Основы разработки на C++. Эффективное использование ассоциативных контейнеров

Оглавление

Эффективное использование ассоциативных контейнеров

1.1 Введение в ассоциативные контейнеры

Ассоциативные контейнеры — это контейнеры, которые хранят значения по различным ключам. Например, контейнер `map` — это их типичный представитель. Пример:

```
map<string, int> counter;
```

В примере ключом является строка. Зная эту строку-ключ, вы можете **добавлять**:

```
counter["apple"] = 1;
```

модифицировать:

```
++counter["apple"];
```

искать:

```
cout << counter["apple"] << endl;
```

и **удалять данные** в контейнере:

```
counter.erase("apple");
```

Другим примером, еще более простым, можно считать контейнер `vector`, где ключи — это просто индексы, целые числа.

```
vector<double> values;
```

Для того чтобы познакомиться с другими, более эффективными контейнерами, давайте решим наглядную задачу. Для задачи нам нужен хороший, длинный текст. Мы выбрали текст про Шерлока Холмса. Давайте проанализируем его и узнаем, какие слова встречались в тексте чаще всего. Обратимся к нашему коду.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <unordered_map>
```

```
#include "head.h"
#include "profile.h"

using namespace std;
```

Мы завели контейнер map от типов `string, int`, чтобы считать количество строк, и текст записан в файле input.txt

```
int main() {
    map<string, int> freqs;
    ifstream fs("input.txt");
```

Давайте прочитаем его и положим содержимое файла в контейнер `freqs` с частотами. Напишем цикл, мы должны это считывать в какую-то переменную строчки. Пока у нас файл не пустой, мы в контейнер `freqs` добавляем эту строчку, если ее нет. А если есть, то увеличиваем значения.

```
    string text;
    while (fs >> text) {
        ++freqs[text];
    }
```

Давайте распечатаем содержимое этого контейнера в цикле и убедимся, что там лежит то, что мы ожидаем. Для этого воспользуемся функцией `head`, которая объявлена в заголовочном файле `head.h`, вы ее можете помнить из наших прошлых курсов. Мы ее использовали затем, чтобы распечатать только первые десять элементов из нашего контейнера.

```
    for (const auto& [k,v] : Head(freqs, 10)) {
        cout << k << '\t' << v << endl;
    }
    cout << endl;
```

Давайте соберем нашу программу и запустим. Что мы видим? Что у нас действительно есть список слов. И у нас есть частоты этих слов, то есть мы знаем, сколько раз каждое слово встречалось в тексте. И это здорово, однако мы хотим решить не ту задачу.

Здесь у нас слова отсортированы по алфавиту, потому что у нас в контейнере `map` ключом является строка, а нам нужно отсортировать их по частоте, чтобы выбрать самые часто используемые. Как нам это сделать?

Давайте просто переложим эти слова из `map` в вектор `pair`. Почему `pair`? Потому что у нас в контейнере `map` хранятся пары ключ/значение. Мы эти самые пары ключ/значение, собственно такого ровно типа, переложим в вектор `words` и скопируем в него все эти пары из контейнера `freqs` от начала до конца.

```
    vector<pair<string, int>> words(freqs.begin(), freqs.end());
```

Затем мы отсортируем этот вектор в нужном нам порядке. Напишем компаратор, который сравнивает частоты. И таким образом после сортировки этого вектора в его начале будет лежать то, что нам необходимо — самые часто используемые слова.

```
    sort(words.begin(), words.end(), [](const auto& l, const auto& r) {
        return l.second > r.second;
    }
```

Что же, давайте выберем эти слова. Возьмем наш цикл, который печатает, и распечатаем, правда, не контейнер `freqs`, а распечатаем вектор `words`, который у нас к этому моменту уже отсортирован.

```
    for (const auto& [k,v] : Head(words, 10)) {
        cout << k << '\t' << v << endl;
    }
    cout << endl;
}
```

Компилируем и запускаем программу, смотрим, что получилось. В первом выводе то, что у нас лежит в контейнере `map`, это первые десять слов по алфавиту. Они отсортированы по ключу. Во втором — это слова, которые отсортированы по частотам. Вот у нас самое часто используемое слово — это артикль the. И дальше идут предлоги, другие артикли, местоимения, собственно, ожидаемо. Так и должно быть в английском языке. Мы можем сделать предположение, что мы сейчас правильно решили нужную нам задачу. Далее мы узнаем, как решить эту задачу эффективнее.

1.2 Размен отсортированности на производительность

Мы собираемся решить эту задачу более эффективно. Что значит более эффективно? Значит мы хотим, чтобы решение работало быстрее, чем у нас сейчас есть, и давайте для начала замерим, сколько же по времени работает наше имеющееся решение. Для этого возьмем код, написанный выше, и для измерения используем макрос `LOG_DURATION`, который вы знаете из прошлых курсов. Этот макрос определен в заголовочном файле `profile.h`.

Здесь мы хотим посмотреть, какие этапы были в нашем решении, и сколько по времени работал каждый этап. Первый этап можем выделить, это когда мы заполняли контейнер `map` словами из файла.

```
    {
        LOG_DURATION("Fill");
        ifstream fs("input.txt");
        string text;
        while (fs >> text) {
            ++freqs[text];
        }
    }
```

И что у нас идет дальше? Дальше мы делали две операции: заполняли вектор копиями из `map` и затем сортировали вектор. Давайте замерим, сколько у нас занимала оставшаяся часть тоже.

```
    LOG_DURATION("Copy and Sort");
    vector<pair<string, int>> words(freqs.begin(), freqs.end());
```

```
sort(words.begin(), words.end(), [](const auto& l, const auto& r) {
    return l.second > r.second;
}
```

Компилируем программу, запускаем ее. Что мы видим? Вот у нас программа работает как раньше. Слова, слова отсортированные в другом порядке, заполнение у нас заняло около секунды, а вот пересортировка почти нисколько, то есть у нас получается почти все время программа заполняла `map` словами из файла.

То есть, если мы хотим ускорить выполнение программы, нам нужно ускорять в первую очередь заполнение контейнера словами из файла, а все остальное оно и так быстро работает.

Давайте посмотрим где же мы тут можем попробовать сэкономить. Мы видим, что слова в контейнере `map`, упорядочены по алфавиту, по ключу, ключ у нас это строка, именно поэтому они упорядочены по алфавиту. Мы не хотим, собственно, иметь эти слова отсортированные по алфавиту, потому что нам этот порядок вообще не интересен. Мы их потом все равно пересортируем в нужном нам порядке по частоте, поэтому было бы здорово, если бы мы смогли от этой сортировки избавиться в пользу какого-нибудь другого контейнера, который делает то же самое, но не тратит время на сортировку, и оказывается, такой контейнер действительно есть, и он называется ожидаемым именем. Он называется `unordered_map`, и чтобы его использовать в этой программе, достаточно просто изменить тип `map` на `unordered_map`.

```
unordered_map<string, int> freqs;
```

Давайте скомпилируем, проверим что все собралось. Изменив всего лишь тип контейнера, мы получили работоспособную программу, запускаем ее и смотрим, что же получилось. Во-первых, к счастью ответ получился точно такой, как нам нужен. Это точно такой же ответ как был в прошлый раз, это самые частотные слова, упорядоченные по чистоте в порядке убывания, вот мы видим тот же самый список предлогов, артиклей и так далее, а вот те слова, которые были получены из файла. Мы видим, что это наверное те же самые слова, только распечатанные сейчас совсем в другом порядке. Они сейчас не упорядочены по алфавиту, и по частоте не упорядочены, они вообще никак не упорядочены, но нам это и не важно, нам тут никакой порядок не нужен, потому что потом мы их скопируем и отсортируем как нам нужно. Но что здесь хорошо? То, что у нас заполнение контейнера стало работать почти в три раза быстрее.

Что у нас в итоге получилось сейчас? Мы сравнили два способа решения задачи:

- один способ — это используя контейнер `map`, когда мы читаем слова из файла, складываем их в контейнер map, потом копируем вектор и сортируем в нужном нам порядке, и такое решение работает по продолжительности около секунды.
- Второе решение, аналогичное, только в качестве контейнера мы используем **не map**, а `unordered_map`. Все остальное то же самое, результат — точно такое же. Но оно работает почти в три раза быстрее и это хорошо, это нам подходит намного больше.

1.3 Внутреннее устройство ассоциативных контейнеров

Мы узнали, что контейнер `unordered_map` в некоторых случаях работает быстрее, чем обычный `map`. Разберем почему он быстрее. Для этого мы решим задачу, в которой нам необходимо реализовать свой простой ассоциативный контейнер, а именно электронную копилку. Мы хотим уметь считать сколько купюр разного номинала у нас накопилось. Давайте напишем код и посмотрим, что у нас получается.

```
#include <iostream>
#include <vector>
#include <array>

using namespace std;

int main() {
```

Значит, купюра у нас задается номиналом и пускай у нас все купюры приходят из входного потока.

```
    int nominal;
    while (cin >> nominal) {

    }
```

Вот мы их все считали. Дальше, нам их надо куда-то сложить. Давайте для этого заведем вектор, и индексом в этом векторе будут номиналы купюр. Этот вектор мы вынуждены завести длиной по 5001, потому что у нас максимальная купюра это 5000 и еще плюс один, потому что индексация начинается с нуля.

```
    vector<int> cash(5001);
```

Когда мы вводим число с клавиатуры или с входного потока, мы складываем его в нашу копилку, то есть увеличиваем счетчик. Таким образом, мы сейчас считали все номиналы купюр из входного потока и положили их вектор. В итоге:

```
    vector<int> cash(5001);
    int nominal;
    while (cin >> nominal) {
        cash[nominal]++;
    }
```

Что делаем дальше? Дальше нам нужно напечатать, что же у нас получилось. Значит, пробежимся по всему вектору, и для тех купюр, которые присутствуют в копилке: присутствует — значит, что у них счетчик не равен нулю, мы напечатаем их количество.

```
    for (int i = 0; i < cash.size(); ++i) {
        if (cash[i] != 0) {
            cout << i << " - " << cash[i] << endl;
        }
    }
```

Запускаем, собрали. Где мы возьмем купюры? Они у нас заранее приготовлены и лежат во входном файле input.txt.

Вот мы видим, что у нас есть набор каких-то купюр. Запустим нашу программу на этом входном файле. Наша программа работает и она действительно подсчитывает для купюр их количество — то, что надо.

Казалось бы, то, что надо, но давайте посмотрим какие недостатки есть у нашей программы:

- Из плюсов отметим то, что **программа очень простая**, она просто использует номинал купюр в качестве индекса и делает инкремент в векторе. Все суперпросто, ошибиться невозможно.
- Какие недостатки? Мы для такой простой задачи выделили **вектор длиной 5001 — это чрезмерно много**, особенно, если мы вспомним, что различных купюр у нас намного меньше (всего лишь 8 в файле input.txt).

Представьте, что у нас максимальная купюра была бы не 5000, а просто какого-нибудь сумасшедше большого номинала, у нас бы даже памяти не хватило выделить такой большой вектор, как бы мы тогда решали задачу? Вернемся к коду, исправим 5001 на 8 и думаем: как же нам быть? Мы можем сложить разные купюры в вектор длины 8, если напишем функцию, которая вычисляет индекс для купюры: для каждого номинала она проверяет, какая купюра пришла на вход, и возвращает соответствующий индекс. Таких сравнений у нас должно быть, соответственно, 8, в них будут фигурировать все наши купюры, и соответственно, они будут возвращать индексы.

```
int GetIndex(int n) {
    if (n == 10) return 0;
    if (n == 50) return 1;
    if (n == 100) return 2;
    if (n == 200) return 3;
    if (n == 500) return 4;
    if (n == 1000) return 5;
    if (n == 2000) return 6;
    if (n == 5000) return 7;
}
```

Там где мы обращаемся к элементам вектора мы используем функцию получения индекса.

```
    while (cin >> nominal) {
        cash[GetIndex(nominal)]++;
    }
```

Казалось бы все, но единственное, что будет некрасиво, что мы будем распечатывать индекс от нуля до семи вместо обозначения купюры. Давайте заведем вспомогательный вектор, тоже длинны 8, в который сложим человеческие названия наших купюр. Это будут, конечно, те же самые номиналы.

```
    static array<int, 8> names = {
        10, 50, 100, 200, 500, 1000, 2000, 5000
    };
```

```
    for (int i = 0; i < cash.size(); ++i) {
        if (cash[i] != 0) {
            cout << names[i] << " - " << cash[i] << endl;
        }
    }
```

Запускаем, собрали. Отлично, теперь мы видим, что программа работает, что она возвращает тот же самый результат, который мы добились с самого начала. И мы сэкономили очень большое количество памяти написав функцию получения индекса.

Это отлично, более того, из кода становится понятно, что мы можем задавать купюры и другим способом, например, используя их названия, а не числовой номинал. Для нас задача практически не меняется, мы просто перепишем функцию получения индекса, где будем сравнивать строки вместо чисел, и таким образом, мы сможем использовать строковые названия в качестве ключей.

Более того, вместо купюр мы можем решать аналогичную задачу для произвольного набора строк. Все что нам понадобится сделать это написать функцию перевода этих строк в индекс. Например, если мы сможем написать такую функцию для набора цветов, то мы сможем использовать эти цвета в качестве ключей.

```
int GetIndex(int n) {
    if (n == "orange") return 0;
    if (n == "red") return 1;
    if (n == "yellow") return 2;
    ...
}
```

Для такой функции получения индекса есть специальное название, она называется **хеш-функцией**. Мы увидели, из примеров, как хеш-функция помогает нам использовать строки в качестве ключей, отображая их в индексы. Что же дальше? Дальше мы отметим, что хеш-функция может, в общем случае, отображать в индексы не только строки, но и произвольные объекты.

Рассмотрим в качестве объекта российский автомобильный номер, он состоит из нескольких цифр, букв и номера региона. Предложим простую хеш-функцию, которая определяет индекс автомобильного номера, как число из его середины. С помощью нее мы можем раскладывать номера по тысяче разных корзин. Обратите внимание, что всевозможных автомобильных номеров существует очень много, их никак не получится разложить всего лишь по тысяче корзин без повторений. Если два номера отличаются между собой только буквами или регионом, то они попадают в одну корзину, такая ситуация называется **коллизией**.

1.4 Внутреннее устройство unordered_map и unordered_set

Изучим, как внутри устроены контейнеры `unordered_map` и `unordered_set`. Они основаны на так называемых **хеш-таблицах**. Давайте рассмотрим как хеш-таблица устроена. Она состоит из нескольких корзин, по которым раскладываются объекты, и как мы уже знаем, для получения индекса корзины применяется хеш-функция. Это все выглядит очень просто, пока мы не вспомним о существовании коллизии.

Попытаемся добавить в хеш-таблицу автомобильный номер, для которого хеш-функция выдает

индекс корзины 227. Пусть эта корзина уже занята другим номером с другими буквами и из другого региона. Что же делать? Не нужно бояться. Существует несколько способов разрешения коллизий, и один из наиболее распространенных — **метод цепочек**. Этот метод заключается в том, что вместо самих объектов в корзинах хранятся списки объектов. При попытке вставить объект в корзину, сначала проверяется, нет ли его уже в списке, и если нет, то объект добавляется. Таким образом в случае возникновения коллизий в корзине оказывается более одного объекта.

Таким образом цепочки могут удлинятся и удлинятся при вставках, но обычно на практике такие цепочки не становятся слишком длинными. Те хеш-таблицы, которые широко распространены, устроены таким образом, чтобы цепочки оставались короткими.

Сейчас мы рассмотрели, что происходит при вставке объекта в хеш-таблицу. Другие операции — поиск и удаление — работают похожим образом. **Все операции состоят из двух основных этапов**:

1. Сначала с помощью хеш-функции для объекта вычисляется индекс корзины;
2. Если корзина не пуста, происходит последовательный поиск объекта по списку (объект сравнивается со всеми имеющимися).

Теперь давайте поговорим о сложности операций в хеш-таблицах, таких как вставка, поиск и удаление. Это **сложность складывается из двух основных слагаемых**:

1. Вычисление хеш-функции — $O(1)$
2. Поиск объекта в корзине. В среднем — $O(1)$

Итак, мы узнали, что в основе `unordered`-контейнеров лежат хеш-таблицы и рассмотрели, как они работают: `unordered_set` хранит в хеш-таблице ключи, а `unordered_map` хранит в них пару: ключ-значение.

1.5 Внутреннее устройство map и set

Теперь давайте рассмотрим контейнеры `map` и `set` и разберемся, почему они работают медленнее. В документации написано, что внутри `map` представляет из себя **сбалансированное двоичное дерево поиска (красно-черное дерево)**. Каждое из этих слов важно; разберемся, что они все значат.

- Двоичное дерево — у каждого узла дерева может быть не более двух потомков;
- Поиска — объекты в дереве упорядочены. Все значения узлов поддерева, меньшие данного узла, попадает в его левое поддерево, а все значение большие данного узла — в правое.
- Сбалансированное — для каждого узла высоты его левого и правого поддеревьев примерно равны. В некоторых реализациях, например, они могут отличаться не более чем на один.

Давайте посмотрим, как же работает поиск в бинарных деревьях поиска. Работает он очень просто: если мы вспомним о том, что узлы упорядочены. Если мы начинаем искать какое-то значение, мы сравниваем его всегда сначала с корнем. Если значение меньше, чем значение в корне, то мы уходим в левое поддерево. Если там мы натыкается на число большее, то мы уходим в правое поддерево и так далее, мы опускаемся все ниже и ниже по дереву, пока не найдем то число, которое нас интересует.

Так же устроен поиск числа, которого в бинарном дереве на самом деле нет. Мы спускаемся все ниже и ниже, и когда доходим до крайних листьев и дальше идти нам некуда, а число мы все еще не нашли, ну значит этого значения в дереве нет, поиск завершился неудачей.

Похожим образом работает вставка. Мы спускаемся ниже и ниже, пока не находим место, где число могло бы быть. Оно могло бы там быть, но его пока нет, поэтому мы со спокойной совестью его туда вставим. Все довольны, у нас получается очень красивое дерево.

Но вы можете задуматься, а что будет, если при серии вставок дерево перекосит, и оно перестанет быть сбалансированным. К счастью на практике такие деревья имеют механизмы балансировки. Суть его в том, что имеющиеся узлы переупорядочиваются таким образом, чтобы заполнить поддеревья равномерно.

Поговорим о **сложности операции в двоичных деревьях поиска**, таких как вставка, поиск и удаление. Во всех этих операциях в худшем случае мы должны пройти путь от корня дерева до какого-то его листа. Вспомнив о том, что деревья сбалансированы, мы можем оценить высоту дерева, как логарифм от количества элементов в нем. Получается, что путь от корня до листа длиной в логарифм.

- Все операции работают за $O(h)$, где h — это высота дерева
- Если дерево сбалансированное, то $h \sim \log_2 N$
- find $\sim \log_2 N$; insert $\sim \log_2 N$; delete $\sim \log_2 N$;

Итак, мы узнали, что в основе контейнеров `map` и `set`, лежат сбалансированные двоичные деревья поиска и рассмотрели как они работают. Собственно, `set` хранит в таком дереве ключи, а `map` хранит там пару: ключ-значение.

Вспомнив о том, что в `unordered`-контейнерах сложности этих же операций в среднем константные, мы понимаем, почему они быстрее.

1.6 Итераторы в map. Почему лучше использовать собственные методы для поиска

Рассмотрим, как работают итераторы в `map`. Для начала установим итератор на `begin`.

```
auto it = m.begin();
```

Мы знаем, что когда мы итерируемся по целому контейнеру `map`, мы получаем отсортированные по возрастанию элементы. Поэтому логично предположить, что `begin` указывает на минимальный элемент, то есть на самый левый узел.

При **инкременте итератора**

```
++it;
```

происходит понятная вещь: мы переходим на следующий по возрастанию элемент, производя обход дерева, и так далее. Каждый вызов оператора инкремента будет нас продвигать все к большим и большим значениям. И мы будем постепенно обходить дерево дальше и дальше.

Аналогично происходит и **декремент итератора**, только в обратном порядке, то есть мы будем двигаться в сторону уменьшения элементов.

Вы можете задаться вопросом: а не являются ли тогда вызовы операторов инкремента и декремента тяжелыми? Ведь некоторые очередные сдвиги итераторов выглядят очень сложными, потому что требуется перескочить очень много уровней.

Давайте рассмотрим этот момент подробнее. Представим, что нам надо обойти все дерево, то есть проитерироваться от `begin` до `end`. Всего в дереве у нас N элементов, поэтому будет $(N-1)$ ребро, так как каждый элемент кроме корня имеет ровно одно входящее в него ребро. И каждое ребро мы проходим дважды по направлению туда и обратно. То есть всего для обхода всех N элементов мы сделаем порядка N проходов ребер. Получается, что в среднем на переход от элемента к элементу мы совершаем какую-то константную работу. Поэтому в среднем вызовы операторов инкремента и декремента стоят недорого, несмотря на то, что некоторые из них тяжелые.

Вы можете спросить, а зачем же мы все это разбирали? Неужели только затем, чтобы сказать, что инкремент работает за константное время в среднем? Но не только. Оказывается, что это знание может пригодиться в весьма неожиданном месте, а именно, при выборе одного из двух похожих вызовов поиска: через глобальный алгоритм

```
lower_bound(begin(m), end(m), ...);
```

либо через встроенный метод

```
m.lower_bound(...);
```

Давайте напишем код и посмотрим, к какой разнице это приводит. У нас есть код, вы его можете помнить по задаче о Шерлоке Холмсе. У нас есть какое-то произведение о нем, оно лежит в файле, мы считываем из него все строки и складываем в `set`.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <unordered_map>
#include "head.h"
#include "profile.h"

using namespace std;

int main() {
    set<string> s_words;

    ifstream fs("input.txt");
```

```
    string text;
    while (fs >> text) {
        s_words.insert(text);
    }
    for (const auto& s : Head(s_words, 5)) {
        cout << s << << endl;
    }
    cout << endl;
}
```

Что мы хотим сделать? Мы хотим вызвать методы поиска в `lower_bound` двумя способами. Сначала вызвать метод `lower_bound` у `set`, а затем глобальный алгоритм `lower_bound`.

Вызываем их с одними и теми же входными данными, то есть пробегаемся по всем буквам от A до Z и ищем слова на эти буквы. Тесты одинаковые, но посмотрим, с какой скоростью они работают.

```
    {
        LOG_DURATION("set::lower_bound method");
        for (char c = 'a'; c < 'z'; ++c) {
            string s(1,c);
            s_words.lower_bound(s);
        }
    }

    {
        LOG_DURATION("global lower_bound in set");
        for (char c = 'a'; c < 'z'; ++c) {
            string s(1,c);
            lower_bound(begin(s_words), end(s_words), s);
        }
    }
```

Что мы видим? Что хоть они и работают одинаково, но время требуют очень разное. **Метод** `lower_bound` выполняется почти мгновенно, а **глобальный** алгоритм `lower_bound` работает сильно дольше, вполне ощутимое время. Мы можем сделать вывод: а может, это у нас глобальный `lower_bound` такой плохой, может быть, он всегда так работал? Но нет, вы же помните из предыдущих курсов, что на отсортированном векторе он работает очень даже хорошо. И давайте это проверим сейчас. Возьмем вектор строк, скопируем в него строки из нашего set, он у нас уже сразу будет отсортированный, потому что в set слова отсортированы. Вызовем на этом векторе точно такой же тест и посмотрим, за какое время выполнится он.

```
    vector<string> v_words(begin(s_words), end(s_words));

    for (const auto& s : Head(v_words, 5)) {
        cout << s << << endl;
    }
    cout << endl;
```

```
{
    LOG_DURATION("global lower_bound in vector");
    for (char c = 'a'; c < 'z'; ++c) {
        string s(1,c);
        lower_bound(begin(v_words), end(v_words), s);
    }
}
```

Запускаем. Что мы видим? Слова вектора лежат те же самые, и работает он очень быстро. Он работает так же быстро, как вызов метода set. То есть намного быстрее, чем этот же алгоритм на контейнере set. Давайте разберемся, почему у нас такая большая разница во времени.

`lower_bound` **в отсортированном векторе использует двоичный поиск**, производя арифметические операции над итераторами. Это возможно, потому что итераторы в векторах являются итераторами произвольного доступа (двоичный поиск и итераторы рассматривались в «Желтом поясе»). Заметим, что **глобальный алгоритм** `lower_bound` **принимает на вход только пару итераторов и ничего не знает об устройстве контейнера**. Поэтому становится важно, насколько мощны эти итераторы, могут ли они позволить, например, обращаться к произвольному элементу между ними. В случае map вряд ли, это видно из структуры дерева. Представьте, что у нас есть два итератора — на `begin` и на `end`, и мы хотим найти элемент ровно между ними посередине. Это довольно трудно сделать, потому что непонятно, где он находится. Посередине как бы находится корень, но это совсем не та середина, это не значит, что элемент строго между `begin` и `end`.

Давайте посмотрим, что у нас итераторы в `set` действительно не такие крутые. Заведем итератор в нашем контейнере, возьмем, скажем, `begin`, пока у нас все хорошо. Но если мы прибавим к нему любое произвольное число, мы просто увидим, что **этот код не компилируется**.

```
auto it = s_words.begin() + 5;
```

Нам компилятор подсказывает, что оператор + не может быть применен к такому итератору.

Мы поняли, что итераторы в `map` не позволяют производить над собой арифметические операции. Они умеют делать только инкремент и декремент. Такие итераторы называются **двунаправленными**. Поэтому для map **глобальный алгоритм** `lower_bound` применяет линейный поиск по порядку. В подтверждение наших слов мы можем открыть документацию для глобального алгоритма `lower_bound` и убедиться, что **он гарантирует логарифмическую сложность только для итераторов произвольного доступа. В противном случае сложность будет линейной**. Заметим, что **встроенный метод** `lower_bound` в отличие от глобального алгоритма знает о внутреннем устройстве контейнера `map`, и это позволяет ему работать за логарифмическое время.

1.7 Итераторы в unordered_map. Инвалидация итераторов в ассоциативных контейнерах

Сейчас мы поговорим о том, как работают итераторы в `unordered_map`, то есть в хеш-таблицах. Посмотрим, например, таблицы, и для начала установим итератор на `begin`. Он указывает на первый элемент списка в первой непустой корзине.

```
auto it = h.begin();
```

Инкремент итератора происходит очень просто, это просто итерирование по односвязному списку.

```
++it;
```

Если при очередном инкременте мы дошли до конца цепочки, то потом мы переходим далее, в следующую непустую корзину, и в том и в другом случае инкремент будет стоить очень дешево. Нам надо лишь перейти по показателям на следующий элемент либо перейти в следующую корзину. Согласитесь, это намного проще, чем вычисления на деревьях.

И казалось бы с декрементом все должно быть аналогично, но нет, оказывается мы вообще **не можем вызывать оператор декремента для таких итераторов**.

То есть итераторы в хеш-таблицах еще менее мощны чем в двоичных деревьях поиска. Для них есть специальное название, **последовательные**, либо, по-простому — **forward-итератор**.

Кажется что мы уже закончили разбираться с итераторами в хеш-таблицах, они же заметно проще, чем итераторы в деревьях, но не совсем, есть важный нюанс: помните, мы затрагивали важное свойство хеш-таблиц, что их цепочки должны быть короткими. Для поддержания этого свойства иногда может случаться **рехеширование**, перекладывание существующих элементов по другим корзинкам, чтобы не держать их переполненными.

Давайте задумаемся, что же будет с порядком элементов хеш-таблицы после рехеширования. Вообще, судя по названию `unordered_map`, элементы и так не были упорядоченны, а мы еще и разложим их по корзинам по-новому.

А что же будет с итераторами после рехеширования? Можно ли ими продолжать пользоваться. И для ответа на этот вопрос обратимся к документации. Откроем cppreference и посмотрим, что нам говорит стандарт. Действительно, при вставке элементов в хеш-таблицу может случиться рехеширование, и в этом случае все итераторы **инвалидируются**, то есть перестают указывать туда, куда должны, и их использование может привести либо к падению программы, либо к неверному результату, либо к неопределенному поведению.

Заметим, что итераторы в контейнере `map`, то есть в двоичном дереве поиска, более устойчивы и живучи в этом смысле. Даже когда при вставке нового элемента в дерево происходит перебалансировка и какой-то элемент, на который указывал итератор, смещается — проблемы нет, он просто меняет связи с соседями, но с ним можно продолжать работать далее.

Давайте еще раз опишем инвалидацию итераторов в `unordered`-контейнерах:

- использовать имеющиеся итераторы после вставки в хеш-таблицу — плохая идея, потому что они могут инвалидироваться;
- удалять можно не опасаясь за итераторы других элементов (при этом, конечно, сам удаленный итератор инвалидируется).

1.8 Использование пользовательских типов в ассоциативных контейнерах

Давайте разберем, как помещать свои собственные типы данных в ассоциативные контейнеры. Для этого напишем какой-нибудь свой собственный тип данных, который будет описывать автомобильный номер:

```
#include <iomanip>
#include <iostream>
#include <tuple>

using namespace std;

struct Plate {
    char C1;
    int Number;
    char C2;
    char C3;
    int Region;
};
```

Напишем для этой структуры оператор помещения в поток, для того, чтобы мы могли распечатывать автомобильные номера

```
ostream& operator << (ostream& out, const Plate& p) {
    out << p.C1;
    out << setw(3) << setfill('0') << p.Number;
    out << p.C2;
    out << p.C3;
    out << setw(2) << setfill('0') << p.Region;
    return out;
}
```

Что мы хотим сделать дальше? Мы хотим завести ассоциативный контейнер `set` и складывать в него эти автомобильные номера

```
#include "generator.h"
#include "profile.h"
#include <set>
#include <unordered_set>

using namespace std;

int main() {
    PlateGenerator pg;
    set<Plate> s_plates;
    const int N = 10;
```

```
    return 0;
}
```

Для того, чтобы генерировать случайные номера, мы написали генератор, в котором есть функция `GetRandomPlate`, которая возвращает случайный автомобильный номер.

```
#include "plates.h"
#include <array>
#include <random>

class PlateGenerator {
public:
    Plate GetRandomPlate() {
        Plate p;
        p.C1 = Letters[LetterDist(RandEng)];
        p.Number = NumberDist(RandEng);
        p.C2 = Letters[LetterDist(RandEng)];
        p.C3 = Letters[LetterDist(RandEng)];
        p.Region = RegionDist(RandEng);
        return p;
    }

private:
    const static int N = 12;
    const array<char, N> Letters = {
        'A', 'B', 'C', 'E', 'H', 'K', 'M', 'O', 'P', 'T', 'X', 'Y'
    };

    default_random_engine RandEng;
    uniform_int_distribution<int> LetterDist{0, N - 1};
    uniform_int_distribution<int> NumberDist{1, 999};
    uniform_int_distribution<int> RegionDist{1, 99};
};
```

Давайте проверим, что это работает

```
    cout << pg.GetRandomPlate();
```

Собираем программу, запускаем — успешно.

Теперь возьмем 10 случайных автомобильных номеров и поместим их в контейнер `set`. Но **код не компилируется**!

```
    for (int i = 0; i < N; ++i) {
        s_plates.insert(pg.GetRandomPlate());
    }
```

Компилятор говорит, что не хватает оператора сравнения $<$ (меньше). Зачем же нам нужен этот оператор, если мы просто хотим поместить номера в контейнер? Для ответа на этот вопрос вспомним, что `set` устроен как красно-черное дерево, а значит элементы в нем упорядочены,

поэтому нужно уметь сравнивать элементы друг с другом.

Давайте напишем оператор $<$ (меньше) для расположения элементов в лексикографическом порядке

```
bool operator < (const Plate& l, const Plate& r) {
    return tie(l.C1, l.Number, l.C2, l.C3, l.Region) < tie(r.C1, r.Number, r.C2, r.C3,
        r.Region);
}
```

Теперь программа собралась успешно. Давайте распечатаем полученные 10 автомобильных номеров

```
    for (const auto& p : s_plates) {
        cout << p << endl;
    }
    cout << endl;
```

Теперь мы видим 10 номеров, отсортированных в лексикографическом порядке.

Решим эту же задачу с `unordered_set`:

```
#include "generator.h"
#include "profile.h"
#include <set>
#include <unordered_set>

using namespace std;

int main() {
    PlateGenerator pg;
    set<Plate> s_plates;
    unordered_set<Plate> h_plates;
    const int N = 10;

    for (int i = 0; i < N; ++i) {
        s_plates.insert(pg.GetRandomPlate());
    }
    for (const auto& p : s_plates) {
        cout << p << endl;
    }
    cout << endl;

    for (int i = 0; i < N; ++i) {
        h_plates.insert(pg.GetRandomPlate());
    }
    for (const auto& p : h_plates) {
        cout << p << endl;
    }
    cout << endl;
```

```
    return 0;
}
```

У нас **снова сообщение об ошибке**! Ошибка связана с тем, что нам не хватает чего-то связанного с хешированием. Давайте напишем хеш-функцию для нашего `unordered_set`, которая будет возвращать целое число для автомобильного номера.

```
struct PlateHasher {
    size_t operator() (const Plate& p) const {
        return p.Number;
     }
};

int main() {
    ...
    unordered_set<Plate, PlateHasher> h_plates;
    ...
```

У нас **снова сообщение об ошибке**! Компилятор говорит, что нам не хватает оператора сравнения. А зачем нам нужен оператор сравниения, если мы просто хотим поместить элементы в хеш-таблицу? Конечно же он нужен! При поиске элементов в хеш-таблице каждый элемент сравнивается с теми, которые уже находятся в хеш-таблице. Давайте напишем оператор сравнения:

```
bool operator == (const Plate& l, const Plate& r) {
    return (l.C1 == r.C1) && (l.Number == r.Number) && (l.C2 == r.C2) && (l.C3 == r.C3) &&
        (l.Region == r.Region);
}
```

Теперь программа успешно компилируется. В итоге мы видим, что нам вывелись 10 номеров из двух контейнеров.

Таким образом,

- если вы хотите поместить произвольный тип в `map` или `set`, то необходимо написать для него оператор сравнения `<` (меньше);
- если вы хотите поместить произвольный тип в `unordered`-контейнер, то необходимо для него определить хеш-функцию и оператор сравнения `==`.

1.9 Зависимость производительности от хеш-функции

Давайте узнаем, как правильно следует писать свою собственную хеш-функцию для хеш-таблицы. Замерим производительность: насколько эффективно наши конструкции работают.

```
...
int main() {
    PlateGenerator pg;
```

```
    set<Plate> s_plates;
    unordered_set<Plate> h_plates;
    const int N = 50000;

    {
        LOG_DURATION("set");
        for (int i = 0; i < N; ++i) {
            s_plates.insert(pg.GetRandomPlate());
        }
        for (int i = 0; i < N; ++i) {
            s_plates.find(pg.GetRandomPlate());
        }
    }
    {
        LOG_DURATION("unordered_set");
        for (int i = 0; i < N; ++i) {
            h_plates.insert(pg.GetRandomPlate());
        }
        for (int i = 0; i < N; ++i) {
            h_plates.find(pg.GetRandomPlate());
        }
    }

    return 0;
}
```

Видим, что у нас тест продолжает работать, но мы наблюдаем какой-то странный эффект. Мы видим, что наш `unordered_set` работает медленнее почти в два раза, чем обычный `set`. А это странно, мы ведь с вами знаем из предыдущих занятий, что хеш-таблица работает очень быстро. Значит, у нас что-то пошло не так. Давайте попытаемся это объяснить.

Как мы знаем, у нас для разрешения коллизий в хеш-таблицах используется метод цепочек. И мы с вами говорили, что важно, чтобы такие цепочки были короткими. Давайте посмотрим, есть ли надежда в нашем коде, что цепочки будут короткими. Такой надежды, получается, что у нас и нет. Потому что у нас эксперимент состоит из 50000 различных автомобильных номеров, а в качестве номеров корзин, то есть в качестве значений хеш-функции, мы используем только числа от 1 до 1000. То есть у нас есть всего лишь 1000 различных корзин для 50000 автомобильных номеров. Этого явно недостаточно. И у нас будет очень много коллизий. Поэтому производительность `unordered_set` и снижается так сильно. Надо что-то предпринять. Каким образом?

Нам нужно использовать более, чем эти 1000 корзин. Мы ведь можем написать более хитрую хеш-функцию. И мы можем написать ее следующим образом. Можем использовать не только центральное число из автомобильного номера, но еще и код региона. С помощью нехитрой арифметической магии мы можем взять номер, умножить его на 100 и прибавить регион. Таким образом, у нас получится пятизначное число, и мы это пятизначное число уже сможем использовать в качестве значения хеш-функции. Давайте это и сделаем. Перепишем наш хеш и посмотрим, изменится ли от этого производительность

```
struct PlateHasher {
```

```
    size_t operator() (const Plate& p) const{
        size_t result = p.Number;
        result *= 100;
        result += p.Region;
        return result;
    }
};
```

Компилируем. Запускаем. Получилось неплохо, у нас `unordered_set` начал выигрывать у `set`. И значит, мы на верном пути. Мы делаем все правильно и правильно понимаем, как работает хеш-таблица.

Но давайте усугубим наш эксперимент. Возьмем побольше, не 50 000 тестовых номеров, а миллион. Что-нибудь изменится или нет? Собираем. Запускаем. `set` у нас отработал за две секунды, `unordered_set` — больше, чем за две секунды. То есть у нас, когда тест становится серьезнее, `unordered_set` все-таки проигрывает по производительности. А мы знаем из теории, что не должен. И опять-таки понятно почему.

Потому что у нас различных корзин 100000 получается (у нас пятизначные числа используются), а тест состоит из миллиона номеров! И у нас опять много коллизий. Окончательно решить эту проблему можно, только использовав всю информацию, которая имеется на госномере. То есть кроме чисел с госномера использовать еще и буквы. Вы спросите: «А как же нам буквы превратить в значение хеш-функции? Ведь буквы — это буквы». Ничего страшного. Мы можем буквы конвертировать в их порядковый номер в алфавите: отняв букву A, например, можно получить порядковый номер. И затем эти порядковые номера использовать в такой арифметической магии. И таким образом получить для этого автомобильного номера большое число. Ну что ж, давайте сделаем это.

```
struct PlateHasher {
    size_t operator() (const Plate& p) const{
        size_t result = p.Number;
        result *= 100;
        result += p.Region;

        int s1 = p.C1 - 'A';
        int s2 = p.C2 - 'A';
        int s3 = p.C3 - 'A';
        int s = (s1*100 + s2)*100 + s3;

        result *= 1000000;
        result += s;

        return result;
    }
};
```

Компилируем и запускаем. Супер. Мы видим, что наш `unordered_set` стал почти в два раза быстрее. Мы сделали все, что от нас требовалось. Мы использовали всю информацию с госномера,

и поэтому у нас получилась отличная хеш-функция.

Таким образом, качество хеш-функции очень сильно влияет на производительность хеш-таблицы, то есть на производительность контейнера `unordered_set`. И хорошая хеш-функция, которую вы пишете для достижения хорошей производительности, должна охватывать широкий диапазон корзин, а также по возможности равномерно распределять по ним объекты.

1.10 Рекомендации по выбору хеш-функции

Давайте напишем еще какую-нибудь собственную структуру. Поместим в нее поля и попробуем написать хешер для нее.

Что там будет? Ну давайте для начала поместим туда число типа `double`, и у нас сразу встает вопрос: как же мы будем писать для хеш-функцию для `double`? Ведь **значения хеш-функции — это целые числа**, а `double` — это дробное число и не очень понятно, как его однозначно перевести. Но еще больше было бы непонятно, если бы у нас было вообще не число, а какая-нибудь строка. Как нам переводить строку в хеш-функцию в число? Неужели снова писать такую же сложную хеш-функцию, как для автомобильных номеров? И это было бы удивительно, потому что тогда мы бы не смогли положить в `unordered_set` по умолчанию ни double, ни string. Но давайте убедимся, что мы на самом деле можем это делать спокойно, без написания собственных хеш-функций. Вот мы создаем `unordered_set` из переменных типа `double` и код компилируется прекрасно, и если мы заменим на `string`, он тоже прекрасно компилируется.

```
...
struct MyType {
    double d;
    string str;
};

int main() {
    unordered_set<double> ht1;
    unordered_set<string> ht2;
    return 0;
}
```

От нас не требуют написать свою собственную хеш-функцию для этих типов. Это значит, что для этих типов стандартные хеш-функции уже написаны, и мы можем их использовать из стандартной библиотеки. Они реализованы в шаблонных структурах `hash`, параметризованных каким-нибудь типом.

Давайте напишем хешер для нашей структуры, в которую поместим в поле тот самый автомобильный номер:

```
struct MyType {
    double d;
    string str;
    Plate plate;
};
```

```
struct MyHasher {
    size_t operator() (const MyType& p) const{
        size_t r1 = dhash(p.d);
        size_t r2 = shash(p.str);
        size_t r3 = phash(p.plate)
        // ax^2 + bx + c
        size_t x = 37;
        return (r1*x*x + r2*x + r3);
    }

    hash<double> dhash;
    hash<double> shash;
    PlateHasher phash;
};

int main() {
    unordered_set<MyType, MyHasher> ht2;
    return 0;
}
```

Все скомпилировалось, мы смогли все правильно сделать.

Таким образом, в качестве распространенных подходов написания хеш-функций для собственных типов можно отметить следующие:

- если ваш класс содержит поля стандартных типов, то для хеширования этих полей можно использовать стандартные хешеры из библиотеки;
- для некоторых типов могут быть уже написаны хешеры, например, вами; для комбинации нескольких хешей, часто используют их как коэффициенты при вычислении некоторого многочлена;
- после применения всех рассмотренных рекомендаций ваша хеш-функция должна давать равномерное распределение по корзинам.

1.11 map::extract и map::merge

Заведем `set` строк и положим в него для начала три каких-нибудь строчки. Напечатаем то, что в них находится, чтобы убедиться в том, что мы сделали все правильно.

```
...
int main() {
    set<string> ss;
    ss.insert("Aaa");
    ss.insert("Bbb");
    ss.insert("Ccc");
```

```
    for (const auto& el : ss) {
        cout << el << ' ';
    }
    cout << endl;
}
```

Что мы хотим сделать дальше? Мы хотим модифицировать дерево, которое хранит эти три строчки. Мы хотим, чтобы все строчки, которые хранятся в нем, начинались с маленькой буквы, а не с большой.

Давайте начнем, скажем, с первого элемента. Установим итератор на `begin`, он указывает на какую-то строчку, и попробуем эту строчку изменить.

```
    auto it = ss.begin();
    string& temp = *it;
    temp[0] = tolower(temp[0]);
```

Но компилятор сообщает нам об ошибке. Он говорит о том, что мы потеряли где-то константность. Что бы это могло значить? По-простому это говорит о том, что мы не можем изменить ключ объекта, который находится в дереве. Вот у нас итератор указывает на `begin`, и мы хотим изменить это значение, то есть мы хотим изменить значение объекта, который лежит в дереве. Но мы знаем, что мы не можем это просто взять и сделать, потому что **элементы у нас в дереве упорядочены**. И если мы захотим поменять ключ, то нам придется поместить этот объект в другое место в дереве. А работая с итератором, мы не можем этого сделать, потому что **итератор указывает на один элемент и про структуру всего дерева ничего не знает**.

Какие есть способы такую задачу решить? Мы можем взять эту строчку, скопировать из дерева, то есть взять копию, а не ссылку, и изменить. Дальше из дерева старое значение удалить и новое измененное значение вставить.

```
    auto it = ss.begin();
    string temp = *it;
    temp[0] = tolower(temp[0]);
    ss.erase(it);
    ss.insert(temp);
```

Так у нас задача решится, решение компилируется. Распечатаем снова дерево, чтобы убедиться, что произошло ожидаемое. Да, мы видим, что у нас строчка, которая раньше начиналась на A, сейчас начинается на a, то есть мы добились, чего хотели, и, казалось бы, задача решена, она очень простая. И в чем же тут подвох?

Давайте разберемся, какие же этапы решения у нас были. У нас сначала было дерево. Потом мы скопировали строку. Потом мы удалили старый объект из дерева, поработали с копией в памяти, скопировали это значение обратно в дерево, и эта копия у нас, получается, была лишней почти все время. Мы зря сделали два копирования: из дерева и в дерево, только для того, чтобы снаружи дерева модифицировать строчку. И, казалось бы, так придется делать всегда, ведь у нас ключи в дереве неизменяемые, это единственный способ. Но нужно понимать, что даже **такой единственный способ вам не подойдет, если у вас объекты очень тяжелые**, вы не хотите тратить на них копирование. **Либо эти объекты вообще нельзя скопировать, потому что они move only**, потому что их можно только перемещать, а копировать нельзя.

И давайте рассмотрим именно такой пример для иллюстрации. Возьмем строчки, которые нельзя скопировать, вы их должны помнить с прошлых задач нашего курса. Здесь у нас имеются конструкторы, унаследованные от строки. Подчеркнуто, что конструктор копирования запрещен, и подчеркнуто, что конструктор перемещения разрешен.

```
struct NCString : public string {
    using string::string;
    NCString(const NCString&) = delete;
    NCString(NCString&&) = default;
};
```

И давайте заведем сейчас дерево из таких некопируемых строк.

```
int main() {
    set<NCString> ss;
    ss.insert("Aaa");
    ss.insert("Bbb");
    ss.insert("Ccc");

    for (const auto& el : ss) {
        cout << el << ' ';
    }
    cout << endl;

    auto it = ss.begin();
    NCString temp = *it;
    temp[0] = tolower(temp[0]);
    ss.erase(it);
    ss.insert(temp);
}
```

Компилятор говорит, что мы не можем вставить в дерево строчку такого типа. Мы ее не можем даже скопировать наружу, потому что она не копируемая, а только перемещаемая. И мы бы хотели ее переместить именно поэтому, но как же перемещать из дерева?

Оказывается, такой способ есть. У дерева есть специальный метод, он называется `extract` и работает так, как нам нужно. Мы можем взять и извлечь из дерева даже не строчку, а весь узел целиком, то есть мы можем оторвать узел от дерева. Для этого воспользуемся словом auto, потому что узел будет иметь специальный тип, отличный от типа строки. У нас есть теперь какой-то node, который мы извлекли из дерева. И этот node внутри себя содержит нужную нам строчку. Мы сейчас можем взять эту строчку и вытащить ее для того, чтобы модифицировать. Заметьте, что мы вытаскиваем ее по ссылке, чтобы не делать лишнее копирование.

```
    auto it = ss.begin();
    auto node = ss.extract(it);
    string& temp = node.value();
```

И мы сейчас вытащили эту строчку по ссылке, произвели ее модификацию. Мы теперь можем это делать, потому что узел у нас находится не в дереве, а отдельно сам по себе. И затем мы хотим

поместить обратно этот узел. Удалять нам уже ничего не нужно, мы же весь узел вытащили уже, а чтобы вставить, нам нужно вставить не строчку, а этот узел целиком.

```
temp[0] = tolower(temp[0]);
ss.insert(move(node));
```

Программа успешно компилируется, она успешно работает. Она работает отлично, она делает то, что и раньше. Она модифицирует значение ключа без избыточного копирования.

В итоге, у нас было дерево из трех узлов, затем мы один узел просто оторвали наружу без копирования — это просто перемещение из дерева. В этом отдельно взятом узле мы изменили значение строки, потом мы вставили этот узел обратно в дерево. Заметьте, что здесь **не было никаких копирований**. Мы просто вытащили узел из дерева и затем вставили его в другое место, чтобы он связался с родителями и сыновьями по-другому.

Давайте рассмотрим еще один интересный пример — как это можно использовать еще удобнее. Допустим, что у нас есть два дерева. Одно дерево строк и другое дерево строк, два сета. Возьмем второй сет, заведем в нем новые объекты, и, допустим, мы хотим слить два этих дерева, взять и переместить все узлы из второго дерева в первое.

```
set<NCString> ss2;
ss2.insert("Xxx");
ss2.insert("Yyy");
ss2.insert("Zzz");
```

И мы можем это сделать! Мы можем вытаскивать каждый узел из дерева 2 и вставлять его в дерево 1, например, в цикле. Либо вместо этого, чтобы это делать не в цикле, использовать функцию `merge` и просто перенести все дерево 2 в дерево 1.

```
ss.merge(ss2);
```

Напечатаем то, что получилось после вызова merge. Шесть элементов. Супер. И, чтобы убедиться, что никаких копирований не было, давайте распечатаем содержимое дерева 2.

```
for (const auto& el : ss) {
    cout << el << ' ';
}
cout << endl;

for (const auto& el : ss2) {
    cout << el << ' ';
}
cout << endl;
```

Конечно, никаких копирований здесь быть не могло, у нас же конструктор копирования запрещен, и мы это видим: у нас распечатана пустая строка там, где мы печатали элементы дерева 2. Когда мы сливали, никаких копирований не произошло, мы просто в дереве связали элементы по-другому, указателями, и сэкономили кучу накладных расходов.

Таким образом, мы убедились, что нельзя напрямую изменять ключ объекта в контейнере `map`. Для того чтобы изменить этот ключ, мы вынуждены скопировать ключ во временный объект,

изменить его там, удалить из старого места, а потом скопировать временную копию обратно. Если мы хотим избежать промежуточных копирований, мы можем использовать функции `extract` и `insert`; а при необходимости перенести все элементы из одного дерева в другое, мы можем использовать метод `merge`.

1.12 Коротко о главном

Пожалуй, самое важное, что следует помнить, это то, что контейнеры `map` и `set` устроены внутри как двоичные деревья поиска, а их `unordered` версии — как хеш-таблицы. Это приводит к тому, что в `map` и `set` элементы хранятся в **отсортированном порядке**, а `unordered` никакой **порядок не гарантируется**.

Когда мы с вами говорили о производительности, то обнаружили, что `unordered`-контейнеры работают заметно быстрее. Операции вставки, поиска и удаления работают в них за **константное в среднем** время, в то время, как в обычных `map` и `set` — за **логарифм**.

Есть еще некоторые нюансы связанные с итераторами: по скорости работы они, можно считать, не отличаются, но в обычных `map` и `set` итераторы переживают операции вставки с последующими перебалансировками деревьев, а в хеш-таблицах может произойти рехеширование и тогда итераторы станут инвалидными. Учитывайте это при решении своих задач.

Основы разработки на C++. Пространства имён и указатель this

Оглавление

Пространства имён

2.1 Знакомство с учебным примером

Поговорим о пространствах имён в языке C++. Начнем с знакомства с учебным примером, на котором мы будем разбирать, что такое пространства имён, и как ими, собственно, пользоваться. Давайте представим, что мы с вами пишем программу управления личными финансами. И мы хотим собирать в одном месте все наши расходы и как-то потом их обрабатывать. И у нас уже есть какой-то код, с которого мы начинаем работу.

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;
```

У нас есть структура `Spending` — собственно, трата, которая состоит из двух полей. Это категория — собственно, что это за трата: продукты, транспорт, одежда или что-то еще, и amount — количество, сколько денег мы на эту категорию потратили.

```
struct Spending {
    string category;
    int amount;
};
```

И вот эти вот самые траты мы как-то в нашей программе умеем обрабатывать. Например, у нас есть функция `CalculateTotalSpendings`, которая по вектору расходов, по вектору трат считает, собственно, сколько всего денег мы потратили. Ну, она делает это предельно просто: она просто проходит по вектору и находит сумму полей `amount`. И возвращает, сколько в сумме денег мы потратили.

```
int CalculateTotalSpendings(const vector<Spending>& spendings) {
    int result = 0;
    for (const Spending& s : spendings) {
        result += s.amount;
    }
    return result;
}
```

Также у нас есть другая функция обработки расходов — это функция `MostExpensiveCategory`. Она находит самую дорогую категорию расходов в наших тратах. Она тоже устроена достаточно просто, она использует стандартный алгоритм `max_element` и проходит по вектору расходов и находит ту категорию, у которой расход, собственно, вот это поле `amount`, максимальный.

```
int MostExpensiveCategory(const vector<Spending>& spendings) {
    auto compare_by_amount = [](const Spending& lhs, const Spending& rhs) {
        return lhs.amount < rhs.amount;
    };
    rerurn max_element(begin(spendings), end(spendings), compare_by_amount)->category;
}
```

И у нас есть функция `main`, в которой представлен пример использования структуры `Spending` и функций. Мы создали какой-то вектор, заполнили его. Вот у нас сказано, что на продукты мы потратили 2500 рублей, на транспорт — 1150, ну и так далее. И мы этот вектор передаем в функцию `CalculateTotalSpendings` и `MostExpensiveCategory`.

```
int main() {
    const vector<Spending> spendings = {
        {"food", 2500},
        {"transport", 1150},
        {"restaurants", 5780},
        {"clothes", 7500},
        {"travel", 23740},
        {"sport", 12000}
    };
    cout << "Total " << CalculateTotalSpendings(spendings) << endl;
    cout << "Most expensive " << MostExpensiveCategory(spendings) << endl;
}
```

Давайте, как обычно, скомпилируем нашу программу, убедимся, что наш стартовый код компилируется, запустим и видим, что суммарно мы потратили 52670 рублей, и самой дорогой категорией расходов оказались путешествия. Ну и если мы посмотрим в наш вектор, то убедимся, что да, наша функция правильно работает, действительно на путешествия мы потратили денег больше всего. Но сейчас мы можем добавлять расходы, только указывая их вот в этом векторе в функции `main` и перекомпилируя программу. Естественно, если мы разрабатываем программу для массового потребителя — это будет неудобно.

И мы решили, что, чтобы наша программа была удобна пользователям, мы хотим добавить в нее возможность загрузки расходов из формата XML и из формата JSON. Естественно, мы не хотим самостоятельно писать свой XML парсер, потому что их уже много написано. Поэтому мы

хотим воспользоваться готовой библиотекой. Ну и то же самое для JSON, мы тоже хотим взять готовый код и просто им воспользоваться. Для этого **в проект нужно добавить соответствующие библиотеки для работы с данными форматами**.

Вот так вот выглядит XML документ, и здесь те же самые расходы, которые у меня в функции `main`, но описанные в формате XML.

```
<july>
    <spend amount = "2500" category = "food"></spend>
    <spend amount = "1150" category = "transport"></spend>
    <spend amount = "5780" category = "restaurants"></spend>
    <spend amount = "7500" category = "clothes"></spend>
    <spend amount = "23740" category = "travel"></spend>
    <spend amount = "12000" category = "sport"></spend>
</july>
```

Собственно, что мы хотим сделать в коде? Мы хотим написать функцию, которая возвращает вектор структур `Spending`, она называется `LoadFromXml`, принимает на вход поток ввода. Ещё одну функцию мы хотим написать для формата JSON с точно таким же интерфейсом. При этом мы хотим воспользоваться нашими готовыми библиотеками, чтобы, собственно, не писать парсеры самим.

```
vector<Spending> LoadFromXml(istream& input) {

}

vector<Spending> LoadFromJson(istream& input) {

}
```

Для того, чтобы вы как следует познакомились с кодом наших готовых библиотек, **написание вот этих функций** — `LoadFromXml` и `LoadFromJson` — **мы вынесли в тренировочные задачи, чтобы вы сами эти функции написали**, сами познакомились с кодом библиотек, потому что так вам будет проще дальше понимать, как мы будем применять и изучать пространства имён.

2.2 Проблема пересечения имён двух разных библиотек

Итак, вы решили две тренировочные задачи, в которых по отдельности добавили в нашу программу управления личными финансами поддержку форматов JSON и XML. Теперь давайте попробуем добавить одновременно поддержку обоих этих форматов в нашу программу. Ведь каким-то пользователям нашего приложения будет удобно загружать свои расходы из XML, а каким-то из JSON. И, конечно, хорошо бы, чтобы наша программа умела одновременно работать с обоими форматами.

Функции `LoadFromXml` и `LoadFromJson` реализованы с помощью библиотек, которые мы вам выдавали.

```
vector<Spending> LoadFromXml(istream& input) {
    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}

vector<Spending> LoadFromJson(istream& input) {
    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().AsArray()) {
        result.push_back(node.AsMap().at("category").AsString(),
            node.node.AsMap().at("amount").AsInt());
    }
    return result;
}
```

Пример использования: создаём поток чтения из файла `spendings.json`, в котором расходы описаны в формате JSON, загружаем оттуда вектор расходов и дальше обрабатываем его с помощью наших функций подсчёта суммарного количества расходов и поиска самого большого расхода.

```
int main() {
    ifstream json_input("spendings.json");
    const auto spendings = LoadFromJson(json_input);

    cout << "Total " << CalculateTotalSpendings(spendings) << endl;
    cout << "Most expensive " << MostExpensiveCategory(spendings) << endl;
}
```

Запускаем компиляцию, и у нас, ожидаемо, ничего не компилируется. Компилятор говорит, что "Document was not declared in this scope". Логично, мы не подключили в главный файл библиотеки для работы с XML и JSON. Поэтому давайте это сделаем. Подключаем `xml.h` и `json.h` и снова запускаем компиляцию программы.

```
#include "xml.h"
#include "json.h"
```

Она снова не компилируется, но ошибка компиляции теперь другая. Давайте посмотрим на неё внимательно. Компилятор пишет: **«redefinition of class Document»**. И давайте мы на неё нажмём. Компилятор ругается, что класс `Document` в файле `json.h` определён заново. Давайте посмотрим, где же находится первое определение.

Мы можем нажать на **«previous definition of class Document»** и ожидаемо попасть в файл `xml.h`, в котором у нас тоже есть класс `Document`.

И, собственно, в чём проблема, почему наша программа не компилируется? Дело в том, что у нас есть два файла, две библиотеки: `xml.h` и `json.h`, и **в обеих этих библиотеках есть**

классы и функции с одинаковыми именами. И там, и там есть классы `Node` и `Document` и есть функция `Load`. И компилятор, собирая наш проект воедино, не может выбрать: **он видит две разные реализации одного и того же имени, и происходит нарушение правила одного определения**. Далее мы посмотрим, как решить эту проблему.

2.3 Знакомство с пространствами имён

Итак, мы столкнулись с проблемой использования двух библиотек, так как в обеих библиотеках есть классы с одинаковыми названиям `Node` и `Document`. И из-за этого, когда мы собирали проект, у нас нарушалось правило одного определения и проект не собирался.

- `xml.h`

```
class Node {...};
class Document {...};
Document Load(istream& ...);
```

- `json.h`

```
class Node {...};
class Document {...};
Document Load(istream& ...);
```

Давайте подумаем, как эту проблему можно решить.

Казалось бы, если имена одинаковые, давайте их сделаем разными и проблема исчезнет. Например, мы могли бы в файле `xml.h` классы `Node`, `Document` и функцию `Load` переименовать, например в `XmlNode`, `XmlDocument` и `LoadXml`. И, в принципе, это нормальное решение, все будет работать, проблема наша исчезнет. И, например, таким образом поступили в библиотеке Qt, которая широко применяется во многих проектах на C++. Но мы с вами не будем делать так, мы пойдем другим путем и воспользуемся сущностью, которая специально существует в C++ для решения вот такой проблемы пересечения имён. Эта сущность — **пространство имён**.

Сделаем следующее: **обернем каждую библиотеку в свое пространство имён**. Начнем с библиотеки `xml`. Вначале перед классом `Node` мы напишем `namespace Xml` и поставим открывающую скобку. Мы только что создали новое пространство имён, которое назвали `Xml`. И с помощью фигурной скобки мы его открыли. Все, что будет внутри этого пространства имён, внутри этой фигурной скобки, будет относиться к пространству имён `Xml`. Поэтому мы переходим к концу нашего заголовочного файла и закрываем фигурную скобку.

```
...
#include <unordered_map>

using namespace std;

namespace Xml {

class Node {...};

class Document {...};

Document Load(istream& input);
...

}
```

Теперь классы `Node`, `Document` и функция `Load` находятся в пространстве имён `Xml`.

Но пока что мы обернули в пространство имён только их объявления. Давайте перейдем в `cpp`-файл и точно так же обернем реализации методов классов и реализацию функции `Load`.

Теперь то же самое давайте сделаем для библиотеки `json`. Точно так же объявим пространство имён, назовем его `Json`, и обернем классы `Node`, `Document` и функцию `Load` в это пространство.

```
...
#include <unordered_map>

using namespace std;

namespace Json {

class Node {...};

class Document {...};

Document Load(istream& input);
...

}
```

Перейдем в `cpp`-файл и обернем в пространство имён все реализации.

Теперь давайте снова соберем наш проект. Запускаем компиляцию. Компиляция завершилась неудачно. Но важно, что ошибка компиляции теперь другая. Теперь компилятор говорит нам: «Document was not declared in this scope». Он говорит это два раза для каждой из функций `LoadFromXml` и `LoadFromJson`.

То есть, если раньше компилятор понимал, что такое `Document`, но видел два его определения, то теперь он не понимает, что это за имя такое `Document`, он его не видит. И чтобы наша программа начала компилироваться, нам нужно указать полное имя класса `Document`. Для этого мы напишем `Xml::Document` и аналогично для всех остальных классов и функций (и также для `Json`).

```
vector<Spending> LoadFromXml(istream& input) {
    Xml::Document doc = Xml::Load(input);
    vector<Spending> result;
    for (const Xml::Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}

vector<Spending> LoadFromJson(istream& input) {
    Json::Document doc = Json::Load(input);
    vector<Spending> result;
    for (const Json::Node& node : doc.GetRoot().AsArray()) {
        result.push_back(node.AsMap().at("category").AsString(),
            node.node.AsMap().at("amount").AsInt());
    }
    return result;
}
```

Запустим компиляцию. Наша программа скомпилировалась и корректно работает.

Таким образом, **мы обернули каждую из библиотек в свое пространство имён**. А в том месте, **где мы эти библиотеки использовали, мы указали полные имена классов и функций**. И теперь у компилятора не возникает неоднозначности. Когда он видит `Json::Document`, он понимает, что это класс `Document` из пространства имён `Json`. И это совсем не тот же самый `Document`, который находится в пространстве имён `Xml`.

2.4 Особенности синтаксиса пространств имён

Давайте подробнее рассмотрим синтаксис, который применяется при работе с пространствами имён. Начнём с простейшего вопроса. Собственно, **как создать свое пространство имён**? Мы это уже сделали в предыдущем примере, и все довольно просто. Мы **пишем ключевое слово `namespace` и за ним указываем имя этого пространства имён. А дальше в фигурных скобках, собственно, в блоке кода, мы перечисляем содержимое этого пространства имён**.

Теперь рассмотрим, как же определять, как же создавать определение, реализацию элементов пространства имён. Здесь есть два варианта синтаксиса.

- пишем имя нашего пространства имён `namespace`, и дальше, в фигурных скобках, реализуем функцию (в данном случае функцию `Load`)

```
namespace Json {
    Document Load(isream& input) {
        ...
    }
}
```

- другой вариант: это не оборачивать реализацию в пространстве имён, а просто указать полные имена используемых объектов.

```
Json::Document Json::Load(isream& input) {
    ...
}
```

Проверим второй способ: возьмем реализацию функции `Load` в пространстве имён `Xml` и вынесем ее в самый конец файла за границы пространства имён `Json`.

```
namespace Xml {
   ...
}

Document Load(isream& input) {
    return Document{LoadNode(input)};
}
```

Запустим компиляцию и увидим, что у нас не компилируется, потому что компилятор не знает, что такое `Document`.

Теперь мы укажем полные имена используемых классов и функций.

```
namespace Xml {
   ...
}

Xml::Document Xml::Load(isream& input) {
    return Xml::Document{Xml::LoadNode(input)};
}
```

Скомпилируем, и у нас компилируется. То есть вот этот второй вариант создания работает.

Но при этом, я думаю, вам очевиден недостаток такого синтаксиса. Смотрите, в этих двух строчках мы написали `Xml::` **четыре раза**. Такой способ часто приводит к загромождению кода.

Теперь давайте отметим важную вещь: **пространства имён расширяемы**, то есть мы можем в разных файлах объявлять одно и то же пространство имён, и все, что мы в этих разных файлах туда поместим, попадет в это пространство имён.

Продемонстрируем это в нашем проекте. Мы создадим новый заголовочный файл. Давайте назовем его `xml_load.h`, а в нем мы подключим файл `xml.h`, объявим пространство имён `Xml` и перенесем функцию `Load` в этот файл. При этом реализацию мы оставим в `xml.ccp`. Только чтобы компилятор знал, какую функцию мы реализуем, мы здесь тоже подключим наш `xml_load.h`.

```
#pragma once;

#include "xml.h"

namespace Xml {
    Document Load(isream& input);
}
```

В главной программе мы тоже подключим `xml_load.h`. Запустим компиляцию. Компиляция прошла успешно.

Теперь давайте поговорим об обращении к элементам пространств имён. Ранее мы показывали, что для того чтобы обратиться к элементу пространства имён какого-то конкретного, **нужно написать имя этого пространства имён, два двоеточия и, собственно, имя класса или функции**. Так вот, это нужно делать, **когда мы обращаемся снаружи**, то есть мы, например, в функции `main` пишем какой-то код и обращаемся к элементам библиотеки `Xml`.

Если же мы обращаемся изнутри пространства имён, как в примере в реализации функции `Load`, — у нас функция `Load` реализована внутри пространства имён `Xml`, поэтому мы можем просто писать `Document`, можем просто вызывать функцию `LoadNode`, которая также находится в этом пространстве имён, — нам **не обязательно** указывать полное имя, потому что **компилятор будет искать это имя**, в данном случае `Document`, **в первую очередь внутри пространства имён**.

Если же мы обращаемся снаружи к каким-то элементам, то нужно указывать полное имя, потому что без него компилятор не будет заглядывать в те пространства имён, которые есть в вашей программе.

2.5 Using-декларация

Давайте рассмотрим юнит-тест `TestDocument` из заготовки решения задачи «Библиотека работы с INI-файлами», которую вы должны были решить ранее. В этом тесте мы объявляем переменную `section`, которая имеет тип указатель на `Ini::Section`.

```
Ini::Section* section = &doc.AddSection("one");
```

При этом ниже, ближе к концу теста, у нас еще объявлены три константы, которые также имеют тип `Ini::Section`, то есть имя `Ini::Section` мы в нашем тесте используем 4 раза. Это юнит-тест на библиотеку работы с INI-файлами, то есть понятно, что он будет обращаться к содержимому пространству имён `Ini`, а не к какому-то другому пространству имён, и поэтому использование полного имени `Section` может быть излишним, оно может затруднять написание кода. Сейчас у нас, конечно, короткое имя у пространства имён, а если оно будет длинное, то код и читать может быть довольно сложно.

В C++ есть возможность **не указывать полное имя объекта из пространства имён** с помощью так называемой **using-декларации**, то есть мы можем написать `using Ini::Section`, и дальше в коде мы можем не использовать префикс `Ini::` при обращении к имени `Section`.

```
using Ini::Section;

Section* section = &doc.AddSection("one");
```

То есть, мы командой `using Ini::Section` сказали компилятору, что когда ты видишь имя `Section`, это значит, что мы имеем в виду вот это полное имя `Ini::Section`. Надо отметить, что **декларация распространяется естественно только на то имя, которое в ней указано**.

Второй важный момент состоит в том, что **using-декларация действует только внутри того блока кода, в котором она находится**, то есть сейчас `using`-декларация находится внутри тела нашей функции, внутри фигурных скобок, которые задают тело функции.

2.6 Директива using namespace

Давайте рассмотрим другой юнит-тест из заготовки решения задачи «Библиотека работы с INI-файлами» — `TestLoadIni`. В нем у нас есть обращение к `Ini::Document`, к `Ini::Load`, и к `Ini::Section`. При этом к `Ini::Section` мы обращаемся даже дважды. На самом деле мы в нашем юнит-тесте обращаемся ко всем именам, которые у нас есть в пространстве имён `Ini`. Поэтому нам хочется писать краткие имена. Мы уже знаем как это сделать. Мы для этого можем воспользоваться `using`-декларацией и написать

```
using Ini::Document;
using Ini::Section;
using Ini::Load;
```

Теперь мы можем спокойно удалить префиксы, запустить компиляцию и увидеть, что всё компилируется. Но на самом деле мы не так много выиграли. Сейчас у нас только три имени, но если бы их было 30, то нам вряд ли было бы удобно писать 30 using-деклараций. Нам нужно какое-то другое средство, которое позволит сказать: «Я хочу использовать все имена из данного пространства имен». И конечно же, в языке C++ такое средство есть, и оно вам хорошо знакомо. Это директива `using namespace`. Мы можем написать `using namespace Ini`, убрать другие `using`-декларации, и наша программа продолжит компилироваться. То есть мы одной этой строчкой сказали компилятору: «Я хочу, чтобы при поиске имён ты бы ещё заглядывал в пространство имён `Ini` и искал там». Как и `using`-декларация, директива `using namespace` действует только в том блоке кода, в котором объявлена.

2.7 Глобальное пространство имён

Если **функция или класс не помещены ни в какое пространство имён**, то говорят, что они находятся в **глобальном пространстве имён**. Мы говорили, что `using`-декларация и директива `using namespace` действуют внутри того блока кода, в котором они объявлены. Однако, их можно использовать и в глобальном пространстве имён, то есть, например, в нашем `cpp`-файле мы

можем написать `using Ini::Document`, и тогда везде в cpp-файле мы можем уже не указывать перед `Document` пространство имён. Или же мы можем прямо здесь написать `using namespace Ini` и вообще не использовать префикс `Ini` в нашем файле. Можно проверить, что при подобном имзенени программа компилируется и всё работает. То есть мы за счет использования `using namespace` в глобальном пространстве имён сократили наш код и избавили себя от необходимости каждый раз указывать префикс.

Но с такими вещами нужно быть осторожными. Пусть, например, есть у вас имя `Document`, и оно видно компилятору. Компилятор понимает, что такое `Document`. **Когда мы помещаем это имя в пространство имён, мы ограничиваем его видимость**, мы говорим, что теперь это имя можно видеть только через специальное окошко. В виде этого окошка выступает префикс `Ini::`. Когда же мы в глобальном пространстве имён вводим `using`-декларация или директиву `using namespace`, мы убираем это окошко и снова имя `Document` становится видно компилятору отовсюду, то есть мы добиваемся эффекта обратного тому, который мы создаем с помощью пространства имён.

А при этом **мы используем пространство имён именно для того, чтобы избежать конфликтов в глобальном пространстве имён**, соответственно, злоупотребляя `using`-декларациями и директивой `using namespace`, мы можем свести на нет все наши усилия от заворачивания классов в функции в пространстве имён. И отсюда следует очень важная, практическая рекомендация: **минимизируйте область действия `using`-деклараций и директивы `using namespace`**, то есть если вы используете их в широкой области видимости, то вы повышаете риск возникновения конфликтов имён. Когда же мы используем `using`-декларацию и директивы `using namespace` только в маленьких блоках кода, в маленьких функциях или даже внутри какого-то блока, который является частью функций, тогда вероятность того, что у нас там возникнут какие-то конфликты имён, очень низкая.

2.8 Using namespace в заголовочных файлах

Использование `using namespace` в заголовочных файлах вызывает проблемы. Давайте вернёмся к примеру, с которого мы начинали, а именно к программе управления личными финансами, которая умеет загружать список расходов из форматов JSON и XML. И давайте для начала мы зайдём в функцию `LoadFromXml` и перепишем её так, как мы научились делать: воспользуемся директивой `using namespace` и не будем писать префикс `Xml::` в этой функции.

```
vector<Spending> LoadFromXml(istream& input) {
    using namespace Xml;

    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}
```

Давайте запустим компиляцию, убедимся что всё хорошо. Здесь мы можем прекрасно использовать `using namespace Xml`, у нас маленькая функция.

Теперь давайте представим, что мы в своей программе решили не только загружать расходы из формата JSON, но и сохранять их в этот формат. Ну, мы развиваем наше приложение, хотим, чтобы у нас было больше пользователей и добавляем новые функции. При этом наша библиотека по работе с форматом JSON умеет только загружать (это не наша библиотека, мы её откуда-то взяли), и она умеет только загружать данные из формата JSON, поэтому мы решили, что сохранение в JSON мы напишем сами. Давайте это сделаем.

Для этого мы добавим в наш проект заголовочный файл, назовём его `json_utils.h`. Затем, мы перенесём в него структуру `Spending`, чтобы нам было удобнее, и добавим следующий набор функций.

```
#pragma once

#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

Json::Node ToJson(const Spending& s);
Json::Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Json::Node& node);
ostream& operator <<(ostream& os, const Json::Document& document);
```

Отлично, теперь давайте мы этот файл подключим в главном файле, в `main.cpp`, и запустим компиляцию. Программа компилируется.

Тут вы могли задуматься, почему мы только объявили эти функции, но не реализовали, и при этом наша программа компилируется? Всё нормально, потому что мы эти функции пока нигде не вызываем, поэтому компилятору достаточно видеть их объявление, он их не вызывает, и определения ему не нужны.

Итак, пока у нас всё хорошо, у нас программа компилируется. Но мы посмотрели вот в этот заголовочный файл и подумали: это же файл про `Json`, поэтому, может быть, нам не стоит много раз использовать префикс `Json::`. Мы же здесь только про `Json` пишем, поэтому давайте воспользуемся директивой `using namespace Json` и не будем писать полные имена содержимого этого пространства имён.

```
#pragma once
```

```
#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

using namespace Json;

Node ToJson(const Spending& s);
Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Node& node);
ostream& operator <<(ostream& os, const Document& document);
```

Вроде бы всё хорошо, но давайте запустим компиляцию. Мы запустили компиляцию и видим, что **программа-то у нас больше не компилируется**, при этом давайте посмотрим на сообщение компилятора, он пишет: «reference to Document is ambiguous». Где это происходит? Это происходит в функции `LoadFromXml`

```
#include "xml.h"
#include "json.h"
#include "json_utils.h"

...

vector<Spending> LoadFromXml(istream& input) {
    using namespace Xml;

    Document doc = Load(input);
    ...
}
```

Теперь наш компилятор не понимает, какому объекту соответствует имя `Document`, потому что у нас в функции используется `using namespace Xml`, а из заголовочного файла `json_utils.h` нам прилетела ещё директива `using namespace Json`. Получается, что вот в этой функции, в точке вызова функции `Load` компилятор видит оба имени `Document`: он видит и `Json::Document`, и `Xml::Document`, и соответственно, у него возникает неоднозначность, и наша программа не компилируется.

Конфликт возник из-за того, что мы воспользовались директивой `using namespace` в заголовочном файле. Чем она плоха? Тем, что мы не знаем, в какие другие файлы будет подключен наш

заголовочный файл, соответственно мы не знаем, какие имена будут видны в тех файлах. А мы берём и в своём заголовочном файле приносим в тот файл все имена из пространства имён, для которого мы воспользовались директивой `using namespace`. Поэтому использование директивы `using namespace` в заголовочных файлах может приводить к неожиданным конфликтам имён, и поэтому **в общем случае нельзя использовать `using namespace` в заголовочных файлах**.

2.9 Пространство имён std

Настало время наконец-то обратить внимание на ту строчку, которую мы писали в наших программах с самого начала — `using namespace std`. **В пространстве имён `std` находится все стандартная библиотека**: все алгоритмы, все контейнеры находятся в этом пространстве имён, то есть `vector`, `deque`, `list`, `map`, алгоритмы, примитивы работы с многопоточностью, `async`, `future`, `mutex` — все они находятся в пространстве имён `std`.

Давайте посмотрим: каково это писать программы без директивы `using namespace std`. Уберем ее. И для примера напишем программу `hello user`, которая спрашивает имя пользователя и здоровается с ним. Она будет выглядеть вот так.

```
#include <iostream>
#include <string>
#include <vector>
#include <map>

int main() {
    std::string name;
    std::cout << "Enter your name" << std::endl;
    std::cin >> name;
    std::cout << "Hello, " << name << std::endl;
    return 0;
}
```

Она у нас компилируется и работает.

Или другой пример: давайте объявим с вами `map`, который отображает строку в вектор строк. Как это будет выглядеть?

```
#include <iostream>
#include <string>
#include <vector>
#include <map>

int main() {
    std::map<std::string, std::vector<std::string>> m;
    return 0;
}
```

Эти два коротких примера демонстрируют нам два важных момента.

- все контейнеры и алгоритмы, которые мы с вами изучали имеют полные имена, которые начинаются с `std::`. И теперь, когда вы будете читать документацию, например на `std::vector`, вы будете понимать, что же это за такое `std::`, потому что в документации обычно для элементов стандартной библиотеки используют их полные имена `std::vector`, `std::string` и так далее.

- использование префикса `std::` в коде, который интенсивно использует стандартную библиотеку, существенно этот код раздувает. Во втором примере мы не просто так объявили этот `map` — видите, в одной строчке в объявлении одной переменной префикс `std::` встречается 4 раза. И, конечно, это может существенно раздувать код, и затруднять его чтение, и, иногда, даже и написание.

Давайте мы с вами сформулируем **рекомендации по использованию пространства имён** `std`. Во-первых, мы сохраняем рекомендацию о том, что **не надо использовать** `using namespace`, **в данном случае** `using namespace std` **в заголовочных файлах**, потому что это может приводить к конфликтам имён. В нашем курсе этих конфликтов не возникает из-за используемого стиля именования функций и классов. Так как в пространстве имён `std` все функции и классы начинаются с маленькой буквы, а мы функции и классы называем с большой буквы. Но в общем случае в других проектах за пределами нашей специализации может использоваться стиль, когда функции и классы именуются с маленькой буквы, и тогда они могут конфликтовать с тем, что есть в пространстве имён `std`.

В `cpp`-файлах ситуация другая. Как мы с вами видели, часто мы в `cpp`-файлах можем интенсивно использовать стандартную библиотеку. И поэтому в них все-таки практичным является использование директивы `using namespace std` даже в глобальном пространстве имён. **В отдельных функциях использование** `using`**-декларации или** `using namespace std` **— в принципе, хорошая практичная рекомендация, но только для** `cpp`**-файлов**.

Давайте перейдем к нашей программе по работе с личными финансами. И давайте перепишем ее в соответствии с рекомендациями, которые мы только что написали. `using namespace Json`, который нельзя использовать, вернем назад, чтобы наша программа компилировалась.

```
#pragma once

#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

Json::Node ToJson(const Spending& s);
```

```
Json::Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Json::Node& node);
ostream& operator <<(ostream& os, const Json::Document& document);
```

Теперь давайте перейдем к `json.h` и поправим его, потому что мы в нем используем `using namespace std`, а мы только что договорились, что не будем использовать эту директиву в заголовочных файлах. И в этом заголовочном файле нам нужно расставить префикс `std` везде, где мы используем элементы стандартной библиотеки.

Конечно, **код** наш именно **в заголовочном файле** от этого немного **раздувается**. Но **зато мы страхуем себя от неприятных проблем**. При этом в `cpp`-файле мы не будем расставлять префикс `std`, а спокойно напишем `using namespace std`, потому что `cpp`-файл никуда не будет включаться и мы можем более безопасно использовать здесь директиву `using namespace`. Запустим компиляцию, и у нас все хорошо. Осталось сделать то же самое для `xml.h`.

Таким образом, директива `using namespace std` существенно сокращает код, который интенсивно используют стандартную библиотеку, но ее нельзя применять в заголовочных файлах и с осторожностью надо применять в `cpp`-файлах.

2.10 Структурирование кода с использованием пространств имён

Все это время мы говорили, что пространства имён в C++ используются только для избежания конфликтов имён, но на самом деле они применяются и для других целей. Посмотрим, как с помощью пространства имён можно улучшить структурирование кода.

Давайте вспомним финальную задачу желтого пояса. В ней вам надо было написать базу данных, которая работала с датами и событиями, при этом вам на вход поступали определенные условия и по этим условиям нужно было отбирать записи в базе данных. На самом деле, если вы не проходили желтый пояс и не решали эту задачу, то ничего страшного, сейчас все станет понятно. Итак, нам на вход поступали условия и мы эти условия разбирали и строили из них абстрактное синтаксическое дерево, которое потом использовали для того, чтобы вычислять условия для каждой записи в базе данных. Абстрактное синтаксическое дерево состояло из некоторых узлов, и каждый узел был экземпляром какого-то класса.

Приведем пример того, в какое дерево выстраивается условие из примера. У нас имеется класс `LogicalOperationNode`, с двумя потомками — `DateComparisonNode` и `EventComparisonNode`, которые, собственно, выполняют вычисление нашего условия.

Давайте посмотрим фрагмент решения финальной задачи желтого пояса, который отвечает как раз за работу с этими условиями. Посмотрим в функцию `main`, она написана специально для нашего примера и работает только с условиями. Она считывает со стандартного ввода строчку условия, разбирает ее с помощью функции `ParseCondition` (она у нас была в финальной задаче желтого пояса) создает абстрактное дерево, которое у нас будет храниться в переменной `condition`, и сейчас, для примера, вычисляет условия для конкретной записи: дата — 31 августа 2018 года и событие — Video.

```
void TestAll();

int main() {
    TestAll();
    string contition_str;
    getline(cin, contition_str);

    istringstream in(contition_str);
    auto condition = ParseCondition(in);

    cout << condition->Evaluate(Date(2018, 8, 31), "Video");
    return 0;
}
```

Давайте мы с вами посмотрим на файл `node.h`. Этот файл содержит классы, которые используются для представления узлов абстрактного синтаксического дерева, при этом этот файл можно открыть в специальном окне, который называется **outline**: в этом окне отображаются все функции, классы и типы, объявленные в файле. И давайте обратим внимание вот на какую особенность: в именах классов в файле `node.h` присутствует слово `Node`, то есть у нас есть базовый класс `Node`, у него есть потомки: `EmptyNode`, `DataComparisonNode`, `EventComparisonNode` и `LogicalOperationNode`. Во всех этих классах используется слово `Node`. Используется и используется — ничего страшного.

Давайте перейдем в `cpp`-файл и посмотрим на него. В нем мы видим что это самое слово `Node`, оно, например, вот в этой строчке используется дважды: для имени класса и еще раз имя класса, потому что это конструктор.

```
...
DateComparisonNode::DateComparisonNode(Comparison comparison, const Date& value) :
    comparison_(comparison), value_(value) {
}
...
```

И вот здесь в конструкторе точно так же.

```
...
EventComparisonNode::EventComparisonNode(Comparison comparison, const string& value) :
    comparison_(comparison), value_(value) {
}
...
```

Давайте вообще ради любопытства посчитаем, сколько раз в этом файла встречается строчка `Node` — во всем файле слово `Node` встречается 12 раз. Ещё давайте заглянем в `node_test`. Этот файл с unit-тестами, который мы запускали в нашей программе, и здесь тоже мы используем эти классы, мы создаем их экземпляры и здесь тоже очень много встречается этот суффикс `Node`.

И понимаете какое дело: возможна ситуация, когда вот это частое использование суффикса `Node` загромождает наш код.

Вы конечно можете сказать, ну подумаешь, что там четыре буквы, но на самом деле этот пример основан на реальной практике, на реальной задаче. Так вот там у классов общий суффикс

имел в длину 14 символов, и часто использование этого суффикса действительно перегружало код и затрудняло его чтение и понимание.

Давайте в нашем учебном примере попробуем избавиться от частого использования общего суффикса в именах классов. У нас есть наши классы в файле `node.h`:

- `Node`;
- `EmptyNode`;
- `DataComparisonNode`;
- `EventComparisonNode`;
- `LogicalOperationNode`;

Мы удалим в их именах общий суффикс `Node`. Дальше мы заведем пространство имён `Nodes` и поместим в него все эти классы. Обратите внимание, что базовый класс мы в пространство имён не помещаем (на самом деле его можно помещать, можно не помещать — это дело вкуса; мы в нашем примере оставим его в глобальном пространстве имён). Давайте осуществим вот это преобразование с нашим проектом.

После того, как мы заменили все имена классов во всех файлах проекта, удалив суффикс `Node` из их имён, давайте сделаем следующий шаг: обернем все классы потомки и их реализации в пространство имён `Nodes`.

Мы выполнили наше преобразование, однако, теперь нам нужно поменять окружающий код, потому что там используются еще имена без учета пространства имён. Мы это сделаем достаточно просто, мы просто запустим компиляцию и будем идти по ошибкам компиляции. Например, первое место, где компилятор не знает, что такое `DateComparison` — это файл `condition_parse.cpp`, в котором реализована это функция `ParseCondition`, разбирающая условия. В этом файле мы укажем полное имя класса, потому что этот файл, так сказать, является клиентом нашего набора узлов, и здесь мы используем их полные имена. Когда мы начинали работу, у нас было `DateComparisonNode`, а теперь будет `Nodes::DateComparison`, так что здесь код изменился совсем чуть-чуть.

И так далее.

В итоге мы просто уменьшаем размеры некоторых файлов в байтах и упрощаем его чтение, потому что нам надо меньше читать дублирующиеся суффиксы. Также, именование классов типа `Nodes::DateComparison` в коде, который их использует, проще понимается, потому что эти «::» подчеркивают структуру.

Мы посмотрели, как с помощью пространства имён можно улучшать структурирование кода. Мы показали, что объединение общих по смыслу классов и функций в пространстве имён подчеркивает логическую структуру кода, уменьшает размер некоторых файлов и может упрощать чтение имён классов, если они длинные.

2.11 Рекомендации по использованию пространств имён

Давайте подведем итоги и сформулируем рекомендации по применению пространства имён в ваших проектах.

- **Пространство имён имеет смысл применять только в больших проектах**, потому что если у вас маленькая программа, которая состоит из одного, двух, максимум трех файлов, то конечно вряд ли у вас возникнет конфликт имён. Когда же у вас большой проект на десятки, сотни, а то и тысячи файлов, то вероятность возникновения конфликтов имён там довольно высока, и поэтому здесь уже возникает смысл использовать пространство имён.

- Пусть вы создали какую-то библиотеку, то есть какой-то обособленный набор функций и классов, который решает не одну какую-то конкретную специфическую задачу, а какой-то набор задач; пусть вы хотите поделиться ею с миром. Тогда **обязательно оберните функции и классы в вашей библиотеке в пространство имён**, для того, чтобы у других пользователей не возникало конфликтов имён, как это было в наших библиотеках по работе с форматами XML и JSON.

- `using`-декларации и директивы `using namespace` позволяют уменьшить объем кода, но при этом повышают вероятность возникновения конфликтов имён, поэтому ими надо пользоваться с осторожностью и стараться минимизировать область их действия (область действия `using`-декларации и директивы `using namespace` — это тот блок кода, в котором они объявлены). **Не надо использовать `using namespace` в заголовочных файлах**. А в `cpp`-файлах, при определенных условиях, это директива отлично работает и упрощает написание кода, поэтому ее **можно использовать в `cpp`-файлах, но с осторожностью, чтобы не возникали неожиданные конфликты имён**.

- `using namespace std` — это специальное пространство имён, в котором находится вся стандартная библиотека; и мы видели, как использование префикса перед каждым членом стандартной библиотеки может раздувать код, поэтому **используйте `using namespace std` в коде, который интенсивно работает со стандартной библиотекой, но если это не заголовочный файл**.

- можно **пробовать объединять общие по смыслу функции и классы в специальное пространство имён**, чтобы подчеркнуть логическую структуру вашей программы, сократить размер отдельных файлов, и в отдельных случаях упростить чтение и понимание вашего кода.

Указатель this

3.1 Присваивание объекта самому себе

Давайте начнем с примера. В «Красном поясе по C++» мы реализовывали шаблон `SimpleVector` — это такая сильно упрощенная реализация стандартного вектора. Мы с вами реализовали набор его конструкторов, деструктор, а также некоторые методы. И сейчас давайте рассмотрим довольно простой, на первый взгляд, способ применения шаблона `SimpleVector`.

```
template <typename T>
ostream& operator << (ostream& os, const SimpleVector<T>& rhs) {
    os << "Size = " << rhs.Size() << " Items:";
    for (const auto& x : rhs) {
        os << ' ' << x;
    }
    return os;
}

int main() {
    SimpleVector<int> source(5);
    for (size_t i = 0; i < source.Size(); ++i) {
        source[i] = i;
    }

    cout << source << endl;
    source = source;
    cout << source << endl;

    return 0;
}
```

Запускаем нашу программу и смотрим, что она вывела. Смотрите, какая интересная вещь: первый вывод ожидаемый, адекватный — размер вектора 5, элементы 0 1 2 3 4. Ровно те элементы, которые мы в него записали в цикле. А вот после присваивания самому себе почему-то наш вектор стал хранить какие-то странные значения. Размер у него не изменился, так и остался 5, а вот элементы почему-то вообще какие-то другие. То есть явно наш оператор копирующего присваивания работает неправильно в том случае, когда мы присваиваем объект самому себе.

Но у вас может возникнуть вопрос: а вообще есть ли смысл в присваивании объекта самому себе? Какая-то странная операция. Но на самом деле такое бывает. Например, у нас есть два

указателя, и оба они указывают на один и тот же объект. И мы эти два указателя разыменовываем и присваиваем один объект другому. Вот в такой ситуации, когда у нас есть только указатели и мы не знаем, на что они на самом деле указывают, может на самом деле произойти присваивание самому себе. Поэтому оно должно работать корректно.

И давайте посмотрим, как сейчас у нас реализован оператор копирующего присваивания в шаблоне `SimpleVector`. Устроен он довольно просто и, вообще говоря, ожидаемо. Мы первым делом освобождаем свои данные, которыми мы владеем, затем выделяем необходимое количество памяти, чтобы сохранить все элементы вектора `other`, копируем его размер, его емкость, и затем, с помощью алгоритма `copy` мы копируем к себе данные другого вектора.

```
template <typename T>
void SimpleVector<T>::operator=(const SimpleVector<T>& other) {
    delete[] data;
    data = new T[other.capacity];
    size = other.size;
    capacity = other.capacity;
    copy(other.begin(), other.end(), begin());
}
```

Вроде бы нормальная реализация, но из нее сразу очевидно, почему у нас некорректно работает присваивание себе. Потому что мы первым же делом удаляем собственные данные. А потом в алгоритме `copy` мы из этой освобожденной памяти читаем. Нам, вообще говоря, везет, что наша программа корректно завершается. Она могла бы падать с ошибкой. Таким образом, нам надо придумать, как поменять реализацию нашего оператора копирующего присваивания, чтобы он корректно обрабатывал присваивание самому себе. И кажется, самый лучший способ это сделать — это проверить, что объект `other`, который нам приходит в качестве параметра, не совпадает с объектом, которому выполняется присваивание. Потому что если он совпадает, то, вообще говоря, делать ничего не надо: мы присваиваем самому себе, свои данные можно не трогать. И у нас возникает вопрос: а как внутри метода класса `SimpleVector`, проверить, что объект `other` — это тот же самый объект, которому выполняется присваивание? В этом нам поможет указатель `this`.

3.2 Знакомство с this

`this` — это специальный указатель, который внутри методов класса указывает на текущий объект этого класса.

Давайте вернёмся к реализации `SimpleVector`, зайдём в его конструктор, который принимает размер, и в нём напишем

```
template <typename T>
SimpleVector<T>::SimpleVector(size_t size) : data(new T[size]), size(size), capacity(size) {
    cout << this << endl;
}
```

Объявим две переменные — `source` и `source2` — обе переменные типа `SimpleVector`. И выведем адреса этих переменных на консоль.

```
int main() {
    SimpleVector<int> source(5);
    SimpleVector<int> source2(5);

    cout << &source << ' ' << &source2 << endl;
}
```

Скомпилируем наш код. И запустим его. Мы видим, что указатель `this` действительно хранит в себе адрес текущего объекта.

Как же теперь нам применить указатель `this`, чтобы решить нашу проблему и ничего не делать, если мы выполняем присваивание самому себе? Очень просто. Пойдём в оператор присваивания и напишем

```
template <typename T>
void SimpleVector<T>::operator=(const SimpleVector<T>& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;
        copy(other.begin(), other.end(), begin());
    }
}
```

Итак, мы видим, что всё теперь работает. Теперь наш код работает правильно, и после присваивания вектора самому себе он не меняется и всё так же выводит те элементы, которые мы в него записали.

И давайте сделаем ещё одну вещь. Мы заглянем в оператор перемещающего присваивания. С ним же, на самом деле, может быть та же самая проблема в строчке

```
    source = move(source);
```

Поэтому давайте зайдём в оператор перемещающего присваивания и сделаем такое же условие

```
template <typename T>
void SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = other.data;
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
}
```

Всё у нас хорошо работает, всё компилируется. Отлично. Таким образом, с помощью указателя `this` мы решили проблему присваивания самому себе в классе `SimpleVector`.

3.3 Ссылка на себя

Давайте рассмотрим такой пример. Допустим, у нас есть много переменных типа `int`. Вот давайте объявим их: переменные `a, b, c, d, e`. И мы хотим вот эти все переменные проинициализировать нулём. Мы это можем сделать так:

```
int main() {
    int a, b, c, d, e;
    a = b = c = d = e = 0;
    return 0;
}
```

Это скомпилируется.

А теперь давайте проверим, можно ли сделать то же самое для нашего шаблона `SimpleVector`. Объявляем эти же переменные, делаем у них тип `SimpleVector` и в присваивании убираем присваивание нуля. И мы поменяли наш код так, что мы переменным `a, b, c, d` присваиваем переменную `e`.

```
int main() {
    SimpleVector<int> a, b, c, d, e;
    a = b = c = d = e;
    return 0;
}
```

Запускаем компиляцию, и у нас не компилируется. При этом смотрим, что нам пишет компилятор. Он говорит, что у него нет оператора присваивания для типов `SimpleVector` от `int` и `void`. Таким образом, у нас не получилось сделать вот такое цепное присваивание.

Однако фундаментальной особенностью языка C++ является возможность создавать пользовательские типы таким образом, чтобы их можно было использовать точно так же, как и встроенные типы. Поэтому должна быть возможность реализовать вот такое вот цепное присваивание для нашего класса `SimpleVector`. И давайте разбираться, как это сделать.

Вернемся к примеру с переменными типа `int`. **Операция присваивания правоассоциативна**. Что это значит? Это значит, что когда у нас выполняется вот такое вот цепное присваивание, то сначала выполняется самое правое присваивание в команде, затем результат этого присваивания присваивается второй справа переменной и так далее.

```
int main() {
    int a, b, c, d, e;
    (a = (b = (c = (d = (e = 0)))));
    return 0;
}
```

Какой здесь самый важный момент? Мы сказали фразу: **результат присваивания**. То есть присваивание должно что-то возвращать. И оно должно возвращать нечто, что мы потом присвоим следующей переменной.

Давайте посмотрим, что возвращает оператор присваивания в нашем классе `SimpleVector`. Он возвращает `void`. Это тот самый `void`, который у нас был в ошибке компиляции в сообщении

компилятора, когда он говорил, что не может для нашего `SimpleVector` сделать цепное присваивание. Значит, нам отсюда, из оператора присваивания, нужно что-то возвращать, чтобы это что-то можно было присвоить следующему объекту. А что мы хотим присвоить? На самом деле себя. То есть тот объект, которому мы выполнили присваивание.

Для начала поменяем тип возвращаемого значения на ссылку на `SimpleVector`. А вернуть ссылку на себя довольно просто: у нас же есть указатель `this`, который указывает на текущий объект. А если у нас есть указатель, то получить ссылку довольно просто (нужно разыменовать этот указатель):

```
template <typename T>
SimpleVector<T>& SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = other.data;
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
    return *this;
}
```

Также нужно не забыть поменять объявление метода (поменяв тип возвращаемого значения). Снова объявляем наши переменные, даём им тип `SimpleVector`.

```
int main() {
    SimpleVector<int> a, b, c, d, e;
    a = b = c = d = e;
    return 0;
}
```

Запускаем компиляцию. Код компилируется. Мы можем даже проверить, что он работает, например, вот таким образом. Мы объявим переменную `e` отдельно и сконструируем ее конструктором, который получает размер. А в конце выведем размер вектора `a`.

```
int main() {
    SimpleVector<int> e(5);
    SimpleVector<int> a, b, c, d;
    a = b = c = d = e;
    cout << a.Size() << endl;
    return 0;
}
```

Скомпилируем, запустим, видим, что вывелась 5.

И у нас в коде осталось еще одно присваивание — перемещающее. И с ним нужно сделать то же самое.

```
template <typename T>
```

```
SimpleVector<T>& SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
    return *this;
}
```

Запускаем компиляцию, у нас всё хорошо.

Вообще говоря, в стандартной библиотеке C++ принято, чтобы операторы присваивания возвращали ссылку на себя. Это делается для того, чтобы работало вот такое цепное присваивание. Мы рекомендуем вам следовать правилам, принятым в стандартной библиотеке, и из операторов присваивания всегда возвращать ссылку на себя.

3.4 this как неявный параметр методов класса

Давайте рассмотрим очень простой код на C++. У нас есть переменная `x`, присваиваем ей значение 3 и выводим. Потом присваиваем значение 8 и снова выводим.

```
int main() {
    int x;

    x = 3;
    cout << x << ' ';
    x = 8;
    cout << x << ' ';
}
```

Давайте запустим программу и увидим что всё работает. Что здесь важно? Что всякий раз, когда мы пишем «`x` равно чему-нибудь», мы записываем это значение в переменную `x`, которая объявлена в начале функции `main`.

Теперь давайте рассмотрим класс `ValueHolder`. В данном случае это структура, у этого класса есть поле `x` и метод `SetValue`, который принимает какое-то значение, записывает в поле `x` и выводит значение этого самого поля.

```
struct ValueHolder {
    int x;

    void SetValue(int value) {
        x = value;
        cout << x << ' ';
    }
};
```

И давайте мы воспользуемся классом `ValueHolder`.

```
int main() {
    ValueHolder a, b;
    a.SetValue(3);
    b.SetValue(5);
}
```

Компилируем, запускаем и в консоль вывелось 3 и 5. Ожидаемо. А как же компилятор догадывается в какую именно переменную, и в какую именно область памяти нужно записать значение при выполнении команды `x = value;`? Ведь в примере с целочисленными переменными все было понятно: есть переменные функция `main`, и мы туда все время записываем. Но здесь у нас есть только одна строчка кода.

Однако, в зависимости от того, для какой переменной `a` или `b` мы вызываем эту команду, значения записываются в разные области памяти. Как же это работает? Как компилятор догадывается, куда писать?

Дело в том, что под капотом указатель `this` является неявным параметром всех методов классов, то есть когда компилятор компилирует ваш код, то он генерирует функцию наподобие следующей.

```
void SetValue(ValueHolder* this_, int Value) {
    this_->x = value;
    cout << this_->x << ' ';
}
```

Таким образом когда вот здесь мы вызываем `SetValue` для `a` и `SetValue` для `b`, в эту функцию передаются разные указатели, и по этим указателей мы обращаемся к разным ячейкам памяти, хранящим поле `x`.

На самом деле, в некоторых языках программирования, например, в Python, даже требуется в методы явно передавать ссылку или указатель на текущий объект класса. Но в C++ этот указатель передается неявно.

Давайте вернемся к традиционному синтаксису для классов. Когда мы обращаемся к полю `x`, мы на самом деле можем написать вот так:

```
struct ValueHolder {
    int x;

    void SetValue(int value) {
```

```
        this->x = value;
        cout << this->x << ' ';
    }
};
```

Таким образом, **указатель `this` является неявным параметром всех методов класса**. И когда внутри метода мы обращаемся к какому-то полю, например, `field`, то на самом деле мы обращаемся к области памяти `this->field`.

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

Оглавление

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»

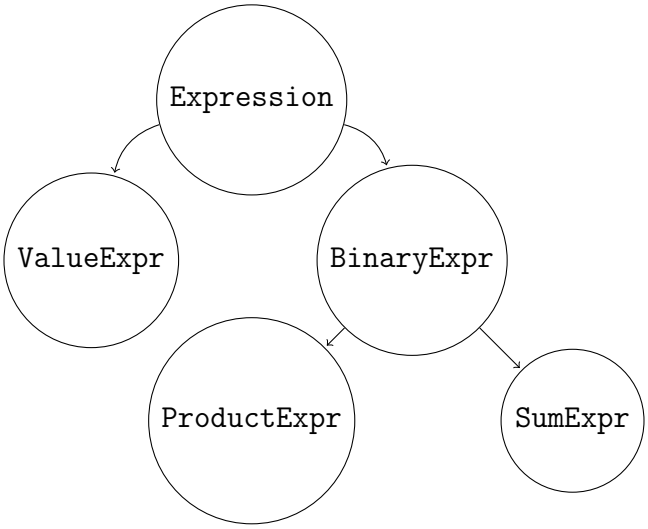
3.1 Разбор задачи «Дерево выражения»

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.

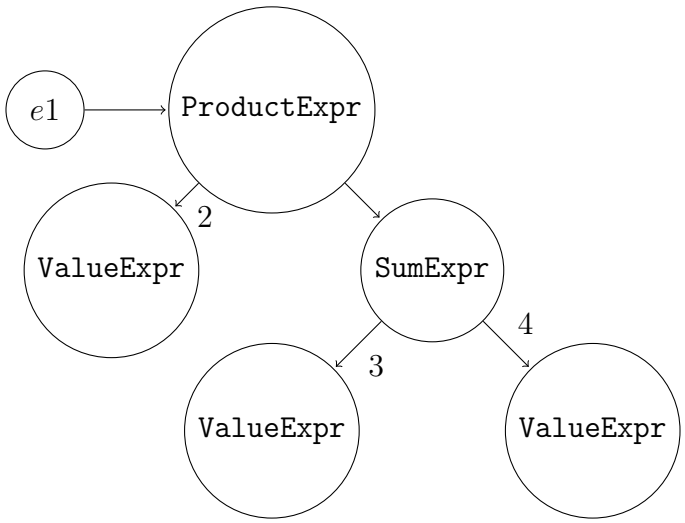
2. Дальше эти временные `unique_ptr` будут переданы в функцию `Sum`, будут перемещены в нее, а та в свою очередь переместит их в новый объект класса `SumExpr`, и они будут сохранены в его полях `left_` и `right_`. Сама же функция `SumExpr` при этом вернет временный `unique_ptr` на вот этот новый созданный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

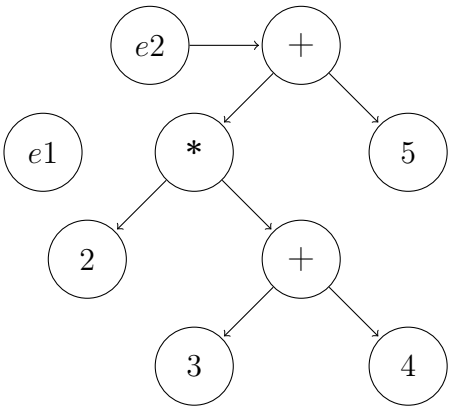
7. И вот только теперь удаляется сам `ProductExpr`.

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.

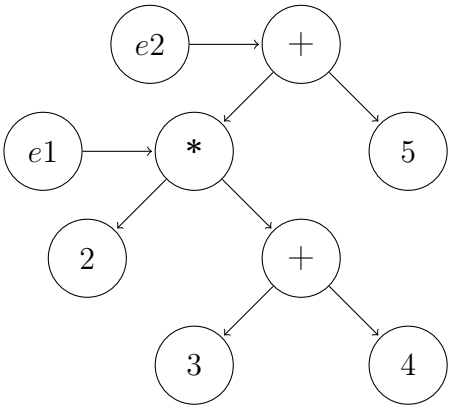
Давайте теперь на сокращенной записи посмотрим, как у нас будет создаваться дерево выражения для e2, которое не просто конструирует выражение с нуля, а берет уже существующее.

```
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.

Основы разработки на C++. shared_ptr и RAII

Оглавление

Умные указатели. Часть 2

1.1 Умный указатель shared_ptr

Давайте, собственно, посмотрим на то, как `shared_ptr` работает. У нас есть некоторый код, который мы использовали для демонстрации возможностей `unique_ptr`. Давайте его запустим.

```
#include <iostream>
#include <memory>

using namespace std;

struct Actor {
    Actor() { cout << "I was born! :)" << endl; }

    ~Actor() { cout << "I died :(" << endl; }

    void DoWork() { cout << "I did some job" << endl; }
};

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

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

Так, мы видим, что он создает `Actor`, дальше его вызывает несколько раз и после этого `Actor` удаляется. Хорошо! В принципе все как и раньше.

Если здесь мы создавали именно `unique_ptr`, то давайте начнем с того, что вместо `unique_ptr` будем создавать `shared_ptr`. И если `unique_ptr` мы создавали с помощью функции `make_unique`, то `shared_ptr`, как вы можете догадаться, мы будем создавать с помощью функции `make_shared`.

```
int main() {
    auto ptr = make_shared<Actor>();
    ...
}
```

Соберем программу таким образом. Она у нас действительно собралась. Запустим, и она отработала точно таким же образом. Что приводит нас к первому важному выводу: `shared_ptr` умеет делать всё то же самое, что и `unique_ptr` и кое-что ещё. Соответственно, давайте посмотрим, что ещё он может делать.

Для начала вспомним немного пример — как то, что у нас выводится соответствует тому, что у нас написано в коде. Значит, сначала у нас создается `Actor`, дальше дважды выполняется работа, а дальше у нас «An actor was expected», то есть передается нулевой `Actor`.

Соответственно `unique_ptr` у нас копировать было нельзя. Главное отличие `shared_ptr` в том, что `shared_ptr` копировать можно, потому что `shared_ptr` разделяемый. Несколько объектов `shared_ptr` могут ссылаться на один и тот же объект, и это совершенно нормально.

Поэтому, чтобы превратить перемещение в копирование мы займемся тем, что удалим функцию `move`. Теперь `ptr2` является копией `ptr`. Давайте соберем такой код.

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

Видим, что он у нас успешно собрался. Запустим, и как мы видим теперь: последний вызов функции `run` у нас точно так же вызывает у `Actor` некоторую работу, то есть он продолжает ссылаться на тот же самый `Actor`. Мы видим, что две функции `run` вызваны для `ptr2` и `ptr` они обе отрабатывают, обе вызывают вывод в консоль, поэтому `ptr` и `ptr2` они оба одновременно указывают на этот объект. Это в общем то то, что нам и нужно было получить. Этих знаний о `shared_ptr` нам с вами на самом деле уже достаточно, для того, чтобы реализовать наши новые требования.

1.2 shared_ptr в дереве выражения

Теперь, когда мы с вами познакомились с возможностями `shared_ptr`, а конкретно с главной его возможностью, что его в отличие от `unique_ptr` можно копировать, мы готовы изменить решение задачи для дерева выражений таким образом, чтобы удовлетворить наше новое требование. В авторском решение с помощью `unique_ptr` все умные указатели переделаем с `unique_ptr` на `shared_ptr`. И дальше у нас есть функции, которые создают новые объекты. Они у нас вызывают `make_unique`, а нам нужно использовать `make_shared`. И как мы помним, `shared_ptr` умеет делать всё то же самое, что и `unique_ptr`. Поэтому такую программу мы уже можем собрать и запустить — она у нас будет выводить все то же самое, но при этом она работает уже на `shared_ptr`. И теперь вопрос: как нужно изменить функцию `main`, чтобы повторное обращение функции `Print` для `e1` у нас выводило не строчку о том, что там нулевой указатель, а чтобы оно точно так же снова выводило содержимое дерева `e1`?

```
int main() {
    ExpressionPtr e1 = Product(Value(2), Sum(Value(3), Value(4)));
    Print(e1.get());

    ExpressionPtr e2 = Sum(move(e1), Value(5));
    Print(e2.get());

    Print(e1.get());
}
```

Почему у нас вообще там выводилось сообщение о том, что у нас нулевое выражение? Потому что мы переместили из указателя `e1`. В случае с `unique_ptr` выхода у нас не было, нам нужно было перемещать. Но у нас же теперь `shared_ptr`, соответственно нам нужно вместо перемещения использовать копирование.

И ответ очень простой: нам нужно удалить функцию `move`. То есть теперь вместо перемещения у нас будет использоваться копирование.

Отлично, давайте посмотрим, как работает такое исправление. Компилируем, запускаем нашу программу и видим, что всё отлично работает. То есть мы снова обратились по указателю `e1` и напечатали исходное дерево. Хорошо.

Давайте теперь проверим, что у нас действительно при удалении указателя `e2` наше поддерево, на которое указывает `e1`, останется без изменений. То есть, что `shared_ptr` в данном случае лучше, чем сырой указатель.

Так. Давайте заключим создание `e2` и его распечатывания в блок, и как мы знаем, у нас `e2` будет уничтожен по выходу из блока. Проверим, что у нас `e1` будет продолжать работать.

```
    ...
    {
        ExpressionPtr e2 = Sum(move(e1), Value(5));
        Print(e2.get());
    }
    ...
```

Соберем такую программу и запустим её. И видим, что у нас вывод корректный, то есть действительно у нас то поддерево, на которое указывает `e1`, не пострадало, несмотря на то, что `e2` удалилось. То есть все работает нормально.

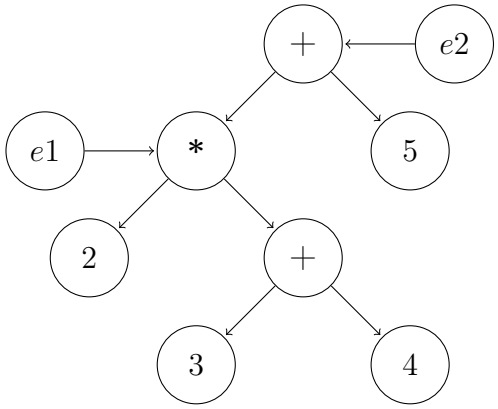
Но вот уберем этот блок. А теперь смотрите, на самом деле мы можем делать даже более интересные вещи. Например, давайте заведем такой `ExpressionPtr e3`, который будет равен сумме `e1` и `e2`, и напечатаем уже вот такой `e3`.

```
    ...
    ExpressionPtr e3 = Sum(e1, e2);
    Print(e3.get());
```

И теперь вопрос к вам: как вы думаете, что напечатает вот эта последняя строчка `Print(e3)`?

Давайте соберем программу и посмотрим, что же она на самом деле напечатает. Так, мы запускаем и видим, что она работает на самом деле абсолютно логично: `e1` мы знаем, `e2` мы тоже знаем. Значит, `e1 + e2` — это просто их сумма. А то, что на самом деле там дерево `e1` входит в

это выражение дважды, это сути дела не меняет, всё и так отлично работает, как раз потому что `shared_ptr` отлично умеет указывать на один и тот же узел из нескольких мест. Хорошо, давайте теперь рассмотрим чуть подробнее, как же у нас эти узлы расположены в памяти и друг на друга ссылаются.



Теперь давайте посмотрим на то, как у нас происходит удаление `e2`.

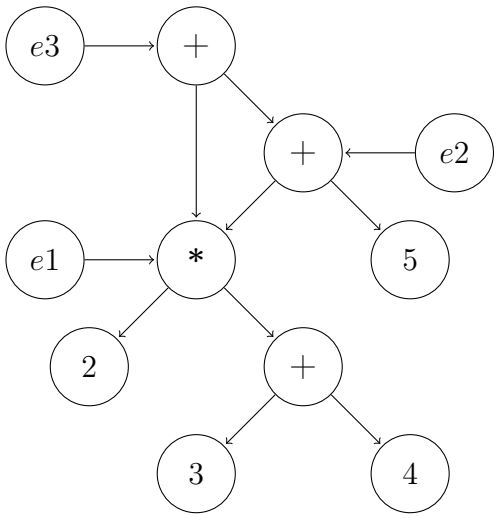
1. Итак, начинается удаление e2. Это `shared_ptr`, он указывает на узел `SumExpr`, начинает удалять `SumExpr`.

2. `SumExpr` при своем удалении начинает удалять свои поля, начинает с поля `right`. Дальше он начинает удалять поле `left`.

3. Поле `left` ссылается на узел умножения. И вот тут самое интересное: `shared_ptr` видит, что на этот узел умножения кроме него ссылается еще другой `shared_ptr` — `e1`, и поэтому он не будет удалять этот узел. Он просто удалится сам, а узел трогать не будет.

На этом удаление `SumExpr` заканчивается, и удаление `e2` тоже заканчивается. А поддерево `e1` у нас остается без изменений.

Теперь давайте посмотрим на вот этот наш пример, когда мы сделали сумму выражений `e1` и `e2`. Если на `unique_ptr` у нас было дерево, то на `shared_ptr` мы смогли сделать направленный ациклический граф. При этом, опять же, если в случае с `unique_ptr` у нас получался рекурсивный обход, абсолютно бесплатный — компилятор сам его делал, здесь точно так же: то, что это граф, и то, что в нем нет циклов, нам гарантирует просто семантика работы с `shared_ptr`. Вот таким, по сути, тривиальным изменением нашей реализации: просто заменив `unique_ptr` на `shared_ptr`, мы смогли реализовать новое требование.



1.3 Внутреннее устройство умных указателей

Теперь давайте посмотрим на то, как умные указатели устроены внутри и как у них получается делать то, что они умеют делать.

Начнем с создания `unique_ptr` на примере вызова функции `make_unique`.

1. Мы вызываем функцию `make_unique`, она создает новый динамический объект T и получает сырой указатель, который на него ссылается.

2. Затем она создает объект собственно `unique_ptr` и помещает в него этот указатель. Сам указатель ей больше не нужен.

3. Этот `unique_ptr` она отдает наружу.

Вот, в общем-то, и все. Так незамысловато это устроено. `unique_ptr` действительно очень простой. Указатель на объект — это единственное, что он хранит внутри себя.

Посмотрим, как происходит перемещение `unique_ptr`.

1. У нас создается новый `unique_ptr`, в который мы перемещаем.

2. Мы берем этот указатель копируем его в новый `unique_ptr` и зануляем его в исходном `unique_ptr`.

Вот так у нас произошло перемещение. Получилось, что в исходном `unique_ptr` у нас ничего не осталось, а новый `unique_ptr` указывает на тот же самый объект.

Хорошо, теперь посмотрим, как происходит разрушение `unique_ptr`.

1. Начнем с `unique_ptr`, который никуда не указывает. Он собственно берет и умирает, потому что у него указатель никуда не указывает, поэтому никаких дополнительных действий не происходит.

2. Теперь, как у нас умирает `unique_ptr`, который куда-то указывает? В своем деструкторе он смотрит: так вот указатель, который у меня лежит, он куда-то указывает? В данном случае да, он указывает на динамический объект. Ну отлично. Значит, он вызывает для него delete и удаляет этот динамический объект, после чего удаляется собственно сам `unique_ptr`.

Вот так, с `unique_ptr` все достаточно просто.

Теперь давайте посмотрим, как у нас устроен `shared_ptr`. Начнем с создания `shared_ptr`, например, при вызове функции `make_shared`.

1. Начало точно такое же: у нас создается динамический объект T, оператор new возвращает нам сырой указатель на этот динамический объект, затем создается объект `shared_ptr`, и этот указатель помещается в `shared_ptr`.

А вот после этого происходит интересное. Смотрите, мы помним, что `shared_ptr` умеет некоторым образом отвечать на вопрос: а существуют ли еще другие `shared_ptr`, которые указывают на тот же самый объект, то есть у него есть некоторая дополнительная информация. Давайте подумаем, где эта дополнительная информация может находиться.

У нас есть сам объект. То есть можно попытаться положить эту информацию в сам объект. Но у нас объект произвольный, по идее его вообще не должно волновать, что он был создан с помощью функции `make_shared`. Поэтому когда мы говорим о стандартных умных указателях, там в сам объект мы ничего добавить не можем, потому что объект абсолютно произвольный.

У нас есть объект `shared_ptr`, который создался. Казалось бы, в принципе, вроде бы да, мы можем положить туда произвольную информацию. Однако мы помним, что `shared_ptr` можно будет скопировать, и скопировать несколько раз. Эти `shared_ptr`, куда мы скопировали, будут жить, а исходный `shared_ptr` у нас уже умрет. Получается, что если мы положим какую-то информацию в этот `shared_ptr`, то тогда она в какой-то момент нам станет уже недоступна. То есть туда ее складывать тоже нельзя.

В сам объект складывать нельзя, в `shared_ptr` складывать нельзя. Что нам остается? Создать некоторое новое место. И действительно, `shared_ptr` **создает в куче новый небольшой утилитарный объект под названием** `ControlBlock`. В этом контрольном блоке он сохраняет счетчик ссылок, то есть текущее количество `shared_ptr`, которые указывают на этот динамический объект. А у себя, в объекте `shared_ptr`, он сохраняет только указатель на этот контрольный блок. Временный указатель нам уже не нужен. Получается, что **в** `shared_ptr` **у нас есть два указателя: и на сам объект, и на контрольный блок**. Это очень важно.

Давайте теперь посмотрим, как происходит копирование `shared_ptr`.

1. Создается новый объект `shared_ptr`. В него копируются указатели на тот же самый объект.

2. В него копируются указатели на тот же самый контрольный блок.

3. Дальше, поскольку у нас создался новый `shared_ptr`, который ссылается на тот же самый объект, нам нужно увеличить счетчик ссылок, что и происходит.

4. Мы увеличим счетчик ссылок, теперь он равен 2. Теперь у нас два `shared_ptr` указывают на один и тот же объект, и что характерно, если мы пойдем в любой из этих `shared_ptr`, мы сможем попасть в один и тот же контрольный блок, где будет записана вот эта информация.

Хорошо, давайте посмотрим, как происходит перемещение `shared_ptr`.

1. Создается новый `shared_ptr`, в него копируется указатель из того `shared_ptr`, из которого мы перемещаем. А в исходном `shared_ptr` этот указатель зануляется, как и в случае с `unique_ptr`.

2. И то же самое происходит с указателем на контрольный блок. Он копируется и зануляется в исходном `shared_ptr`.

3. При этом обратите внимание, что счетчик ссылок у нас не меняется, поскольку действительно у нас один `shared_ptr` стал указывать на этот объект, а другой перестал указывать на этот объект. Поэтому перемещение `shared_ptr` — это более эффективная операция, чем его копирование. И в принципе, как всегда, **перемещение — это в некотором смысле оптимизация операции копирования**.

Давайте теперь посмотрим на разрушение `shared_ptr`. Начнем с простой ситуации, когда разрушается `shared_ptr`, который никуда не указывает.

1. Как и в случае с `unique_ptr`, он просто берет и уничтожается. Это не влечет за собой никаких последствий.

Следующим примером рассмотрим удаление `shared_ptr`, когда существует какой-то другой `shared_ptr`, указывающий на тот же самый объект.

1. Начинает разрушаться `shared_ptr`, и в своем деструкторе он идет в контрольный блок, и на этот раз он уменьшает счетчик ссылок. Их было две, он уменьшает до одной.

2. Смотрит: а есть там еще какие-то ссылки? Да, счетчик не упал до нуля. Если счетчик не упал до нуля, значит, кто-то еще указывает на тот же самый объект, и значит, нам ничего делать больше не нужно.

3. После этого мы просто берем и уничтожаем текущий объект `shared_ptr`.

И самый интересный пример — это у нас удаляется последний `shared_ptr`, указывающий на данный объект.

1. Он начинает удаляться, он идет в контрольный блок и уменьшает счетчик ссылок.

2. Видит, что счетчик ссылок на этот раз упал до нуля. Значит, он последний, все, больше никого нет. За нами Москва. Нужно за собой прибраться. Первым делом он удаляет контрольный блок. В конце концов, контрольный блок он сам же и завел, для того чтобы отследить количество ссылок.

3. После этого он, очевидно, удаляет объект, поскольку именно этим занимаются умные указатели — они удаляют объекты.

4. И вот теперь, когда он за собой прибрался, он может быть удален сам.

На самом деле всё, конечно же, не так просто. На самом деле `unique_ptr` работает несколько сложнее, `shared_ptr` работает намного сложнее, чем было описано. Но это некоторая модель, которой на самом деле вам более чем достаточно для понимания большинства случаев работы с умными указателями, и которая вам позволит решать большинство практических задач.

1.4 Владение

На текущий момент мы с вами уже довольно подробно познакомились с управлением памятью в C++. Настало время несколько структурировать наши знания. Давайте посмотрим, какие вообще в C++ бывают виды объектов с точки зрения времени их жизни.

1. У нас бывают **автоматические** объекты, временем жизни автоматических объектов управляет компилятор. К автоматическим объектам относятся локальные переменные, глобальные переменные и члены классов.

2. Также у нас бывают **динамические** объекты. Временем жизни динамических объектов управляет программист. Мы уже знаем, что динамические объекты создаются с помощью ключевого слова `new`, которое используется в недрах функций `make_unique` и `make_shared`. А удаляются они с помощью ключевого слова `delete`, которое опять же используется в деструкторах умных указателей.

3. Также в C++ существует понятие **владения**. Считается, что некоторая сущность владеет объектом, если она отвечает за его удаление.

Давайте посмотрим, кто владеет разными видами объектов.
Для **автоматических** объектов:

- если говорить о локальных переменных, то локальными переменами владеет окружающий их блок кода. Действительно, когда поток управления программы выходит из блока кода, мы знаем, что уничтожаются все локальные переменные, которые были объявлены в этом блоке.

- Глобальные переменные: можно считать, что ими владеет сама программа. Мы знаем, что когда программа завершается, уже после завершения функции `main`, происходит удаление всех глобальных переменных.

- А членами класса владеет сам объект класса. Действительно, мы никогда не задумываемся о том, когда нам нужно удалять члены класса. Мы знаем, что в любой момент, как бы это ни произошло, когда будет уничтожен сам объект класса, он позаботится о том, чтобы в своем деструкторе удалить свои члены.

И с автоматическим объектами: эти правила, которые мы здесь перечислили, они очень четкие, и им следует компилятор. Они всегда работают ровно так.

С **динамическими** объектами ситуация несколько интереснее. Как вы думаете, кто владеет динамическими объектами? Как вы можете догадаться по названию данного блока, динамическими объектами владеют умные указатели. При этом

- `unique_ptr` обеспечивает уникальное, эксклюзивное владение объектами. На один объект может указывать только один `unique_ptr`, и один `unique_ptr` может указывать только на один объект.

- `shared_ptr` — это разделяемое, или совместное, владение. На один и тот же объект может указывать несколько `shared_ptr`.

- А вот **сырой указатель не владеет объектом**. Считается, что если у нас есть сырой указатель, мы его получили каким-то образом, то мы никаким образом не отвечаем за время жизни этого объекта, на который он указывает.

И вот тут ситуация достаточно интересна с динамическими объектами в том плане, что то, что мы сейчас перечислили, это соглашение. Это не совсем правила. И этим соглашениям должен следовать программист. При этом программист, вообще говоря, может их нарушить.

Давайте посмотрим на эти соглашения в действии на конкретном примере кода.

```
unique_ptr<Animal> CreateAnimal(const string& name) {
    if (name == "Tiger")
        return make_unique<Tiger>();
    if (name == "Wolf")
        return make_unique<Wolf>();
    if (name == "Fox")
        return make_unique<Fox>();
    return nullptr;
}
```

Нам даже не нужно заглядывать внутрь этой функции, чтобы понять, что эта функция создает объект и возвращает нам его во владение. Потому что она возвращает нам `unique_ptr`. Теперь мы у себя сохраняем `uniqie_ptr`, мы как-то им пользуемся, и владение находится на нашей стороне. Эта функция следует данным соглашениям.

Следующий пример

```
class Shape {
    shared_ptr<Texture> texture_;

public:
    Shape(shared_ptr<Texture> texture) : texture_(move(texture)) {}
};
```

Из сигнатуры конструктора мы можем понять, что новый объект фигуры, который создан с помощью этого конструктора, он будет находиться в совместном владении этой текстурой. Этот пример также следует нашим соглашениям.

Следующий пример с использованием сырого указателя.

```
void Print(const Expression* e) {
    if (!e) {
        cout << "Null expression provided" << endl;
        return;
    }
    cout << e->ToString() << "=" << e->Evaluate() << endl;
}
```

Функция, разумеется, не удаляет объект и ничего не делает с ним, что относилось бы к времени жизни этого объекта. То есть указатель, который она получила, — не владеющий.

Таким образом, существуют соглашения по владению динамическими объектами. Этим соглашениям следует программист. И эти соглашения могут быть нарушены. Они могут быть нарушены по ошибке, например, из-за простого незнания, или они могут быть нарушены по необходимости. Необходимость может возникнуть в том случае, если нам нужно ручное, низкоуровневое управление динамической памятью. Например, мы пишем свой контейнер.

В случае соблюдения этих соглашений, вы можете гарантировать, и это очень важно, и это то, зачем люди придерживаются данных соглашений, что в вашей программе не может быть никаких утечек памяти и не может быть двойного удаления объекта. То есть если вы следуете этим соглашениям, у вас такие ситуации просто теоретически невозможны. Нельзя написать такой код.

Однако если вы эти соглашения нарушаете, то все может сработать корректно, но при определенных ситуациях вы можете создать программы, в которых будут утечки памяти или двойное удаление. Давайте с вами посмотрим примеры, какие именно могут возникать проблемы.

Вот первый пример такой программы. Вопрос: как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
}
```

Это очень простая программа, здесь мы создаем динамический объект и потом его не удаляем. Очевидно, здесь будет утечка. Причем в данной программе мы нарушили наше соглашение — мы получили владеющий сырой указатель, но нам нужно было на нём вызвать `delete`, что мы забыли сделать.

Посмотрим следующий пример. Вот такая программа, чуть-чуть посложнее. Как вы думаете, как она отработает?

```
struct A {/*...*/};

void UseA(int x) {
    A* ptr = new A;
    if (x < 0) {
        return;
    }
    delete ptr;
}

int main() {
    UseA(-1);
}
```

В этой программе мы уже вызываем `delete`, после того как мы вызвали `new`, но дело в том, что до вызова `delete` у нас присутствует некоторое условие, причем в результате выполнения этого условия мы можем выйти из функции, и `delete` не будет вызван. Здесь функция вызывается как

раз с таким аргументом, что у нас условие будет выполнено и до `delete` мы не дойдем. То есть в данном случае у нас опять же будет утечка.

Этот пример может выглядеть немного надуманным, но на практике вот именно такая проблема и встречается чаще всего. Как это происходит? Функция обычно начинается с того, что она достаточно небольшая. Где-то в начале этой функции мы объект создаем, в конце мы его удаляем, сама функция в пять строчек — очевидно, что всё будет хорошо. Никакой утечки не будет. Потом ваш проект эволюционирует, эта функция увеличивается. В ней в какой-то момент становится уже сотни строчек, на ней начинают работать люди, которые даже не знают тех, кто изначально ее написал, и дальше в какой-то момент кто-то вставляет посередине вот такое условие — с досрочным выходом из функции. И забывает посмотреть, что у нас где-то там находится `delete`, его тоже нужно вызвать. И у нас получается вот такая ситуация — возникает утечка. Здесь мы нарушили то же самое соглашение, что и в предыдущем примере: мы создали владеющий сырой указатель, потому что здесь мы как бы сами его удаляем, то есть мы оставляем на нем ответственность за то, чтобы объект был удален.

Посмотрим следующий пример. Здесь у нас уже появляются умные указатели. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
    unique_ptr<A> up1(ptr);
    unique_ptr<A> up2(ptr);
}
```

Несмотря на то, что мы здесь используем умные указатели, все равно программа отработает неправильно. Недостаточно просто использовать умные указатели, нужно использовать их правильно. Что у нас здесь происходит? Мы создаем новый динамический объект, сохраняем его в сыром указателе. А потом передаем этот сырой указатель в конструктор двух `unique_ptr`. И у нас получается, что два `unique_ptr` указывают на один и тот же объект. Но `unique_ptr` — это очень простой класс, он в своем деструкторе смотрит: объект есть какой-то, на который я указываю? Есть. Значит, я его удалю. И оба `unique_ptr` в данном случае попытаются удалить объект. Это приведет к некорректной работе программы. Здесь мы нарушили соглашение об эксклюзивном владении `unique_ptr`. Мы знаем, что одним объектом должен владеть только один `unique_ptr`. Здесь мы его нарушили и получили проблему.

Посмотрим следующий пример. Здесь все то же самое, но вместо `unique_ptr` используются `shared_ptr`. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
    shared_ptr<A> sp1(ptr);
    shared_ptr<A> sp2(ptr);
}
```

Несмотря на то, что, казалось бы, `shared_ptr` как раз и нужен для того, чтобы организовать разделяемое владение и несколько `shared_ptr` действительно могут ссылаться на один и тот же объект, здесь ситуация гораздо более интересная. Мы создали новый объект, поместили его в сырой указатель. И опять же, мы передаем сырой указатель в конструктор двух `shared_ptr`. Когда мы создаем первый `shared_ptr`, он видит: сырой указатель, отлично, создам для него контрольный блок, все хорошо. Когда мы создаем второй `shared_ptr`, у него нет никакой возможности узнать, что есть уже какой-то другой `shared_ptr`, который указывает на тот же самый объект, поэтому он спокойно создаст еще один контрольный блок и будет указывать на тот же самый объект. И у нас получится два никак не связанных `shared_ptr`, каждый со своим контрольным блоком, которые указывают на один и тот же объект. И каждый из них, когда будет удаляться, будет считать, что на этот объект больше никто не указывает, и попытается его удалить. У нас, опять же, здесь произойдет, как и в предыдущем примере, двойное удаление. В этой программе мы более тонко нарушили наше соглашение, на самом деле `ptr` в данном случае является у нас владеющим. Мы об этом чуть подробнее поговорим попозже.

Давайте рассмотрим следующий пример. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

A* makeA() {
    return new A;
}

int main() {
    unique_ptr<A> up(makeA());
    shared_ptr<A> sp(makeA());
}
```

Эта программа похожа на предыдущую, но она уже отработает корректно, потому что здесь у нас создаются два умных указателя, оба создаются из сырых указателей, но каждый раз у нас создается новый динамический объект. То есть здесь у нас всё будет хорошо: `unique_ptr` будет указывать на свой объект, `shared_ptr` будет указывать на свой объект. Каждый из них его удалит. То есть вроде бы проблем нет.

Но давайте подумаем, что будет, если кто-то вызовет функцию `MakeA` и забудет передать этот указатель в конструктор умного указателя. Вот в этом случае у нас может произойти утечка, может быть точно такая же ситуация, которую мы рассмотрели в первых нескольких примерах. Поэтому несмотря на то, что эта программа работает корректно, она нарушает соглашение. Она создает владеющий сырой указатель, который возвращает функция `MakeA`. Поэтому её следует переписать с выполнением наших соглашений, например, вот таким образом:

```
struct A {/*...*/};

unique_ptr<A> makeA() {
    return make_unique<A>();
}

int main() {
```

```
    unique_ptr<A> up(makeA());
    shared_ptr<A> sp(makeA());
}
```

Это абсолютно корректная ситуация. И вот такая программа работает точно так же, корректно, но уже следует нашим соглашениям. Именно так и следует писать.

И давайте посмотрим ещё один, самый интересный пример. Как вы думаете, как отработает такая программа?

```
struct A {/*...*/};

int main() {
    auto up1 = make_unique<A>();
    unique_ptr<A> up2(up1.get());
}
```

В этой программе, несмотря на то, что мы даже не писали в явном виде `new` и `delete`, мы с вами всё равно умудрились нарушить соглашение и создать такую ситуацию, что произойдет двойное удаление.

Смотрите, что получается. Мы создаем новый динамический объект с помощью функции `make_unique`. Всё замечательно. Получаем `unique_ptr up1`. После этого мы достаем из него сырой указатель и подаем его в конструктор второго `unique_ptr`. И у нас опять получается ситуация, когда два `unique_ptr` указывают на один и тот же объект. В итоге это, конечно, приведет к двойному удалению. Идея здесь в том, что указатель, который передается в конструктор умного указателя, трактуется как владеющий. То есть в данном случае сырой указатель, который возвращает `get()`, был трактован как владеющий.

Давайте подведем небольшой итог. Мы рассмотрели соглашение по владению динамическими объектами. И после этого привели множество примеров, как эти соглашения можно нарушить. Причем некоторые из них были не совсем очевидными, как, например, последний пример. Давайте теперь сформулируем несколько практических положений, то есть что на практике означают эти соглашения, что это значит на уровне написания кода. На самом деле это означает

- Не нужно использовать `new` и `delete`, как мы это уже знаем.
- Поскольку нельзя использовать `new` для создания объектов, нужно использовать функции `make_unique` и `make_shared`.
- Нельзя использовать конструкторы умных указателей, которые принимают сырые указатели. То есть нельзя создавать умные указатели напрямую из сырых. А это ровно то, что мы сделали в последнем примере, потому что вот такой конструктор — он трактует сырой указатель, который ему передают, как владеющий, ведь он же сейчас создаст умный указатель, который будет владеть этим объектом. А кто им владел до того? Значит, это сырой указатель.

new и конструкторы скрываются от нас в недрах функций `make_unique` и `make_shared`. То есть `new` и конструкторы — это некоторый низкоуровневый инструмент управления, которым пользоваться не нужно. Вместо этого нужно пользоваться высокоуровневым — функциями `make_unique`

и `make_shared`. Это ровно то, что они делают: создают объект и конструируют умные указатели из указателя на динамический объект. А вот вызов `delete` скрывается от нас в деструкторах умных указателей. То есть, опять же, `delete` мы напрямую не вызываем. Мы полагаемся на то, что этим займутся за нас умные указатели.

Таким образом, мы с вами выяснили, что есть два вида объектов, с точки зрения времени их жизни: автоматические и динамические. Мы знаем, что **владение автоматических объектов берет на себя компилятор и существуют строгие правила, которым он следует**. А вот **владение динамическими объектами описывается соглашениями, которым следует программист**. Но эти соглашения могут быть нарушены. И их нарушение может привести к проблемам, как мы видели. Если же им следовать, то тогда мы гарантируем, что в нашей программе будут отсутствовать утечки памяти и отсутствовать двойные удаления.

1.5 Присваивание умных указателей

До сих пор мы с вами инициализировали умные указатели в момент их создания, и после этого их не меняли. Однако умные указатели можно присваивать, и периодически это бывает полезно. В римере, который мы использовали для демонстрации возможностей `shared_ptr`, мы создаем один `Actor`. Для того чтобы показать, как умные указатели присваиваются, нам понадобится парочка. Поэтому давайте дадим актору имя и будем передавать его в конструкторе, и сделаем так, чтобы у нас, когда `Actor` что-то говорит — он выводил информацию о том, кто это делает, то есть будем вводить имя.

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;

struct Actor {
    Actor(string name) : name_(move(name)){
        cout << name << "I was born! :)" << endl;
    }

    ~Actor() {
        cout << name << "I died :(" << endl;
    }

    void DoWork() {
        cout << name << "I did some job" << endl;
    }

    string name_;
};
```

Теперь давайте изменим нашу функцию `main`: продемонстрируем возможности присваивания на примере `unique_ptr`

```
int main() {
    auto ptr1 = make_unique<Actor>("Alice");
    auto ptr2 = make_unique<Actor>("Boris");
    run(ptr1.get());
    run(ptr1.get());
    return 0;
}
```

Мы видим достаточно ожидаемый вывод. У нас сначала создалась Алиса, потом создался Борис, они оба выполнили некоторую работу и после этого оба дружно умерли в обратном порядке. Все вполне ожидаемо. Хорошо.

Давайте теперь выполним присваивание, то есть напишем

```
    ...
    ptr1 = move(ptr2);
    run(ptr1.get());
    run(ptr1.get());
    return 0;
```

И теперь, прежде чем мы запустим программу, вопрос к вам: как вы думаете, что напечатают вот эти три строчки? Для того чтобы нам с вами было лучше видно, что же эти строчки напечатают, давайте сделаем вот такую отбивку. Чтобы в выходе их явно было видно.

```
    ...
    cout << "----" << endl;
    ptr1 = move(ptr2);
    run(ptr1.get());
    run(ptr1.get());
    cout << "----" << endl;
    return 0;
```

Соберем программу в таком виде и запустим. Итак, что же мы видим? Что первым делом после присваивания у нас удаляется Алиса. Как же так произошло?

Смотрите. У нас `ptr1` указывал на Алису, а потом `ptr1` присваивается тому, на что указывал `ptr2`. То есть получается, что на Алису в этот момент уже больше никто не указывает. Раз на нее больше никто не указывает, но она при этом управлялась умным указателем, она соответственно должна быть удалена, иначе бы она утекла. Соответственно, в этот момент она удаляется. Смотрим дальше на вывод. У нас какую-то работу выполняет Борис, и после этого вызывается `run` с нулевым актором. Ну действительно, `ptr1` у нас теперь указывает туда, куда указывал ptr2, то есть на Бориса, а из `ptr2` у нас было выполнено перемещение. То есть в нем у нас ничего не осталось. И поэтому при этом втором вызове `run` у нас выводится сообщение, что актора там нет. Все вполне логично.

Давайте теперь попробуем сформулировать точное правило: в какой момент у нас удаляются объекты, которые управляются умными указателями. До этого момента мы как бы говорили,

что умные указатели удаляют объект в своем деструкторе. И это правда, но не вся. Дело в том, что умные указатели могут не удалить объект в своем деструкторе, например, у нас есть один `shared_ptr`, он указывает на объект, он удаляется, но при этом другой `shared_ptr` продолжает указывать на тот же самый объект. Как мы прекрасно знаем, в этом случае объект не будет удален.

Кроме того, умные указатели могут удалить объект не в деструкторе, как мы это с вами только что видели, умный указатель удалил объект в момент присваивания, то есть после присваивания у нас Алиса была удалена. Наиболее точное правило будет звучать подобным образом: **динамический объект удаляется, когда им перестают владеть умные указатели**. При этом перестать владеть объектом умные указатели могут в несколько моментов.

- Во-первых, при разрушении, с этого мы собственно и начали.
- Во-вторых, при перемещении из умного указателя, это мы тоже подробно рассмотрели.
- И при присваивании умного указателя чему-то. Это мы с вами только что видели на примере присваивания `unique_ptr`.

Давайте теперь подробнее поговорим именно про присваивание, потому что в C++ их существует несколько разных видов

- Для начала **перемещающее присваивание** — ровно то, что мы сейчас с вами делали в программе. У нас `ptr1` указывает на Алису, `ptr2` указывает на Бориса. После того как мы выполняем перемещающее присваивание, у нас `ptr1` начинает указывать туда, куда указывал `ptr2`, то есть на Бориса. При этом `ptr2` у нас не указывает больше никуда, потому что мы из него переместили, а на Алису в этот момент никто не указывает. То есть в соответствии с нашей формулировкой умные указатели прекратили владение Алисой. Следовательно, в этот момент она должна быть удалена, раз на нее больше никто не указывает. Вот она и удаляется. Собственно, то, что мы с вами и видели. Это перемещающее присваивание, и оно будет одинаково работать и для `unique_ptr`, и для `shared_ptr`.
- Теперь посмотрим на **копирующее присваивание**. Копирующее присваивание, очевидно, применимо только к `shared_ptr`, потому что `unique_ptr` копировать нельзя. Пусть у нас вот такая ситуация: опять же, у нас Алиса и Борис, и три `shared_ptr`. Первый указывает на Алису, а остальные два указывают на Бориса. Мы присваиваем `ptr2 = ptr1`. После этого присваивания у нас `ptr2` начинает указывать вместо Бориса на Алису, потому что ровно туда указывал `ptr1`. Но при этом у нас на все объекты продолжают указывать какие-то умные указатели, то есть больше ничего в этот момент не происходит.

  Однако дальше давайте выполним еще одно присваивание. Теперь `ptr3` присваивается `ptr1`, и `ptr3` тоже начинает указывать на Алису вместо Бориса. Теперь Борисом больше не владеет ни один умный указатель, следовательно, в этот момент он должен быть удален, что и происходит.
- Теперь давайте посмотрим на еще один интересный вид присваивания — это **присваивание** `nullptr`. Оно, опять же, применимо и к `unique_ptr`, и к `shared_ptr`. Когда мы присваиваем умному указателю `nullptr`, он просто становится простым и прекращает владение объектом,

на который он указывал. И если на объект больше никто не указывает, следовательно, объект удаляется. **Подобное присваивание очень полезно, если нам нужно освободить владение объектом до того, как умный указатель прекратил свое существование**. То есть здесь `ptr1` у нас продолжает существовать, но больше уже никуда не указывает.

1.6 shared_ptr и многопоточность

На текущий момент мы с вами рассматривали только программы, которые работают в однопоточном режиме. Однако умные указатели и в частности `shared_ptr` можно очень часто встретить в многопоточных программах. Поэтому давайте сейчас напишем небольшой пример многопоточной программы, где `shared_ptr` используется для разделения некоторого ресурса.

```
#include <future>
#include <iostream>
#include <memory>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

class Data {
public:
    Data(string data) : data_(data) {
        cout << "Data constructed\n";
    }
    ~Data() {
        cout << "Data destructed\n";
    }

    const string& Get() const {
        return data_;
    }
    string& Get() {
        return data_;
    }
private:
    string data_;
};

void ShareResource(shared_ptr<Data> ptr) {
    cout << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << endl;
}

vector<future<void>> spawn() {
    vector<future<void>> tasks;
```

```
    auto data = make_shared<Data>("meow");

    for (int i = 0; i < 10; ++i) {
        tasks.push_back(async([=]() {
            ShareResource(data);
        }));
    }

    return tasks;
}

int main() {
    cout << "Spawning tasks...\n";
    auto tasks = spawn();
    cout << "Done spawning.\n";
}
```

Причём `data` в нашу лямбда функцию мы захватываем по значению. Потому что, как мы уже ожидаем, у нас эти задачи будут выполняться после того, как данная функция завершится, потому что мы возвращаем вектор из `future`. То есть захватывать по ссылке мы не можем. Если бы мы захватили по ссылке, мы могли бы обратиться к этому `data`, когда у нас он уже был разрушен, ведь это локальная переменная функции.

Давайте теперь запустим. Так, что у нас происходит? Чего-то он нам тут выводит, на самом деле не очень понятно чего выводит, как-то оно все вперемешку, и на самом деле он выводит нам это все вперемешку, потому что у нас несколько потоков одновременно начинают писать в `cout`. Одновременно писать в `cout` можно. То есть доступ к `cout` в принципе синхронизирован. Там не будет data race. Проблема в том, что у нас каждый вот этот оператор вывода `<<` может перекрываться в разных потоках. Получается у нас такая чересполосица. Соответственно, для того, чтобы этой чересполосицы избежать, нам нужно использовать ровно один оператор вывода.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << endl;
    cout << ss.str();
}
...
```

Соберём и запустим. Так, вот теперь мы видим, что у нас программа делает что-то осмысленное, значит, давайте читать. Мы начали создавать задачи. Затем зашли в функцию `spawn`. У нас вывод, что создались данные, а дальше у нас уже начали работать наши задачи, которые мы создали с помощью `async`. Вот мы видим, что у нас был расшарен наш ресурс из тредов с разными `id`. Дальше мы закончили исполнять, но при этом задачи продолжили выполняться, потому что это как раз ровно то, на что мы рассчитывали, что задачи будут продолжаться уже продолжать выполняться уже после того как отработала наша функция.

Итак, давайте теперь сделаем такую интересную вещь. Давайте выведем сколько у нас в дан-

ный момент существует на наши данные ссылок из различных `shared_ptr`. Для того, чтобы это сделать мы воспользуемся методом `use_count()`, который есть у `shared_ptr`. Он на практике не то, чтобы очень полезен. Потому что как раз в данном случае у нас многопоточное выполнение, и как только мы спросили некоторые `use_count()`, сразу же после этого этот `use_count()` у нас может измениться, поскольку у нас несколько потоков с ним взаимодействуют. Но при этом это будет достаточно полезно, чтобы примерно прикинуть некоторый масштаб количества использований.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << ", counter =
        " << ptr.use_count() << endl;
    cout << ss.str();
}
...
```

Запустим теперь. Мы видим, что у нас происходит с `counter`. 11, 10, 11, 8. Мы видим, что в процессе выполнения у нас счетчик действительно меняется, то есть `shared_ptr` у нас как-то копируются, умирают, то есть что-то происходит. Так, можно еще запустить. Картинка немного меняется, но общая суть остается примерно одинаковой: из разных тредов мы меняем этот счетчик. Какой отсюда можно сделать важный вывод? Несмотря на то, что мы не предпринимали никаких дополнительных усилий к тому чтобы обеспечить синхронизированный доступ к этому счетчику, программа у все равно работает корректно. Дело в том, что `shared_ptr` сам синхронизирует доступ к своему счетчику,и абсолютно нормально: из разных потоков этот счетчик менять. Поэтому мы как раз безопасно это можем сделать. А вот если мы попытаемся проделать аналогичную операцию с нашим `data`, вот тут у нас могут быть проблемы. Потому что доступ к `data` у нас не особо синхронизирован.

Так что давайте в нашем `ShareResource` попробуем взять и `data` поменять. Конкретно давайте в нашей строчке чего-нибудь допишем. И теперь давайте еще включим оптимизацию (Release), потому что, если оптимизации нет, то у нас это может не сработать.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << ", counter =
        " << ptr.use_count() << endl;
    cout << ss.str();
    ptr->Get().push_back('x');
}
...
```

Так. Соберем программу. Мы видим, что наши данные начинают меняться и в принципе пока вроде бы оно выглядит даже нормально. Однако! Давайте увеличим количество итераций — поставим, скажем, тысячу, и давайте уберем вывод в выходной поток, потому что вывод в выходной поток на самом деле нам обеспечивает синхронизацию собственно через этот вывод, потому что сам доступ к потоку синхронизирован, и запустим теперь. Вроде сработало. А вот теперь уже не сработало. Мы видим, что наша программа упала, а упала она ровно потому, что мы из нескольких потоков

модифицируем одни и те же данные, а это как мы уже знаем состояние гонки, так делать нельзя. Поэтому наша программа упала.

Соответственно, здесь для того, чтобы избежать этой проблемы нам нужно либо синхронизировать доступ к этим данным как мы это, собственно, делали раньше, либо в явном виде запретить модификацию этого объекта из нескольких потоков, и самый удобный способ это сделать — это на самом деле сказать, что объект, который нам приходит, он константный (`const Data`).

Так, давайте вернем небольшое количество итераций и соберем программу. Запускаем. Программа у нас работает как раньше, при этом мы на уровне интерфейса `ShareResource` мы как бы ограничили себя — мы не можем модифицировать данные через функцию `ShareResource`, — то есть обезопасили себя от состояния гонки. Этот приём достаточно часто применяется на практике.

Таким образом, мы узнали, что `shared_ptr` обеспечивает потокобезопасный доступ к счетчику ссылок. То есть вы можете безопасно копировать `shared_ptr` из нескольких потоков и удалять `shared_ptr` в этих потоках. Все это приводит к модификации счетчика, но вам не нужно беспокоиться о его синхронизации.

А вот о чем вам нужно беспокоиться, так это о синхронизации доступа к самому объекту, потому что здесь `shared_ptr` никак не влияет и никак не модифицирует правила игры. Если вы хотите менять объект из нескольких потоков, тогда обеспечьте некоторую синхронизацию для этого объекта. Однако же простой способ сделать всё-таки безопасный доступ — это в явном виде запретить модификацию объекта из нескольких потоков. То есть передавать `shared_ptr` на константный объект, тогда вы не сможете его модифицировать. Нет модификации — нет проблем.

И сейчас должно быть понятно, что перемещение `shared_ptr` — это некоторая оптимизация. Как вы помните, она позволяет избежать изменений счетчика, а изменение счетчика к тому же еще и синхронизировано, а синхронизированные изменения счетчика — это несколько более дорогая операция, чем простое изменение счетчика. И поэтому за счёт перемещения `shared_ptr` мы опять же можем избежать вот такой чуть более дорогой операции.

1.7 Умный указатель weak_ptr

На текущий момент мы с вами познакомились с основными и наиболее полезными возможностями умных указателей, которые приходится часто использовать на практике. Теперь мы поговорим о некоторых дополнительных возможностях, которые на практике используются не так часто, но если они нужны, то лучше знать о том, как их делать, иначе без них будет достаточно тяжело. Конкретно, мы поговорим о двух вещах: о ещё одном умном указателе `weak_ptr` и о пользовательском `deleter`, который можно использовать в `shared_ptr`.

Начнём с `weak_ptr`. Пусть у нас есть следующий пример

```
class Cache {
public:
    shared_ptr<Widget> GetWidget(const string& name) {
        if (!map_[name]) {
            map_[name] = make_shared<Widget>(name);
        }
```

```
        return map_[name];
    }

private:
    map<string, shared_ptr<Widget>> map_;
};
```

Это отлично работает, но есть одна небольшая проблема. Даже если мы создали какой-то `Widget` и в программе его уже больше не используем, этот `Widget` всё равно будет существовать, потому что на него сохраняется `shared_ptr` внутри этого объекта кеша, то есть `Widget` мы не используем, но он всё равно у нас висит и отжирает некоторые ресурсы. Нам бы хотелось сделать так, чтобы если в остальной программе мы перестали пользоваться этим `Widget`, то этот `Widget` удалялся бы. Как это можно сделать?

Первая мысль, которая приходит в голову, почему бы нам не использовать `use_count()`? Мы уже видели, что он вернет нам текущее количество ссылок из `shared_ptr` на данный объект. Если мы видим, что этот `use_count() == 1` — это значит, что только внутри нашего кеша хранится `shared_ptr` на этот объект, а больше в программе нигде не используется, и соответственно, мы можем его удалить. Казалось бы да, но есть некоторая проблема. Мы же не знаем точный момент, когда в остальной программе у нас уничтожится последний `shared_ptr`, который указывает на этот `Widget`, то есть непонятно когда этот `use_count()` вызывать.

Мы можем подумать, ладно, давайте заведем какой-нибудь фоновый поток в котором будем по таймеру смотреть на `use_count()`. Хорошо, так в принципе можно сделать, но тут возникает другая проблема: как мы уже знаем `use_count()` в многопоточной среде использовать очень ненадёжно, потому что значение, которое возвращает `use_count()` устаревает ровно в тот момент, когда мы его получили, ибо ровно в этот же самый момент из другого потока кто-то может у нас запросить этот же самый `Widget`, и, соответственно, счетчик ссылок у нас увеличится на одиночку. Получается, что если нам нужен `use_count()`, то нам нужен фоновый поток, но если фоновый поток — мы не можем использовать `use_count()`.

Есть другое решение, более подходящее в данном случае — это как раз использование `weak_ptr`: `weak_ptr` — **это невладеющий умный указатель**. Казалось бы, он вроде бы умный, но почему он тогда не владеющий? Чем он лучше обычного сырого указателя? Дело в том, что из `weak_ptr` можно корректно создать `shared_ptr`, который уже будет владеющим. Как мы знаем, из сырого указателя создавать `shared_ptr` — это гиблое дело. Это вообще считается низкоуровневым управлением динамической памятью, потому что каждый раз, когда мы создаем `shared_ptr` из сырого указателя у нас для этого объекта заводится свой контрольный блок, и созданные таким образом `shared_ptr` оказываются не связанными друг с другом, и они скорее всего приведут к двойному удалению объекта. А вот из `weak_ptr` можно абсолютно корректно создавать `shared_ptr`, и все эти `shared_ptr` будут иметь один и тот же контрольный блок. Давайте посмотрим, как у нас изменится код с использованием `weak_ptr`.

Метод `lock()` у `weak_ptr` как раз создает `shared_ptr`, для того объекта, на который указывает этот `weak_ptr`, если такой объект есть. Соответственно, если такой объект есть, то мы получаем у него него `shared_ptr` и возвращаем. А вот если этого объекта нет, то что происходит?

Объекта может не быть по двум причинам: либо этот объект ещё не был создан вообще, в принципе мы только создали кэш и в кэше `Widget` с этим именем нет, либо — и это самое главное,

этот объект когда-то был создан, но с тех пор им перестали пользоваться, то есть мы когда-то отдали на него `shared_ptr`, но потом этот `shared_ptr` и все его копии были уничтожены, и объектом больше никто не пользуется — счетчик ссылок упал до нуля, тогда этот объект будет удален и `weak_ptr` сможет понять, что объект был удален.

Соответственно, в обоих этих случаях метод `lock()` вернет нам пустой `shared_ptr` — это будет обозначать, что объекта у нас нет по той или иной причине.

В этом случае мы зайдем в условие, и, поскольку у нас объекта нет, но соответственно, его нужно создать, поэтому мы точно таким же образом вызываем функцию `make_shared` для `Widget` с данным именем, она возвращает `shared_ptr`, этот `shared_ptr` мы неявно конвертируем в `weak_ptr` и складываем его в `map`. После этого мы этот `shared_ptr` возвращаем.

```
class Cache {
public:
    shared_ptr<Widget> GetWidget(const string& name) {
        auto ret = map_[name].lock();
        if (!ret) {
            map_[name] = make_shared<Widget>(name);
        }
        return map_[name];
    }

private:
    map<string, weak_ptr<Widget>> map_;
};
```

Вот таким несложным исправлением мы смогли добиться ровно того поведения этой программы, которое нам нужно, за счёт использования ещё одного специфического умного показателя `weak_ptr`.

1.8 Пользовательский deleter

Пусть у нас есть следующая задача.

```
Widget* GetNonOwningPtr();
shared_ptr<Widget> GetOwningPtr();

/*?..*/ GetWidget(bool owning) {
    if (owning) {
        return GetOwningPtr();
    } else {
        return GetNonOwningPtr();
    }
}
```

Казалось бы, а какой тип будет возвращать эта наша написанная функция? Давайте подумаем.

- Допустим, она возвращает сырой указатель. Тогда вызвать функцию, которая возвращает сырой указатель, никаких проблем нет, мы просто вернем тот же самый сырой указатель.

Но что делать, если нам нужно вызвать функцию, которая возвращает `shared_ptr`? Мы можем, конечно, из этого `shared_ptr` достать сырой указатель с помощью `get()`, мы так уже делали. И это действительно сработает. Но мы вернем сырой указатель, который не будет участвовать во владении этим объектом. А нам бы хотелось, чтобы этот указатель смог поучаствовать во владении. Получается, что сырой указатель нам не подходит.

- Допустим, будем возвращать тоже `shared_ptr`. Тогда у нас не будет никаких проблем вызвать функцию, которая сама возвращает `shared_ptr` — мы просто вернем тот же самый `shared_ptr`. Но что делать с функцией, которая возвращает сырой указатель? Ведь мы знаем, что нельзя просто так взять и засунуть сырой указатель в `shared_ptr`. Тем более, если мы вернем `shared_ptr`, мы знаем, что `shared_ptr` — это владеющий умный указатель, и когда он будет удален или когда его копии будут удалены, он попытается удалить этот объект. А нам не нужно, чтобы он удалялся, ведь нам возвращают невладеющий указатель. Что же делать?

На самом деле мы можем сделать хитрый ход конем и всё-таки использовать `shared_ptr`. Смотрите, за счёт чего.

```
Widget* GetNonOwningPtr();
shared_ptr<Widget> GetOwningPtr();

shared_ptr<Widget> GetWidget(bool owning) {
    if (owning) {
        return GetOwningPtr();
    } else {
        Widget* ptr = GetNonOwningPtr();
        auto dummyDeleter = [](Widget*) {};
        return shared_prt<Widget>(ptr, dummyDeleter);
    }
}
```

Что делать, если нам нужно вызвать функцию, которая возвращает обычный указатель? Смотрите, мы его вызываем, получаем обычный указатель, и дальше мы этот указатель засунем в новый `shared_ptr`, который мы создали. Как же так, скажете вы, он же попытается его удалить. И мы знаем, что действительно, умные указатели удаляют объект, на который они ссылаются, когда они заканчивают владение этим объектом. Но это же C++. То есть на самом деле всё немного не так, всё немного все хитрее. **Умные указатели не удаляют объект, когда заканчивают владение объектом, на самом деле они для этого объекта вызывают `deleter`, который по умолчанию этот объект удаляет**; `deleter` — это просто некоторая функция. И эта функция по умолчанию реализована таким образом, что она просто вызывает `delete` для переданного указателя.

А здесь мы с вами завели свой `deleter`, в которм мы не делаем ничего. То есть получается, мы создаем такой `shared_ptr`, который указывает на некоторый объект и который, когда он заканчивает владение этим объектом, вызывает для него `deleter`, который не делает ничего. То есть по сути мы создали `shared_ptr`, который этим объектом не владеет. Получается, что в первой ветке мы возвращаем владеющий `shared_ptr`, который нам вернула функция, а во второй ветке мы возвращаем невладеющий `shared_ptr`, то есть мы возвращаем некоторый `shared_ptr`, который

условно владеет объектом. Обратите внимание, что в данном случае мы нарушили соглашение по владению динамическими объектами. Ну потому что как бы предполагается, что `shared_ptr` безусловно владеет объектами, на которые он ссылается. Однако здесь у нас была определенная причина. Мы решили, что для реализации такого поведения программы мы нарушим эти соглашения. И это может быть оправдано в зависимости от задачи, которую мы решаем.

Таким образом, использование `deleter` на самом деле позволяет нам создать условно владеющий `shared_ptr`, хотя это приведет к нарушению соглашения по владению. И на самом деле `deleter` доступен не только для `shared_ptr`, но для `unique_ptr` тоже можно указать свой `deleter`. Правда, работать это будет немножко по-другому.

Небольшое напутствие: всегда старайтесь следовать соглашению по владению динамическими объектами. На практике их нужно будет нарушать разве что в тех случаях, когда вам придется взаимодействовать с компонентами, которые уже по тем или иным причинам нарушают эти соглашения. Например, это может быть компонента, написанная с использованием старого стандарта C++, еще до C++11, когда не было умных указателей. Или это может быть компонента, написанная на чистом C, где, очевидно, нет никаких умных указателей. В этом случае рекомендуется эти компоненты в явном виде изолировать от всего остального кода и во всем остальном коде придерживаться этих соглашений.

Идиома RAII

2.1 Знакомство с редактором vim и консольным компилятором

Давайте познакомимся с той средой, в которой будем писать программы, а также контролировать их и запускать. Работать будем в операционной системе Linux, используя консольный текстовый редактор vim. Давайте с его помощью напишем простейшую программу, которая печатает «Hello, world!» на экране.

Пишем знакомый нам текст. К vim можно подключить различные плагины, которые позволят нам использовать и автодополнения кода и поиск по коду, но не будем сейчас это делать.

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!\n";
}
```

Итак, выполняем команду `g++`, то есть вызываем компилятор, указываем ему опцию `-std=c++17`, хотя в программе нет ничего, что требовало бы C++17, указываем имя файла и указываем с помощью опции `-o` имя того выходного исполняемого файла, в который компилятор запишет скопмилированный результат. Пусть это будет файл `hello.out`. Выполняем команду, и если вы не увидели на экране никаких сообщений — это значит, что программа скомпириловалась успешно. Теперь запускаем файл `hello.out` и видим на экране «Hello, world!».

```
# g++ --std=c++17 hello.cpp -o hello.out
# ./hello.out
```

2.2 Жизненный цикл объекта

Вспомним, что такое **жизненный цикл объекта**. Блок кода, грубо говоря, — это фрагмент программы, ограниченный операторными или фигурными скобками. Например, тело функции — это блок кода. Тело условного оператора или оператор цикла — это тоже блок кода. Блоки кода могут быть вложены друг в друга, как матрёшки. Если внутри какого-то блока кода объявляется переменная, то обычно такая переменная считается автоматической. Если тип этой переменной —

это какой-то класс, то такую переменную мы называем объектом. Автоматические объекты создаются на стеке и живут до конца блока. Это значит, что как только мы из блока выходим, для таких созданных к этому моменту переменных компилятор вызывает деструктор у соответствующего класса, а также генерирует и исполняет код, который очищает память, которую эта переменная на стеке занимала. Заметьте, что порядок удаления таких переменных обратен порядку их создания. В этом примере в искусственном блоке кода написаны две переменных:

```
{
    // ...
    string s;
    // ...
    vector<int> v = {1, 2, 3};
    //...
}
```

И, как бы мы из этого блока не вышли, гарантированно будут вызваны для них деструкторы. Эти деструкторы будут очищать ту память, в которой вектор и строка хранили свои данные.

В противовес автоматическим переменным, бывают переменные, которые можно создать в динамической памяти. Время жизни таких переменных управляется программистом. Создаются они с помощью конструкции `new`, удаляются с помощью `delete`, но дополнительная гибкость с управлением времени и жизни разменивается здесь на потенциальный набор проблем, который может быть связан с утечкой памяти. Ведь программисту приходится не забывать всякий раз освобождать переменную, когда она уже не нужна. Вот еще один искусственный пример.

```
string* p1;

{
    string* p2 = new string;
    p1 = p2
    // ...
}

delete p1;
```

Заметим, что умирает лишь сам указатель `p2`, но никак не та память, на которую он указывал. Мы нарочно сохраним этот указатель в переменной `p1`, тоже типа указатель, который переживет этот блок. И вот в конце вызовем `delete` для этой переменной `p1`. Именно в этот момент мы вручную удалим тот объект, который мы породили в динамической памяти.

Вспомним также про исключения. **Если в программе произошло исключение, то текущий блок покидается аварийно**. Это означает, что для переменных, которые к этому моменту автоматически были созданы на стеке, автоматически же будут генерироваться и вызываться деструкторы. Это явление называется **раскруткой стека**. Важно: **можно генерировать исключения в конструкторах. Более того, это единственный способ сообщить внешнему миру о том, что объект не может быть создан в конструкторе**. Однако генерировать исключения, которые покидают пределы деструкторов, крайне опасно. Считается, что все деструкторы должны отработать без сбоя. Представьте себе, что будет если в деструкторе, который

работает в момент раскрутки стека, произойдет еще одно исключение. Вложенные поля сложных объектов ведут себя похожим на стек образом. Перед телом конструктора они инициализируются, после выполнения тела деструктора они уничтожаются. Поэтому если внутри конструктора произошло какое-то исключение, то все проинициализированные, к этому моменту вложенные, поля — несмотря на то что сам объект, который конструируется, не будет считаться созданным, — будут корректно уничтожены с помощью деструкторов. Давайте рассмотрим вот такой пример.

```
class C {
    vector<int> vals;
public:
    C(const vector<int>& given) : vals(given) {
        if (any_of(begin(vals), end(vals), [](int v) {return v < 0;})) {
            throw runtime_error("negative values!");
        }
    }

    ~C() {
    }
};
```

Давайте напишем код, который все это иллюстрирует.

```
#include <iosream>

using namespace std;

class C {
public:
    C() {
        cout << "C()\n";
    }
    ~C() {
        cout << "~C()\n";
    }
};

int main() {
    {
        C c;
    }
    {
        C c;
    }
}
```

Скомпилируем эту программу. Запустим. И мы увидим, что сначала создан объект типа `C`, потом уничтожен. И опять создан объект типа `C` и снова уничтожен, то есть при выходе из блока автоматически вызывается деструктор.

Пусть теперь у нас есть какая-то вспомогательная функция, которую мы будем вызывать из функции `main`.

```
#include <iostream>
#include <stdexcept>
...
void foo() {
    C c1;
    throw runtime_error("");
    C c2;
}

int main() {
    try {
        foo();
    } catch (...) {
        cout << "exception\n";
    }
}
```

Скомпилируем такую программу, запустим её. Смотрите: несмотря на то, что в функции `foo` должны были создаваться два объекта, на самом деле, конечно же, был создан только один, который `c1` и который создаётся до генерации исключения.

Обратите внимание, что мы покинули функцию `foo` аварийно и для этого объекта деструктор был также вызван автоматически.

Таким образом, для автоматических объектов, то есть объектов, которые создаются на стеке, гарантированно будут вызываться деструкторы, каким бы способом мы не покинули блок. То же самое относится и к полям вложенных объектов классов.

2.3 Идея RAII

Вообще, **RAII расшифровывается как Resource Aquisition Is Initialization**. Считается, что это не очень удачное название. Перевести на русский его можно как «выделение ресурса (или получение ресурса, или захват ресурса) есть инициализация», то есть такая операция должна являться инициализацией в правильно написанном коде, инициализация какой-то переменной. Но давайте сначала разберемся, что такое ресурс.

Ресурс нам предоставляется нам напрокат какой-то третьей стороной, например, операционной системой. Этот ресурс надо не забыть вернуть, когда он нам больше не нужен, потому что ресурс ограничен. Типичный пример ресурсов — это файл, мьютекс или память. Так вот, эта идиома предлагает с каждым фактом запроса, захвата ресурса связывать какую-нибудь автоматическую переменную. Такая переменная с помощью своего деструктора будет автоматически возвращать этот ресурс, когда он окажется не нужен. Можно сказать, что идиома RAII — это такой рай для программиста, который позволяет легко следить за ресурсами. Проведем аналогию с обычной жизнью.

Пусть блок кода — это комната в каком-то помещении, в которой ходит программист. Выход из блока — это значит выход из комнаты через какую-то дверь. Исключение — это срочная эвакуация через запасной выход. Что в этой терминологии является ресурсом? Можно считать, что ресурсом является электричество, а освобождение ресурса — это требование выключить свет, когда мы выходим из комнаты. Нам нужно не забыть выключить свет, даже если объявлена эвакуация.

Что предлагает подход RAII? Подход RAII говорит, что выключать свет каждый раз вручную утомительно. Очень легко забыть это сделать, особенно в чрезвычайной ситуации, поэтому давайте поставим автоматический датчик, который будет в комнате следить за тем, что никого нет. Включать свет мы по-прежнему будем вручную, но если датчик сказал, что комната пуста и её все покинули, свет будет автоматически выключатся. Наверное, вы знаете, что есть языки с так называемой «сборкой мусора». Они предлагают совершенно другой подход. В этих языках по комнатам периодически ходит вахтер, который вручную выключает свет в тех комнатах, где никого нет. C++ не такой язык. Здесь всякая работа с ресурсом, следуя этой идиоме, должна быть обернута в какой-то блок кода, в начале которого специальные переменные инициализируются, а в ее деструкторах этот ресурс автоматически освобождается.

2.4 RAII-обёртка над файлом

Попробуем применить идиому RAII на практике к конкретному ресурсу. Давайте в качестве ресурса выберем банальный файл. Для начала попробуем поработать с файлом так, как мы это бы делали с помощью библиотеки языка C, где никакой идиомы RAII нет. Вот пример программы.

```
#include <cstdio>
#include <iostream>

using namespace std;

int main() {
    FILE* f = fopen("output.txt", "w");

    if (f != nullptr) {
        fputs("Hello, world!\n", f);
        fputs("This file is written with fputs\n", f);
        fclose(f);
    } else {
        printf("Cannot open file\n");
    }
}
```

У нас в программе может быть много мест, из которых мы можем выйти. В каждом из них, если мы работали с файлом, нам надо не забыть вернуть этот ресурс операционной системе. Закрывался файл с помощью функции `fclose`. Давайте запустим компилятор, программа успешно скомпилирована, запускаем её и видим, что у нас теперь появился файл `output.txt`, в который мы действительно что-то записали.

Теперь давайте попробуем переписать эту программу в духе идиомы RAII. Для этого объявим

специальный класс, который будет оборачивать в себе этот ресурс.

```
class File {
private:
    FILE* f;

public:
    File(const string& filename) {
        f = fopen(filename.c_str(), "w");
        if (f == nullptr) {
            throw runtime_error("cannot open " + filename);
        }
    }

    void Write(const string& line) {
        fputs(line.c_str(), f);
    }

    ~File() {
        fclose(f);
    }
};

int main() {
    try {
        File f("output.txt");
        f.Write("Hello, world!\n");
        f.Write("This is RAII file\n");
    } catch (...) {
        cout << "cannot open file\n";
    }
}
```

Посмотрите, мне нигде не пришлось писать явно функцию `fclose`, которая этот файл закрывает, кроме как в деструкторе класса. Она написана один раз, потому что моя переменная автоматическая. Каким бы способом она не вышла бы из этого блока, вот если бы было бы написано посередине `return`, или если бы здесь произошло еще какое-то исключение, созданная к этому моменту переменная будет автоматически освобождена.

Давайте скомпилируем нашу программу и убедимся, что она работает. Успешно скомпилировалась и запустилась, и мы видим, что она действительно записала в файл `output.txt` эти строки.

2.5 Копирование и перемещение RAII-обёрток

Итак, мы с вами написали класс `File`, который в духе идиомы RAII, являлся оберткой над низкоуровневой файловой переменной. В его конструкторе мы пытались выделить этот ресурс,

запросить его у операционной системы, то есть открыть файл. В его деструкторе мы этот файл закрывали. Давайте попробуем написать небольшой кусок кода, который покажет, что не все так просто и на самом деле наш файл нуждается в некоторых улучшениях.

```
int main() {
    try {
        File f("output.txt");
        File f2 = f;
        ...
    }
    ...
}
```

Давайте попробуем скомпилировать такую безобидную программу и посмотрим как она себя поведет. Скомпилировалась, но при запуске она как-то очень странно себя повела. Мы видим на экране какой-то непонятный дамп. Что же произошло? Написано «double free» — двойное освобождение.

Действительно, давайте разберемся: мы с вами создали копию нашей файловой переменной. Для переменной `f2` был неявно вызван конструктор копирования. Мы не написали в нашем классе никакого конструктора копирования, поэтому компилятор предоставил его нам по умолчанию. Что же делает этот конструктор копирования по умолчанию? А он просто копирует все поля, которые были в классе. В нашем классе единственное поле — это указатель. Он и был скопирован. Теперь получается, что два объекта `f` и `f2` хранят внутри себя указатель на одну и ту же область памяти, два одинаковых указателя. Что произойдет, когда эти переменные начнут выходить из блока? Как вы помните, первой умирает та переменная, которая была создана последней, то есть `f2`, и в момент работы деструктора для `f2` ничего плохого не происходит, закрывается наш файл. А вот в тот момент, когда умирает переменная `f`, мы пытаемся этот файл закрыть дважды. В этот момент и происходит ошибка. **Возвращение ресурса дважды — это большая ошибка, которую нельзя допускать**. Что же делать?

Давайте разберемся, в чем вообще причина этой проблемы? На самом деле причина в том, что мы не определили семантику копирования. Можно было бы делать по разному, например, можно было бы предположить, что в случае такого копирования объектов у нас должен создаваться какой-то новый файл, рядышком, с каким-то дополнительным другим именем, в который бы копировалось содержимое предыдущего файла и с которыми мы бы теперь работали независимо.

А можно сделать проще. Можно сказать, что объекты типа `File` мы просто запрещаем копировать. На самом деле, если мы что то подобное делаем с конструктором копирования, скорее всего, наверняка нам надо сделать что-то похожее и с оператором присваивания. Это можно написать так.

```
...
class File {
private:
    FILE* f;

    File(const File&) = delete;
    void operator = (const File&) = delete;
```

```
public:
    ...
};
```

Таким образом мы предостерегли себя от неосторожного использования этого класса.

2.6 RAII вокруг нас

Давайте обратим внимание, что **идиома RAII**, которую мы изучаем, на самом деле **повсеместно встречается в стандартной библиотеке языка C++**. Типичный пример — это, стандартные классы `string` и `vector`. Эти классы хранят свои элементы в динамической памяти. В конструкторе это динамическая память выделяется. В каких-то функциях, например, если мы в вектор добавляем новые элементы и требуется реаллокация, мы эту динамическую память перераспределяем. В деструкторе этих классов эта память непременно освобождается. Пользователю этих классов нет необходимости думать об этом. Всё это скрыто под капотом внутри. Именно в этом и удобство этой идиомы. Мы просто пользуемся этими классами, создаем такие переменные на стеке и совершенно не задумываемся о том, что когда эти переменные выходят из соответствующего блока, где-то выполняется код, который эту память освобождает.

Другой типичный пример идиомы RAII — это умный указатель `unique_ptr`. Здесь, в отличие от `vector`, динамическая память не выделяется в конструкторе. Наоборот, мы захватываем владение уже существующим указателем, который указывает на какую-то память, и который нам предоставил пользователь. А вот деструктор точно так же эту динамическую память освобождает.

Ещё в качестве примера можно привести файловый поток `fstream`, который чем-то напоминает нашу обертку над файлом, которую мы написали в прошлый раз. Есть небольшое отличие: у него другой интерфейс для ввода-вывода, он по другому обрабатывает ошибки в случае, если файл открылся неуспешно, но суть точно такая же. В своём деструкторе он пытается закрыть файл.

Ещё классический пример идиомы RAII в стандартной библиотеке, это работа с `mutex`: `mutex` — это тоже ресурс. Давайте попробуем посмотреть на голый `mutex` непосредственно. У него в интерфейсе есть две функции: заблокировать `mutex` и разблокировать: `lock()` и `unlock()`. Если бы мы пользовались им непосредственно, то вначале каждой функции нам бы приходилось брать блокировку, при выходе из этой функции её снимать, возвращать `mutex` обратно. Давайте рассмотрим пример, который вам уже знаком. Вы уже писали такой код, где в несколько потоков вызывается функция `Spend` которая пытается уменьшить баланс на какую-то величину.

```
struct Account {
    int balance = 0;

    bool Spend(int value) {
        if (value <= balance) {
            balance -= value;
            return true;
        }
```

```
        return false;
    }
};
```

Я напомню, что если этот код скомпилировать непосредственно, то может произойти такое, что у нас в итоге баланс окажется отрицательным, поскольку несколько потоков, которые не договорившись друг с другом, одновременно начнут уменьшать переменную `balance`. Для того, чтобы этого не было, надо внутри этой функцию взять блокировку. Попробуем это сначала сделать с помощью голого `mutex`.

```
#include <mutex>

...

struct Account {
    int balance = 0;

    bool Spend(int value) {
        m.lock();
        if (value <= balance) {
            balance -= value;
            m.unlock();
            return true;
        }
        m.unlock();
        return false;
    }
};
```

Программа скомпилируется и будет работать, баланс будет всё время нулевым, но обратите внимание, помнить о том, что при каждом выходе с функций надо не забыть написать `unlock` — это довольно утомительно, это чревато ошибками. Если наша функция окажется сложнее, из нее может быть очень много точек выхода, причем как явных, как здесь, так и не явных. Дело в том, что какие-то функции внутри могут сгенерировать исключение, которые мы не обработаем и тогда мы функцию `Spend` будем аварийно покидать, `mutex` в этом случае окажется в состоянии блокировки. Блокировку мы забудем снять. Что же делать?

Давайте воспользуемся таким вспомогательным классом `lock_guard`, также вам знакомым, который позволяет написать всё то же самое, но только в одну строчку.

```
#include <mutex>

...

struct Account {
    int balance = 0;

    bool Spend(int value) {
        lock_guard<mutex> guard(m);
```

```
        //m.lock();
        if (value <= balance) {
            balance -= value;
            //m.unlock();
            return true;
        }
        //m.unlock();
        return false;
    }
};
```

Давайте убедимся, что этот код тоже работает. Да, баланс все время получается нулевым.

На самом деле, классы, которые вы писали на занятиях, тоже в каком-то смысле исповедуют идиому RAII. `ObjectPool`, который хранил объекты, и даже `LogDuration`, который вы использовали для того, чтобы заметить время работы какого-то блока кода, можно считать примерами RAII. В `LogDuration`, конечно же, никакой ресурс не захватывался в самом начале, но его деструктор вызывался всякий раз, когда переменная выходила из блока, и он выполнял нетривиальные действия по замеру времени.

Давайте рассмотрим, почему в языке C++ нет блока `try/finally` как в некоторых других языках, например в Java. Блок `finally` нужен в тех языках для того, чтобы гарантированно освободить ресурсы, как бы ни завершился основной код, с исключением или без. Обычно пишется обертка `try`, в который помещается какой-то код, потенциально опасный, в котором производится работа с каким-то ресурсом, например, открывается файл. После этого в блоке `catch` перехватываются быть может какие-то исключения, и в самом конце в блоке `finally` вы пытаетесь закрыть файл, что бы ни произошло.

Можно провести такую аналогию из жизни, продолжая те аналогии, которые мы уже приводили. Если блок кода — это какая-то комната с множеством выходов, то в таких языках все выходы ведут в какой-то общий коридор, единый коридор, в котором есть один выключатель, который выключает свет. Этот выключатель непременно надо выключить руками. Почему этого нет **в языке C++**? Потому что полностью **блок `finally` заменен на идиому RAII**. Код, который мог бы быть написан при каждой обработке ошибки в таком блоке `finally`, пишется на самом деле ровно один раз и ровно в одном месте: в деструкторе соответствующего класса, который оборачивает этот ресурс.

Разбор задачи

3.1 Разбор задачи с использованием идиомы RAII

Давайте посмотрим на вспомогательные классы

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class FlightProvider {
public:
    using BookingId = int;

    struct BookingData {
        string city_from;
        string city_to;
        string date;
    };

    BookingId Book(const BookingData& data) {
        ++counter;
        cerr << "Flight booking: " << counter << "\n";
        return counter;
    }

    void Cancel(const BookingId& id) {
        --counter;
        cerr << "Cancel flight: " << id << "\n";
    }

private:
    int counter = 0;
};
```

По аналогии с этим классом `FlightProvider`, есть очень похожий класс `HotelProvider` для бронирования гостиниц.

```
class HotelProvider {
public:
    using BookingId = int;

    struct BookingData {
        string city;
        string date_from;
        string date_to;
    };

    BookingId Book(const BookingData& data) {
        ++counter;
        cerr << "Hotel booking: " << counter << "\n";
        return counter;
    }

    void Cancel(const BookingId& id) {
        --counter;
        cerr << "Cancel hotel: " << id << "\n";
    }

private:
    int counter = 0;
};
```

Давайте также предположим, что функция `Book` и в том, и в другом классе, вообще говоря, может генерировать исключения, например, если мест в гостинице нет или ни на какой рейс не удалось забронировать билет. Давайте сымитируем эту ситуацию

```
...
class FlightProvider {
    ...
    BookingId Book(const BookingData& data) {
        ++counter;
        if (counter > 1)
            throw runtime_error("Overbooking");
        cerr << "Flight booking: " << counter << "\n";
        return counter;
    }
    ...
```

Давайте также заметим, что отмена бронирования никогда не приводит к сбоям, а отменить бронирование какой-либо поездки или проживание в гостинице, которое было в прошлом, то есть уже совершено, это тоже нормальная, законная ситуация, которая не приводит к какой-либо ошибке.

Теперь давайте попробуем воспользоваться этими классами для того, чтобы написать систему управления командировками сотрудников. Мы хотим посылать сотрудников в командировки, и нам надо знать, на каких рейсах они летят в другие города и в каких гостиницах живут. Поэто-

му, пользуясь этими классами, мы напишем класс `TripManager`, у которого тоже будет похожий интерфейс.

```
struct Trip {
    vector<HotelProvider::BookingId> hotels;
    vector<FlightProvider::BookingId> flights;
};

class TripManager {
public:
    using BookingId = int;

    struct BookingData {
        string city_from;
        string city_to;
        string date_from;
        string date_to;
    };

    Trip Book(const BookingData& data) {
        Trip trip;
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        {
            HotelProvider::BookingData data;
            trip.hotels.push_back(hotel_provider.Book(data));
        }
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        return trip;
    }

    void Cancel(Trip& trip) {
        for (auto& id : trip.hotels) {
            hotel_provider.Cancel(id);
        }
        trip.hotels.clear();
        for (auto& id : trip.flights) {
            flight_provider.Cancel(id);
        }
        trip.flights.clear();
    }
```

```
private:
    HotelProvider hotel_provider;
    FlightProvider flight_provider;
};
```

Ну, что ж, давайте посмотрим, как теперь заработает такая функция `main`.

```
int main() {
    TripManager tm;
    auto trip = tm.Book({});
    tm.Cancel(trip);
}
```

Запускаем программу. Ой, и внезапно видим на экране, что перелёт забронирован, гостиница забронирована, а вот попытка забронировать обратный перелёт обернулась исключением, которое мы не перехватили. Ну, действительно, мы же там специально предусмотрели, что в случае, когда перелётов больше одного, то происходит исключение, мы это смоделировали. Да, мы вспомнили, что функция `Book`, вообще говоря, может выкидывать исключение, если бронирование не удалось, поэтому этот кусочек кода давайте обернём в `try/catch`, чтобы исключения цивилизованно ловить, чтобы наша программа не завершалась аварийно с такой ошибкой.

```
int main() {
    try {
        TripManager tm;
        auto trip = tm.Book({});
        tm.Cancel(trip);
    } catch (...) {
        cout << "Exception\n";
    }
}
```

Ещё раз запускаем, смотрим. И что же мы видим? Посмотрите, мы смогли забронировать перелёт, забронировать гостиницу, поймали даже это исключение. Но почему мы не видим сообщение об отмене этих перелётов? Ведь если всю командировку не получилось забронировать целиком, то те её составные части, которые мы уже забронировали, надо отменить. Предположим, что бронирование устроено так, что в случае, если мы его не отменяем, автоматически списываются деньги с какой-то карты. Это не здорово. Давайте посмотрим, как грамотно отменять такие результаты в случае каких-то внутренних сбоев, как сделать нашу командировку транзакционной.

Бронирование — это такой ресурс, который нам предоставляют соответствующие провайдеры `HotelProvider` и `FlightProvider`. Получается, что этот ресурс утекает. Для того чтобы справиться с этой проблемой, нам надо поправить функцию `Book`. Давайте обернем опасный блок кода, а именно вызовы функции `Book` у этих провайдеров, в `try/catch`.

```
class TripManager {
    ...
    Trip Book(const BookingData& data) {
        Trip trip;
```

```
      try {
        {
          FlightProvider::BookingData data;
          trip.flights.push_back(flight_provider.Book(data));
        }
        {
          HotelProvider::BookingData data;
          trip.hotels.push_back(hotel_provider.Book(data));
        }
        {
          FlightProvider::BookingData data;
          trip.flights.push_back(flight_provider.Book(data));
        }
      } catch (...) {
          Cancel(trip);
          throw;
      }
      return trip;
  }
  ...
```

Давайте посмотрим, как теперь изменится программа. Посмотрите, теперь действительно в случае исключения мы отменяем два предыдущих бронирования. Это ожидаемое поведение, которого мы и хотели достичь.

Однако поглядите на эту функцию `Book`. Кажется, что она устроена слишком сложно. Она теперь превращается в такие макароны кода, которые обернуты вот этими блоками `try/catch` на случай, как бы чего не вышло. На самом деле, мы можем элементарно забыть написать этот `try/catch` в какой-нибудь аналогичной функции, которая, например, добавляет новые города к командировке. Более того, интерфейс нашего класса, нашей структуры `Trip` вообще открыт, и кто угодно может добавить новую командировку, если у него есть доступ к соответствующему провайдеру. В случае, если произойдет такая же ошибка, он может позабыть просто отменить предыдущее бронирование. Такой код чреват утечками наших ресурсов.

Давайте посмотрим, что предлагает нам идиома RAII взамен. **Идиома RAII гласит: каждый ресурс следует обернуть в объект, который за него отвечает**. RAII, напомним, расшифровывается как Resource Acquisition Is Initialization. И там, казалось бы, речь идет про инициализацию, про выделение ресурса. Но **гораздо важнее в этой идиоме помнить про деструктор**, ведь именно в деструкторе класса будет написан код, который этот ресурс возвращает. Давайте, следуя этой идиоме, перепишем эту структуру `Trip`.

```
class Trip {
private:
    HotelProvider& hotel_provider;
    FlightProvider& flight_provider;

public:
    vector<HotelProvider::BookingId> hotels;
```

```
    vector<FlightProvider::BookingId> flights;

    Trip(HotelProvider& hp, FlightProvider& fp)
        : hotel_provider(hp), flight_provider(fp) {}

    Trip(const Trip&) = delete;
    Trip(Trip&&) = default;

    Trip& operator=(const Trip&) = delete;
    Trip& operator=(Trip&&) = default;

    void Cancel() {
        for (auto& id : hotels) {
            hotel_provider.Cancel(id);
        }
        hotels.clear();
        for (auto& id : flights) {
            flight_provider.Cancel(id);
        }
        flights.clear();
    }

    ~Trip() {
        Cancel();
    }
};
```

еперь давайте попробуем переписать наш `TripManager`, чтобы он смог работать с этой новой версией класса `Trip`. Нам уже не нужен блок `try/catch`, мы можем смело его убрать. И код становится таким же, как был в нашей первой, наивной версии. Действительно, если в какой-то момент одно из бронирований окажется неудачным, и произойдет исключение, то раскрутка стека гарантирует нам, что все созданные к этому моменту автоматические объекты будут уничтожены, для них будет вызван деструктор. И поэтому для объекта `trip` будет вызван деструктор, который вызовет `Cancel`, такая поездка будет отменена.

```
class TripManager {
    ...
    Trip Book(const BookingData& data) {
        Trip trip(hotel_provider, flight_provider);
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        {
            HotelProvider::BookingData data;
            trip.hotels.push_back(hotel_provider.Book(data));
        }
```

```
    {
        FlightProvider::BookingData data;
        trip.flights.push_back(flight_provider.Book(data));
    }
    return trip;
}

void Cancel(Trip& trip) {
    trip.Cancel();
}
...
```

Попробуем скомпилировать нашу программу. Наша программа запустилась, и она ведет себя так же ожидаемо, как и в версии с обёрткой `try/catch` внутри функции `Book`.

Обратите внимание, что нам теперь нигде не пришлось писать этот `try/catch`. В коде класса `Trip` мы его не написали. Давайте посмотрим, что нам это дало. Мы заметили, что у нас есть определенный ресурс — это командировка. Он, в свою очередь, состоит из каких-то составных элементарных ресурсов, которые нам предоставляются провайдерами: `HotelProvider` и `FlightProvider`. Если проводить аналогию, скажем, с памятью или с файлами, то в роли таких провайдеров системных ресурсов обычно выступает операционная система, которая не представлена каким-то объектом явно, она находится за кулисами. Но вот здесь у нас особый тип ресурса, и поэтому эти провайдеры должны быть нам известны явно. Можно было бы развивать эту идею дальше и переделать интерфейс наших провайдеров, чтобы они в функции бронирования возвращали бы не идентификатор бронирования, а тоже какой-нибудь объект, который умеет сам себя отменять в случае, если вызван его деструктор или что-то пошло не так. Но мы ограничились тем, что написали такую обертку `Trip` для составного бронирования.

Таким образом, оборачивайте ваши ресурсы в классы, следуя идиоме RAII, пишите деструкторы этих классов. **Именно в деструкторах должно возвращаться владение этими ресурсами тем, кто их предоставил**.

Основы разработки на C++. Функции: принципы понятного кода

Оглавление

Зачем нужны функции?

1.1 Зачем нужны функции?

Что такое хороший код? Это очень субъективное понятие, и все зависит от контекста, если у вас маленькая команда и маленький проект, то можете писать более-менее как угодно, но если у вас в команде 20 человек, они пишут один и тот же код, то очень важно, чтобы все понимали его одинаково и придерживались одинаковых договоренностей. Поэтому вам будут даны рекомендации по написанию хорошего кода. Вы можете им следовать, а можете не следовать, но при этом вы будете понимать последствия нарушения этих рекомендаций. Итак, прежде чем приступить к обсуждению того, как же писать хорошие функции, мы начнем с обсуждения того, зачем же вообще в языке C++ нужны функции.

Для этого давайте возьмем задачу из второго курса — «Демографические показатели». Вот ее решение, которое было опубликовано.

```
enum class Gender {
    FEMALE,
    MALE
};

struct Person {
    int age;
    Gender gender;
    bool is_employed;
};

template <typename InputIt>
int ComputeMedianAge(InputIt range_begin, InputIt range_end) {
    if (range_begin == range_end) {
      return 0;
    }
    vector<typename InputIt::value_type> range_copy(range_begin, range_end);
    auto middle = begin(range_copy) + range_copy.size() / 2;
    nth_element(
        begin(range_copy), middle, end(range_copy),
        [](const Person& lhs, const Person& rhs) {
          return lhs.age < rhs.age;
        }
    );
```

```
    return middle->age;
}

void PrintStats(vector<Person> persons) {
    auto females_end = partition(
        begin(persons), end(persons), [](const Person& p) {
            return p.gender == Gender::FEMALE;
        }
    );
    auto employed_females_end = partition(
        begin(persons), females_end, [](const Person& p) {
            return p.is_employed;
        }
    );
    auto employed_males_end = partition(
        females_end, end(persons), [](const Person& p) {
            return p.is_employed;
        }
    );

    cout << "Median age = "
         << ComputeMedianAge(begin(persons), end(persons)) << endl;
    cout << "Median age for females = "
         << ComputeMedianAge(begin(persons), females_end) << endl;
    cout << "Median age for males = "
         << ComputeMedianAge(females_end, end(persons)) << endl;
    cout << "Median age for employed females = "
         << ComputeMedianAge(begin(persons), employed_females_end) << endl;
    cout << "Median age for unemployed females = "
         << ComputeMedianAge(employed_females_end, females_end) << endl;
    cout << "Median age for employed males = "
         << ComputeMedianAge(females_end, employed_males_end) << endl;
    cout << "Median age for unemployed males = "
         << ComputeMedianAge(employed_males_end, end(persons)) << endl;
}

int main() {
    int person_count;
    cin >> person_count;
    vector<Person> persons;
    persons.reserve(person_count);
    for (int i = 0; i < person_count; ++i) {
        int age, gender, is_employed;
        cin >> age >> gender >> is_employed;
        Person person{age, static_cast<Gender>(gender), is_employed == 1};
        persons.push_back(person);
    }
```

```
    PrintStats(persons);
    return 0;
}
```

А теперь давайте представим, что нам запретили пользоваться функциями, ну, кроме разве что функции `main`, и весь код оказался в функции `main`. Что тогда с этим кодом случится?

Давайте перейдем в конструктив и поймем, что же в этом коде такого плохого — что случилось от того, что мы избавились от функций? Итак,

- Во-первых, совершенно очевидно, что копируя код вычисления медианного возраста несколько раз, мы явно закрыли себе возможность переиспользования этого кода. Если мы где-то еще захотим вычислить медианный возраст набора людей, мы этот код должны скопировать и как-то его поменять, при этом высока вероятность ошибиться, конечно. То есть первое **преимущество функции** — это **возможность переиспользовать код**.

- Дальше, давайте ещё посмотрим, что же плохого в этом коде? Как мы будем его тестировать? Сейчас у нас огромнейшая функция `main`, и если мы хотим этот код протестировать, мы должны написать какую-то внешнюю программу, которая будет нашу запускать — подавать ей на вход входные данные в стандартный поток ввода и смотреть, что же она вывела в стандартный поток вывода. То есть на вход — набор людей, на выходе — несколько чисел. Мы так не сможем протестировать непосредственно функцию вычисления медианного возраста. Когда же эта функция была, мы могли написать на нее unit-тест, например, с помощью нашего unit-тест фреймворка, подав конкретные входные данные в функцию и посмотрев, что же она вычислила. Итак, второе **преимущество функций** — это **возможность писать на них unit-тесты**.

- Третье **преимущество функций** — это **отсутствие необходимости или даже минимальная необходимость в комментариях**. Потому что, если посмотреть на то, как выглядел хороший код с функцией `ComputeMedianAge`, тут было явно видно, что мы вычисляем медианный возраст от такого диапазона людей. В плохом же коде — это просто некоторый блок кода. То есть понятное название функции помогает нам понять, что же в этом коде происходит — она документирует этот код.

- Наконец, смежное преимущество — это фиксированный ввод и вывод функции. Когда функция есть, вот функция `ComputeMedianAge`, мы явно видим, что она принимает на вход — в данном случае два итератора на людей, и явно видим, что она возвращает — она возвращает целое число. Если же мы посмотрим соответствующий блок кода в плохом варианте, то будет совершенно неочевидно, что же здесь приходит на вход этого блока, а что же на выходе — совершенно непонятно, какими данными манипулирует этот участок кода. Итого, четвертое **преимущество функции** — это **фиксированная сигнатура**. Функция явно декларирует свой ввод и вывод.

1.2 Функции или методы классов?

Давайте обсудим разницу между функциями и методами класса. Итак, есть, во-первых, понятные технические отличия. У класса всё-таки есть поля, и любые методы имеют доступ к этим

полям, возможно, даже на запись, если эти методы не константные. Поэтому в эти поля можно уносить некоторый глобальный контекст методов класса. Соответственно, если у вас есть набор функций, и им постоянно нужно менять какие-то определённые данные или их читать, то вы можете сделать их методами класса, а весь глобальный контекст увести в поля этого класса. Это, с одной стороны, удобно, а, с другой стороны, конечно, может усложнять понимание, потому что метод, получается, ещё имеет какие-то дополнительные данные в полях, и неизвестно, что вообще с ними будет происходить.

С другой стороны, есть важное семантическое отличие, смысловое. А именно, в C++ принято, что какой-то конкретный класс или, вообще говоря, любой тип олицетворяет собой какой-то объект. Поэтому просто так объединить набор функций в какой-то класс просто потому, что вам это удобно, может быть довольно сомнительным действием. Давайте рассмотрим пример. Опять же, задача «Демографические показатели». Если написать некоторые функции, то функция `main` будет устроена предельно просто

```
int main() {
    PrintStats(ComputeStats(ReadPersons()));
    return 0;
}
```

У вас может возникнуть желание объединить функции с похожим смыслом, с похожим назначением в один клас или структуру. Структуру, скажем, `StatsManager`, и в неё внести все эти функции в виде методов.

```
struct StatsManager {
    static vector<Person> ReadPersons(istream& in_stream = cin);
    static AgeStats ComputeStats(vector<Person> persons);
    static void PrintStats(const AgeStats& stats, ostream& out_stream = cout);
};
```

Теперь давайте обновим функцию `main`. И здесь мы тоже должны к вызову каждого метода добавить `StatsManager::`.

```
int main() {
    StatsManager::PrintStats(
        StatsManager::ComputeStats(StatsManager::ReadPersons()));
    return 0;
}
```

И, в принципе, наверное, ради этого вы могли это сделать. Потому что сейчас явно видно, что все эти функции относятся к задаче работы со статистикой.

Но, с другой стороны, если мы всё делали ради того, чтобы добавить какую-то строчку и два двоеточия перед всеми функциями, какой-то общий префикс, который их как-то объединяет, почему бы нам не использовать пространство имён. С тем же успехом я мог весь этот код заключить в одноимённый `namespace`. Ну и получилось бы примерно то же самое, зато никаких лишних сущностей.

Итак, основная рекомендация: если вы хотите просто так объединить набор смежных функций в класс, не имеющий никаких полей, остановитесь и используйте пространство имён. Если же вы

делаете класс, потому что считаете, что в дальнейшем появится глобальный контекст, нет, не плодите, пожалуйста, лишние сущности и всё равно используйте пространство имён или просто свободные функции, в них нет ничего плохого.

1.3 Какими должны быть функции?

Какими же важными свойствами функции должны обладать, чтобы считаться хорошими, и чтобы код был понятным?

1. Первое свойство — это **понятность сигнатуры функции**. Вы должны уметь посмотреть на сигнатуру функции и из этого понять, что же функция делает. По ее названию, по входным параметрам, по типу выходного параметра.

2. Во-вторых, эта **функция должна быть понятна в месте вызова**. То есть когда функция вызывается, должно быть понятно примерно, что каждый параметр означает и чего ожидать от этой функции в привязке к этим параметрам.

3. **Функции надо писать такими, чтобы их было удобно тестировать**. Понятно, что есть функции, которые вы вообще не протестируете, но и написать функцию можно настолько плохо, что тестировать ее будет неудобно.

4. **Функция должна быть поддерживаемой**. То есть если придет какой-то другой разработчик, или вы, и вы захотите как-то обновить функциональность этой функции, расширить ее, это должно быть легко.

5. И наконец, **связанное свойство — это сфокусированность функции**. То есть функция должна решать одну конкретную задачу. Не несколько маленьких как-то переплетенно, а одну конкретную.

И давайте мы сейчас все эти важные свойства проиллюстрируем несколько гипертрофированными, но тем не менее, понятными примерами на примере известных вам уже задач.

Начнем с **понятности сигнатуры**. Задача «Экспрессы». Представьте, что вы увидите чье-то решение этой задачи и там есть функция `Process`.

```
void Process() {...}
```

Совершенно непонятно, что делает эта функция, если не посмотреть внутрь этой функции.

Еще один пример функции с плохой, непонятной сигнатурой — вот такой. Задача «Трекер задач». Опять же, представьте, что вы видите чье-то решение этой задачи, и там вот такая функция — `PerformPersonTasks`, которая принимает, во-первых, словарь из строки в `TasksInfo`, по неконстатной ссылке, затем она принимает имя человека по константной ссылке — это, допустим, понятно. Она принимает количество задач, которые надо выполнить, и еще две не константных ссылки `first` и `second` на объекты типа `TasksInfo`.

```
void PerformPersonTasks(map<string, TaskInfo>& person_tasks, const string& person, int
   task_count, TaskInfo& first, TaskInfo& second) {...}
```

Понятно ли, что это за `first` и `second`?

Еще какая-то не константная ссылка на `person_tasks`. То есть функция как минимум меняет данные по трем ссылкам — это первый аргумент функции и два последних. При этом она еще ничего не возвращает. То есть она принимает что-то на вход, а на самом деле, `TasksInfo` — это тоже словари, и как-то она эти три словаря меняет. Кошмар. Непонятно, что происходит.

На самом деле, изначально решение было довольно приятным, с приятным интерфейсом. Там был класс `TeamTasks`, у него было приватное поле `person_tasks_` с тем самым словарем и метод — `PerformPersonTasks`, который принимал имя человека и количество задач и возвращал через кортеж два словаря задач со статистикой — задачи, которые сделаны, и задачи, которые не затронуты.

```
class TeamTasks {
public:
    ...

    tuple<TaskInfo, TaskInfo> PerformPersonTasks(const string& person, int task_count);

private:
    map<string, TaskInfo> person_tasks_;
};
```

Вот такой интерфейс был понятным.

Второе важное свойство функции — это **понятность** этих функций **в месте вызова**. Давайте для примера возьмем задачу «Hotel manager». Представьте, что мы хотим вызвать метод `Book` с какими-то конкретными значениями с целочисленных полей, и у нас получается такая вот строка

```
manager.Book(0, "Mariott", 1, 2);
```

Понятно ли чем здесь отличается 1, 2 и 0 — какие-то 3 числа? Их можно совершенно спокойно перепутать, и код все равно продолжит компилироваться. Здесь могла бы помочь среда разработки, если навести на метод `Book`, можно увидеть, что у меня сначала `client_id`, потом `room_count`. Но ничто не мешает ошибиться при написании этой функции, не посмотрев подсказку, или просто не понять при чтении этого кода, что же здесь происходит. Итак, будьте осторожны с функциями, которые принимают несколько целочисленных аргументов.

Следующая проблема, это **удобство тестирования функций**. Вернемся к нашему замечательному примеру с функцией `Process` в экспрессах. Это функция не принимает ничего и не возвращает ничего.

```
void Process() {
    ...
    cin >> ...
    ...
    cout << ...
}
```

Как написать на нее unit-тест — непонятно. Непонятно, что вообще с ней делать, как для нее переопределить входные данные. Если бы она хотя бы принимала поток по ссылке, можно было

бы его переопределить, то еще куда ни шло, но сейчас глобальная переменные — потоки `cin` и `cout` не позволяют написать нормальный unit-тест на эту функцию.

Следующее свойство — это **поддерживаемость функции**. Давайте рассмотрим пример неподдерживаемой функции, даже неподдерживаемой архитектуры класса, на примере системы бронирования отелей.

Давайте представим, что внутри всех классов мы решили отказаться от структуры `Booking`.

```
struct Booking {
    int64_t time;
    int client_id;
    int room_count;
};
```

У нас теперь 3 отдельные очереди вместо одной: есть очередь времен, очередь `client_id` и очередь количеств комнат, и везде нужно писать три раза `push`, потом надо написать три раза `pop` — и, наверное, такой код может показаться нормальным, но как только мы будем добавлять какое-то новое свойство бронирования, нужно будет добавить еще одну очередь, `push`, и, самое важное — то место, где очень легко ошибиться, это забыть добавить `pop`. То есть, **структура нам здесь помогала сделать код более поддерживаемым, более устойчивым к ошибкам, которые могут возникнуть при расширении функциональности**.

И наконец про **сфокусированность**. Опять же хороший пример несфокусированной функции — это функция `Process`, потому что она делает сразу несколько вещей. Во-первых, она считывает данные, во-вторых, она выводит данные, ну и наконец она взаимодействует с переменной `routes`. Получается, что смотря на какой-то произвольный кусок функций, давайте представим, что она очень большая, нельзя заранее угадать, а не считает ли она ещё какие-то данные. Она может быть даже уже что-то считала и вывела, и потом еще раз что-то считала и вывела, то есть несколько задач одной функции в ней как-то могут перемещаться — и это очень неприятно и очень мешает понимать функции, особенно если они довольно большие.

Вы можете увидеть несколько примеров хороших функций, чтобы вы понимали на что вам ориентироваться.

Функция, которая вычисляет остаток от деления двух вещественных чисел.

```
double remainder(double x, double y);
```

Метод вектора `size`

```
template<typename T>
size_t vector<T>::size() const;
```

функция `to_string`, которая принимает число и возвращает строчку.

```
string to_string(int value);
```

Алгоритм `count`, который подсчитывает сколько в данном диапазоне элементов встречается конкретных объектов.

```
template<typename InputIt, typename T>
size_t count(InputIt first, InputIt last, const T& value);
```

1.4 Философия понятного кода

Мы довольно много поговорили о том, какими же должны быть хорошие функции. И наверняка у некоторых из вас начал зарождаться вопрос, почему же мы такие зануды? На самом деле мы, команда курса, считаем, что особенно когда вы работаете над большим проектом, в большой команде нужно максимально щепетильно относиться к качеству вашего кода. Когда вы проектируете интерфейс вашего кода, нужно быть параноиком и перфекционистом. **Очень важно выработать навык предвидения того, как можно неправильно интерпретировать ваш замысел, который вы вложили в ваш код**.

И вы здесь можете спросить: зачем мне быть идеальным разработчиком? Почему мне нельзя быть просто хорошим? И вот без этого всего, без всех этих страшных рекомендаций, просто буду писать код как-нибудь и все будет нормально.

Хорошо, представьте, что вы работаете в большой команде и вы хороши, но так процентов на 60. И вот у вас есть коллеги, скажем 10 человек, вы написали какой-то код и дальше ваши коллеги иногда приходят и его как-то читают, исправляют. И вот 6 человек, они правильно поняли ваш замысел, как-то код обновили, все довольны, вы довольны и коллеги довольны, код понятный, но остальные 4 ничего не поняли. Каждый из них либо прочитал код и страдает, либо как-то его кое-как исправил и все разнес, и в итоге страдают и эти 4 человека, и вы страдаете, потому что ваш код теперь совершенно непонятен и тем 6 людям тоже, им сначала было хорошо, а теперь они тоже в печали, потому что непонятно что случилось с вашим кодом. Именно поэтому планка качества к коду очень высока, особенно когда вы работаете в большой команде.

Однако, некоторые из вас могут сказать: зачем мне писать понятный код, если всегда есть комментарии? То есть написал код как-нибудь, прокомментировал, и все замечательно. Даже непонятный код будет понятен, потому что есть комментарии. Но знаете если вы издаете книгу с вашим кодом, то есть вы написали, все там пояснили, распечатали и она зафиксирована, ну почему бы и нет, пишите как хотите, но всё таки, если речь идет о реальном коде, который работает в продакшене, его как-то меняют, он как-то развивается, то нужно думать еще и о том, что ваш код будет кто-то менять. И вряд ли ему будет дело до изменения комментариев. Вообще главная проблема комментариев в том, что если вы забыли их обновить, то код продолжит компилироваться и нормально работать. Если же вы допустили ошибку в коде, то он сломается. **Поэтому комментарии часто рассинхронизируются с основным кодом и нужно стараться их количество минимизировать, например, за счет понятности функций**.

В некоторых командах, на самом деле, есть хорошая практика — документирование не совсем всего кода, а только заголовка функции, только сигнатуры, то есть там буквально, в нескольких строчках написано, что эта функция делает. Очень замечательно если ваша команда к этому приучена и обновляет эту документацию, но если команда так не делает, то довольно тяжело будет заставить ребят обновлять комментарии к функциям. Гораздо проще научиться все таки сами функции поддерживать в адекватном состоянии, поэтому комментарии, они иногда нужны если речь идет о сложном коде, но далеко не везде. **Старайтесь в первую очередь делать код понятным**.

Детали проектирования функций

2.1 Как передать объект в функцию

Как же передать в функцию один объект, точнее, как его принять?

Вопрос первый: **по значению или по константной ссылки надо принять этот объект**? Во-первых, стоит понимать, что достаточно легкие объекты дешевле скопировать, чем принять по ссылке, а потом работать со ссылкой. Например, целые числа.

И здесь правило такое: **если размер объекта не больше 16 байт, то его дешевле скопировать**. Здесь правда есть ловушка, если у вас есть совершенно произвольная структура и ее размер пока что 16 байт, то пусть вас это не обманывает, возможно в ней потом появятся другие поля и размер станет больше чем 16 байт. Поэтому вы можете принимать по значению только те легкие объекты, размер которых именно такой по смыслу, и никогда не изменится. Например, целое число.

Еще один случай, когда **надо принимать по значению, если вы хотите этот объект, его данные скопировать, и как-то их менять**.

Давайте рассмотрим несколько примеров

```
vector<Query> ReadQueries(int query_count);
```

`string_view` тоже можно принимать по значению, потому что это всегда 16 байт.

```
bool CheckName(string_view name);
```

```
bool CheckBooking(const Booking& booking);
```

```
vector<int> BuildSorted(vector<int> numbers);
```

Что если ваша функция хочет принимать не все параметры всегда? Например,

```
vector<Query> ReadQueries(int query_count, ReadMode mode = ReadMode::Normal);
```

В этом случае нас спасает конечно значения по умолчанию — вы их прекрасно знаете.

Другой случай, если объект, который вы хотите сделать необязательным, принимался по константной ссылке, или просто по ссылке, и вы хотите иметь возможность эту ссылку не передавать, вот здесь такой интересный момент, что можно эту ссылку сделать указателем и принимать объект по указателю.

```
vector<Query> ReadQueries(int query_count, ReadSettings* settings = nullptr);
```

Понятно, что в современном C++, если не вспоминать о том, что указателем можно ходить по памяти, указатель отличается от ссылки только тем, что он может быть нулевым. Именно это свойство здесь можно использовать. То есть вы можете принять объект по указателю, а не по ссылке, и дать этому указателю значение по умолчанию `nullptr`, и дальше можно внутри функции это обработать. Если указатель нулевой, значит объект не передали.

Еще один важный момент: когда вы читаете сигнатуру каких-то функций, как правило в документации, вы можете видеть там двойные ссылки перед параметрами. Вы уже знаете, что например, как в конструкторе перемещения у вектора, это означает, что вот конкретно этот вариант этого конструктора принимает обязательно временный вектор, ну и дальше данные из него перемещает.

```
vector<T>::vector<T>(vector<T>&& other);
```

Но есть и другой интересный случай, когда вы можете увидеть два амперсанда в документации, например

```
template<typename M>
/*...*/ map<K, V>::insert_or_asiign(const K& k, M&& obj);
```

Пока просто запомните, что такая конструкция, когда у вас есть шаблонный тип и потом два амперсанда, означает, что эта функция может принять как временный объект, так и обычный, постоянный объект, и если там обычный объект, она не будет из него забирать данные, ну и не сможет, понятное дело, а если временный, то заберет.

Наконец, еще один важный нюанс, если речь идет о методах классов, то `this`, указатель на текущий объект по сути является неявным параметром метода, и нужно понимать, что если вы видите какой-то **метод, он принимает данные не только из своих параметров, но еще и из полей класса**, в частности из этого следует, что если у класса очень много полей, то они все, все эти поля являются по сути входными параметрами для всех методов, и если у параметров методов много, и у класса много полей, то получается перегруженность с большим количеством входных объектов.

Таким образом, передавать объект через глобальные переменные плохо, потому что непонятно и нетестируемо. Можно передать по назначению, если легко скопировать, то есть 16 байт или меньше, причем не только сейчас, но и всегда в будущем, можно передать по контактной ссылке.

Если вы хотите, чтобы можно было параметр не передавать, используйте значение по умолчанию.

Если вы хотите передавать в функцию ссылку или ничего, то есть сделать ссылку не обязательно, то сделать ее указателем, со значением по умолчанию `nullptr`.

И наконец, просто имейте в виду, что `this` это неявный параметр любого метода.

2.2 Как передать в функцию набор объектов

Давайте обсудим, как передать в функцию много объектов, если они имеют один тип. Общепринятый универсальный способ, в том числе в стандартной библиотеке C++, это передать функцию

два итератора. Например, алгоритм `sort` принимает два итератора, которые обозначают начало и конец диапазона объектов, которые надо сортировать.

```
template<typename RandomIt>
void sort(RandomIt begin, RandomIt end);
```

В чём же плюсы такого подхода? Конечно, универсальность, которую нам здесь дарют шаблоны функций. Вы можете передать итераторы произвольного типа и на элементы произвольного типа, что очень актуально для универсальных алгоритмов типа `sort`

Соответственно, универсальность — это плюс, а в чём же минус? Смотря на сигнатуру функции, вы не можете сходу предположить, особенно если там будут непонятные названия аргументов, например, вы не сможете предположить: функция принимает объекты произвольного или конкретного типа. Эта непрозрачность безусловно является минусом.

В каком же случае можно сделать себе послабления и вместо двух итераторов передать в функцию какой-то конкретный контейнер? Например, если вы от этого контейнера чего-то конкретного ожидаете, например, вы передайте функцию `unordered_set` только потому что вы хотите искать в этом наборе объектов за константное время. Или если вам не нужна такая универсальность, которую вам дают итераторы, если вы хотите максимальной понятности сигнатуры.

Что же делать с константностью? Если вам при вызове этой функции важно, чтобы она не меняла ваши объекты, передайте константные итераторы.

Ещё один интересный момент заключается в том, что вы можете в документации увидеть случаи, когда набор объектов передается вот в таком виде, вот, например,

```
basic_string(const CharT* s,
             size_type count,
             const Allocator& alloc = Allocator());
```

Если вы знакомы с языком C или просто проявляли некоторую любопытность в процессе предыдущих курсов, то вы знаете, что это способ передать набор символов, которые лежат в памяти подряд. В виде указателя на начало этого набора и в виде количества символов в этом наборе. Такое можно встретить не только в конструкторе строки, но и, например, в конструкторе `string_view`.

В чем же проблемы такой сигнатуры функции? В том что человек не особо близко знакомый с C++ или с C, откуда это вообще пришло, может не понять, указатель и `count` каким-то образом другом с другом связаны, что это вообще количество объектов, если отсчитывать от этого указателя столько-то элементов.

Вы уже привыкли, что если вы хотите вызвать какой-то алгоритм от набора элементов, как правило от контейнера, вы должны вызвать `begin`, должны вызвать `end` и вызвать, собственно, нужную вам функцию от этих двух итераторов. Конечно, хотелось бы как в других языках писать просто и компактно, например `sort` от контейнера, и такое скоро можно будет делать. А именно, в ближайшем стандарте C++ скорее всего появится так называемый модуль `Ranges`, который позволит вам не только вызывать алгоритмы от конкретного контейнера, не вызывая `begin` и `end`, но и даже передавать, например, в функцию `sort` ту функцию, которую надо применить к объектам перед их сравнением.

2.3 Как вернуть объект из функции

Настала пора обсудить, как же объекты из функции возвращать. Хотя постойте, мы же это обсуждали в первом курсе. Объекты из функции возвращаются через `return`. Ну то есть здесь ничему новому вы не научитесь...

Ладно, давайте договоримся, что вы будете использовать `return` и только `return`, а мы рассмотрим преимущества этого подхода.

Итак, `return` — это всем известный, максимально удобный и понятный способ вернуть данные из функции. Потому что

- Во-первых, прямо по сигнатуре функции видно, что она возвращает.
- Внутри функции видно выражение `return` что-то, функцию завершили, данные вернули.
- И наконец, специально для `return` придуманы оптимизации `copy elision`, `Named Return Value Optimization (NRVO)`, и, собственно, `move`-семантика, которую мы обсуждали в предыдущем курсе.

Именно поэтому вам стоит максимально использовать `return`.

Однако, конечно, здесь не без подводных камней, и есть ситуации, когда при возврате некоторых объектов из функции у вас копирование неизбежно. А именно, если у объекта много данных на стеке. Давайте рассмотрим такой пример.

```
...
using namespace std;

struct UserInfo {
    string Name;
    uint8_t age;
};

UserInfo ReadUserInfo(istream& in_stream) {
    UserInfo info;
    in_stream >> info.name;
    in_stream >> info.age;
    //...
    return Info;
}

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

Если вы из функции будете возвращать конкретную переменную, то все тоже будет в порядке, у вас сработает `NRVO`.

Но если вдруг у нас функция будет устроена по-другому

```
...
UserInfo ReadUserInfo(istream& in_stream) {
    /*UserInfo info;
    in_stream >> info.name;
    in_stream >> info.age;*/
    //...
    return ReadUser().info;
}
```

то есть будет возвращаться не локальная переменная, а какой-то другой, пусть даже временный объект, точнее, поле временного объекта, то в этом случае не сработает ни `copy ellision`, ни `NRVO`, а `move`-семантика нам не поможет, потому что у этой структуры со временем будет всё больше и больше данных на стеке. Что же делать в этой ситуации?

Мы могли бы сказать, что нужно отказаться от `return`, но нет. Потому что если вы имеете дело с такими объектами, объектами, у которых много данных на стеке, то у вас будет от них гораздо больше проблем, чем просто при возврате из функции. Например,

```
int main() {
    vector<UserInfo> users;
    sort(begin(users), end(users));
}
```

Выглядит хорошо, но на самом деле сортировка внутри будет переставлять элементы местами, а у этих элементов много данных на стеке, и поэтому это будет долго. Объекты, у которых много данных на стеке, опасны в любом случае. Вам все равно рано или поздно придется переписывать очень много кода, избавляясь от копирований этих объектов. Поэтому есть такой интересный совет по работе с такими объектами, а именно — оборачивать их в `unique_pointer`.

```
...
#include <memory>
...

using InfoPtr = unique_ptr<UserInfo>;

InfoPtr ReadUserInfo(istream& in_stream) {
    const auto info_ptr = make_unique<UserInfo>();
    UserInfo& info = *info_ptr;
    in_stream >> info.name;
    in_stream >> info.age;
    //...
    return info_ptr;
}

int main() {
    vector<InfoPtr> users;
    sort(begin(users), end(users));
}
```

Преимущество этого подхода в том, что мы отселяем объект, у которого много данных на стеке, в кучу. И, благодаря этому, у нас всё становится хорошо. Объекты хранятся в одном конкретном месте и никуда оттуда не перемещаются, только если мы явно не захотим как-то их скопировать.

И теперь у нас и сортировка будет в порядке, потому что будем переставлять указатели на кучу, и вот такие сценарии вида: когда мы хотим оставить каких-то пользователей из диапазона, будут максимально простыми.

2.4 Как вернуть несколько объектов из функции

Что же делать, если вы хотите вернуть из функции несколько объектов, возможно разных типов? На самом деле мы это уже обсуждали в начале второго курса, и на эту тему даже была задача «Трекер задач». В этой задаче нужно было реализовать, в том числе, среди прочих, метод `PerformPersonTasks`, который возвращает два словаря задач. Это мы делали с помощью кортежа.

```
tuple<TaskInfo, TaskInfo> TeamTasks::PerfomPersonTasks(...) {
    ...
    return {move(updated_tasks), move(untouched_tasks)};
}
```

Почему это удобно? Потому что мы могли при вызове этой функцией использовать structured bindings:

```
auto [updated_tasks, untouched_tasks] = tasks.PerformPersonTasks(...);
```

Именно structured bindings позволяют нам здесь использовать возврат с помощью кортежа из нескольких объектов, причем, что ещё хорошо, что даже до structured bindings, если у вас эти переменные уже есть, они как-то объявлены, даже когда не было structured bindings мы могли использовать функцию `tie`:

```
tie(updated_tasks, untouched_tasks) = tasks.PerformPersonTasks(...);
```

Но как правило, конечно, вы используете structured bindings, потому что этих переменных у вас ещё нет.

Это все хорошо и замечательно, но давайте посмотрим на этот код с точки зрения его понятности. Если вы возвращаете из функции наборы объектов совершенно разных типов, и по ним совершенно понятно, какой из них что означает, например:

```
pair<int, CurrencyType> ComputeCost(...);
```

Вот к такой сигнатуре функции вопросов не возникает.

Если же у нас `PerformPersonTasks` возвращает 2 `TaskInfo`, то в принципе, из контекста, не очень понятно, чем они вообще отличаются. И для этих случаев удобно вместо пар и кортежей с непонятными названиями полей, использовать структуры.

```
struct PerformResult {
    TaskInfo updated_tasks;
    TaskInfo untoucned_tasks;
};
```

```
PerformResult TeamTasks::PerfomPersonTasks(...) {
    ...
    return {move(updated_tasks), move(untouched_tasks)};
}
```

У нас никак не меняется `return`-выражение, и, что самое приятное в structured binding, что они умеют распаковывать и структуру тоже.

Хорошо, что ещё можно сказать про возврат нескольких объектов из функции? Не боитесь ли вы, что в `return` выражениях у вас случатся копирования? Конечно, в `return` всё будет хорошо, но здесь есть маленький нюанс, который был виден в коде: если при возврате одной переменной из функций вас спасал `NRVO`, то когда вы возвращаете набор переменных, объединяя их в фигурные скобки, чтобы у вас получилась пара или кортеж, никакого `NRVO` не будет, и вам надо явно вызвать `move` от этих переменных при передаче в конструктор пары или кортежа. Это не очень удобно, но плюсы возврата через `return` и затем связки с помощью structured bindings этот минус перевешивают.

2.5 Возврат данных через исключения

Некоторые из вас могли бы задаться вопросом, почему бы для возврата из функции некоторой дополнительной информации просто не использовать исключения? Что ж, давайте мы на этот вопрос ответим, но с точки зрения производительности.

```
#include "profile.h"
#include <variant>
#include <vector>

using namespace std;

enum class FailReason {
    Bad, Invalid, Ugly, Buggy, Wrong
};

variant<int, FailReason> ComputeCostVariant(int i) {
    if (i % 10 > 0) {
        return i * 100;
    } else {
        return static_cast<FailReason>(i / 10 % 5);
    }
}

int ComputeCostThrow(int i) {
    if (i % 10 > 0) {
        return i * 100;
    } else {
        throw static_cast<FailReason>(i / 10 % 5);
```

```
    }
}

const int ITER_COUNT = 1000000;

int main() {
    vector<int> stats(6);

    {
        LOG_DURATION("variant");
        for (int i = 0; i < ITER_COUNT; ++i) {
            if (const auto res = ComputeCostVariant(i); holds_alternative<FailReason>(res)) {
                ++stats[static_cast<size_t>(get<FailReason>(res))];
            } else {
                stats.back() += get<int>(res);
            }
        }
    }

    {
        LOG_DURATION("throw");
        for (int i = 0; i < ITER_COUNT; ++i) {
            try {
                stats.back() += ComputeCostThrow(i);
            } catch(const FailReason reason) {
                ++stats[static_cast<size_t>(reason)];
            }
        }
    }
    return 0;
}
```

Код скомпилировался, запустился, и мы видим, что версия с `variant` работает быстрее, чем версия с исключениями. То есть версия с исключениями чуточку лаконичнее, но при этом дольше. Какой из этого можно сделать вывод?

Вывод очень простой: **исключения в ваших программах нужны только для действительно исключительных, неожиданных ситуациях или просто не в «горячих» с точки зрения производительности местах**. Когда вы хотите где-то использовать исключения, вспомните о том, не стоит ли вам подумать о производительности и использовать `variant` или `optional`.

Вызовы конструкторов

3.1 Понятность вызовов конструкторов

Давайте обсудим, как же делать вызовы конструкторов более понятными. Вернёмся к задаче «Электронная книга». Давайте представим, что `MAX_USER_COUNT` перестало быть константой и теперь стало параметром конструктора, а также количество страниц у нас тоже как-то ограничено, и это ограничение нам тоже приходит в конструкторе. И давайте представим, что у нас есть некоторый параметр, который как-то меняет рейтинг, как-то на него влияет.

```
class ReadingManager {
public:
    ReadingManager(int max_user_count,
                   int max_page_count,
                   double cheer_factor) : user_page_counts_(max_user_count + 1, -1),
                       page_achieved_by_count_(max_page_count + 1, 0) {}

    void Read(int user_id, int page_count) {
        UpdatePageRange(user_page_counts_[user_id] + 1, page_count + 1);
        user_page_counts_[user_id] = page_count;
    }

    double Cheer(int user_id) const {
        const int pages_count = user_page_counts_[user_id];
        if (pages_count == -1) {
            return 0;
        }
        const int user_count = GetUserCount();
        if (user_count == 1) {
            return 1;
        }
        return (user_count - page_achieved_by_count_[pages_count]) * 1.0
            / (user_count - 1);
    }
private:
    vector<int> user_page_counts_;

    vector<int> page_achieved_by_count_;
```

```
    int GetUserCount() const {
        return page_achieved_by_count_[0];
    }

    void UpdatePageRange(int lhs, int rhs) {
        for (int i = lhs; i < rhs; ++i) {
            ++page_achieved_by_count_[i];
        }
    }
};
```

Как нам создать объект такого класса?

```
ReadingManager manager(20000, 1000, 2);
```

Здравствуй криптографичное создание класса, ничего не понятно — 20000, 1000 и 2. А если поменять их местами, изменится что-то или нет? В общем, непонятно.

Как эту проблему решать? Самый простой способ — это комментарии.

```
ReadingManager manager(/*max_user_count*/ 20000,
                       /*max_page_count*/ 1000,
                       /*cheer_factor*/ 2);
```

На самом деле, уже понятно, что нам не хватает именованных аргументов функции, которые есть в других языках, а в C++ пока, к сожалению, нет, поэтому приходится как-то выкручиваться. И комментарии — это один из способов сделать такой вызов конструктора более понятным. Но и тут есть нюансы. Код стал понятнее, но проблемы остаются.

3.2 Как рефакторить конструкторы с непонятными сигнатурами

Продолжим обсуждать конструкцию со сложными сигнатурами. Мы уже рассмотрели способ комментирования отдельных аргументов конструктора. Давайте научимся это делать более безопасно и прозрачно для компилятора. А именно, если нам не хватало именованных полей, именованных аргументов функций, то придется обходиться существующими средствами языка, и как вариант, можно завести дополнительные методы в этом классе, которые будут устанавливать нужные нам параметры.

```
class ReadingManager {
public:
    ReadingManager() {}

    void SetMaxUserCount(int max_user_count) {
        user_page_counts_.assign(max_user_count + 1, 0);
        user_positions_.assign(max_user_count + 1, 0);
    }
```

```
    void SetMaxPageCount(int max_page_count) {}

    void SetCheerFactor(double cheer_factor) {}
    ...
};
...

int main() {
    ReadingManager manager;
    manager.SetMaxUserCount(20000);
    manager.SetMaxPageCount(1000);
    manager.SetCheerFactor(2);
}
```

Однако есть нюансы: если мы допускаем, что наши старые методы `Read` и `Cheer` могут быть вызваны в том числе и до `Set`-методов, то везде придется добавлять проверки следующего вида

```
class ReadingManager {
public:
    ...
    void Read(int user_id, int page_count) {
        if (max_user_count_ <= 0) {
            // ...
        }
        ...
    }

    double Cheer(int user_id) {
        if (max_user_count_ <= 0) {
            // ...
        }
        ...
    }
    ...
private:
    int max_user_count_ = 0;
    ...
};
```

Выглядит не очень удобно, поэтому давайте думать, что же нам делать с этим классом. Получается, что у его жизненного цикла сейчас есть две явные стадии, которые хорошо видно на примере функции `main`. Мы сначала этот класс инициализируем, с помощью `Set`-методов, а затем используем, и эти стадии хотелось бы явно как-то разграничить, и способ для этого есть: давайте `Set`-методы выделим в отдельный класс. Этот класс мы назовем `Builder`-классом.

```
class ReadingManagerBuilder {
public:
    void SetMaxUserCount(int max_user_count) {
```

```
        max_user_count_ = max_user_count;
    }

    void SetMaxPageCount(int max_page_count) {
        max_page_count_ = max_page_count;
    }

    void SetCheerFactor(double cheer_factor) {
        cheer_factor_ = cheer_factor;
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

Также нам нужно будет как-то обозначить базовый переход из `Builder`-класса в наш класс, в `ReadingManager`, для этого мы напишем метод `Build`, который будет возвращать `ReadingManager`, принимать ничего и будет константным, и в этом методе мы создадим как раз объект `ReadingManager`.

```
class ReadingManager {
public:
    ReadingManager(int max_user_count, int max_page_count, double cheer_factor) :
        user_page_counts_(max_user_count + 1, 0) {

    }

};

class ReadingManagerBuilder {
public:
    ...
    ReadingManager Build() const {
        return {max_user_count_, max_page_count_, cheer_factor_};
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

Получается, что класс нужно создавать теперь следующим образом

```
int main() {
    ReadingManagerBuilder builder;
    builder.SetMaxUserCount(20000);
```

```
    builder.SetMaxPageCount(1000);
    builder.SetCheerFactor(2);
    ReadingManager manager = builder.Build();
}
```

Однако, возникает резонный вопрос, казалось бы нас огорчил вот такой конструктор, с кучей непонятных полей, его всё еще можно вызвать непонятно и печалиться. С этим надо что-то сделать. И еще проверки у нас остались в методах. Проверки можно унести в метод `Build` — и это первое очевидное преимущество

```
    ...
    ReadingManager Build() const {
        if (max_user_count_ <= 0) {
            // ...
        }
        return {max_user_count_, max_page_count_, cheer_factor_};
    }
```

Что же делать с конструктором? Раз уж он такой непонятный и у нас теперь есть способ этот конструктор не вызывать, его можно сделать приватным, то есть не дать его вызывать снаружи.

```
class ReadingManager {
public:
    ...

private:
    ReadingManager(int max_user_count, int max_page_count, double cheer_factor) :
        user_page_counts_(max_user_count + 1, 0) {}
    ...
};
```

Код не компилируется, потому что компилятор ругается на приватный конструктор. Действительно, конструктор приватный, поэтому его вызвать в методе `Build` нельзя. Что же в этом случае делать? Как спрятать тот конструктор от внешних пользователей, но разрешить его вызывать классу `ReadingManagerBuilder`.

Для этого этим двум классам нужно подружиться, а именно нужно в классе `ReadingManager` добавить `friend`**-декларацию**, то есть **класс** `ReadingManager` **будет разрешать методам класса** `ReadingManagerBuilder` **обращаться к своим приватным методам**.

```
class ReadingManager {
public:
    friend class ReadingManagerBuilder;
    ...
```

Попробуем снова скомпилировать код. И он успешно скомпилировался.

Запись в функции `main` можно сделать проще, потому что здесь слишком много раз употребляется этот самый `Builder`, поэтому давайте сделаем такую интересную вещь: мы будем из всех этих `Set`-методов возвращать не `void`, а ссылку на текущий объект `Builder`, неконстантную ссылку.

```
class ReadingManagerBuilder {
public:
    ReadingManagerBuilder& SetMaxUserCount(int max_user_count) {
        max_user_count_ = max_user_count;
        return *this;
    }

    ReadingManagerBuilder& SetMaxPageCount(int max_page_count) {
        max_page_count_ = max_page_count;
        return *this;
    }

    ReadingManagerBuilder& SetCheerFactor(double cheer_factor) {
        cheer_factor_ = cheer_factor;
        return *this;
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

И благодаря этому можно объединять вызовы `Set`-методов в цепочки, не указывая несколько раз название переменной.

```
int main() {
    ReadingManager manager = ReadingManagerBuilder()
        .SetMaxUserCount(20000)
        .SetMaxPageCount(1000)
        .SetCheerFactor(2)
        .Build();
}
```

Проверим, что код компилируется. Код действительно, компилируется.

Таким образом, по сути мы рассмотрели некоторую вариацию **паттерна проектирования** `Builder`, который позволяет нам разграничить инициализацию класса и его последующее использование.

Итак, когда у конструктора, да и вообще у любой функции много параметров, во-первых, рассеивается внимание, во-вторых, очень легко их местами перепутать, поэтому рекомендуется в случае, когда у вас действительно много параметров и это важно для вашего проекта (конечно не во всех случаях) использовать паттерн `Builder`, который позволяет вам, создавая объект типа `Builder` и вызывая у него `Set`-методы и затем метод `Build`, более понятно и прозрачно создавать объекты нужного вам типа.