но давайте вспомним другой пример и другое свойство константности. Мы говорили, что у нас в программах часто бывают объекты, которые обладают какими-то дополнительными свойствами. И мы это разбирали на примере отсортированного вектора. Так вот, для объектов, которые обладают какими-то дополнительными свойствами, не присущими их классам, делать такие объекты константными очень и очень полезно, потому что вы позволяете проще понимать ваш код и вы делаете так, что вам язык, вам компилятор, гарантирует, что вот это дополнительное свойство у объекта не пропадет в течение его жизни.

Дальше. Мы с вами познакомились с идиомой IILE (immediately invoked lambda expression). И мы вам **советуем пробовать использовать эту идиому, когда вам нужно создать константный объект с какой-то нетривиальной**. Эта идиома не всегда приводит к тому, что у вас получается хороший, понятный, легко читаемый код, поэтому совет именно такой: пробуйте. Попробуйте, посмотрите, что получится. Если это вас не устраивает, то воспользуйтесь какими-то другими способами, например, напишите отдельную функцию, которая выполняет вот эту нетривиальную инициализацию.

Где **идиома IILE незаменима, так это в ситуациях, когда нам нужно померить время, которое уходит на конструирование объекта**. Мы начинали наш модуль с примера, в котором мы стали изменять поле объекта и из-за этого убрали константность у метода. Так вот, наша рекомендация состоит в том, чтобы **никогда не снимать константность у методов, которые по своему смыслу не должны изменять объект**.

Если вам нужно в константных методах что-то изменять, то, во-первых, возможно, вы делаете что-то не так, а во-вторых, есть ключевое слово `mutable`, которое позволяет изменять поля в константных методах. Однако хочется сразу вас предостеречь от соблазна объявлять все свои поля как `mutable`. В этом случае вы нарушаете гарантии, которые даются константными методами. Мы говорили: константный метод не меняет наблюдаемое состояние объекта. Поэтому если вы пометите ключевым словом`mutable` поле, которое относится к наблюдаемому состоянию объекта, вы будете врать своим пользователям, и вашим классом нельзя будет пользоваться из-за того, что константные методы не будут выполнять своих гарантий.

В моем опыте есть только два сценария, когда ключевое слово `mutable` применимо: это кэши и это мьютексы. Оба примера мы с вами рассмотрели. И вот если вы хотите применить ключевое слово `mutable` в какой-то другой ситуации, не для кэширования каких-то результатов и не для обеспечения внутренней синхронизации объекта, три раза подумайте, правильно ли вы делаете, нельзя ли поступить как-то иначе.

Ну и давайте еще раз вспомним, что в многопоточных программах нам нет необходимости выполнять синхронизацию доступа к константным объектам. Поэтому **если ваш класс предназначен для использования в многопоточной среде и в нем есть `mutable` поля, гарантируйте, обеспечьте, что эти `mutable` поля потокобезопасны**, потому что вам необходимо обеспечить свойство вашего класса, которое заключается в том, что константность гарантирует потокобезопасность и при доступе к константным объектам нет обходимости выполнять внешнюю синхронизацию.

Умные указатели. Часть 1

2.1 Введение

Умные указатели нужны для того, чтобы автоматизировать управление динамической памятью в C++. В предыдущем курсе, в «Красном поясе», мы уже обсуждали динамическую память, и как ей пользоваться в C++, и вы уже знаете, что управляется динамическая память с помощью ключевых слов `new` и `delete`: `new` создает динамический объект, а `delete` удаляет этот объект. И вы знаете, что в явном виде использовать эти ключевые слова в вашем коде плохо. Мы разбирали пример, в котором у вас из-за явного использования ключевого слова `delete` утекали объекты. Это нехорошо. Соответственно, умные указатели позволяют сделать так, чтобы вы использовали динамические объекты в вашем коде, но вам не приходилось в явном виде писать ключевые слова `new` и `delete`.

В качестве примера мы возьмем задачу «ObjectPool» из предыдущего курса.

2.2 Обнаружение утечки памяти в ObjectPool

Итак, вы вспомнили про задачу ObjectPool и посмотрели на авторское решение.

```
#include "test_runner.h"

#include <algorithm>
#include <string>
#include <queue>
#include <stdexcept>
#include <set>

using namespace std;

template <class T>
class ObjectPool {
public:
    T* Allocate();
    T* TryAllocate();

    void Deallocate(T* object);

    ~ObjectPool();
```

```
private:
    queue<T*> free;
    set<T*> allocated;
};

template <typename T>
T* ObjectPool<T>::Allocate() {
    if (free.empty()) {
        free.push(new T);
    }
    auto ret = free.front();
    free.pop();
    allocated.insert(ret);
    return ret;
}

template <typename T>
T* ObjectPool<T>::TryAllocate() {
    if (free.empty()) {
        return nullptr;
    }
    return Allocate();
}

template <typename T>
void ObjectPool<T>::Deallocate(T* object) {
    if (allocated.find(object) == allocated.end()) {
        throw invalid_argument("");
    }
    allocated.erase(object);
    free.push(object);
}

template <typename T>
ObjectPool<T>::~ObjectPool() {
    for (auto x : allocated) {
        delete x;
    }
    while (!free.empty()) {
        auto x = free.front();
        free.pop();
        delete x;
    }
}
```

Давайте теперь попробуем в этом решении найти ошибку. Для этого напишем некоторую те-

стирующую программу.

```
#include "ObjectPool.h"

#include <iostream>

using namespase std;

void run() {
    ObjectPool<char> pool;
    for (int i = 0; i < 1000; ++i) {
        cout << "Allocating object #" << i << endl;
        poll.Allocate();
    }
}

int main() {
    run();
    return 0;
}
```

Давайте её соберём. Так, программа у нас успешно собралась, и давайте теперь её запустим. Запускаем. Вот мы видим, что программа у нас занимается тем, что просто выделяет некоторые объекты и пока ничего с ними не делает. По крайней мере, мы видим, что она работает, цикл крутится. Уже не плохо. Наша же задача сейчас состоит в том, чтобы понять: объекты, которые у нас в пуле: на самом ли деле они у нас удалились в тот момент, когда удалился собственно пул. А сделаем мы это с использованием небольшого трюка.

Мы заведём такой глобальный счётчик, в котором мы будем считать, сколько сейчас у нас объектов существует в программе, и заведём специальный класс под названием `Counted`. В своём конструкторе `Counted` будет увеличивать этот счётчик. А в своём деструкторе он будет этот счётчик уменьшать.

```
int counter = 0;

struct Counted {
    Counted() {
        ++counter;
    }
    ~Counted() {
        --counter;
    }
};
```

И затем мы, собственно, взглянем на количество объектов по завершению работы программы.

```
void run() {
    ObjectPool<Counted> pool;
```

```
    cout << "counter before loop = " << counter << endl;
    for (int i = 0; i < 1000; ++i) {
        cout << "Allocating object #" << i << endl;
        poll.Allocate();
    }
    cout << "counter after loop = " << counter << endl;
}

int main() {
    run();
    cout << "counter before exit = " << counter << endl;
    return 0;
}
```

Мы её собираем. Давайте теперь запустим. Когда она завершается, она выводит нам информацию о количестве объектов — то, что нам было нужно. Сразу после цикла этих объектов у нас 1000. Логично. У нас был цикл на тысячу итераций, мы выделили тысячу объектов. А перед завершением этих объектов осталось ноль. То есть они все удалились. Собственно, не забывайте, что, когда мы выходим из функции `run`, все локальные объекты этой функции уничтожаются. В том числе уничтожается наш пул объектов. А этот пул объектов в своем деструкторе, как мы помним, занимается тем, что уничтожает все объекты, которые он выделил. То есть это абсолютно корректное поведение программы. Ну и вроде бы пока всё работает неплохо, да? Хорошо.

Однако давайте теперь создадим для программы специальные условия. То есть объекты мы же выделяем в памяти? А давайте сделаем так, что памяти у нас на самом деле не хватает. Это стандартная, в принципе, ситуация. Память — конечный ресурс. На сервере память вообще разделяется между многими приложениями, которые там запускаются, поэтому память в какой-то момент может кончиться. Чтобы нам было проще создать такую ситуацию, мы её сэмулируем. Давайте напишем вот такую магическую строчку (в командной строке).

```
>appverif /verify pool.exe /faults 10000 200
```

И вот теперь запустим нашу программу. И мы видим, что наша программа начала падать, и она пишет, что случилось исключение под названием «std::bad_alloc».

Давайте посмотрим, что же произошло. Мы с вами сейчас сэмулировали нехватку памяти. То есть в системе-то память была, но вот нашей программе она эту память не отдала. Мы это сделали с помощью утилиты `appverifier`. Запускается она с помощью исполнимого файла `appverif`. Она входит в стандартную поставку Windows SDK. И у нее есть следующие параметры: сначала указываем `/verify` и имя исполнимого файла, для которого мы применяем подобное поведение, дальше `/faults` и указываем два числа. Первое — это вероятность того, что попытка выделения динамической памяти на самом деле вернёт нам ошибку со стороны операционной системы. Она указывается как целое число из миллиона. Здесь мы указали 10000 из миллиона, то есть 1 из 100, то есть с вероятностью 0.01 у нас попытка увеличить память вернёт ошибку. И дальше указали число 200 — это количество миллисекунд, в течение которых наша программа будет работать нормально.

Но чтобы её нормально протестировать, она вообще-то должна сначала загрузиться, она должна загрузить `runtime`, должна запуститься функция `run`, и уже только после этого нам нужно

делать так, чтобы у нас возникали какие-то проблемы в выделении. Это мы делали под Windows. Под Linux вы можете воссоздать очень похожую ситуацию с помощью утилиты `ulimit`. Она работает немножко по-другому.

Обратите внимание, что, **чтобы воспроизвести подобное поведение на вашей машине, вам, может быть, придётся немножко изменить константы или увеличить количество итераций в цикле выделения объектов**. Потому что нехватка памяти — это такая тонкая материя... Такие баги нужно еще уметь отловить.

Итак, сейчас нам среда написала, что у нас выбрасывается исключение «std::bad_alloc». Давайте попробуем его отловить и как-то на него отреагировать.

```
void run() {
    ObjectPool<Counted> pool;

    cout << "counter before loop = " << counter << endl;
    try {
        for (int i = 0; i < 1000; ++i) {
            cout << "Allocating object #" << i << endl;
            poll.Allocate();
        }
    } catch(const bad_alloc& e) {
        cout << e.what << endl;
    }
    cout << "counter after loop = " << counter << endl;
}
```

Давайте соберём программу. Давайте теперь её запустим. Вот мы видим, что у нас программа перестала падать. Теперь вместо падения она выводит сообщения из текста исключения. Она говорит, что она успела выделить 49 объектов, и, самое интересное, это количество объектов, которое у нас находится перед завершением программы, количество объектов, которые сейчас живут в программе. И мы видим, что внезапно это количество стало равно 1. То есть какой-то объект у нас не удалился. Этот объект у нас утёк.

Давайте запустим ещё несколько раз программу... А вот теперь смотрите: мы запустили ещё несколько раз — ещё интереснее! В этот раз у нас все объекты удалились. То есть теперь никто не утёк. Получается достаточно странное поведение программы, плохо предсказуемое. Объект то утекает, то не утекает, и подобное поведение у нас возникло из-за того, что мы ограничили память, доступную программе. Это не очень хорошо. Программа, вообще говоря, должна быть готова к тому, что ей не хватает памяти, и корректно реагировать на данные ситуации.

2.3 Откуда берётся утечка памяти?

Поскольку в основном мы в нашей тестовой программе вызываем функцию `Allocate`, ну и еще деструктор, который занимается удалением, нас будет интересовать в первую очередь функция `Allocate`. Давайте разберем просто по шагам, как она работает.

```
template <typename T>
```

```
T* ObjectPool<T>::Allocate() {
    if (free.empty()) {
        free.push(new T);
    }
    auto ret = free.front();
    free.pop();
    allocated.insert(ret);
    return ret;
}
```

- Первое, что мы проверяем — есть ли у нас сейчас свободные объекты. Очередь пуста, поэтому свободных объектов нет. Тогда нам нужно создать новый объект. Мы вызываем `new T` и создаем в куче новый объект типа `T`. А дальше указатель на него мы помещаем в очередь свободных объектов.

- После этого нам нужно обозначить этот объект как занятый. Мы же вызываем функцию `Allocate`, то есть он у нас будет занятый. Для этого нам нужно указатель из очереди переместить в множество. Это мы сделаем в несколько этапов.

- Для начала сохраним этот указатель в локальном объекте. У нас есть локальный указатель `ret`, он ссылается на тот же самый объект.

- После этого мы удаляем указатель с вершины очереди вот таким образом, теперь очередь у нас снова пуста.

- И, наконец, мы помещаем этот указатель в множество занятых объектов. Дальше функция завершает работу с помощью функции `return`, и этот указатель передается на вызывающую сторону, куда-то там, и там уже, собственно, его используют так, как нужно. Состояние программы, состояние нашего класса `ObjectPool`, как мы видим: у нас есть один объект, и указатель на этот объект у нас содержится в множестве занятых объектов.

Давайте теперь, собственно, посмотрим, что же произойдет, когда у нас не хватит памяти. Итак, мы вызываем функцию `Allocate`, успешно выделяем объект, начинаем перекладывать указатель на этот объект в множество занятых объектов. А теперь давайте вспомним, что же такое множество. Множество, как вы знаете, это, по сути, дерево. Дерево — это динамическая структура. Отдельные элементы дерева, они тоже выделяются в куче. То есть для того чтобы добавить указатель в это множество, нам нужно под этот указатель создать элемент. А элемент-то этот, он тоже берется из кучи, он создается динамически. То есть в этот момент нам может не хватить памяти, мы можем не суметь выделить новый элемент для множества. И вот если ровно в этот момент нам не хватает памяти, то тогда у нас у бросается исключение `std::bad_alloc`.

В случае исключения, как мы помним, мы начинаем раскрутку стека, то есть из этой функции мы выходим и после этого уничтожаем все локальные переменные функции. В данном случае это одна переменная `ret`, и мы ее уничтожаем. Все, теперь данная функция у нас завершила свою работу, началась раскрутка стека, мы ушли выше по стеку. Посмотрим на то, в каком состоянии остался наш, собственно, `ObjectPool`.

Объектов у нас 49, а указателей на эти объекты всего 48 из множества занятых. И последний объект, который мы создали на данном вызове, получается, утек, на него не указывает ни один указатель. У нас нет шансов его удалить, мы просто не знаем, как это сделать. И вот он у нас как раз и останется в конце работы программы.

Теперь, когда мы разобрались с тем, почему у нас возникает утечка объектов, вопрос к вам: а как вы думаете, почему в некоторых случаях утечка все-таки не возникает? Давайте теперь разберемся с тем, почему иногда утечка всё-таки может не возникать. Еще раз прокрутим последнюю итерацию работы функции. Итак, мы заходим, проверяем, есть ли у нас свободные объекты — объектов нет — и выделяем объект. Собственно, вот: мы выделяем объект в памяти. А памяти может не хватить, мы же с этого начали? То есть исключение может быть выброшено именно в этот момент: пытаемся выделить объект — выскакивает исключение. Начинается раскрутка стека, мы выходим из функции, локальных переменных нет, и это то состояние, в котором остается наша программа. Это абсолютно корректное состояние. То есть в данном случае исключение не привело ни к каким проблемам.

Вообще это нормально, что в программе происходит исключение, важно просто быть к этому готовым. Компилятор, когда он создавал новый объект, он был готов, он знал, что может памяти не хватить, и он позаботился о том, чтобы этот объект не был создан и память не утекла. Абсолютно аналогичная ситуация произойдет, если вы в конструкторе объекта выбросите исключение. То же самое, компилятор увидит: «Ага, не смог создать объект. Ну ладно, хорошо, ничего страшного. Выброшу исключение, ничего утекать не будет». А мы, со своей стороны, к сожалению, оказались не готовы, потому что у нас объект утек. Теперь мы разобрались, что у нас происходит.

Давайте, собственно, попробуем это исправить. Мы знаем, какая строчка у нас бросает исключение. Строчка, где у нас происходит вставка указателя в множество занятых объектов. Давайте сначала попробуем исправить наивно. Возникнет вопрос: а что блок `catch()` должен возвращать в этом случае? Первая мысль, которая может прийти нам в голову: давайте вернем ноль, нулевой указатель, то есть у нас же не получилось выделить объект. Но нет. Функция `Allocate` должна вернуть указатель на существующий объект. Она не может вернуть нулевой указатель. Функция `TryAllocate` может вернуть нулевой указатель, `Allocate` — не может. Если мы вернем нулевой указатель, мы нарушим контракт. Это будет очень плохо.

Получается, что мы оказываемся в ситуации, когда мы не можем выполнить контракт функции. Существует только один вариант, что делать в этом случае: бросать исключение. Возникает тогда следующий вопрос: а какое исключение бросать? Здесь все просто, у нас уже есть некоторое исключение, вот этот `bad_alloc`, и нам нужно просто пробросить его дальше. Это будет наиболее логичное поведение. И вот такая **операция проброса исключения дальше** в языке C++ **обозначается просто** `throw`, без параметров. То есть когда мы не указываем, что именно мы делаем `throw`, значит, мы берем текущее исключение, которое у нас прилетело в текущий `catch block`, и просто пробрасываем его дальше. **Вне блока** `catch` **писать** `throw` **без параметров нельзя**. Будет ошибка компиляции.

```
template <typename T>
T* ObjectPool<T>::Allocate() {
    if (free.empty()) {
        free.push(new T);
    }
```

```
    auto ret = free.front();
    free.pop();
    try {
        allocated.insert(ret);
    } catch(const bad_alloc&) {
        delete ret;
        throw;
    }
    return ret;
}
```

Давайте соберем нашу программу и посмотрим, что у нас происходит сейчас. Итак, нам действительно удалось исправить эту утечку. Однако давайте поймем, какой ценой нам удалось это сделать. Мы добавили `try/catch` вокруг функции добавления элемента в множество.

- **Во-первых**, добавление этого `try/catch` в нашу небольшую функцию, оно затрудняет восприятие логики. То есть у нас был очень простой код, мы по шагам брали указатель, присваивали, перекладывали его в множество. Теперь у нас появился какой-то `try/catch`, в который мы в явном виде вызываем `delete`, а потом еще перебрасываем исключение — смотрится это жутковато.

- **Во-вторых**, на самом деле, на самом деле, это не полное решение, и проблема у нас все еще может быть. Вспомните, что у нас есть очередь, в которую мы складываем свободные объекты. А очередь — это тоже динамический объект. Очередь, ее элементы тоже выделяются в куче. То есть, вообще говоря, у нас есть еще одно место, которое может бросить исключение, которое приведет к утечке. Его мы здесь просто не исправляли. И, в принципе, подобные исправление, его очень легко забыть сделать, и очень легко изначально ошибиться. То есть в том решении, которое изначально мы разбирали, эта ошибка действительно была, и она была чертовски неочевидна. Если бы мы не написали такую специфическую тестовую программу, мы бы ее даже не нашли.

- **И в целом** это просто не идиоматический C++. Мы используем контейнеры и очередь, множества. И вроде бы эти контейнеры должны скрыть от нас заботу об управлении динамической памятью, но при этом почему-то нам пришлось задумываться о том, а что, если сейчас мы не сможем добавить элемент в множество? Это очень плохо. Нужно решать это дело по-другому. Как вы уже можете догадаться, решать с помощью умных указателей.

2.4 Умный указатель unique_ptr

Давайте подумаем, как же нам исправить эту проблему без использования блока `try/catch`. Заметим, что проблемы изначально начались из-за того, что мы в явном виде вызываем оператор `delete`. В самом деле, это ровно то, что мы сделали в блоке `catch`. Мы вызвали оператор `delete` и отправили исключение дальше по стеку.

Взглянем вообще на нашу систему. У нас есть наш вызывающий код, у нас есть объект, которым мы создаем, и есть указатель, который из нашего кода на этот объект указывает. Мы попробовали вызывать `delete` на нашей стороне, в коде. Получилось плохо, следовательно, нам не нужно

вызывать `delete`. Что же нам остается? Может быть, нам следует вызвать `delete` на стороне объекта, который мы создаем? На самом деле такие техники действительно существуют. То есть есть такой подход, когда объект сам себя удаляет, но это очень специфическая и продвинутая техника, и сейчас мы о ней разговаривать не будем. Самое главное, что она нестандартная. Остается что? Остается, на самом деле, указатель, правильно? Давайте сделаем так, чтобы указатель, который у нас есть на объект, чтобы он занимался удалением этого объекта. И мы действительно можем это сделать, но не с обычным указателем, обычный указатель так не умеет. Нам понадобится умный указатель. И этим мы сейчас и займемся, мы посмотрим на умный указатель под названием `unique_ptr`.

Давайте продемонстрируем его возможности в небольшой тестовой программе. Для начала мы заведем некоторый класс, который у нас будет уметь говорить нам о том, что с ним происходит. То есть в конструкторе он нам будет выводить, что он создался. Дальше в деструкторе он будет нам говорить, что он умер. И дальше у него будет некоторая функция, чтобы он делал что-нибудь полезное.

```
#include <iostream>

using namespace std;

struct Actor {
    Actor() {
        cout << "I was born! :)" << endl;
    }
    ~Actor() {
        cout << "I died :(" << endl;
    }
    void DoWork() {
        cout << "I did some job" << endl;
    }
};

int main() {
    auto ptr = new Actor;
    ptr->DoWork();
    delete ptr;
    return 0;
}
```

Давайте запустим нашу программку. Что она выводит? Объект создался, затем что-то сделал и затем умер. Все логично, все нормально. Давайте теперь заведем небольшую функцию, которая... Мы не всегда работаем с объектами напрямую, мы очень часто объект куда-то передаем, и там уже с ним, собственно, как-то работаем. Давайте заведем функцию, которая будет принимать указатель на `Actor` и будет проверять, если указатель ненулевой, то будет, собственно, выполнять какую-то работу, а если нулевой, то она нам напишет «An actor was expected».

```
void run(Actor* ptr) {
    if(ptr) {
        ptr->DoWork();
    } else {
        cout << "An actor was expected :(" << endl;
    }
}

int main(){
...
run(ptr);
...
}
```

Так, хорошо. Программка скомпилировалась, запускаем. Вывод не изменился, мы просто взяли этот вызов и перенесли в функцию.

Теперь. Вот допустим... Что мы сделали? Мы создали объект, и, допустим, мы забыли вызвать оператор `delete`. Давайте его просто удалим. Как, я думаю, вы уже прекрасно понимаете, в этом случае объект у нас удаляться не будем, и мы не увидим строчки о том, что объект умирает. Запускаем — да, действительно, объект был создан, что-то сделал и после этого не умер.

Давайте теперь вспомним о том, зачем мы здесь собрались. Нам нужен какой-то указатель, который будет уметь сам удалять объект. И вот, собственно, используем `unique_ptr`. Для этого нам нужен будет заголовочный файл `memory`.

```
#include <memory>
...
int main() {
    auto ptr = unique_ptr<Actor>(new Actor);
    // run(ptr);
    return 0;
}
```

Давайте скомпилируем программу. И обратите внимание, что у нас сейчас в этой программе все еще нет явного вызова оператора `delete`. Но давайте ее запустим. И мы видим, что объект был создан и объект был успешно уничтожен. То есть, действительно, `unique_ptr` сам позаботился о том, чтобы уничтожить объект, а мы ничего не делали. И это замечательно. Похоже, что мы двигаемся в правильном направлении. Давайте посмотрим, что еще мы можем делать с умным указателем.

Попробуем вызвать функцию `run`, в которую мы передавали обычный указатель, и скомпилируем код. У нас будет ошибка компиляции. Компилятор нам говорит, что не может преобразовать `unique_ptr<Actor>` к `Actor*`, то есть к обычному указателю. В принципе, логично, это же все-таки разные сущности. Поэтому нам нужно из умного указателя как-то достать обычный указатель на тот объект, на который он указывает. Для этого существует метод `get`.

```
...
run(ptr.get());
```

Скомпилируем программу, запустим ее. И да, мы видим, что функция успешно вызвана. То есть функция, которая принимает обычный указатель. Действительно, когда мы пишем какой-то код, который использует объект, нам не нужно задумываться о том, каким образом хранится этот объект. Или это объект какой-то локальный; или он хранится в обычном указателе и мы вызываем `delete`; или он управляется умным указателем — нам это не нужно знать, то есть мы просто пишем код и принимаем указатель.

Давайте посмотрим, что еще мы можем делать с `unique_ptr`. Давайте для начала попробуем, попробуем его скопировать.

```
...
auto ptr2 = ptr;
```

Соберем такую программу. И что она нам скажет? Она нам скажет, что нет, «не могу я такое скомпилировать, потому что вы пытаетесь использовать конструктор копирования `unique_ptr`, а у `unique_ptr` конструктора нет». Он `unique` не просто так, он уникальный. **На один и тот же динамический объект может указывать только один** `unique_ptr`. А здесь мы попытались как бы скопировать указатель, то есть чтобы два `unique_ptr` указывалои на один и тот же объект. Так делать нельзя. Мы не можем копировать `unique_ptr`, но мы можем его перемещать.

```
...
auto ptr2 = move(ptr);
```

Так, программа, программа успешно компилируется, всё хорошо.

Теперь попробуем что-нибудь сделать с этим указателем, который мы получили. Например, запустим на нем ту же самую функцию `run`.

```
int main() {
    auto ptr = unique_ptr<Actor>(new Actor);
    run(ptr.get());
    auto ptr2 = move(ptr);
    run(ptr2.get())
    return 0;
}
```

Теперь мы видим, что у нас дважды была выполнена работа: первый раз на исходном указателе, второй раз на скопированном указателе. И теперь вопрос к вам: а как вы думаете, что происходит с исходным указателем после того, как мы из него переместили? Как вы, возможно, догадались, когда мы перемещаем из одного `unique_ptr` в другой `unique_ptr`, исходный `unique_ptr` оказывается пустым, потому что других возможностей, на самом деле, практически нет. Указатели должны оставаться уникальными, при этом указатель не будет заниматься копированием объекта. То есть единственное, что мы можем сделать — это обнулить то, на что ссылается исходный умный указатель, исходный `unique_ptr`. И давайте, собственно, попробуем вызвать функцию `run` на исходном указателе после того, как мы его переместили.

```
...
run(ptr2.get());
run(ptr.get());
```

Запустим такой код. И, действительно, у нас появляется новая строчка, которая говорит о том, что мы вообще-то ожидали некоторый объект, а передали-то нам нулевой указатель. То есть мы убедились, что исходный `unique_ptr` у нас действительно оказался нулевым после перемещения.

Ну и последнее замечание о `unique_ptr`, которое нам сейчас понадобится, — это то, что для создания `unique_ptr` используется специальная функция. Например, смотрите, какая у нас есть проблема в записи. Мы написали `unique_ptr<Actor>`, то есть в явном виде указали тип, и снова написали `new Actor` же. То есть мы эффективно как бы повторили тип объекта. Это, на самом деле, если тип большой, это может быть реальной проблемой. И кроме того, с такой записью связаны еще некоторые проблемы, о которых мы поговорим попозже. Но сейчас мы воспользуемся, для создания умного указателя на новый объект, специальной функцией `make_unique`. Она создана именно для этого. То есть вы не должны писать `unique_ptr` от `new` какой-то объект, вам следует писать `make_unique`. Как минимум вы уже понимаете, что это сэкономит вам место на экране. Но кроме того, она обладает еще некоторыми полезными свойствами, о которых мы поговорим позже.

```
auto ptr = make_unique<Actor>();
```

Попробуем собрать такую программу. Программа, на самом деле, получается практически эквивалентной. Запустим ее, и да, у нас все в порядке.

Таким образом,

- основная функция `unique_ptr` заключается в том, что в своем деструкторе он сам удаляет тот объект, на который он ссылается. Если он не ссылается ни на какой объект, то есть он был обнулен, то в деструкторе он и делать ничего не будет, он просто умрет. И всё. То есть `unique_ptr`, из которого мы переместили, он просто умирает и ничего за собой не удаляет;
- `unique_ptr` нельзя копировать, его можно только перемещать, потому что он уникальный. Мы поняли, что, для того чтобы достать сырой указатель из умного указателя `unique_ptr`, используется метод `get`;
- для того чтобы создавать `unique_ptr` на новый объект, нам нужно использовать функцию `make_unique`.

2.5 unique_ptr для исправления утечки

Давайте применим `unique_ptr` для того, чтобы сделать хорошее исправление в нашей задаче с `ObjectPool`.

Но для того, чтобы использовать `unique_ptr` в качестве ключа ассоциативного контейнера, нам нужно знать о некоторой возможности этого самого ассоциативного контейнера, которую мы пока не проходили. Поэтому мы сделаем реализацию, которая опирается уже на известные нам возможности ассоциативных контейнеров и для этого мы будем использовать `unique_ptr` не в качестве ключа, а в качестве значения. Для этого мы изменим `set` на `map`, `unique_ptr` сделаем значением, а в качестве ключа будем использовать сырой указатель, который будет указывать на тот же самый элемент. В этом случае у нас получится некоторое очевидное дублирование. Но

наша задача здесь не написать оптимальный `ObjectPool`, а просто показать как можно решить проблему с которой мы столкнулись с помощью использования умных указателей.

И дальше мы с вами уже знаем, что использовать упорядоченный контейнер у которого ключами будут являться указатели, большого смысла нет, потому что порядок на указателях не несет особого значения. Поэтому мы используем здесь неупорядоченный контейнер `unordered_map`. Заменим наш `set` на `unordered_map` и также подключим файл `memory`, в котором у нас находится `unique_ptr`.

```
#include "test_runner.h"

#include <algorithm>
#include <string>
#include <queue>
#include <stdexcept>
#include <unordered_map>
#include <memory>

using namespace std;

template <class T>
class ObjectPool {
public:
    T* Allocate();
    T* TryAllocate();

    void Deallocate(T* object);

    ~ObjectPool();

private:
    queue<unique_ptr<T>> free;
    unordered_map<T*, unique_ptr<T>> allocated;
};
```

Давайте теперь посмотрим, что делать с нашей реализацией.

```
template <typename T>
T* ObjectPool<T>::Allocate() {
    if (free.empty()) {
        free.push(make_unique<T>());
    }
    auto ptr = move(free.front());
    free.pop();
    T* ret = ptr.get();
    allocated[ret] = move(ptr);
    return ret;
}
```

Дальше у нас есть функция `TryAllocate`, в ней менять ничего не нужно.

```
template <typename T>
T* ObjectPool<T>::TryAllocate() {
    if (free.empty()) {
        return nullptr;
    }
    return Allocate();
}
```

У нас есть функция `Deallocate`. Она ищет переданный объект в хеш-таблице занятых объектов. Она это делает правильно, но вот `find` нам еще понадобится. Мы его сохраним в отдельный итератор.

```
template <typename T>
void ObjectPool<T>::Deallocate(T* object) {
    auto it = allocated.find(object);
    if (it == allocated.end()) {
        throw invalid_argument("");
    }
    free.push(move(it->second));
    allocated.erase(it);
}
```

Теперь посмотрим на деструктор. Деструктор нам нужен был для того, чтобы удалять объекты, которые мы выделили, но, теперь у нас есть `unique_ptr`, который будет заниматься удалением за нас. Поэтому с деструктором мы сделаем лучшее, что вообще можно сделать с кодом. Мы его удалим.

```
...
class ObjectPool {
public:
    T* Allocate();
    T* TryAllocate();

    void Deallocate(T* object);

private:
...
```

Давайте посмотрим как будет работать наша программа теперь. Мы ее соберем. Так, программа собралась. Давайте еще прежде чем запускать проверим, что мы правильно написали функцию `Deallocate`. Потому что поскольку она является частью шаблонного класса, она у нас просто не будет компилироваться.

```
void run() {
    ObjectPool<Counted> pool;

    pool.Deallocate(nullptr);
    ...
}
```

Так, она успешно скомпилировалась. Давайте теперь запустим нашу программу. Так, мы видим, что у нас после цикла значение 59, перед выходом 0, значит всё в порядке, всё удалилось. Запустим ещё раз. Запустили — всё в порядке. В общем мы вполне успешно справились с утечкой без использования `try/catch`, а с использованием `unique_ptr`.

Давайте посмотрим чуть подробнее, как же у нас работает это наше исправление? Давайте разберем опять по шагам как будет работать функция `Allocate`.

- Заходим в функцию `Allocate`, проверяем, есть ли у нас что-то в очереди свободных объектов. Ничего нет. Поэтому мы создаем новый динамический объект с помощью `make_unique`; `make_unique` возвращает нам `unique_ptr`, который мы сразу же перемещаем в очередь свободных объектов.

- `unique_ptr` перемещается из очереди свободных объектов в локальную `unique_ptr` под названием `ptr`. Обратите внимание, что в этот момент исходный `unique_ptr`, который у нас находится в очереди уже больше никуда не указывает, он пустой.

- Удаляем элемент из очереди. Это приводит к удалению пустого `unique_ptr`, а поскольку он пустой это не влечет за собой никаких дополнительных операций.

- из `ptr` вытаскиваем сырой указатель и сохраняем его локальной переменной `ret`.

- После этого у нас `unique_ptr` перемещается в таблицу занятых объектов. Здесь мы видим, что у нас есть элемент ключ значения и ключ является сырым указателем, значением `unique_ptr` и оба они указывают на один и тот же элемент. После этого мы возвращаем сырой указатель вызывающей стороне.

Отлично! Все отработало корректно. Обратите внимание, как в каждый момент работы функции будет существовать ровно один указатель, который указывает на динамической объект и это очень важно.

Теперь давайте посмотрим, что произойдет если у нас случится нехватка памяти. Так у нас создается новый объект `unique_ptr`, этот объект мы перемещаем в локальную переменную, указываем сырой указатель и вот мы доходим до добавления `unique_ptr` в таблицу. Раньше у нас здесь было множество, сейчас у нас здесь с вами хеш-таблица. Хеш-таблица тоже динамическая структура, поэтому при добавлении элементов в хеш-таблицу у нас тоже может не хватить памяти. Пусть у нас не хватает памяти, у нас возникает исключение, оно вылетает, начинается раскрутка стека. Первым делом у нас удаляется сырой указатель, потому что он объявлен последним в нашей функции. Он удаляется. Это сырой указатель, его удаление не влечет за собой никаких дополнительных действий. А вот дальше, дальше у нас удаляется локальный `unique_ptr`. И как вы уже возможно заметили, этот `unique_ptr` указывает на объект и при своем удалении он удалит

этот объект. Получается, что несмотря на то, что у нас выбросилось исключение, у нас не произошло никакой утечки и наш `ObjectPool` остался в консистентном состоянии и можно свободно продолжать пользоваться, он будет работать и в частности, он абсолютно корректно удалит за собой все объекты, которые он выделил — и это очень важно.

Таким образом, исправление с помощью `unique_ptr`

- упрощает восприятие логики программы;
- решает проблему полностью (в каждый момент времени на объект ссылается ровно один `unique_ptr`);
- сложно забыть или ошибиться (будет ошибка компиляции);
- полностью идиоматический подход в C++ к работе с динамическими объектами

Разбор задачи «Дерево выражения»

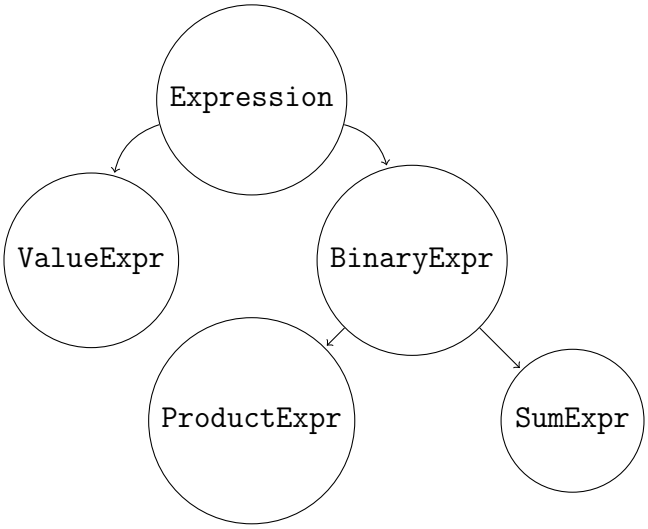
3.1 Разбор задачи «Дерево выражения»

Теперь давайте подробнее рассмотрим задачу построения дерева выражений, потому что на ее примере мы дальше будем развивать и смотреть другие умные указатели, а конкретно рассмотрим авторское решение. Если вы проходили «Желтый пояс», то можете помнить, что на пятой неделе разбиралась очень похожая задача, где мы вручную строили дерево выражения. Но тогда у нас был упор на полиморфизм, то есть мы смотрели, как ведут себя полиморфные объекты, и мы не использовали `unique_ptr`, потому что еще даже не знали семантики перемещения. Мы использовали `shared_ptr` и особо не вдавались в детали его работы. Сейчас же предполагается, что мы, наоборот, уже знаем прекрасно, как работает полиморфизм, и будем акцентировать свое внимание именно на умных указателях. И покажем, как в данном случае работает `unique_ptr`.

- Базовым классом выступает `Expression`.

    1. У него есть наследник `ValueExpr`, который соответствует узлу-константе, то есть когда у вас какое-то конкретное число встречается в выражении, и, соответственно, это число хранится в нем как поле `value_`.
    2. Другой наследник — `BinaryExpr`, он соответствует некоторому двоичному выражению — сложению и умножению.
        (a) У него в свою очередь есть наследник `ProductExpr`, который представляет умножение.
        (b) И есть наследник `SumExpr`, который представляет сложение.

        Причем в этих классах никаких дополнительных полей нет. Всё, что они делают, это переопределяют некоторые виртуальные функции для того, чтобы правильным образом кастомизировать поведение базового класса.



Давайте теперь посмотрим подробнее, как у нас происходит создание дерева выражения, например, вот для такого примера кода

```
ExpressionPtr e1 = Product(Value(2), Sum(Value(3), Value(4)));
```

Посмотрим с самого низу, как у нас отработает функция

1. `Value(3)`: она создаст нам новый объект `ValueExpr` и вернет временный `unique_ptr` на этот объект. В итоге будет сохранено число три. Дальше `Value(4)` аналогичным образом создаст `ValueExpr`, в котором сохраняется четверка, и вернет временный `unique_ptr` на этот объект.

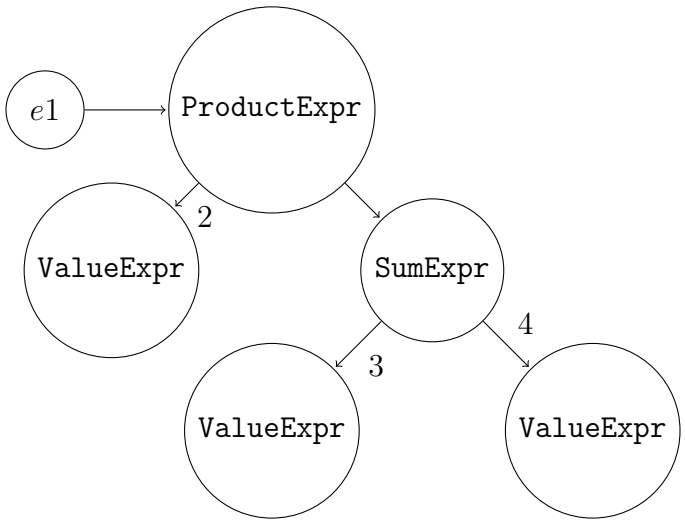
2. Дальше эти временные `unique_ptr` будут переданы в функцию `Sum`, будут перемещены в нее, а та в свою очередь переместит их в новый объект класса `SumExpr`, и они будут сохранены в его полях `left_` и `right_`. Сама же функция `SumExpr` при этом вернет временный `unique_ptr` на вот этот новый созданный `SumExpr`.

3. Дальше `Value(2)` создаст временный `unique_ptr` на `ValueExpr` с двоечкой.

4. И теперь два временных `unique_ptr` будут перемещены в функцию `Product`, которая в свою очередь создаст новый объект `ProductExpr` и переместит эти умные показатели — `unique_ptr` — в его поля `left_` и `right_`.

5. И вот она уже, наконец, вернет некоторый `unique_ptr`, который будет сохранен в локальную переменную `e1`.

Таким образом, в результате нескольких перемещений у нас создается полное дерево выражения.



Теперь мы создали дерево выражения, давайте посмотрим, как оно у нас будет разрушаться.

1. Разрушение дерева выражения начинается с того, что локальная переменная `e1` выходит из `scope` и начинает разрушаться. Начинает работать деструктор `unique_ptr`, он смотрит: так, есть ли какой-то объект, на который я указываю? Да, есть объект, `ProductExpr`. Замечательно. Вызывается `delete` для этого объекта. Начинается разрушение этого объекта, вызывается деструктор `ProductExpr`.

2. Как мы знаем, **деструктор удаляет все поля объекта, причем начиная с конца**. То есть первое, что он начнет делать, он начнет удалять поле `right_`. Итак, он начал удалять поле `right_`, а это, в свою очередь, тоже `unique_ptr`, поэтому он пойдет удалять тот объект, на который он указывает. То есть, начнется удаление объекта `SumExpr`.

3. Тот в свою очередь тоже начнет удалять свои поля, и первым делом начнет удалять поле `right_` — это тоже `unique_ptr`, и он в свою очередь запустит удаление объекта `ValueExpr`.

4. И вот только `ValueExpr`, который как бы «лист» в нашем дереве, он за собой ничего не потянет. Соответственно, здесь у нас продолжится разрушение объекта `SumExpr`, теперь у него поле `left_` будет уничтожаться — это `unique_ptr`, который уничтожит объект `ValueExpr`. Всё, на этом у нас закончилось удаление объекта `SumExpr`, мы возвращаемся назад по стеку.

5. И теперь у нас удаляется поле `left_` для `ProductExpr`.

6. Он тянет за собой `ValueExpr`.

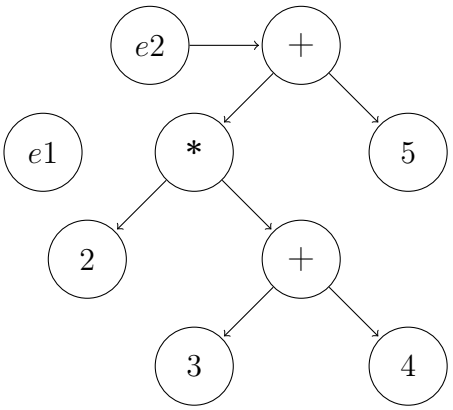
7. И вот только теперь удаляется сам `ProductExpr`.

8. И наконец-то удаляется исходный `unique_ptr e1`, который у нас был локальной переменной.

То есть что произошло? Смотрите: у нас компилятор за нас сделал нам рекурсивный обход этого дерева, хотя мы ничего подобного вообще не писали. Все, что мы сделали, — это просто завели два поля `unique_ptr`. Семантика работы компилятора такая, что когда он начал удалять, он просто рекурсивно прошелся и все дерево за нас удалил в абсолютно правильном порядке, а мы этого даже не писали — а если мы этого не писали, мы не могли ошибиться — это очень хорошо.

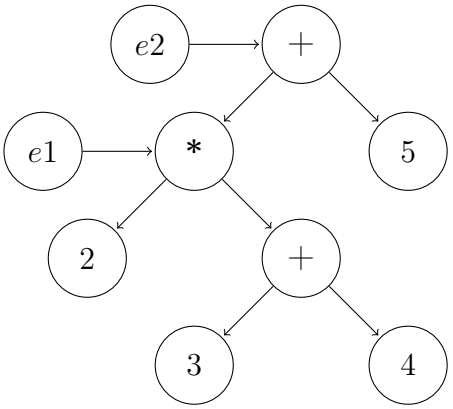
Давайте теперь на сокращенной записи посмотрим, как у нас будет создаваться дерево выражения для e2, которое не просто конструирует выражение с нуля, а берет уже существующее.

```
ExpressionPtr e2 = Sum(move(e1), Value(5));
```



Поскольку мы переместили из указателя `e1`, как мы знаем, в нем у нас ничего не осталось. Если мы попытаемся его распечатать, то нам скажут, что такого выражения просто нет. Это абсолютно корректная работа, так нас, собственно, и просили сделать.

Однако теперь давайте подумаем. Изменились некоторые требования. Это абсолютно корректная ситуация, у нас были исходные требования, мы их реализовали, потом нам говорят: да, это очень хорошо, но мы хотим уметь делать кое-что еще. А конкретно мы хотим продолжать уметь пользоваться вот этим `e1` для того, чтобы иметь доступ к поддереву, на которое он указывал. Мы хотим примерно вот такую картину:



Но из такой картины становится понятно, что `unique_ptr` нам здесь уже не подойдет. Потому что смотрите: действительно, у нас на узел умножения здесь будет указывать уже два `unique_ptr`, а мы прекрасно понимаем, что так делать нельзя — `unique_ptr` **может ссылаться только на один объект**. Если мы сделаем такую ситуацию, то у нас каждый из этих `unique_ptr` попытается удалить этот узел умножения, и, конечно, ничего хорошо из этого не получится. Значит, `unique_ptr` нам не подходят. Хорошо, что делать?

Если `unique_ptr` не подходит, какие мы еще знаем указатели? Есть сырой указатель. Давайте попробуем завести некоторый сырой указатель `ptr`, который указывает на узел умножения. И в какой-то степени да, это даже будет работать. Пока жив `e2`, мы действительно сможем возвращаться по этому указателю `ptr`. Проблема в том, что **когда `e2` удалится, он потянет за**

собой удаление всего дерева. Если `e2` удаляется, то и все дерево удаляется за ним. И `ptr` у нас получается висячим, то есть по нему мы уже не сможем безопасно обратиться, а нам бы хотелось, чтобы то поддерево, на которое мы указывали, продолжало существовать. И как раз для того чтобы этого добиться, нам с вами понадобится новый умный указатель под названием `shared_ptr`, о котором мы поговорим далее.