

## 1.4. Условный оператор и циклы

### 1.4.1. Условный оператор if

Условный оператор if позволяет указать операции, которые должны выполняться при соблюдении некоторого условия, либо не выполняться, если это условие неверно.

Синтаксис оператора if в простейшем случае имеет вид:

```
if ( <условие> )  
    <команда если верно>
```

Пусть пользователь вводит с консоли два числа, которые потом сравниваются между собой.

```
int a, b;  
  
cin >> a >> b;  
  
if (a == b)  
    cout << "equal";
```

Если ввести два одинаковых числа, программа выводит «equal», иначе — ничего не выводит.

Оператор else позволяет указать утверждение, которое будет выполнено в случае, если условие не верно. Оператор else всегда идет в паре с оператором if и имеет следующий синтаксис:

```
if ( <условие> )  
    <команда если верно>  
else  
    <команда если неверно>
```

В результате, программу можно дополнить следующим образом.

```
int a, b;  
  
cin >> a >> b;  
  
if (a == b)  
    cout << "equal" << endl;  
else  
    cout << "not equal" << endl;
```

Если ввести два одинаковых числа, программа выводит «equal», иначе — «not equal».

Если необходимо выполнить больше одной операции при выполнении условия, нужно использовать фигурные скобки:

```
if ( <условие> ) {  
    ...  
}
```

Например, можно вывести значения чисел: оба значения, если числа различны, и одно, если совпадают.

```
int a, b;  
  
cin >> a >> b;  
  
if (a == b) {  
    cout << "equal" << endl;  
    cout << a;  
}  
else {  
    cout << "not equal" << endl;  
    cout << a << " " << b;  
}
```

Здесь endl (end of line) — оператор, который делает перенос строки.

При работе с оператором if следует иметь в виду следующую особенность. Пусть дан такой код:

```
int a = -1;  
  
if (a >= 0)  
    if (a > 0)  
        cout << "positive";  
else  
    cout << "negative";
```

Из-за отступов могло показаться, что оператор else относится к внешнему if, а на самом деле в такой записи он относится к внутреннему if. В C++, в отличие от Python, отступы не определяют вложенность. В итоге программа ничего не выводила в консоль.

Если явно расставить скобки, получится:

```
int a = -1;  
  
if (a >= 0) {  
    if (a > 0)  
        cout << "positive";  
}  
else {
```

```
    cout << "negative";  
}
```

В данном случае, как и ожидается, выведено «negative».

Из последнего примера можно сделать вывод, что следует всегда явно расставлять фигурные скобки, даже если выполнить необходимо всего одну команду.

## 1.5. Цикл while

Цикл while может быть полезен, если необходимо выполнять некоторые условия много раз, пока истинно некоторое условие.

```
while ( <условие> )  
    <команда>
```

Пусть пользователь вводит число n. Требуется подсчитать сумму чисел от 1 до n.

```
int n = 5;  
int sum = 0;  
int i = 1;  
while (i <= n) {  
    sum += i;  
    i += 1;  
}  
cout << sum;
```

Аналогом цикла while является так называемый цикл do-while, который имеет следующий синтаксис:

```
do {  
    <команда>  
} while ( <условие> );
```

Следующая программа является интерактивной игрой, в которой пользователь пытается угадать загаданное число.

```
int a = 5;  
int b;  
  
do {  
    cout << "Guess the number: ";  
    cin >> b;  
} while (a != b);  
  
cout << "You are right!";
```

## 1.6. Цикл for

Цикл for используется для перебора набора значений. В качестве набора значений можно использовать некоторые типы контейнеров:

```
vector    vector<int> a = {1, 4, 6, 8, 10};
```

```
    int sum = 0;
    for (auto i : a) {
        sum += i;
    }
```

```
    cout << sum;
```

```
map        map<string, int> b = {{"a", 1}, {"b", 2}, {"c", 3}};
```

```
    int sum = 0;
    string concat;
    for (auto i : b) {
        concat += i.first;
        sum += i.second;
    }
```

```
    cout << concat << endl;
    cout << sum;
```

```
string     string a = "asdfasdfasdf";
```

```
    int i = 0;
    for (auto c : a) {
        if (c == 'a') {
            cout << i << endl;
        }
        ++i;
    }
```

Простой цикл for позволяет создавать цикл с индексом:

```
    string a = "asdfasdfasdf";
```

```
    for (int i = 0; i < a.size(); ++i) {
        if (a[i] == 'a') {
            cout << i << endl;
        }
    }
```

С помощью оператора **break** можно прервать выполнение цикла:

```
string a = "sdfasdfasdf";

for (int i = 0; i < a.size(); ++i) {
    if (a[i] == 'a') {
        cout << i << endl;
        break;
    }
}

cout << "Yes";
```

3

Yes

# Неделя 2

## Функции и контейнеры

### 2.1. Функции

#### 2.1.1. Объявление функции. Возвращаемое значение.

Прежде код программы записывался внутри функции `main`. В данном уроке будет рассмотрено, как определять функции в C++. Для начала, рассмотрим преимущества разбиения кода на функции:

- Программу, код которой разбит на функции, проще понять.
- Правильно выбранное название функции помогает понять ее назначение без необходимости читать ее код.
- Выделение кода в функцию позволяет его повторное использование, что ускоряет написание программ.
- Функции — это единственный способ реализовать рекурсивные алгоритмы.

Теперь можно приступить к написанию первой функции. Объявление функции содержит:

- Тип возвращаемого функцией значения.
- Имя функции.
- Параметры функции. Перечисляются через запятую в круглых скобках после имени функции. Для каждого параметра нужно указать не только его имя, но и тип.
- Тело функции. Расположено в фигурных скобках. Может содержать любые команды языка C++. Для возврата значения из функции используется оператор `return`, который также завершает выполнение функции.

Например, так выглядит функция, которая возвращает сумму двух своих аргументов:

```
int Sum(int x, int y) {  
    return x + y;  
}
```

Чтобы воспользоваться функцией, достаточно вызвать ее, передав требуемые параметры:

```
int x, y;  
cin >> x >> y;  
cout << Sum(x, y);
```

Данный код считывает два значения из консоли и выведет их сумму.

Важной особенностью оператора `return` является то, что он завершает выполнение функции. Рассмотрим функцию, проверяющую, входит ли слово в некоторый набор слов:

```
bool Contains(vector <string> words, string w) {  
    for (auto s : words) {  
        if (s == w) {  
            return true;  
        }  
    }  
    return false;  
}
```

Функция принимает набор строк и строку `w`, для которой надо проверить, входит ли она в заданный набор. Возвращаемое значение — логическое значение (входит или не входит), имеет тип `bool`.

Результат работы функции для нескольких случаев:

```
cout << Contains({"air", "water", "fire"}, "fire"); // 1  
cout << Contains({"air", "water", "fire"}, "milk"); // 0  
cout << Contains({"air", "water", "fire"}, "water"); // 1
```

Функция работает так, как ожидалось. Стоит отметить, что в C++ значения логического типа выводятся как 0 и 1. Чтобы лучше понять, как выполнялась функция, запустим отладчик.

Далее представлен код программы с указанием номеров строк и результат пошагового исполнения в виде таблицы. Первый столбец — номер шага, второй — строка, которая исполняется на данном шаге, а третий — значение переменной `s`, определенной внутри функции.

```

1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  using namespace std;
6
7  bool Contains(vector<string> words, string w)
8  {
9      for (auto s : words) {
10         if (s == w) {
11             return true;
12         }
13     }
14     return false;
15 }
16
17 int main() {
18     cout << Contains({"air", "water", "fire"},
19                      "water") << endl; // 1
20     return 0;
21 }

```

№	LN	string s
0	16	-
1	17	-
2	7	-
3	8	air
4	9	air
5	8	water
6	9	water
7	10	water
8	17	-
9	18	-

Видно, что программа в цикле успевает перебрать только первые два значения из вектора, после чего возвращает `true` и выполнение функции прекращается. Этот пример демонстрирует то, что оператор `return` завершает выполнение функции.

## Ключевое слово `void`

Рассмотрим функцию, которая выводит на экран некоторый набор слов, который был передан в качестве параметра.

```

??? PrintWords(vector<string> words) {
    for (auto w : words) {
        cout << w << " ";
    }
} /* */

```

Остается вопрос: что следует написать в качестве типа возвращаемого значения. Функция по своей сути ничего не возвращает и не понятно, какой тип она должна возвращать.

В случаях, когда функция не возвращает никакого значения, в качестве возвращаемого типа используется ключевое слово `void` (*англ.* пустой). Таким образом, код функции будет следующим:



```
void PrintWords(vector<string> words) {  
    for (auto w : words) {  
        cout << w << " ";  
    }  
}
```

После того, как функция была определена, ее можно вызвать:

```
PrintWords({"air", "water", "fire"})
```

### 2.1.2. Передача параметров по значению

Рассмотрим следующую функцию, которая была написана, чтобы устанавливать значение 42 передаваемому ей аргументу:

```
void ChangeInt(int x) {  
    x = 42;  
}
```

В функции `main` эта функция вызывается с переменной `a` в качестве параметра, значение которой до этого было равно 5.

```
int a = 5;  
ChangeInt(a);  
cout << a;
```

После вызова функции `ChangeInt` значение переменной `a` выводится на экран. Вопрос: что будет выведено на экран, 5 или 42?

Верный способ проверить — запустить программу и посмотреть. Запустив ее, можно убедиться, что программа выводит «5», то есть значение переменной `a` внутри `main` не поменялось в результате вызова функции `ChangeInt`.

Этот пример призван продемонстрировать то, что параметры функции передаются по значению. Другими словами, в функцию передаются копии значений, переданные ей во время вызова.

Посмотрим на то, как это происходит с помощью пошагового выполнения.

```
1  #include <iostream>
2  using namespace std;
3
4  void ChangeInt(int x) {
5      x = 42;
6  }
7
8  int main() {
9      int a = 5;
10     ChangeInt(a);
11     cout << a;
12     return 0;
13 }
```

№	LN	int a	int x
0	9	-	-
1	10	5	-
2	4	5	5
3	5	5	42
4	11	5	-

Видно, что меняется значение переменной внутри функции `ChangeInt`, а значение переменной в `main` остается тем же.

### 2.1.3. Передача параметров по ссылке

Поскольку параметры функции передаются по значению, изменение локальных формальных параметров функции не приводит к изменению фактических параметров. Объекты, которые были переданы функции на месте ее вызова, останутся неизменными. Но что делать в случае, если функция по смыслу должна поменять объекты, которые в нее передали.

Допустим, нужно написать функцию, которая обменивает значения двух переменных. Проверим, подходит ли такая функция для этого:

```
void Swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Определим две переменные, `a` и `b`:

```
int a = 1;
int b = 2;
```

От правильно работающей функции ожидается, что значения переменных поменяются местами, а именно переменная `a` будет равна двум, а `b` — одному. Применим функцию `Swap`.

```
Swap(a, b);
```

Выведем на экран значения переменных:

```
cout << "a == " << a << '\n'; // 1
cout << "b == " << b << '\n'; // 2
```

Мы видим, что значения переменных не изменились. Действительно, поскольку параметры функции при вызове были скопированы, изменение переменных `x` и `y` никак не привело к изменению переменных внутри функции `main`.

Чтобы реализовать функцию `Swap` правильно, параметры `x` и `y` нужно передавать по ссылке. Это соответствует тому, что в качестве типа параметров нужно указывать не `int`, а `int&`. После исправления функция принимает вид:

```
void Swap(int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Запустив программу снова, можно убедиться, что функция отработала так, как и ожидалось.

Таким образом, для модификации передаваемых в качестве параметров объектов, их нужно передавать не по значению, а по ссылке. Ссылка — это особый тип языка C++, является синонимом переданного в качестве параметра объекта. Ссылка оформляется с помощью знака `&` после типа передаваемой переменной.

Можно привести еще один пример, в котором оказывается полезным передача параметров функции по ссылке. Уже говорилось, что в библиотеке `algorithm` существует функция сортировки. Например, отсортировать вектор из целых чисел можно так:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};
sort(begin(nums), end(nums));
```

Чтобы проверить, что все работает, также выведем элементы вектора на экран:

```
for (auto x : nums) {
    cout << x << " ";
}
```

Запускаем программу. Программа выводит: «0 1 2 2 3 6», то есть вектор отсортировался.

Однако, у данного способа есть недостаток: при вызове `sort` дважды указывается имя вектора, что увеличивает вероятность ошибки из-за невнимательности при написании кода. В результате опечатки программа может не скомпилироваться или, что гораздо хуже, работать неправильно. Поэтому хотелось бы написать такую функцию сортировки, при вызове которой имя вектора нужно указывать лишь раз.

Без использования ссылок такая функция выглядела бы примерно так:

```
vector<int> Sort(vector<int> v) {  
    sort(begin(v), end(v));  
    return v;  
}
```

Она принимает в качестве параметра вектор из целых чисел и возвращает также вектор целых чисел, а внутри выполняет вызов функции `sort`. Запустим программу:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};  
nums = Sort(nums);  
for (auto x : nums) {  
    cout << x << " ";  
}
```

Убеждаемся, что программа дает тот же результат. Но мы не избавились от дублирования: мало того, что в месте вызова мы также указываем имя вектора дважды, так и в определении функции `Sort` тип вектора указывается также два раза.

Перепишем функцию, используя передачу параметра по ссылке:

```
void Sort(vector<int>& v) {  
    sort(begin(v), end(v));  
}
```

Такая функция уже ничего не возвращает, а изменяет переданный ей в качестве параметра объект. Поэтому при ее вызове указывать имя вектора нужно один раз:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};  
Sort(nums);  
for (auto x : nums) {  
    cout << x << " ";  
}
```

Именно это и хотелось получить.

## 2.1.4. Передача параметров по константной ссылке

Раньше было показано, как в C++ можно создавать свои типы данных. А именно была определена структура Person:

```
struct Person {  
    string name;  
    string surname;  
    int age;  
};
```

Допустим, что была проведена перепись Москвы и вектор из Person, который содержит в себе данные про всех жителей Москвы, можно получить с помощью функции GetMoscowPopulation:

```
vector<Person> GetMoscowPopulation();
```

Здесь специально не приводится тело этой функции, которое может быть устроено очень сложно, отправлять запросы к базам данных и так далее. Вызвать эту функцию можно так:

```
vector<Person> moscow_population = GetMoscowPopulation();
```

Требуется написать функцию, которая выводит на экран количество людей, живущих в Москве. Эта функция ничего не возвращает, принимает в качестве параметра вектор людей и выводит красивое сообщение:

```
void PrintPopulationSize(vector<Person> p) {  
    cout << "There are " << p.size() <<  
        " people in Moscow" << endl;  
}
```

Воспользуемся этой функцией:

```
vector<Person> moscow_population = GetMoscowPopulation();  
PrintPopulationSize(moscow_population);
```

Программа вывела, что в Москве 12500000 людей: «There are 12500000 people in Moscow».

Замерим время выполнение функции GetMoscowPopulation и функции PrintPopulationSize. Подключим специальную библиотеку для работы с промежутками времени, которая называется chrono:

```
#include <chrono>  
#include <iostream>  
#include <vector>  
#include <string>  
  
using namespace std;  
using namespace std::chrono;
```

После этого до и после места вызова каждой из интересующих функций получим текущее значение времени, а затем выведем на экран разницу:

```
auto start = steady_clock::now();
vector<Person> moscow_population = GetMoscowPopulation();
auto finish = steady_clock::now();
cout << "GetMoscowPopulation "
    << duration_cast<milliseconds>(finish - start).count()
    << " ms" << endl;

start = steady_clock::now();
PrintPopulationSize(moscow_population);
finish = steady_clock::now();
cout << "PrintPopulationSize "
    << duration_cast<milliseconds>(finish - start).count()
    << " ms" << endl;
```

В результате получаем:

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 1034 ms
```

Получается, что функция, которая возвращает вектор из 12 миллионов строк, работает быстрее функции, которая всего-то печатает размер этого вектора. Функция `PrintPopulationSize` ничего больше не делает, но работает дольше.

Но мы уже говорили, что при передаче параметров в функции происходит полное глубокое копирование передаваемых переменных, в данном случае — вектора из 12 500 000 элементов. Фактически, чтобы вывести на экран размер вектора, мы тратим целую секунду на его полное копирование. С этим нужно как-то бороться.

Избежать копирования можно с помощью передачи параметров по ссылке:

```
void PrintPopulationSize(vector<Person>& p) {
    cout << "There are " << p.size() <<
        " people in Moscow" << endl;
}
```

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 0 ms
```

Теперь все работает хорошо, но у данного способа есть несколько недостатков:

- Передача параметра по ссылке — способ изменить переданный объект. Но в данном случае функция не меняет объект, а просто печатает его размер. Объявление этой функции

```
void PrintPopulationSize(vector<Person>& p)
```

может сбивать с толку. Может создаться впечатление, что функция как-то меняет свой аргумент.

- В случае, если промежуточная переменная не создается:

```
PrintPopulationSize(GetMoscowPopulation());)
```

программа даже не скомпилируется. Дело в том, что в C++ результат вызова функции не может быть передан по ссылке в другую функцию (почему это так будет сказано позже в курсе).

Получается, что при передаче по значению, мы вынуждены мириться с глубоким копированием всего вектора при каждом вызове функции печати размера, а при передаче по ссылке — мириться с вышеназванными двумя проблемами. Существует ли идеальное решение без всех этих недостатков?

Выход заключается в использовании передачи параметров по так называемой константной ссылке. Это делается с помощью ключевого слова **const**, которое добавляется слева от типа параметра. Символ & остается на месте и указывает, что происходит передача по ссылке.

Определение функции принимает вид:

```
void PrintPopulationSize(const vector<Person>& p) {  
    cout << "There are " << p.size() <<  
        " people in Moscow" << endl;  
}
```

В результате PrintPopulationSize выполняется за 0 мс, а также работает передача результата вызова функции в качестве параметра другой функции по константной ссылке:

```
PrintPopulationSize(GetMoscowPopulation());)
```

Также мы не вводим в заблуждение пользователей нашей функции и явно указываем, что параметр не будет изменен, так как он передается по константной ссылке.

Такой подход также защищает от случайного изменения фактических параметров функций. Допустим, по ошибке в функцию печати количества

людей в Москве попал код, добавляющий туда одного жителя Санкт-Петербурга.

```
void PrintPopulationSize(const vector<Person>& p) {
    cout << "There are " << p.size() <<
        " people in Moscow" << endl;
    p.push_back({"Vladimir", "Petrov", 40});
}
```

В случае передачи по ссылке такая ошибка могла бы остаться незамеченной, но при передаче по константной ссылке такая программа даже не скомпилируется:

```
main.cpp: In function 'void PrintPopulationSize(const std::vector<
    Person>&)':
main.cpp:20:41: error: passing 'const std::vector<Person>' as 'this
    ' argument discards qualifiers [-fpermissive]
    p.push_back({"Vladimir", "Petrov", 40});
                                ^
```

Компилятор в таком случае выдает ошибку, так как нельзя изменять принятые по константной ссылке фактические параметры.

## 2.2. Модификатор `const` как защита от случайных изменений

На самом деле `const` — специальный модификатор типа переменной, запрещающий изменение данных, содержащихся в ней.

Например, рассмотрим следующий код:

```
int x = 5;
x = 6;
x += 4;
cout << x;
```

В этом коде переменная `x` изменяется в двух местах. Как несложно убедиться, в результате будет выведено «10». Добавим ключевое слово `const`, не меняя ничего более:

```
const int x = 5;
x = 6;
x += 4;
cout << x;
```

При попытке скомпилировать этот код, компилятор выдает следующие сообщения об ошибках:



```
main.cpp: In function 'int main()':
main.cpp:9:7: error: assignment of read-only variable 'x'
    x = 6;
    ^
main.cpp:10:8: error: assignment of read-only variable 'x'
    x += 4;
    ^
```

Обе строчки, в которых переменная подвергается изменениям, приводят к ошибкам. Закомментируем их:

```
const int x = 5;
//x = 6;
//x += 4;
cout << x;
```

Теперь программа компилируется как надо и выводит «5». Чтение переменной является немодифицирующей операцией и не вызывает ошибок при компиляции.

Рассмотрим пример со строковой переменной `s`:

```
string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s << endl;
```

Здесь представлены операции: получение длины строки, добавление текста в конец строки, инициализация другой строки значением `s+'!'`, вывод значения строки в консоль. Запускаем программу:

```
5
hello, world
hello, world!
```

Теперь добавляем модификатор `const`.

```
const string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s;
```

При компиляции только в одном месте выводится ошибка:

```
basic_string.h:1131:7: note:   in call to 'std::__cxx11::
basic_string<_CharT, _Traits, _Alloc>& std::__cxx11::
basic_string<_CharT, _Traits, _Alloc>::operator+=(const _CharT*)
[with _CharT = char; _Traits = std::char_traits<char>; _Alloc =
std::allocator<char>]'
    operator+=(const _CharT* __s)
    ~~~~~~
```

Вывод длины строки, использование строки при инициализации другой строки и вывод строки в консоль — немодифицирующие операции и ошибок не вызывают. А вот добавление в конец строки еще как-либо текста — уже нет. Закомментируем соответствующую строку:

```
const string s = "hello";
cout << s.size() << endl;
//s += ", world";
string t = s + "!";
cout << s;
```

```
5
hello
hello!
```

Более сложный пример: рассмотрим вектор строк и попытаемся изменить первую букву первого слова этого вектора с прописной на заглавную:

```
vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

Программа успешно компилируется и выводит «Hello» как и ожидалось. Установим модификатор `const`.

```
const vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

В итоге компиляция завершается ошибкой:

```
main.cpp:9:13: error: assignment of read-only location '(& w.std::
vector<std::__cxx11::basic_string<char> >::operator [] (0)) ->std::
__cxx11::basic_string<char>::operator [] (0) '
w[0][0] = 'H';
```

Здесь важно отметить следующее: мы не меняем вектор непосредственно (не добавляем элементы, не меняем его размер), а модифицируем только его элемент. Но в C++ модификатор `const` распространяется и на элементы контейнеров, что и демонстрируется в данном примере.

## Зачем вообще в C++ нужен модификатор `const`?

Главное предназначение модификатора `const` — помочь программисту не допускать ошибок, связанных с ненамеренными модификациями переменных. Мы можем пометить ключевым словом `const` те переменные, которые не хотим изменять, и компилятор выдаст ошибку в том месте, где происходит ее изменение. Это позволяет экономить время при написании кода, так как избавляет от мучительных часов отладки.

## 2.3. Контейнеры

### 2.3.1. Контейнер vector

Тип `vector` представляет собой набор элементов одного типа. Тип элементов вектора указывается в угловых скобках. Классический сценарий использования вектора — сохранение последовательности элементов.

#### Создание вектора требуемой длины. Ввод и вывод с консоли

Напишем программу, которая считывает из консоли последовательность строк, например, имен лекторов. Сначала на вход подается число элементов последовательности:

```
int n;  
cin >> n;
```

Поскольку известно количество элементов последовательности, его можно указать в конструкторе вектора (то есть в круглых скобках после названия переменной):

```
vector<string> v(n);
```

После этого можно с помощью цикла `for` перебрать все элементы вектора по ссылке:

```
for (string& s : v) {  
    cin >> s;  
}
```

Каждый очередной элемент `s` — ссылка на очередной элемент вектора. С помощью этой ссылки считывается очередная строчка.

Теперь остается вывести вектор на экран, чтобы проверить, что все было считано правильно. Для этого удобно написать специальную функцию, которая выводит все значения вектора. Вызываем функцию следующим образом:

```
PrintVector(v);
```

А само определение функции `PrintVector` располагаем над функцией `main`:

```
void PrintVector(const vector<string>& v) {  
    for (string s : v) {  
        cout << s << endl;  
    }  
}
```

Запустим программу и проверим, что она работает:

```
> 2
> Anton
> Ilia
Anton
Ilia
```

Отлично: мы успешно считали элементы вектора и успешно их вывели.

### Добавление элементов в вектор. Методы `push_back` и `size`

Можно реализовать эту программу несколько иначе с помощью цикла `while`. Также считаем число элементов вектора `n`, но создадим пустой вектор `v`.

```
int n;
cin >> n;
vector<string> v;
```

Создадим переменную `i`, в которой будет храниться индекс считываемой на данной итерации строки.

```
int i = 0;
```

В цикле `while` считываем строку из консоли в локальную вспомогательную переменную `s`, которая добавляется к вектору с помощью метода `push_back`:

```
while (i < n) {
    string s;
    cin >> s;
    v.push_back(s);
    cout << "Current size = " << v.size() << endl;
    ++i;
}
```

В конце каждой итерации значение `i` увеличивается на 1. Чтобы продемонстрировать, что размер вектора меняется, на каждой итерации его текущий размер выводится на экран.

После завершения цикла чтения, как и в предыдущем примере, выводим значения вектора на экран с помощью функции `PrintVector`:

```
PrintVector(v);
```

```
> 2
> first
Current size = 1
> second
Current size = 2
first
second
```

Как и ожидалось, после ввода первой строки текущий размер стал равным 1, а после ввода второй — равным 2.

Вернемся к прошлой программе, в которой размер вектора задавался через конструктор в самом начале, и добавим туда также вывод размера на каждой итерации цикла:

```
int n;
cin >> n;
vector<string> v(n);
for (string& s: v) {
    cin >> s;
    cout << "Current size = " << v.size() << endl;
}
PrintVector(v);
```

Запустим программу и убедимся, что в таком случае размер вектора постоянен:

```
> 2
> first
Current size = 2
> second
Current size = 2
first
second
```

Так происходит, потому что в самом начале программы вектор создается сразу нужного размера.

### Задание элементов вектора при его создании

Бывают случаи, когда содержимое вектора заранее известно. В этом случае указать заранее известные значения при создании вектора можно с помощью фигурных скобок. Например, числовой вектор, содержащий количество дней в каждом месяце (для краткости: в первых 5 месяцах), можно создать так:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};
```

Такой вектор можно распечатать:

```
PrintVector(days_in_months);
```

Правда, сперва следует подправить функцию PrintVector так, чтобы она принимала числовой вектор, а не вектор строк:

```
void PrintVector(const vector<int>& v) {  
    for (auto s : v) {  
        cout << s << endl;  
    }  
}
```

Запустим программу, убеждаемся, что она работает как надо.

Иногда бывает необходимым изменить значения вектора после его создания. Например, в високосных годах количество дней в феврале — 29, и чтобы это учесть, слегка допишем нашу программу:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};  
if (true) { // if year is leap  
    days_in_months[1]++;  
}  
PrintVector(days_in_months);
```

Здесь для простоты проверка на високосность опущена. Замечу, что в C++ элементы вектора нумеруются с нуля, поэтому количество дней в феврале хранится в первом элементе вектора.

Из этого примера можно сделать вывод, что вектор также можно использовать для хранения элементов в привязке к их индексам.

## Создание вектора, заполненного значением по умолчанию

Допустим, нужно создать вектор, который для каждого дня в феврале хранит, является ли данный день праздничным. В этом случае следует использовать вектор булевых значений. Поскольку большинство дней праздничными не являются, хотелось бы, чтобы при создании вектора все его значения по умолчанию были false.

Значение по умолчанию можно указать, передав его в качестве второго аргумента конструктора:

```
vector<bool> is_holiday(28, false);
```

В качестве первого аргумента конструктора указывается длина вектора, как и в первом примере. После этого заполним элементы вектора. Например, известно, что 23 февраля — праздничный день:

```
is_holiday[22] = true;
```

Вывести вектор в консоль можно с помощью функции PrintVector:

```
PrintVector(is_holiday);
```

Функцию PrintVector все же предстоит сперва доработать, чтобы она принимала вектор булевых значений.

```
void PrintVector(const vector<bool>& v) {  
    for (auto s : v) {  
        cout << s << endl;  
    }  
}
```

Заметим, что изменилось только определение типа при задании параметра функции, а ее тело осталось неизменным. В будущем это позволит обобщить эту функцию для вывода векторов разных типов. Но пока мы не обсудили этот вопрос, приходится довольствоваться только функциями, каждая из которых работает с векторами определенного типа.

## Изменение длины вектора

Для удобства сперва доработаем функцию вывода, чтобы кроме значений выводились также и индексы элементов.

```
void PrintVector(const vector<bool>& v) {  
    int i = 0;  
    for (auto s : v) {  
        cout << i << ": " << s << endl;  
        ++i;  
    }  
}
```

Иногда бывает необходимым изменить длину вектора. Например, если необходимо (по тем или иным причинам) созданный в предыдущей программе вектор использовать для хранения праздничных мартовских дней, его нужно сперва расширить и заполнить значением по умолчанию.

Попытаемся сделать это с помощью функции `resize`, которая может выполнить то, что надо. Попробуем это сделать:

```
is_holiday.resize(31);  
PrintVector(is_holiday);
```

Метод `resize` сделал не то, что мы хотели, потому что старые значения остались и 23 марта оказалось праздничным. Если мы хотим переиспользовать этот вектор и сделать его нужной длины, нам понадобится метод `assign`:

```
is_holiday.assign(31, false);
```

В качестве первого аргумента передается желаемый размер вектора, а в качестве второго — какими элементами проинициализировать его элементы. Теперь можно указать, что 8 марта — праздничный день:

```
is_holiday[7] = true;
```

Запустив код, убеждаемся, что «упоминание о 23 марта» пропало, как и хотелось.

```
PrintVector(is_holiday);
```

Очистить вектор можно с помощью метода `clear`:

```
is_holiday.clear();
```

## 2.3.2. Контейнер `map`

### Создание словаря. Добавление элементов

Допустим, требуется хранить важные события в привязке к годам, в которые они произошли. Для решения этой задачи лучше всего подходит такой контейнер как словарь. Словарь состоит из пар ключ-значение, причем ключи не могут повторяться. Для работы со словарями нужно подключить соответствующий заголовочный файл:

```
#include <map>
```

Создадим словарь с ключами типа `int` и строковыми значениями:

```
map<int, string> events;  
events[1950] = "Bjarne Stroustrup's birth";  
events[1941] = "Dennis Ritchie's birth";  
events[1970] = "UNIX epoch start";
```

Напишем функцию, которая позволяет вывести словарь на экран:

```
void PrintMap(const map<int, string>& m) {  
    cout << "Size = " << m.size() << endl;  
    for (auto item: m) {  
        cout << item.first << ": " << item.second << endl;  
    }  
}
```

Обратиться к ключу очередного элемента `item` при итерировании можно как `item.first`, а к значению — как к `item.second`. Также добавим в функцию вывода словаря вывод его размера (используя метод `size`).



## Итерирование по элементам словаря

Выведем получившийся словарь на экран с помощью написанной функции:

```
PrintMap(events);
```

На экран будут выведены три элемента:

```
Size = 3
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
1970: UNIX epoch start
```

Словарь не просто вывелся на экран в формате ключ-значение. В выводе ключи оказались отсортированными в порядке возрастания целых чисел.

Этот пример демонстрирует одно из важных свойств словаря: элементы в нем хранятся отсортированными по ключам, а также выводятся отсортированными в цикле `for`.

## Обращение по ключу к элементам словаря

Кроме того, можно обращаться к конкретным значениям из словаря по ключу. Например, можно узнать событие, которое произошло в 1950 году:

```
cout << events[1950] << endl;
```

```
Bjarne Stroustrup's birth
```

Отдельно отметим, что такой синтаксис очень напоминает синтаксис для получения значения элемента вектора по индексу. В некотором смысле, словарь позволил расширить функционал вектора: теперь в качестве ключей можно указывать сколь угодно большие целые числа.

## Удаление по ключу элементов словаря

Элементы словаря можно не только добавлять в него, но и удалять. Для удаления элемента словаря по ключу используется метод `erase`:

```
events.erase(1970);
PrintMap(events);
```

```
Size = 2
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
```

## Построение «обратного» словаря

Ключи словаря могут иметь тип `string`. Продемонстрируем это, обратив построенный нами словарь. Словарь, который получится в результате, позволит получать по названию события год, когда это событие произошло.

Для построения такого словаря, напомним функцию `BuildReversedMap`:

```
map<string, int> BuildReversedMap(
    const map<int, string>& m) {
    map<string, int> result;
    for (auto item: m) {
        result[item.second] = item.first;
    }
    return result;
}
```

Реализация этой функции достаточно проста. Сперва нужно приготовить итоговый словарь, типы ключей и значений в котором переставлены по сравнению с исходным словарем. Затем в цикле `for` нужно пробежаться по всем элементам исходного словаря и записать в итоговый, используя в качестве ключа бывшее значение, а в качестве значения — ключ. После цикла нужно вернуть получившийся словарь с помощью `return`.

Для вывода на экран получившегося словаря необходимо написать функцию `PrintReversedMap`, поскольку мы пока не научились писать функцию, выводящую на печать словарь любого типа:

```
void PrintReversedMap(const map<string, int>& m) {
    cout << "Size = " << m.size() << endl;
    for (auto item: m) {
        cout << item.first << ": " << item.second << endl;
    }
}
```

Еще раз отметим, что тело функции уже довольно общее и в нем нигде не содержатся типы ключей и значений.

Теперь можно запустить следующий код:

```
map<string, int> event_for_year = BuildReversedMap(events);
PrintReversedMap(event_for_year);
```

```
Size = 2
Bjarne Stroustrup's birth: 1950
Dennis Ritchie's birth: 1941
```

Также по названиям событий можно получить год, в котором они произошли:

```
cout << event_for_year["Bjarne Stroustrup's birth"];
```

```
1950
```

## Создание словаря по заранее известным данным

Создание словаря по заранее известному набору пар ключ-значение можно произвести следующим образом с помощью фигурных скобок:

```
map<string, int> m = {"one", 1}, {"two", 2}, {"three", 3};
```

Выведем словарь на экран и убедимся, что он создан правильно:

```
PrintMap(m);
```

```
one: 1
three: 3
two: 2
```

Все ключи здесь отсортировались лексикографически, то есть в алфавитном порядке.

Следует также отметить, что функцию печати словаря можно улучшить, итерируясь по нему по константной ссылке:

```
void PrintMap(const map<string, int>& m) {
    for (const auto& item: m) {
        cout << item.first << ": " << item.second << endl;
    }
}
```

В таком случае получается избежать лишнего копирования элементов словаря.

Еще раз отметим, как удалять значения из словаря, например для ключа «three»:

```
map<string, int> m = {"one", 1}, {"two", 2}, {"three", 3};
m.erase("three");
PrintMap(m);
```

```
one: 1
two: 2
```

## Подсчет количества различных элементов последовательности

Словари могут быть полезными, если необходимо подсчитать, сколько раз встречаются элементы в некоторой последовательности.

Допустим, дана последовательность слов:

```
vector<string> words = {"one", "two", "one"};
```

Строки могут повторяться. Необходимо подсчитать, сколько раз встретилась каждое слово из этой последовательности. Для этого создадим словарь:

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

После этого пробежимся по всем элементам последовательности. Случай, когда слово еще не встречалось, нужно будет рассматривать отдельно, например так:

```
for (const string& word : words) {  
    if (counters.count(word) == 0) {  
        counters[word] = 1;  
    } else {  
        ++counters[word];  
    }  
}
```

Проверка на то, содержится ли элемент в словаре, может быть произведена с помощью метода count, как показано в коде. Такой код, безусловно, работает, но оказывается, что он избыточен. Достаточно написать так:

```
for (const string& word : words) {  
    PrintMap(counters);  
    ++counters[word];  
}  
PrintMap(counters);
```

Дело в том, что как только происходит обращение к конкретному элементу словаря с помощью квадратных скобок, компилятор уже создает пару для этого ключа со значением по умолчанию (для целого числа значение по умолчанию — 0).

Здесь мы сразу добавили вывод всего словаря для того, чтобы продемонстрировать как меняется размер словаря (в функцию PrintMap также добавлен вывод размера словаря):

```
Size = 0  
Size = 1  
one: 1
```

```
Size = 2
one: 1
two: 1
Size = 2
one: 2
two: 1
```

Продemonстрируем, что от простого обращения к элементу словаря происходит добавление к нему пары с этим ключом и значением по умолчанию:

```
map<string, int> counters;
counters["a"];
PrintMap(counters);
```

```
Size = 1
a: 0
```

## Группировка слов по первой букве

Приведем еще один пример, показывающий, как можно использовать свойство изменения размера словаря при обращении к несуществующему ключу. Предположим, что необходимо сгруппировать слова из некоторой последовательности по первой букве. Решение данной задачи может выглядеть следующим образом:

```
vector<string> words = {"one", "two", "three"};
map<char, vector<string>> grouped_words;
for (const string& word : words) {
    grouped_words[word[0]].push_back(word);
}
```

В цикле `for` сначала идет обращение к несуществующему ключу (первой букве каждого слова). При этом ключ добавляется в словарь вместе с пустым вектором в качестве значения. Далее, с помощью метода `push_back` текущее слово присваивается в качестве значения текущего ключа. Выведем словарь на экран и убедимся, что слова были сгруппированы по первой букве:

```
for (const auto& item: grouped_words) {
    cout << item.first << endl;
    for (const string& word : item.second) {
        cout << word << " ";
    }
    cout << endl;
}
```

```
o
one
t
two three
```

## Стандарт C++17

Недавно комитет по стандартизации языка C++ утвердил новый стандарт C++17. Говоря простым языком, были утверждены новые возможности языка. Но, к сожалению, изменения в стандарте только спустя некоторое время отражаются в свежих версиях компиляторов. Также свежие версии компиляторов не всегда просто использовать, так как они еще не появились в дистрибутивах для разработки. Тем не менее, имеет смысл рассказывать о свежих возможностях языка, даже если пока они не поддерживаются компиляторами и их еще нельзя использовать.

Чтобы попробовать новые возможности компиляторов, существуют различные ресурсы, например [gsc.goldbolt.org](http://gsc.goldbolt.org). Он представляет собой окно ввода кода на C++ и панель для выбора версии компилятора. На данной панели можно выбрать еще не вышедшую версию компилятора gsc 7. Чтобы сказать компилятору, что код будет соответствовать новому стандарту, нужно указать флаг компиляции `|--std=c++17|`.

Среди новых возможностей — новый синтаксис для итерирования по словарю. Например, так бы выглядел код итерирования с использованием старого стандарта:

```
#include <map>

using namespace std;

int main() {
    map<string, int> m = {{"one", 1}, {"two", 2}};
    for (const auto& item : m) {
        item.first, item.second;
    }

    return 0;
}
```

В данном коде имеются следующие проблемы:

- Переменная `item` имеет «странный» тип с полями `first` и `second`.
- Нужно либо помнить, что `first` соответствует ключу, а `second` — значению текущего элемента, либо заводить временные переменные.

В новом стандарте появляется возможность писать такой код более понятно:

```
map<string, int> m = {{"one", 1}, {"two", 2}};
for (const auto& [key, value] : m) {
    key, value;
}
```

### 2.3.3. Контейнер set

Допустим, необходимо сохранить для каждого человека, является ли он известным. В этом случае можно было бы завести словарь, ключами в котором были бы строки, а значениями — логические значения:

```
map<string, bool> is_famous_person;
```

Теперь, чтобы указать, что какие-то люди являются известными, можно написать следующий код:

```
is_famous_person["Stroustrup"] = true;
is_famous_person["Ritchie"] = true;
```

Имеет ли смысл добавлять в этот словарь людей, которые являются неизвестными? Наверное, нет: таких людей слишком много и их нет нужды хранить, когда можно хранить только известных людей. А в этом случае значениями в таком словаре являются только true.

#### Создание множества. Добавление элементов.

Для решения такой задачи более естественно использовать другой контейнер — множество (set). Для работы с множествами необходимо подключить соответствующий заголовочный файл:

```
#include <set>
```

Теперь можно создать множество известных людей:

```
set<string> famous_persons;
```

Добавить в это множество элементы можно с помощью метода insert:

```
famous_persons.insert("Stroustrup");
famous_persons.insert("Ritchie");
```

## Печать элементов множества

Функция PrintSet, позволяющая печатать на экране все элементы множества строк, реализуется следующим образом:

```
void PrintSet(const set<string>& s) {  
    cout << "Size = " << s.size() << endl;  
    for (auto x : s) {  
        cout << x << endl;  
    }  
}
```

В эту функцию сразу добавлен вывод размера множества — он может быть получен с помощью метода size.

Теперь можно вывести на экран элементы множества известных людей:

```
PrintSet(famous_persons);
```

```
Size = 2  
Ritchie  
Stroustrup
```

Элементы множества выводятся в отсортированном порядке, а не в порядке добавления.

Также гарантируется уникальность элементов. То есть повторно никакой элемент не может быть добавлен в множество.

```
set<string> famous_persons;  
famous_persons.insert("Stroustrup");  
famous_persons.insert("Ritchie");  
famous_persons.insert("Stroustrup");  
PrintSet(famous_persons);
```

```
Size = 2  
Ritchie  
Stroustrup
```

## Удаление элемента

Удаление из множества производится с помощью метода erase:

```
set<string> famous_persons;  
famous_persons.insert("Stroustrup");  
famous_persons.insert("Ritchie");  
famous_persons.insert("Anton");  
PrintSet(famous_persons);
```



```
Size = 3
Anton
Ritchie
Stroustrup
```

```
famous_persons.erase("Anton");
PrintSet(famous_persons);
```

```
Size = 2
Ritchie
Stroustrup
```

## Создание множества с известными значениями

С помощью фигурных скобок можно создать множество, заранее указывая значения содержащихся в нем элементов. Например, множество названий месяцев может быть инициализировано как:

```
set<string> month_names =
    {"January", "March", "February", "March"};
PrintSet(month_names);
```

```
Size = 3
February
January
March
```

## Сравнение множеств

Как и другие контейнеры, множества можно сравнивать:

```
set<string> month_names =
    {"January", "March", "February", "March"};
set<string> other_month_names =
    {"March", "January", "February"};

cout << (month_names == other_month_names) << endl;
```

В результате будет выведено «1», то есть эти множества равны.

## Проверка принадлежности элемента множеству

Для того, чтобы быстро проверить, принадлежит ли элемент множеству, можно использовать метод count:

```
set<string> month_names =
    {"January", "March", "February", "March"};
cout << month_names.count("January") << endl;
```

## Создание множества по вектору

Чтобы создать множество по вектору, не обязательно писать цикл. Реализовать это можно следующим образом:

```
vector<string> v = {"a", "b", "a"};  
set<string> s(begin(v), end(v));  
PrintSet(s);
```

Size = 2

a

b

С помощью аналогичного синтаксиса можно создать и вектор по множеству.

## Неделя 3

# Переменные в C++. Пользовательские типы данных

### 3.1. Алгоритмы. Лямбда-выражения

#### 3.1.1. Вычисление минимума и максимума

Напишем функцию Min, которая будет принимать два числа, вычислять минимальное и возвращать его:

```
int Min(int a, int b){  
    if (a < b) {  
        return a;  
    }  
    return b;  
}
```

Аналогично реализуем функцию, которая будет возвращать максимум из двух чисел:

```
int Max(int a, int b){  
    if (a > b) {  
        return a;  
    }  
    return b;  
}
```

Проверим, как эти функции работают:

```
cout << Min(2, 3) << endl; // 2  
cout << Max(2, 3) << endl; // 3
```

Чтобы реализовать функцию нахождения минимума и максимума других типов, их пришлось бы определять дополнительно.

Но в стандартной библиотеке C++ существуют встроенные функции вычисления минимума и максимума, которые могут работать с переменными различных типов, которые могут сравниваться друг с другом.

Для работы со стандартными алгоритмами нужно подключить заголовочный файл:

```
#include <algorithm>
```

Теперь остается изменить первую букву в вызовах функции с большой на маленькую, чтобы использовать встроенные функции:

```
cout << min(2, 3) << endl;  
cout << max(2, 3) << endl;
```

Использование встроенных функций позволяет избежать ошибок, связанных с повторной реализацией их функциональности.

Точно также можно искать минимум и максимум двух строк:

```
string s1 = "abc";  
string s2 = "bca";  
cout << min(s1, s2) << endl;  
cout << max(s1, s2) << endl;
```

Точно так же можно искать минимум и максимум всех типов, которые можно сравнивать между собой, то есть для которых определен оператор <.

### 3.1.2. Сортировка

Пусть необходимо отсортировать вектор целых чисел:

```
vector<int> v = {  
    1, 3, 2, 5, 4  
};
```

Для удобства определим функцию, выводящую значения вектора в консоль:

```
void Print(const vector<int>& v, const string& title){  
    cout << title << ": ";  
    for (auto i : v) {  
        cout << i << ' ';  
    }  
}
```

Вторым параметром передается строка `title`, которая будет выводиться перед выводом вектора.

Распечатаем вектор до сортировки с «заголовком» `"init"`:

```
Print(v, "init");
```

После этого воспользуемся функцией сортировки. Чтобы это сделать, ей нужно передать начало и конец интервала, который нужно отсортировать. Взять начало и конец интервала можно с помощью встроенных функций `begin` (возвращает начало вектора) и `end` (возвращает конец вектора):

```
sort(begin(v), end(v));
```

После этого распечатаем вектор с меткой «sort»:

```
cout << endl;
Print(v, "sort");
```

Результат работы программы:

```
init: 1 3 2 5 4
sort: 1 2 3 4 5
```

Программа работает так, как и ожидалось.

### 3.1.3. Подсчет количества вхождений конкретного элемента

Допустим, необходимо подсчитать сколько раз конкретное значение встречается в контейнере.

Например, необходимо подсчитать количество элементов «2» в векторе из целых чисел. Для этого можно воспользоваться циклом range-based for:

```
vector<int> v = {
    1, 3, 2, 5, 4
};
int cnt = 0;
for (auto i : v) {
    if (i == 2) {
        ++cnt;
    }
}
cout << cnt;
```

Несмотря на то, что этот код работает, не следует подсчитывать число вхождений таким образом, поскольку в стандартной библиотеке есть специальная функция.

Функция `count` принимает начало и конец интервала, на котором она работает. Третьим аргументом она принимает элемент, количество вхождений которого надо подсчитать.

```
vector<int> v = {
    1, 3, 2, 5, 4
};
cout << count(begin(v), end(v), 2);
```

### 3.1.4. Подсчет количества элементов, которые удовлетворяют некоторому условию

Подсчитать количество элементов, которые обладают некоторым свойством, можно с помощью функции `count_if`. В качестве третьего аргумента в этом случае нужно передать функцию, которая принимает в качестве аргумента элемент и возвращает `true` (если условие выполнено) или `false` (если нет). Чтобы подсчитать количество элементов, которые больше 2, определим внешнюю функцию:

```
bool Gt2(int x) {
    if (x > 2) {
        return true;
    }
    return false;
}
```

Теперь эту функцию можно передать в `count_if`:

```
vector<int> v = {
    1, 3, 2, 5, 4
};
cout << count_if(begin(v), end(v), Gt2);
```

По аналогии можно определить функцию «меньше двух»:

```
bool Lt2(int x) {
    if (x < 2) {
        return true;
    }
    return false;
}
```

Которую также можно использовать в `count_if`:

```
cout << count_if(begin(v), end(v), Lt2);
```

Недостаток такого подхода заключается в следующем: функция `Gt2` является достаточно специализированной функцией, и вряд ли она будет повторно использоваться. Также определение функции расположено далеко от места ее использования.

### 3.1.5. Лямбда-выражения

Лямбда-выражения позволяют определять функции на лету — сразу в месте ее использования. Синтаксис следующий: сначала идут квадратные скобки, после которых — аргументы в круглых скобках и тело функции.

```
cout << count_if(begin(v), end(v), [](int x) {  
    if (x > 2) {  
        return true;  
    }  
    return false;  
});
```

В этом примере лямбда-выражение принимает на вход целое число и возвращает `true`, если переданное число больше 2.

Пусть необходимо сделать так, чтобы число, с которым происходит сравнение, например, приходило из консоли.

```
int thr;  
cin >> thr;
```

Если попытаться воспользоваться этой переменной в лямбда-выражении:

```
cout << count_if(begin(v), end(v), [](int x) {  
    if (x > thr) {  
        return true;  
    }  
    return false;  
});
```

компилятор выдаст ошибку «`thr` is not captured». Непосредственно использовать в лямбда-выражении переменные из контекста нельзя. Чтобы сообщить, что переменную следует взять из контекста как раз используются квадратные скобки.

```

cout << count_if(begin(v), end(v), [thr](int x) {
    if (x > thr) {
        return true;
    }
    return false;
});

```

### 3.1.6. Mutable range-based for

Допустим, необходимо увеличить все значения в некотором массиве на 1.

```

vector<int> v = {
    1, 3, 2, 5, 4
};
Print(v, "init");

```

Вывод вектора на экран производится в функции Print:

```

void Print(const vector<int>& v, const string& title){
    cout << title << ": ";
    for (auto i : v) {
        cout << i << ' ';
    }
}

```

Для этого можно воспользоваться обычным циклом for:

```

for (int i = 0; i < v.size(); ++i) {
    ++v[i];
}
cout << endl;
Print(v, "inc");

```

Такая программа отлично работает и выдает ожидаемый от нее результат. Но все же хочется использовать цикл range-based for, так как в этом случае значительно меньше вероятность внести ошибку.

```

for (auto i : v) {
    ++i;
}

```

Но такой код не работает: значения вектора не изменяются, так как по умолчанию на каждой итерации берется копия объекта из контейнера. Получить доступ к объекту в цикле range-based for можно добавив после ключевого слова auto символ &, обозначающий ссылку.



```
for (auto& i : v) {  
    ++i;  
}
```

## 3.2. Видимость и инициализация переменных

### 3.2.1. Видимость переменной

Ссылаться на переменную (и использовать) можно только после того, как она была объявлена:

```
cout << x;  
int x = 5;
```

Такой код даже не скомпилируется. Компилятор выдаст ошибку «'x' was not declared in this scope». Если же поменять строчки местами, программа заработает.

Другой пример:

```
{  
    int x = 5;  
    {  
        cout << x;  
    }  
    cout << x;  
}  
cout << x;
```

Здесь переменная `x` определена внутри операторных скобок, и в трех местах кода имеет место попытка вывести ее на экран. При этом программа не компилируется: компилятор указывает на ошибку при попытке вывести на экран `x` за пределами первых операторных скобок. Если эту строчку закомментировать, программа успешно скомпилируется:

```
{  
    int x = 5;  
    {  
        cout << x;  
    }  
    cout << x;  
}  
//cout << x;
```

Таким образом, переменные в C++ видны только после своего объявления и до конца блока, в котором были объявлены. Например, следующий код с условным оператором не скомпилируется:

```
if (1 > 0) {  
    int x = 5;  
}  
cout << x;
```

Это связано с тем, что переменная объявлена внутри тела условного оператора.

То же самое имеет место для цикла `while`:

```
while (1 > 0) {  
    int x = 5;  
}  
cout << x;
```

И для цикла `for`:

```
for (int i = 0; i < 10; ++i) {  
    int x = 5;  
}  
cout << x;
```

Может возникнуть вопрос: видна ли переменная `i`, которая была объявлена как счетчик цикла:

```
for (int i = 0; i < 10; ++i) {  
    int x = 5;  
}  
cout << i;
```

Оказывается, что она также не видна.

Еще один пример:

```
string s = "hello";  
{  
    string s = "world";  
    cout << s << endl;  
}  
cout << s << endl;
```

Здесь переменная `s` определена как внутри операторных скобок, так и вне их. Программа компилируется и выводит:

```
world  
hello
```

Однако использование одинаковых имен, хоть не вызывает ошибку компиляции, считается плохим стилем, так как усложняет понимание кода и увеличивает вероятность ошибиться.

### 3.2.2. Инициализация переменной

Пусть функция `PrintInt` объявляет переменную типа `int` и выводит на экран:

```
void PrintInt() {  
    int x;  
    cout << x << endl;  
}
```

Пусть также определена функция `PrintDouble`, в которой определяется переменная типа `double` и ей сразу присваивается значение. Переменная также выводится на экран.

```
void PrintDouble() {  
    double pi = 3.14;  
    cout << pi << endl;  
}
```

Пусть в `main` сначала вызывается функция `PrintInt`, а за ней — `PrintDouble`:

```
PrintInt();  
PrintDouble();
```

Такая программа компилируется без ошибок. Интерес представляет то, какое значение `x` будет выведено на экран, поскольку значение этой переменной установлено не было. Можно предположить, что по умолчанию значение переменной `x` будет равно 0. Программа выводит:

```
0  
3.14
```

Теперь рассмотрим другую ситуацию, когда после функции `PrintDouble` еще раз вызывается `PrintInt`:

```
PrintInt();  
PrintDouble();  
PrintInt();
```

В результате работы программы:

```
0  
3.14  
1074339512
```

Вместо нуля при втором вызове `PrintInt` выводится какое-то большое странное число, а не ноль.

Этот пример показывает, что значение неинициализированной переменной не определено. У этого есть рациональное объяснение. Автор C++ руководствовался принципом «zero overhead principle»<sup>1</sup>:

---

<sup>1</sup> «Не платить за то, что не используется.»

```
int value; // мне все равно, что будет в переменной value
int value = 0; // мне необходимо, чтобы в value был ноль
```

Отсюда следует вывод: переменные нужно инициализировать при их объявлении. Так получится защититься от ситуации, что в переменной неожиданно окажется мусор, а не ожидаемое значение.

### 3.2.3. Инициализация переменной при объявлении: примеры

Следующая функция печатает, является ли переданное в качестве параметра число четным:

```
void PrintParity(int x) {
    string parity;

    if (x % 2 == 0) {
        parity = "even";
    } else {
        parity = "odd";
    }

    cout << x " is " << parity;
}
```

В этом случае, казалось бы, не получается инициализировать переменную при ее объявлении. Однако этого можно добиться с помощью тернарного оператора:

```
void PrintParity(int x) {
    string parity = (x % 2 == 0) ? "even": "odd";
    cout << x " is " << parity;
}
```

Следующая функция выводит на экран, является ли число положительным, отрицательным или нулем:

```
void PrintPositivity(int x) {
    string positivity;

    if (x > 0) {
        positivity = "positive";
    } else if (x < 0) {
        positivity = "negative";
    } else {
```

```
    positivity = "zero";  
}  
  
    cout << x " is " << positivity;  
}
```

В этом случае также хочется добиться инициализации переменной при ее объявлении.

Для этого можно вынести часть кода в отдельную функцию:

```
string GetPositivity(int x){  
    if (x > 0) {  
        return "positive";  
    } else if (x < 0) {  
        return "negative";  
    } else {  
        return "zero";  
    }  
}
```

В этом случае переменная positivity может быть инициализирована в месте ее объявления:

```
void PrintPositivity(int x){  
    string positivity = GetPositivity(x);  
    cout << x " is " << positivity;  
}
```

## 3.3. Структуры. Классы

### Структуры

#### 3.3.1. Зачем нужны структуры?

Ядром ООП является создание программистом собственных типов данных. Для начала следует обсудить вопрос, зачем вообще такое может понадобиться.

Допустим, программа должна работать с видеолекциями, в том числе с их названиями и длительностями (в секундах). Можно написать такую функцию, которая будет работать с данными характеристиками видеолекции:

```
void PrintLecture(const string& title,
                  int duration) {
    cout << "Title: " << title <<
          ", duration: " << duration << "\n";
}
```

Эта функция выводит на экран информацию о видеолекции, принимая в качестве параметров ее название и продолжительность.

Если нужно вывести информацию о курсе, то есть о серии видеолекций, можно написать функцию PrintCourse. Эта функция должна принять на вход набор видеолекций, но поскольку информация о них хранится в виде характеристик, функция принимает в качестве параметров вектор названий и вектор длительностей видеолекций:

```
PrintCourse(const vector<string>& titles,
            const vector<int>& durations) {
    int i = 0;
    while (i < titles.size()) {
        PrintLecture(titles[i], durations[i]);
        ++i;
    }
}
```

Может возникнуть необходимость хранить и обрабатывать дополнительно имя лекторов, которые читают лекции. Код постепенно разбухает. В функцию PrintLecture нужно передавать еще один параметр:

```
void PrintLecture(const string& title,
                  int duration,
                  const string& author) {
```

```

        cout << "Title: "    << title <<
            ", duration: " << duration <<
            ", author: "    << author << "\n";
    }

```

Функцию PrintCourse также нужно модифицировать:

```

void PrintCourse(const vector<string>& titles,
                const vector<int>& durations,
                const vector<string>& authors) {
    int i = 0;
    while (i < titles.size()) {
        PrintLecture(titles[i],
                    durations[i],
                    authors[i]);
        ++i;
    }
}

```

Основные недостатки представленного подхода:

- Хочется работать с объектами (лекциями), а не отдельно с каждой из составляющих характеристик (название, продолжительность, имя лектора). Другими словами, в коде неправильно выражается намерение: вместо того, чтобы передать в качестве параметра лекцию, передается название, продолжительность и имя автора.
- При добавлении или удалении характеристики нужно менять заголовки функций, а также все их вызовы.
- Отсутствует единый список характеристик. Не существует единого места, где указаны все характеристики объекта.

### 3.3.2. Структуры

Для создания нового типа данных используется ключевое слово `struct`. После него идет название нового типа данных, а затем в фигурных скобках перечисляются поля.

```

struct Lecture { // Составной тип из 3 полей
    string title;
    int duration;
    string author;
};

```



Синтаксис объявления полей похож на синтаксис объявления переменных.

Обратиться к определенному полю объекта можно написав после имени переменной точку, после которой записывается название требуемого поля. Теперь можно переписать функции, чтобы они использовали новый тип данных:

```
void PrintLecture(const Lecture& lecture) {  
    cout << "Title: "    << lecture.title <<  
        ", duration: " << lecture.duration <<  
        ", author: "    << lecture.author << "\n";  
}
```

Здесь лекция передается по ссылке, чтобы избежать копирования.

В функции PrintCourse все еще понятнее: она будет принимать то, что и задумывалось изначально — набор видеолекций, в виде вектора из элементов типа Lecture:

```
void PrintCourse(  
    const vector<Lecture>& lectures) {  
    for (Lecture lecture : lectures) {  
        PrintLecture(lecture);  
    }  
}
```

Особо отметим, что хоть и был определен пользовательский тип данных, можно создавать контейнеры, элементы которого будут иметь такой тип. Более того, итерирование в данном случае уже можно производить с помощью цикла range-based for, а не while.

Код стал более понятным, более компактным, лучше поддерживаемым. Если нужно добавить новую характеристику видеолекции, достаточно поправить определение структуры, а менять заголовки и вызовы функций не потребуется. Разве что может понадобится добавление вывода нового поля в функцию PrintLecture, что вполне ожидаемо.

### 3.3.3. Создание структур

Существует несколько способов создания переменной пользовательского типа с определенными значениями полей. Самый простой из них — объявить переменную желаемого типа, а после — указать значения каждого поля вручную. Например:

```
Lecture lecture1;  
lecture1.title = "OOP";  
lecture1.duration = 5400;  
lecture1.author = "Anton";
```

Проблема такого способа заключается в том, что название переменной постоянно повторяется, а также такой код занимает целых 4 строчки даже в таком простом примере.

Более короткий способ создания структур с требуемыми значениями полей: при инициализации после знака равно записать в фигурных скобках желаемые значения полей в том же порядке, в котором они были объявлены:

```
Lecture lecture2 = {"OOP", 5400, "Anton"};
```

Более того, такой способ годится даже для вызова функций без создания промежуточных переменных:

```
PrintLecture({"OOP", 5400, "Anton"});
```

Точно также, с помощью фигурных скобок можно вернуть объект из функции:

```
Lecture GetCurrentLecture() {  
    return {"OOP", 5400, "Anton"};  
}
```

```
Lecture current_lecture = GetCurrentLecture();
```

### 3.3.4. Вложенные структуры

Поле некоторого пользовательского типа может иметь тип, который также является пользовательским. Другими словами, можно создавать вложенные структуры.

Например, если название лекции представляет собой не одну, а три строки (название специализации, курса и название недели), можно создать структуру LectureTitle:

```
struct LectureTitle {  
    string specialization;  
    string course;  
    string week;  
};  
  
struct DetailedLecture {  
    LectureTitle title;  
    int duration;  
};
```

Новый тип можно использовать везде, где можно было использовать встроенные типы языка C++. В том числе указывать как тип поля при создании других типов.

Создать вложенную структуру можно используя уже известный синтаксис:

```
LectureTitle title = {"C++", "White belt", "OOP"};
DetailedLecture lecture1 = {title, 5400};
```

Этот код можно записать короче и без использования временной переменной:

```
DetailedLecture lecture2 = {
    {"C++", "White belt", "OOP"},
    5400
};
```

Обращаться к внутренним полям можно ожидаемым образом:

```
cout << lecture2.title.specialization << "\n";
// Выведет «C++»
```

### 3.3.5. Область видимости типа

Использовать тип можно только после его объявления. Поэтому поменять местами объявления DetailedLecture и LectureTitle не получится: будет ошибка компиляции.

```
struct DetailedLecture {
    LectureTitle title; // Не компилируется:
    int duration;       // тип LectureTitle
};                     // пока неизвестен

struct LectureTitle {
    string specialization;
    string course;
    string week;
};
```

## Классы

### 3.3.6. Приватная секция

Пусть требуется написать программу, которая работает с маршрутами между городами. Каждый маршрут будет представлять собой название

двух городов, где маршрут начинается и где маршрут заканчивается. Объявим структуру:

```
struct Route {  
    string source;  
    string destination;  
};
```

Кроме того, пусть дана функция для расчета длины пути.

```
int ComputeDistance(  
    const string& source,  
    const string& destination);
```

Эта функция уже написана кем-то и ее реализация может быть достаточно тяжелой: функция может в ходе исполнения обращаться к базе данных и запрашивать данные оттуда.

В любом случае, в программе иногда возникает необходимость вычислить длину маршрута. Каждый раз вычислять длину затратно, поэтому ее нужно где-то хранить. Можно создать еще одно поле в существующей структуре.

```
struct Route {  
    string source;  
    string destination;  
    int length;  
};
```

Теперь, в принципе, можно написать программу, которая будет делать то, что требуется, и она может отлично работать. Однако поле `length` доступно публично, то есть нельзя быть уверенным, что `length` — это расстояние между `source` и `destination`:

- Можно случайно изменить значение переменной `length`
- Можно изменить один из городов и забыть обновить значение `length`

Хочется минимизировать количество возможных ошибок при написании кода. Для этого нужно запретить прямой, то есть публичный, доступ к полям.

Таким образом можно объявить приватную секцию:

```
struct Route {  
    private:  
        string source;  
        string destination;  
        int length;  
};
```

Теперь к данным полям нет доступа снаружи класса:

```
Route route;
route.source = "Moscow";
    // Раньше компилировалось, теперь нет
cout << route.length;
    // Так тоже нельзя: запрещён любой доступ
```

Теперь структура абсолютно бесполезна, потому что в публичном доступе ничего нет. Для того, чтобы обратиться к приватным полям, нужно использовать методы.

### 3.3.7. Методы

Можно дописать методы к структуре, чтобы она стала более функциональной:

```
struct Route {
    public:
        string GetSource() { return source; }
        string GetDestination() { return destination; }
        int GetLength() { return length; }

    private:
        string source;
        string destination;
        int length;
};
```

Методы очень похожи на функции, но привязаны к конкретному классу. И когда эти методы вызываются, они будут работать в контексте какого-то конкретного объекта.

Определение метода похоже на определение функции, но производится внутри класса. Нужно сначала записать возвращаемый тип, затем название метода, а после, в фигурных скобках, тело метода.

Теперь созданные методы можно использовать следующим образом:

```
Route route;

route.GetSource() = "Moscow";
    // Бесполезно, поле не изменится

cout << route.GetLength();
    // Так теперь можно: доступ на чтение
```

```
int destination_name_length =  
    route.GetDestination().length();  
    // И так можно
```

Отличия методов от функций:

- Методы вызываются в контексте конкретного объекта.
- Методы имеют доступ к приватным полям (и приватным методам) объекта. К ним можно обращаться просто по названию поля.

На самом деле, структура с добавленными приватной, публичной секциями и методами — это формально уже не структура, а класс. Поэтому вместо ключевого слова `struct` лучше использовать `class`:

```
class Route { // class вместо struct  
public:  
    string GetSource() { return source; }  
    string GetDestination() { return destination; }  
    int GetLength() { return length; }  
  
private:  
    string source;  
    string destination;  
    int length;  
};
```

Программа будет работать точно так же, как если бы это не делать. Но это увеличит читаемость кода, так как существует следующая договоренность:

**Структура (struct)** — набор публичных полей. Используется, если не нужно контролировать консистентность. Типичный пример структуры:

```
struct Point {  
    double x;  
    double y;  
};
```

**Класс (class)** скрывает данные и предоставляет определенный интерфейс доступа к ним. Используется, если поля связаны друг с другом и эту связь нужно контролировать. Пример класса — класс `Route`, описанный выше.

### 3.3.8. Контроль консистентности

В обсуждаемом примере поля класса `Route` были сделаны приватными, чтобы использование класса было более безопасным. Планируется, что класс сам при необходимости будет, например, обновлять длину маршрута. Чтобы предоставить способ для изменения полей, нужно написать еще несколько публичных методов:

**SetSource** — позволяет изменить начало маршрута.

**SetDestination** — позволяет изменить пункт назначения.

В каждом из этих методов нужно не забыть обновить длину маршрута. Это лучше всего сделать с помощью метода `UpdateLength`, который будет доступен только внутри класса, то есть будет приватным методом.

В итоге код класса будет выглядеть следующим образом:

```
class Route {
public:
    string GetSource() {
        return source;
    }
    string GetDestination() {
        return destination;
    }
    int GetLength() {
        return length;
    }
    void SetSource(const string& new_source) {
        source = new_source;
        UpdateLength();
    }
    void SetDestination(const string& new_destination) {
        destination = new_destination;
        UpdateLength();
    }

private:
    void UpdateLength() {
        length = ComputeDistance(source, destination);
    }
    string source;
    string destination;
    int length;
};
```

Таким образом, создан полноценный класс, который можно использовать, например, так:

```
Route route;
route.SetSource("Moscow");
route.SetDestination("Dubna");
cout << "Route from " <<
    route.GetSource() << " to " <<
    route.GetDestination() << " is " <<
    route.GetLength() << " meters long";
```

Итак, смысловая связь между полями класса контролируется в методах.

### 3.3.9. Константные методы

Попробуем написать функцию, которая будет что-то делать с нашим классом. Например, функцию, которая распечатает информацию о маршруте:

```
void PrintRoute(const Route& route) {
    cout << route.GetSource() << " - " <<
        route.GetDestination() << endl;
}
```

Маршрут принимается по константной ссылке, чтобы лишний раз не копировать объект.

Создадим маршрут и вызовем эту функцию:

```
int main() {
    Route route;
    PrintRoute(route);
    return 0;
}
```

При попытке запуска кода появляется ошибка.

Дело в том, что в методе `GetSource` нигде явно не указано, что он не меняет объект. С другой стороны, в функцию `PrintRoute` объект `route` передается по константной ссылке, то есть функция `PrintRoute` не имеет право изменять этот объект. Поэтому компилятор не дает вызывать те методы, для которых не указано явно, что объект они не меняют.

Чтобы указать, что метод не меняет объект, нужно объявить метод константным. То есть дописать ключевое слово `const`:

```
class Route {
public:
```



```

string GetSource() const {
    return source;
}
string GetDestination() const {
    return destination;
}
int GetLength() const {
    return length;
}

```

Также давайте добавим начало и конец маршрута, чтобы вывод был интереснее:

```

int main() {
    Route route;
    route.SetSource("Moscow");
    route.SetDestination("Vologda");
    PrintRoute(route); // Выведет Moscow - Vologda
    return 0;
}

```

Теперь все работает. Итак, константными следует объявлять все методы, которые не меняют объект.

Если попытаться объявить константным метод, который меняет объект, компилятор выдаст сообщение об ошибке. Сообщения об ошибках, как правило, понятны, но в случае ошибок с константностью — не всегда. Поэтому следует запомнить, что ошибка «passing ... discards qualifiers» значит, что имеет место проблема с константностью. Также нельзя вызывать не константные методы для объекта, переданного по константной ссылке.

Следующая функция переворачивает маршрут. Она принимает значение по не константной ссылке, потому что объект будет изменен.

```

void ReverseRoute(Route& route) {
    string old_source = route.GetSource();
    string old_destination = route.GetDestination();
    route.SetSource(old_destination);
    route.SetDestination(old_source);
}

```

Этот пример демонстрирует то, что по не константной ссылке можно вызывать как константные, так и не константные методы.

```

ReverseRoute(route);
PrintRoute(route);

```

### 3.3.10. Параметризованные конструкторы

Чтобы сделать классы более удобными в использовании, можно использовать так называемые конструкторы.

Допустим, нужно создать маршрут между конкретными городами. Можно сделать это, например, с помощью уже известного синтаксиса:

```
Route route;  
route.SetSource("Zvenigorod");  
route.SetDestination("Istra");
```

Недостаток такого способа: для такой простой задачи нужно написать три строчки кода. Избавиться от этого недостатка можно написав функцию, которая будет создавать маршрут:

```
Route BuildRoute(  
    const string& source,  
    const string& destination) {  
    Route route;  
    route.SetSource(source);  
    route.SetDestination(destination);  
    return route;  
}
```

```
Route route = BuildRoute("Zvenigorod", "Istra");
```

Такое решение этой очень распространенной проблемы весьма искусственно и выглядит подозрительным. Действительно, имя BuildRoute не стандартизировано: может быть функция CreateTrain или MakeLecture. По названию класса становится невозможно понять, как называется та самая функция, которая создает объекты данного класса.

В C++ существует готовое решение для этой проблемы — конструкторы, которые уже встречались ранее для встроенных типов данных:

```
vector<string> names(5);    // Вектор из 5 пустых строк  
string spaces(10, ' ');    // Строка из 10 пробелов
```

Хочется, чтобы и в случае пользовательского типа можно было сделать как-то похоже:

```
Route route("Zvenigorod", "Istra");    // Не умеем
```

Чтобы такой код работал, нужно написать конструктор.

Конструктор — это специальный метод класса без возвращаемого значения, название которого совпадает с названием класса.

Конструктор, который принимает названия двух городов, можно написать, вызывая в его теле методы SetSource и SetDestination:

```

class Route {
public:
    Route(const string& new_source,
          const string& new_destination) {
        SetSource(new_source);
        SetDestination(new_destination);
    }
};

Route route("Zvenigorod", "Istra");
    // Теперь работает
cout << "Route from Zvenigorod to Istra " <<
     "has length " << route.GetLength() << "\n";

```

Строго говоря, в таком случае метод `UpdateLength` вызывается дважды, причем один раз еще до того, как значение конечного пункта маршрута не было установлено. Поэтому в конструкторе не стоит использовать методы, созданные для использования вне класса. Внутри конструктора можно просто проинициализировать поля нужными значениями непосредственно, а после этого вызывать метод `UpdateLength` всего один раз.

```

class Route {
public:
    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
        UpdateLength();
    }
    // ...
};

```

### 3.3.11. Конструкторы по умолчанию, использование конструкторов

Если для класса был написан параметризованный конструктор, создание переменной без параметров уже не будет работать.

```

class Route {
public:
    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
    }
};

```

```

        UpdateLength();
    }
    // ...
};

Route route;  // Теперь не компилируется

```

Чтобы исправить это, нужно дописать так называемый конструктор по умолчанию.

```

class Route {
public:
    Route() {}  // Раньше компилятор делал это сам
    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
        UpdateLength();
    }
};

```

Если по умолчанию не нужно как-то инициализировать поля, тело конструктора по умолчанию можно оставить пустым. Если для класса (никакой) конструктор не указан, компилятор создает пустой конструктор самостоятельно.

Если необходимо, чтобы по умолчанию поля были заполнены определенными значениями, это можно указать в конструкторе по умолчанию:

```

class Route {
public:
    Route() {
        source = "Moscow";
        destination = "Saint Petersburg";
        UpdateLength();
    }
    // ...
};

Route route;  // Маршрут от Москвы до СПб

```

Если переменная объявляется без указания параметров, то используется конструктор по умолчанию:

```

Route route1;
    // По умолчанию: Москва - Петербург

```

Если после названия переменной в круглых скобках указаны некоторые параметры, то вызывается параметризованный конструктор:

```
Route route2("Zvenigorod", "Istra");  
    // Параметризованный
```

Если маршрут по умолчанию нужно передать в функцию, которая принимает объект по константной ссылке:

```
void PrintRoute(const Route& route);
```

то в качестве объекта можно передать пустые фигурные скобки:

```
PrintRoute(Route()); // По умолчанию  
PrintRoute({});      // Тип понятен из заголовка функции
```

Если же нужно передать произвольный объект, аргументы параметризованного конструктора можно перечислить в фигурных скобках без указания типа:

```
PrintRoute(Route("Zvenigorod", "Istra"));  
PrintRoute({"Zvenigorod", "Istra"});
```

На самом деле, такой синтаксис можно использовать и для встроенных в язык функций и методов:

```
vector<Route> routes;  
routes.push_back({"Zvenigorod", "Istra"});
```

А также, когда необходимо вернуть объект в результате работы функции:

```
Route GetRoute(bool is_empty) {  
    if (is_empty) {  
        return {};  
    } else {  
        return {"Zvenigorod", "Istra"};  
    }  
}
```

Компилятор уже видит, объект какого типа функция возвращает, поэтому в return параметры конструктора можно указать в фигурных скобках, или написать пустые фигурные скобки для использования конструктора по умолчанию.

Аналогично можно делать и в случае встроенных типов:

```
vector<int> GetNumbers(bool is_empty) {  
    if (is_empty) {  
        return {};  
    } else {  
        return {8, 6, 9, 6};  
    }  
}
```

### 3.3.12. Значения по умолчанию для полей структур

Как правило, конструкторы в структурах не нужны. Создавать объект можно и с помощью синтаксиса с фигурными скобками:

```
struct Lecture {
    string title;
    int duration;
};

Lecture lecture = {"ООР", 5400};
// ОК, работало и без конструкторов
```

Но в некоторых случаях могло бы быть полезным использование конструктора по умолчанию для структур. Оказывается, что если нужен только конструктор по умолчанию, достаточно задать значения по умолчанию для полей:

```
struct Lecture {
    string title = "C++";
    int duration = 0;
};
```

Тогда при создании переменной без инициализации будут использоваться значения по умолчанию:

```
Lecture lecture;
cout << lecture.title << " " << lecture.duration << "\n";
// Выведет <<C++ 0>>
```

При этом все еще доступен синтаксис с фигурными скобками:

```
Lecture lecture2 = {"ООР", 5400};
```

Также можно не указывать несколько последних полей:

```
Lecture lecture3 = {"ООР"};
```

В этом случае для них будут использоваться значения по умолчанию.

### 3.3.13. Деструкторы

Деструктор — специальный метод класса, который вызывается при уничтожении объекта. Его назначение — откат действий, сделанных в конструкторе и других методах: закрытие открытого файла и освобождение выделенной вручную памяти. Название деструктора состоит из символа тильды (~) и названия класса.

Также в деструкторе можно осуществлять любые другие действия, например, вывод информации. На практике писать деструктор самому нужно очень редко. Как правило, достаточно использовать деструктор, который генерируется компилятором.

Рассмотрим созданный ранее класс Route:

```
class Route {
public:
    Route() {
        source = "Moscow";
        destination = "Saint Petersburg";
        UpdateLength();
    }
    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
        UpdateLength();
    }
    string GetSource() const {
        return source;
    }
    string GetDestination() const {
        return destination;
    }
    int GetLength() const {
        return length;
    }
    void SetSource(const string& new_source) {
        source = new_source;
        UpdateLength();
    }
    void SetDestination(const string& new_destination) {
        destination = new_destination;
        UpdateLength();
    }
}

private:
    void UpdateLength() {
        length = ComputeDistance(source, destination);
    }
    string source;
    string destination;
```

```

    int length;
};

```

Для демонстрационных целей в качестве ComputeDistance можно использовать простую заглушку:

```

int ComputeDistance(const string& source,
                   const string& destination) {
    return source.length() - destination.length();
}

```

Реально же ComputeDistance может содержать запросы к базе данных, сложные вычисления и так далее, то есть может выполняться долго. Поэтому при написании программы имеет смысл минимизировать количество вызовов ComputeDistance.

Создадим лог вызовов функции ComputeDistance:

```

private:
    void UpdateLength() {
        length = ComputeDistance(source, destination);
        compute_distance_log.push_back(
            source + " - " + destination);
    }
    string source;
    string destination;
    int length;
    vector<string> compute_distance_log;
};

```

В деструкторе объекта теперь можно сделать так, чтобы этот лог выводился в печать перед уничтожением объекта.

```

class Route {
public:
    Route() {
        source = "Moscow";
        destination = "Saint Petersburg";
        UpdateLength();
    }
    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
        UpdateLength();
    }
}

```



```

~Route() {
    for (const string& entry : compute_distance_log) {
        cout << entry << "\n";
    }
}

```

Теперь посмотрим, что выведет такой код:

```

Route route("Moscow", "Saint Petersburg");
route.SetSource("Vyborg");
route.SetDestination("Vologda");

```

```

Moscow — Saint Petersburg
Vyborg — Saint Petersburg
Vyborg — Vologda

```

### 3.3.14. Время жизни объекта

С помощью отладочной информации изучим то, как и когда уничтожаются объекты в разных ситуациях. Добавим отладочную печать во все конструкторы и деструкторы:

```

class Route {
public:
    Route() {
        source = "Moscow";
        destination = "Saint Petersburg";
        UpdateLength();
        cout << "Default constructed\n";
    }

    Route(const string& new_source,
          const string& new_destination) {
        source = new_source;
        destination = new_destination;
        UpdateLength();
        cout << "Constructed\n";
    }

    ~Route() {
        cout << "Destructed\n";
    }

    string GetSource() const {
        return source;
    }
}

```

```

    string GetDestination() const {
        return destination;
    }
    int GetLength() const {
        return length;
    }
    void SetSource(const string& new_source) {
        source = new_source;
        UpdateLength();
    }
    void SetDestination(const string& new_destination) {
        destination = new_destination;
        UpdateLength();
    }

private:
    void UpdateLength() {
        length = ComputeDistance(source, destination);
    }
    string source;
    string destination;
    int length;
};

```

Выполним следующий код:

```

for (int i : {0, 1}) {
    cout << "Step " << i << ": " << 1 << "\n";
    Route route;
    cout << "Step " << i << ": " << 2 << "\n";
}
cout << "End\n";

```

Результат его выполнения:

```

Step 0: 1
Default constructed
Step 0: 2
Destructed
Step 1: 1
Default constructed
Step 1: 2
Destructed
End

```

На каждой итерации, как только объект выходит из своей зоны видимости, он уничтожается. При уничтожении объекта вызывается деструктор.

```

int main() {
    cout << 1 << "\n";
    Route first_route;
    if (false) {
        cout << 2 << "\n";
        return 0;
    }
    cout << 3 << "\n";
    Route second_route;
    cout << 4 << "\n";
    return 0;
}

```

```

1
Default constructed
3
Default constructed
4
Destructed
Destructed

```

Компилятор уничтожает объекты в обратном порядке относительно того, как они создавались. Объект, который был создан вторым, уничтожается первым.

Теперь отправим на выполнение такой код:

```

void Worthless(Route route) {
    cout << 2 << "\n";
}

int main() {
    cout << 1 << "\n";
    Worthless({});
    cout << 3 << "\n";
    return 0;
}

```

Результат будет следующий:

```

1
Default constructed
2
Destructed
3

```

```

Route GetRoute() {
    cout << 1 << "\n";
}

```

```

    return {};
}

int main() {
    Route route = GetRoute();
    cout << 2 << "\n";
    return 0;
}

```

```

1
Default constructed
2
Destructed

```

Если результат вызова функции не сохраняется, результат получается иной:

```

Route GetRoute() {
    cout << 1 << "\n";
    return {};
}

int main() {
    GetRoute();
    cout << 2 << "\n";
    return 0;
}

```

```

1
Default constructed
Destructed
2

```

Это связано с тем, что созданная в функции переменная не может быть использована после выполнения этой функции. Она никуда не была сохранена, поэтому она сразу же была уничтожена.

## Неделя 4

# Потоки. Исключения. Перегрузка операторов

### 4.1. ООП: Примеры

#### 4.1.1. Практический пример: класс «Дата»

Часто приходится иметь дело с датами. Дата представляет собой три целых числа. Логично написать структуру:

```
struct Date {  
    int day;  
    int month;  
    int year;  
}
```

Эту структуру можно использовать следующим образом:

```
Date date = {10, 11, 12};
```

Напишем функцию PrintDate, которая принимает дату по константной ссылке (чтобы избежать лишнего копирования):

```
void PrintDate(const Date& date) {  
    cout << date.day << "." << date.month << "."  
        << date.year << "\n";  
}
```

Вывести созданную выше дату можно следующим образом:

```
PrintDate(date);
```

Существует некоторая путаница при таком способе инициализации переменной типа Date. Не понятно, если не видно определения структуры,

какая дата имеется в виду. По такой записи нельзя сразу сказать, где день, месяц и год.

Решить эту проблему можно, создав конструктор. Причем конструктор должен будет принимать не три целых числа в качестве параметров (так как будет точно такая же путаница), а «обертки» над ними: объект типа Day, объект типа Month и объект типа Year.

```
struct Day {  
    int value;  
};  
  
struct Month {  
    int value;  
};  
  
struct Year {  
    int value;  
};
```

Эти типы представляют собой простые структуры с одним полем. Конструктор типа Date имеет вид:

```
struct Date {  
    int day;  
    int month;  
    int year;  
  
    Date(Day new_day, Month new_month, Year new_year) {  
        day = new_day.value;  
        month = new_month.value;  
        year = new_year.value;  
    }  
};
```

После этого прежняя запись перестает работать:

```
error: could not convert '{10, 11, 12}' from '<brace-enclosed  
initializer list>' to 'Date'
```

Это вынуждает записать такой код:

```
Date date = {Day(10), Month(11), Year(12)};
```

По этому коду мы явно видим, где месяц, день или год. Если перепутать местами месяц и день, компилятор выдаст сообщение об ошибке:

```
could not convert '{Month(11), Day(10), Year(12)}' from '<brace-  
enclosed initializer list>' to 'Date'
```

Однако легко забыть, что все это делалось для лучшей читаемости кода, и «улучшить» код, удалив явные указания типов Day, Month, Year.

```
Date date = {{10}, {11}, {12}};
```

При этом он продолжает компилироваться.

Чтобы сделать код более устойчивым к таким «улучшениям», напишем конструкторы для структур Day, Month, Year.

```
struct Day {
    int value;
    Day(int new_value) {
        value = new_value;
    }
};

struct Month {
    int value;
    Month(int new_value) {
        value = new_value;
    }
};

struct Year {
    int value;
    Year(int new_value) {
        value = new_value;
    }
};
```

Пока что лучше не стало: нежелательный синтаксис все еще можно использовать. Стало даже еще хуже: теперь можно опустить внутренние фигурные скобки (как было в исходном варианте).

```
Date date = {10, 11, 12};
```

Написав такие конструкторы, мы разрешили компилятору неявно преобразовывать целые числа к типам Day, Month, Year. Чтобы избежать неявного преобразования типов, нужно указать компилятору, что так делать не надо, используя ключевое слово `explicit`.

```
struct Day {
    int value;
    explicit Day(int new_value) {
        value = new_value;
    }
};
```

```

    }
};

struct Month {
    int value;
    explicit Month(int new_value) {
        value = new_value;
    }
};

struct Year {
    int value;
    explicit Year(int new_value) {
        value = new_value;
    }
};

```

Ключевое слово `explicit` не позволяет вызывать конструктор неявно.

#### 4.1.2. Класс `Function`: Описание проблемы

Допустим, при реализации поиска по изображениям ставится задача упорядочить результаты поисковой выдачи, учитывая качество и свежесть картинок. Таким образом, каждая картинка характеризуется двумя полями:

```

struct Image {
    double quality;
    double freshness;
};

```

Учет этих двух полей при формировании поисковой выдачи производится с помощью функции `ComputeImageWeight` и зависит от набора параметров:

```

struct Params {
    double a;
    double b;
};

```

Вес изображения мы определяем следующим образом:

$$weight = quality - freshness * a + b$$

Ниже дан код функции `ComputeImageWeight`:



```
double ComputeImageWeight(const Params& params,
                          const Image& image) {
    double weight = image.quality;
    weight -= image.freshness * params.a + params.b;
    return weight;
}
```

После этого оказывается, что нужно также учесть рейтинг изображения, то есть каждая картинка характеризуется уже тремя полями:

```
struct Image {
    double quality;
    double freshness;
    double rating;
};
```

Учет рейтинга при формировании веса изображения контролируется с помощью нового параметра:

```
struct Params {
    double a;
    double b;
    double c;
};
```

И производится также в функции ComputeImageWeight:

```
double ComputeImageWeight(const Params& params,
                          const Image& image) {
    double weight = image.quality;
    weight -= image.freshness * params.a + params.b;
    weight += image.rating * params.c;
    return weight;
}
```

Если вдруг кроме функции ComputeImageWeight существует функция ComputeQualityByWeight:

```
double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
    double quality = weight;
    quality += image.freshness * params.a + params.b;
    return quality;
}
```

то нужно не забыть внести изменения и в нее тоже:

```
double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
    double quality = weight;
    quality -= image.rating * params.c;
    quality += image.freshness * params.a + params.b;
    return quality;
}
```

В данном случае присутствует неявное дублирование кода: существует некоторый способ вычисления веса изображения от его качества. Он представлен в коде два раза: в функции `ComputeImageWeight` и в функции `ComputeQualityByWeight`.

### 4.1.3. Класс `Function`: Описание

Чтобы убрать дублирование, нужно для сущности «способ вычисления веса изображения от его качества» написать некоторый класс. По сути, эта сущность есть некоторая функция, поэтому реализовывать нужно класс `Function` и функцию `MakeWeightFunction`, которая будет возвращать нужную функцию.

```
Function MakeWeightFunction(const Params& params,
                           const Image& image) {
    ...
}
```

Функции `ComputeImageWeight` и `ComputeQualityByWeight` следует переписать следующим образом:

```
double ComputeImageWeight(const Params& params,
                          const Image& image) {
    Function function = MakeWeightFunction(params, image);
    return function.Apply(image.quality);
}

double ComputeQualityByWeight(const Params& params,
                              const Image& image,
                              double weight) {
    Function function = MakeWeightFunction(params, image);
    function.Invert();
    return function.Apply(weight);
}
```

При этом в функции `ComputeQualityByWeight` нужно использовать обратную функцию.

Функция `MakeWeightFunction`, таким образом, должна быть реализована следующим образом:

```
Function MakeWeightFunction(const Params& params,
                           const Image& image) {
    Function function;
    function.AddPart('-',
                    image.freshness * params.a + params.b);
    function.AddPart('+', image.rating * params.c);
    return function;
}
```

Метод `AddPart` добавляет часть функции к объекту типа `Function`.

#### 4.1.4. Класс `Function`: Реализация

Реализуем класс `Function`. Этот класс обладает методами:

**Метод `AddPart`** — добавляет очередную часть в функцию. Принимает два аргумента: символ операции и вещественное число.

**Метод `Apply`** — возвращает вещественное число, применяя текущую функцию к некоторому числу. Это константный метод, так как он не должен менять функцию.

**Метод `Invert`** — заменяет текущую функцию на обратную.

**Приватное поле `parts`** — набор элементарных операций. Каждая элементарная операция представляет собой объект типа `FunctionPart`.

В качестве конструктора используется конструктор по умолчанию.

В классе `FunctionPart` понадобится:

**Конструктор** — принимает на вход символ операции и операнд (вещественное число). Сохраняет эти значения в приватных полях.

**Метод `Apply`** — применяет операцию к некоторому числу. Константный метод.

**Метод `Invert`** — инвертирует элементарную операцию.

Теперь можно привести реализации обоих классов:

```

class FunctionPart {
public:
    FunctionPart(char new_operation, double new_value) {
        operation = new_operation;
        value = new_value;
    }
    double Apply(double source_value) const {
        if (operation == '+') {
            return source_value + value;
        } else {
            return source_value - value;
        }
    }
    void Invert() {
        if (operation == '+') {
            operation = '-';
        } else {
            operation = '+';
        }
    }
};

private:
    char operation;
    double value;
};

```

```

class Function {
public:
    void AddPart(char operation, double value){
        parts.push_back({operation, value});
    }
    double Apply(double value) const {
        for (const FunctionPart& part : parts) {
            value = part.Apply(value);
        }
        return value;
    }
    void Invert() {
        for (FunctionPart& part : parts) {
            part.Invert();
        }
        reverse(begin(parts), end(parts));
    }
}

```

```
private:
    vector<FunctionPart> parts;
};
```

#### 4.1.5. Класс Function: Использование

Проверим, как работают созданные классы. Создадим изображение (объект класса Image) и проинициализируем его поля:

```
Image image = {10, 2, 6};
```

Вычисление веса изображения невозможно без задания параметров формулы. Создадим объект типа Params:

```
Params params = {4, 2, 6};
```

Подсчитаем вес изображения с помощью функции ComputeImageWeight:

```
// 10 - 2 * 4 - 2 + 6 * 6 = 36
cout << ComputeImageWeight(params, image) << endl;
```

В результате получим:

36

Теперь протестируем функцию ComputeQualityByWeight:

```
// 20 - 2 * 4 - 2 + 6 * 6 = 46
cout << ComputeQualityByWeight(params, image, 46) << endl;
```

На выходе имеем, чему должно быть равно качество изображения:

20

Таким образом, код работает успешно.

## 4.2. Работа с текстовыми файлами

### 4.2.1. Потоки в языке C++

Стандартная библиотека обеспечивает гибкий и эффективный метод обработки целочисленного, вещественного, а также символьного ввода через консоль, файлы или другие потоки. А также позволяет гибко расширять способы ввода для типов, определенных пользователем.

Существуют следующие базовые классы:

**istream** поток ввода (cin)

**ostream** поток вывода (cout)

**iostream** поток ввода/вывода

Все остальные классы, о которых пойдет речь далее, от них наследуются.

Классы, которые работают с файловыми потоками:

**ifstream** для чтения (наследник istream)

**ofstream** для записи (наследник ostream)

**fstream** для чтения и записи (наследник iostream)

### 4.2.2. Чтение из потока построчно

Чтение из потока производится с помощью оператора ввода ( $\gg$ ) или функции `getline`, которая позволяет читать данные из потока построчно.

Пусть заранее создан файл со следующим содержимым:

```
hello world!  
second line
```

Для работы с файлами нужно подключить библиотеку `fstream`:

```
#include <fstream>
```

Чтобы считать содержимое файла следует объявить экземпляр класса `ifstream`:

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

В качестве аргумента конструктора указывается путь до желаемого файла.

Далее можно создать строковую переменную, в которую будет записан результат чтения из файла:

```
string line;
```

Функция `getline` первым аргументом принимает поток, из которого нужно прочитать данные, а вторым — переменную, в которую их надо записать. Чтобы проверить, что все работает, можно вывести переменную `line` на экран:

```
getline(input, line);  
cout << line << endl;
```

Чтобы считать и вторую строчку, можно попробовать запустить следующий код:

```
getline(input, line);  
cout << line << endl;  
  
getline(input, line);  
cout << line << endl;
```

Если вызвать `getline` в третий раз, то она не изменит переменную `line`, так как уже достигнут конец файла и из него ничего не может быть прочитано:

```
getline(input, line);  
cout << line << endl;  
  
getline(input, line);  
cout << line << endl;  
  
getline(input, line);  
cout << line << endl;
```

Чтобы избежать таких ошибок, следует помнить, что `getline` возвращает ссылку на поток, из которого берет данные. Поток можно привести к типу `bool`, причем `false` будет в случае, когда с потоком уже можно дальше не работать.

Переписать код так, чтобы он выводил все строчки из файла и ничего лишнего, можно так:

```
ifstream input("hello.txt");  
string line;  
while (getline(input, line)) {  
    cout << line << endl;  
}
```

Следует обратить внимание, что переводы строки при выводе добавлены искусственно. Это связано с тем, что функция `getline`, на самом деле, считывает данные до некоторого разделителя, причем по умолчанию до символа перевода строки, который в считанную строку не попадает.

### 4.2.3. Обработка случая, когда указанного файла не существует

Рассмотрим ситуацию, когда по некоторым причинам неверно указано имя файла или файла с таким именем не может существовать в файловой системе. Например, внесем опечатку:

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

При запуске этого кода оказывается, что он работает, ничего не выводит, но никак не сигнализирует о наличии ошибки.

Вообще говоря, желательно, чтобы программа не умалчивала об этом, а явно сообщала, что файла не существует и из него нельзя прочесть данные.

У файловых потоков существует метод `is_open`, который возвращает `true`, если файловый поток открыт и готов работать. Программу, таким образом, следует переписать так:

```
ifstream input("helol.txt");
string line;

if (input.is_open()){
    while (getline(input, line)) {
        cout << line << endl;
    }
    cout << "done!" << endl;
} else {
    cout << "error!" << endl;
}
```

Следует также отметить, что файловые потоки можно приводить к типу `bool`, причем значение `true` соответствует тому, что с потоком можно работать в данный момент. Другими словами, код можно переписать в следующем виде:

```
ifstream input("helol.txt");
string line;

if (input){
    while (getline(input, line)) {
        cout << line << endl;
    }
    cout << "done!" << endl;
} else {
    cout << "error!" << endl;
}
```



#### 4.2.4. Чтение из потока до разделителя

Научимся считывать данные с помощью `getline` поблочно с некоторым разделителем. Например, в качестве разделителя может выступать символ «минус». Допустим, считать нужно дату из следующего текстового файла `date.txt`:

2017–01–25

Для этого создадим:

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

Объявим строковые переменные `year`, `month`, `day`.

```
string year;  
string month;  
string day;
```

Нужно считать файл таким образом, чтобы соответствующие части файла попали в нужную переменную. Воспользуемся функцией `getline` и укажем разделитель:

```
if (input) {  
    getline(input, year, '-');  
    getline(input, month, '-');  
    getline(input, day, '-');  
}
```

Чтобы проверить, что все работает, выведем переменную на экран через пробел:

```
cout << year << ' ' << month << ' ' << day << endl;
```

#### 4.2.5. Оператор чтения из потока

Решим ту же самую задачу с помощью оператора чтения из потока (`>>`). Записывать считанные данные будем в переменные типа `int`.

```
ifstream input("date.txt");  
int year = 0;  
int month = 0;  
int day = 0;  
if (input) {  
    input >> year;  
    input.ignore(1);  
    input >> month;
```

```

        input.ignore(1);
        input >> day;
        input.ignore(1);
    }
    cout << year << ' ' << month << ' ' << day << endl;

```

После того, как из потока будет считан год, следующим символом будет «минус», от которого нужно избавиться. Это можно сделать с помощью метода `ignore`, который принимает целое число — сколько символов нужно пропустить. Аналогично считываются месяц и день. Получается такой же результат.

То, каким методом пользоваться, зависит от ситуации. Иногда бывает удобнее сперва считать всю строку целиком.

#### 4.2.6. Оператор записи в поток. Дозапись в файл.

Данные в файл можно записывать с помощью класса `ofstream`:

```

const string path = "output.txt";

ofstream output(path);
output << "hello" << endl;

```

После проверим, что записалось в файл, открыв его и прочитав содержимое:

```

ifstream input(path);
if (input) {
    string line;
    while (getline(input, line)) {
        cout << line << endl;
    }
}

```

Чтобы избежать дублирования кода, имеет смысл создать переменную `path`.

Для удобства можно создать функцию, которая будет считывать весь файл:

```

void ReadAll(const string& path) {
    ifstream input(path);
    if (input) {
        string line;
        while (getline(input, line)) {
            cout << line << endl;
        }
    }
}

```

```
    }
  }
}
```

Предыдущая программа примет вид:

```
const string path = "output.txt";

ofstream output(path);
output << "hello" << endl;

ReadAll(path);
```

Следует отметить, что при каждом запуске программы файл записывается заново, то есть его содержимое удалялось и запись начиналась заново.

Для того, чтобы открыть файл в режиме дозаписи, нужно передать специальный флажок `ios::app` (от англ. *append*):

```
ofstream output(path, ios::app);
output << " world!" << endl;
```

#### 4.2.7. Форматирование вывода. Файловые манипуляторы.

Допустим, нужно в определенном формате вывести данные. Это могут быть имена колонок и значения в этих колонках.

Сохраним в векторе `names` имена колонок и после этого создадим вектор значений:

```
vector<string> names = {"a", "b", "c"};
vector<double> values = {5, 0.01, 0.000005};
```

Выведем их на экран:

```
for (const auto& n : names) {
    cout << n << ' ';
}
cout << endl;
for (const auto& v : values) {
    cout << v << ' ';
}
cout << endl;
```

При этом читать значения очень неудобно.

Для того, чтобы решить такую задачу, в языке C++ есть файловые манипуляторы, которые работают с потоком и изменяют его поведение. Для того, чтобы с ними работать, нужно подключить библиотеку `iomanip`.

**fixed** Указывает, что числа далее нужно выводить на экран с фиксированной точностью.

```
cout << fixed;
```

**setprecision** Задаёт количество знаков после запятой.

```
cout << fixed << setprecision(2);
```

**setw** (set width) Указывает ширину поля, которое резервируется для вывода переменной.

```
cout << fixed << setprecision(2);  
cout << setw(10);
```

Этот манипулятор нужно использовать каждый раз при выводе значения, так как он сбрасывается после вывода следующего значения:

```
for (const auto& n : names) {  
    cout << setw(10) << n << ' '  
}  
cout << endl;  
cout << fixed << setprecision(2);  
for (const auto& v : values) {  
    cout << setw(10) << v << ' '  
}
```

Здесь колонки были выведены в таком же формате.

**setfill** Указывает, каким символом заполнять расстояние между колонками.

```
cout << setfill(' ');
```

**left** Выравнивание по левому краю поля.

```
cout << left;
```

Для удобства напомним функцию, которая будет на вход принимать вектора имен и значений, и выводить их в определенном формате:

```

void Print(const vector<string>& names,
           const vector<double>& values, int width) {
    for (const auto& n : names) {
        cout << setw(width) << n << ' ';
    }
    cout << endl;
    cout << fixed << setprecision(2);
    for (const auto& v : values) {
        cout << setw(width) << v << ' ';
    }
    cout << endl;
}

```

Покажем как пользоваться манипуляторами setfill и left:

```

cout << setfill('.');
cout << left;
Print(names, values, 10);

```

## 3.4. Перегрузка операторов для пользовательских типов

В данном видео будет рассмотрено, как сделать работу с пользовательскими структурами и классами более удобной и похожей на работу со стандартными типами. Например, когда целое число считывается из консоли или выводится в консоль, это можно сделать очень удобно — с помощью операторов ввода и вывода.

### 3.4.1. Тип Duration (Интервал)

Рассмотрим структуру Интервал, которая включает поля: час и минута.

```
struct Duration {  
    int hour;  
    int min;  
}
```

Напишем функцию, которая будет возвращать интервал, считывая значения из потока:

```
Duration ReadDuration(istream& stream) {  
    int h = 0;  
    int m = 0;  
    stream >> h;  
    stream.ignore(1);  
    stream >> m;  
    return Duration {h, m};  
}
```

Также определим функцию PrintDuration, которая будет выводить интервал в поток.

```
void PrintDuration(ostream& stream, const Duration&  
    ↪ duration) {  
    stream << setfill('0');  
    stream << setw(2) << duration.hour << ':'  
        << setw(2) << duration.min;  
}
```

Воспользуемся функциями, сперва заведя и инициализируя строковый поток:

```
stringstream dur_ss("01:40");  
Duration dur1 = ReadDuration(dur_ss);  
PrintDuration(cout, dur1);
```

Использовать функции `ReadDuration` и `PrintDuration`, в принципе, удобно, но было бы удобнее использовать операторы ввода из потока и вывода в поток.

### 3.4.2. Перегрузка оператора вывода в поток

Определим оператор вывода в поток, который принимает в качестве первого аргумента поток, а в качестве второго константную ссылку на экземпляр объекта. Пусть (пока) он возвращает `void`:

```
void operator<<(ostream& stream, const Duration& duration) {
    stream << setfill('0');
    stream << setw(2) << duration.hour << ':'
        << setw(2) << duration.min;
}
```

Заметим, что сигнатуры функции `PrintDuration` и оператора вывода очень похожи, поэтому реализацию можно скопировать без каких-либо изменений.

Мы сделали класс гораздо удобнее для работы:

```
cout << dur1;
```

Но если мы попытаемся добавить перенос на новую строку, программа не скомпилируется.

```
cout << dur1 << endl;
```

Попытаемся понять, почему так происходит. Рассмотрим, например, следующий код:

```
cout << "hello" << " world";
```

Оператор вывода `operator<<` первым аргументом принимает поток, а вторым — строку для вывода, и возвращает поток, в который делал вывод.

Можно вызвать по цепочке два оператора вывода:

```
operator<<(operator<<(cout, "hello"), " world");
```

Поэтому оператор вывода должен возвращать не `void`, а ссылку на поток:

```
ostream& operator<<(ostream& stream, const Duration&
→ duration) {
    stream << setfill('0');
    stream << setw(2) << duration.hour << ':'
        << setw(2) << duration.min;
    return stream;
}
```

После этого код начинает работать.

### 3.4.3. Перегрузка оператора ввода из потока

Аналогичным образом определим оператор ввода из потока:

```
istream& operator>>(istream& stream, Duration& duration) {  
    stream >> duration.h;  
    stream.ignore(1);  
    stream >> duration.m;  
    return stream;  
}
```

Теперь считывать интервалы можно прямо из потока:

```
stringstream dur_ss("02:50");  
Duration dur1 {0, 0};  
dur_ss >> dur1;  
cout << dur1 << endl;
```

Оператор ввода также возвращает ссылку на поток, чтобы была возможность считывать сразу несколько переменных.

### 3.4.4. Конструктор по умолчанию

Дополнительно стоит отметить, что язык C++ позволяет задать значения структуры по умолчанию. Этого можно добиться с помощью создания конструктора по умолчанию:

```
struct Duration {  
    int hour;  
    int min;  
  
    Duration(int h = 0, int m = 0) {  
        hour = h;  
        min = m;  
    }  
}
```

### 3.4.5. Перегрузка арифметических операций

Реализуем возможность складывать интервалы естественным образом:

```
Duration dur1 = {2, 50};  
Duration dur2 = {0, 5};  
cout << dur1 + dur2 << endl;
```



Такой код еще не компилируется, поскольку не определен оператор плюс. Оператор плюс на вход принимает два объекта и возвращает их сумму:

```
Duration operation+(const Duration& lhs, const Duration&
    ↪ rhs) {
    return Duration(lhs.hour + rhs.hour, lhs.min + rhs.min);
}
```

Сокращения lhs и rhs обозначают Left/Right Hand Side.

После этого код начинает работать.

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 5};
cout << dur1 + dur2 << endl;
// OUTPUT: 02:55
```

Однако, запустим этот код для другой пары интервалов:

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
cout << dur1 + dur2 << endl;
// OUTPUT: 02:85
```

Интервал «02:85» — достаточно странный интервал. Следует сделать так, чтобы минуты всегда были от 0 до 59. Логично исправить для этого конструктор типа Duration:

```
struct Duration {
    int hour;
    int min;

    Duration(int h = 0, int m = 0) {
        int total = h * 60 + m;
        hour = total / 60;
        min = total % 60;
    }
}
```

После такого определения конструктора:

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
cout << dur1 + dur2 << endl;
// OUTPUT: 03:25
```

### 3.4.6. Сортировка. Перегрузка операторов сравнения.

Допустим, необходимо для вектора интервалов

```
Duration dur1 = {2, 50};
Duration dur2 = {0, 35};
Duration dur3 = dur1 + dur2;
vector<Duration> v {
    dur1, dur2, dur3
}
```

расположить элементы этого вектора по возрастанию.

Для удобства напомним функцию PrintVector:

```
void PrintVector(const vector<Duration>& durs) {
    for (const auto& d : durs) {
        cout << d << ' ';
    }
    cout << endl;
}
```

Использовать эту функцию можно следующим образом:

```
vector<Duration> v {
    dur1, dur2, dur3
}
PrintVector(v); // => 03:25 02:50 00:35
```

Попробуем отсортировать вектор:

```
sort(begin(v), end(v));
PrintVector(v);
```

При компиляции возникают ошибки, которые говорят о том, что оператор сравнения не определен. Можно исправить эту ошибку двумя способами:

- Определить функцию-компаратор и передать ее в качестве третьего аргумента в функцию sort.

```
bool CompareDurations(const Duration& lhs, const
    ↪ Duration& rhs) {
    if (lhs.hour == rhs.hour) {
        return lhs.min < rhs.min;
    }
    return lhs.hour < rhs.hour
}
```

Пример использования функции компаратора:

```
Duration dur1 { 1, 12 };
Duration dur2 { 1, 13 };
cout << boolalpha << CompareDurations(dur1, dur2) <<
    ↪ endl;
// OUTPUT: true
```

- Перегрузить оператор «меньше» для типа Duration. Если третий аргумент функции sort не указан, при сортировке используется он.

```
bool operator<(const Duration& lhs, const Duration&
    ↪ rhs) {
    if (lhs.hour == rhs.hour) {
        return lhs.min < rhs.min;
    }
    return lhs.hour < rhs.hour;
}
```

С помощью манипулятора потока boolalpha можно выводить в консоль значения логических переменных как true/false.

### 3.4.7. Использование перегруженных операторов в собственных структурах

Решим для примера практическую задачу. Пусть дан текстовый файл с результатами забега нескольких бегунов:

```
0:32 Bob
0:15 Mary
0:32 Jim
```

Необходимо создать файл, где бегуны будут отсортированы согласно их результату, а также дополнительно вывести бегунов, которые бежали дольше всех.

Для решения этой задачи удобно использовать структуру типа map, поскольку в этом случае автоматически поддерживается упорядоченность данных:

```
ifstream input("runner.txt");
Duration worst;
map<Duration, string> all;
if (input) {
    Duration dur;
    string name;
    while (input >> dur >> name) {
```

```

        if (worst < dur) {
            worst = dur;
        }
        all[dur] += (name + " ");
    }
}
ofstream out("result.txt");
for (const auto durationNames& : all) {
    out << durationNames.first << '\t' << durationNames.second
        << endl;
}
cout << "Worst runner: " << all[worst] << endl;
// OUTPUT: "Worst runner: Bob Jim"

```

Результирующий файл:

```

0:15  Mary
0:32  Bob Jim

```

### 3.3. Исключения в C++ (введение)

Исключение — это нестандартная ситуация, то есть когда код ожидает определенную среду и инварианты, которые не соблюдаются.

Банальный пример: функции, которая суммирует две матрицы, переданы матрицы разных размерностей. В таком случае возникает исключительная ситуация и можно «бросить» исключение.

#### 3.3.1. Практический пример: парсинг даты в заданном формате

Допустим необходимо парсить даты

```
struct Date {  
    int year;  
    int month;  
    int day;  
}
```

из входного потока.

Функция ParseDate будет возвращать объект типа Date, принимая на вход строку:

```
Date ParseDate(const string& s){  
    stringstream stream(s);  
    Date date;  
    stream >> date.year;  
    stream.ignore(1);  
    stream >> date.month;  
    stream.ignore(1);  
    stream >> date.day;  
    stream.ignore(1);  
    return date;  
}
```

В этой функции объявляется строковый поток, создается переменная типа Date, в которую из строкового потока считывается вся необходимая информация.

Проверим работоспособность этой функции:

```
string date_str = "2017/01/25";  
Date date = ParseDate(date_str)  
cout << setw(2) << setfill('0') << date.day << '.'  
      << setw(2) << setfill('0') << date.month << '.'  
      << date.year << endl; // OUTPUT: "25.01.2017"
```

Код работает. Но давайте защитимся от ситуации, когда данные на вход приходят не в том формате, который ожидается:

2017a01b25

Программа выводит ту же дату на экран. В таких случаях желательно, чтобы функция явно сообщала о неправильном формате входных данных. Сейчас функция это не делает.

От этой ситуации можно защититься, изменив возвращаемое значение на `bool`, передавая `Date` в качестве параметра для его изменения, и добавляя внутри функции нужные проверки. В случае ошибки функция возвращает `false`. Такое решение задачи очень неудобное и существенно усложняет код.

### 3.3.2. Выброс исключений в C++

В C++ есть специальный механизм для таких ситуаций, который называется исключения. Что такое исключения, можно понять на следующем примере:

```
Date ParseDate(const string& s){
    stringstream stream(s);
    Date date;
    stream >> date.year;
    if (stream.peek() != '/') {
        throw exception();
    }
    stream.ignore(1);
    stream >> date.month;
    if (stream.peek() != '/') {
        throw exception();
    }
    stream.ignore(1);
    stream >> date.day;
    return date;
}
```

Если формат даты правильный, такой код отработает без ошибок:

```
string date_str = "2017/01/25";
Date date = ParseDate(date_str);
cout << setw(2) << setfill('0') << date.day << '.'
    << setw(2) << setfill('0') << date.month << '.'
    << date.year << endl;
```

Если сделать строчку невалидной, программа упадет:

```

string date_str = "2017a01b25";
Date date = ParseDate(date_str);
cout << setw(2) << setfill('0') << date.day << '.'
      << setw(2) << setfill('0') << date.month << '.'
      << date.year << endl;

```

Чтобы избежать дублирования кода, создадим функцию, которая проверяет следующий символ и кидает исключение, если это необходимо, а затем пропускает его:

```

void EnsureNextSymbolAndSkip(stringstream& stream) {
    if (stream.peek() != '/') {
        throw exception();
    }
    stream.ignore(1);
}

```

Функция ParseDate примет вид:

```

Date ParseDate(const string& s){
    stringstream stream(s);
    Date date;
    stream >> date.year;
    EnsureNextSymbolAndSkip(stream);
    stream >> date.month;
    EnsureNextSymbolAndSkip(stream);
    stream >> date.day;
    return date;
}

```

### 3.3.3. Обработка исключений. Блок try/catch

Ситуация когда программа падает во время работы не очень желательна, поэтому нужно правильно обрабатывать все исключения. Для обработки ошибок в C++ существует специальный синтаксис:

```

try {
    /* ...код, который потенциально
       может дать исключение... */
} catch (exception&) {
    /* Обработчик исключения. */
}

```

Проверим это на практике:

```

string date_str = "2017a01b25";
try {
    Date date = ParseDate(date_str);
    cout << setw(2) << setfill('0') << date.day << '.'
         << setw(2) << setfill('0') << date.month << '.'
         << date.year << endl;
} catch (exception& ex) {
    cout << "exception happens";
}

```

Хорошо бы донести до вызывающего кода, что произошло и где произошла ошибка. Например, если отсутствует какой-то файл, указать, что файл не найден и путь к файлу. Для этого есть класс `runtime_error`:

```

void EnsureNextSymbolAndSkip(stringstream& stream) {
    if (stream.peek() != '/') {
        stringstream ss;
        ss << "expected / , but has: " << char(stream.peek());
        throw runtime_error(ss.str());
    }
    stream.ignore(1);
}

```

Если у исключения есть текст, его можно получить с помощью метода `what` исключения.

```

} catch (exception& ex) {
    cout << "exception happens: " << ex.what();
}

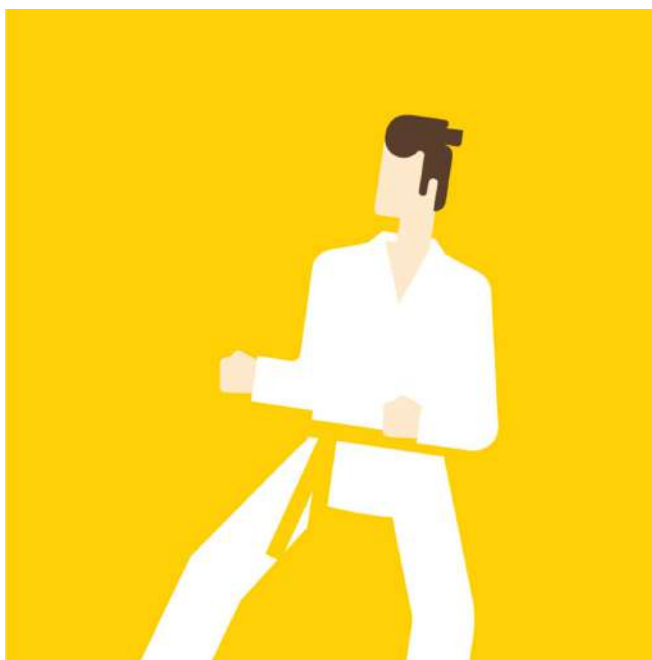
```



# Основы разработки на C++: жёлтый пояс

Неделя 1

Целочисленные типы, кортежи, шаблонные функции



# Оглавление

<b>Целочисленные типы, кортежи, шаблонные функции</b>	<b>2</b>
1.1 Целочисленные типы . . . . .	2
1.1.1 Введение в целочисленные типы . . . . .	2
1.1.2 Преобразования целочисленных типов . . . . .	5
1.1.3 Безопасное использование целочисленных типов . . . . .	7
1.2 Кортежи и пары . . . . .	10
1.2.1 Упрощаем оператор сравнения . . . . .	10
1.2.2 Кортежи и пары . . . . .	12
1.2.3 Возврат нескольких значений из функций . . . . .	15
1.3 Шаблоны функций . . . . .	17
1.3.1 Введение в шаблоны . . . . .	17
1.3.2 Универсальные функции вывода контейнеров в поток . . . . .	19
1.3.3 Рефакторим код и улучшаем читаемость вывода . . . . .	22
1.3.4 Указание шаблонного параметра-типа . . . . .	24

# Целочисленные типы, кортежи, шаблонные функции

## 1.1. Целочисленные типы

### 1.1.1. Введение в целочисленные типы

Вы уже знаете один целочисленный тип – это тип `int`. Начнём с проблемы, которая может возникнуть при работе с ним. Для этого вспомним задачу «Средняя температура» из первого курса. Нам был дан набор наблюдений за температурой, в виде вектора `t` (значения 8, 7 и 3). Нужно было найти среднее арифметическое значение температуры за все дни и затем вывести номера дней, в которые значение температуры было больше, чем среднее арифметическое. Должно получиться  $(8 + 7 + 3)/3 = 6$ .

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> t = {8, 7, 3}; // вектор с наблюдениями
    int sum = 0; // переменная с суммой
    for (int x : t) { // проитерировались по вектору и нашли суммарную температуру
        sum += x;
    }
    int avg = sum / t.size(); // получили среднюю температуру
    cout << avg << endl;
    return 0;
} // вывод программы будем писать последним комментарием листинга
// 6
```

Но в той задаче было ограничение: вам гарантировалось, что все значения температуры не

отрицательные. Если в таком решении у вас в исходном векторе будут отрицательные значения температуры, например,  $-8$ ,  $-7$  и  $3$  (ответ  $(-8 - 7 + 3)/3 = -4$ ), код работать не будет:

```
int main () {
    vector<int> t = {-8, -7, 3};    // сумму -4
    ...
    int avg = sum / t.size();    // t.size() не умеет хранить отрицательные числа
    cout << avg << endl;
    return 0;
}
// 1431655761
```

Это не  $-4$ . На самом деле мы от незнания неаккуратно использовали другой целочисленный тип языка C++. Он возникает в `t.size()` – это специальный тип, который не умеет хранить отрицательные числа. Размер контейнера отрицательным быть не может, и это беззнаковый тип. Какая еще бывает проблема с целочисленными типами? Очень простой пример:

```
int main () {
    int x = 2'000'000'000;    // для читаемости разбиваем на разряды кавычками
    cout << x << " ";    // выводим само число
    x = x * 2;
    cout << x << " ";    // выводим число, умноженное на 2
    return 0;
}
// 2000000000 -294967296
```

Итак, запускаем код и видим, что 4 миллиарда в переменную типа `int` не поместилось.

### Особенности целочисленных типов языка C++:

1. В языке C++ память для целочисленных типов ограничена. Если вам не нужны целые числа размером больше 2 миллиардов, язык C++ для вас выделит вот ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.
2. Некоторые целочисленные типы языка C++ беззнаковые. Используя их, вы сможете хранить больше положительных значений, но не сможете хранить отрицательные.

### Виды целочисленных типов:

- `int` – стандартный целочисленный тип.
  1. `auto x = 1;` – как и любая комбинация цифр имеет тип `int`;
  2. Эффективен: операции с ним напрямую транслировались в инструкции процессора;
  3. В зависимости от архитектуры имеет размер 64 или 32 бита, и диапазон его значений от  $-2^{31}$  до  $(2^{31} - 1)$ .
- `unsigned int (unsigned)` – беззнаковый аналог `int`.
  1. Диапазон его значений от 0 до  $(2^{32} - 1)$ . Занимает 8 или 4 байта.
- `size_t` – тип для представления размеров.
  1. Результат вызова `size()` для контейнера;
  2. 4 байта (до  $(2^{32} - 1)$ ) или 8 байт (до  $(2^{64} - 1)$ ). Зависит от разрядности системы.
- Типы с известным размером из модуля `cstdint`.
  1. `int32_t` – знаковый, всегда 32 бита (от  $-2^{31}$  до  $(2^{31} - 1)$ );
  2. `uint32_t` – беззнаковый, всегда 32 бита (от 0 до  $(2^{32} - 1)$ );
  3. `int8_t` и `uint8_t` всегда 8 бит; `int16_t` и `uint16_t` всегда 16 бит; `int64_t` и `uint64_t` всегда 64 бита.

Тип	Размер	Минимум	Максимум	Стоит ли выбрать его?
<code>int</code>	4 (обычно)	$-2^{31}$	$2^{31} - 1$	по умолчанию
<code>unsigned int</code>	4 (обычно)	0	$2^{32} - 1$	только положительные
<code>size_t</code>	4 или 8	0	$2^{32} - 1$ или $2^{64} - 1$	размер контейнеров
<code>int8_t</code>	1	$-2^7$	$2^7 - 1$	сильно экономить память
<code>int16_t</code>	2	$-2^{15}$	$2^{15} - 1$	экономить память
<code>int32_t</code>	4	$-2^{31}$	$2^{31} - 1$	нужно ровно 32 бита
<code>int64_t</code>	8	$-2^{63}$	$2^{63} - 1$	недостаточно <code>int</code>

## Узнаём размеры и ограничения типов:

Как узнать размеры типа? Очень просто:

```
cout << sizeof(int16_t) << " "; // размер типа в байтах. Вызывается от переменной
cout << sizeof(int) << endl;
// 2 4
```

Узнаём ограничения типов:

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main() {
    cout << sizeof(int16_t) << " ";
    cout << numeric_limits<int>::min() << " " << numeric_limits<int>::max() << endl;
    return 0;
}
// 4 -2147483648 2147483647
```

### 1.1.2. Преобразования целочисленных типов

Начнём с эксперимента. Прибавим 1 к максимальному значению типа `int`.

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main() {
    cout << numeric_limits<int>::max() + 1 << " ";
    cout << numeric_limits<int>::min() - 1 << endl;
    return 0;
}
// -2147483648 2147483647
```

Получилось, что  $\text{max} + 1 = \text{min}$ , а  $\text{min} - 1 = \text{max}$ . Теперь попробуем вычислить среднее арифметическое  $1000000000 + 2000000000$ .

```
int x = 2'000'000000;
int y = 1'000'000000;
cout << (x + y) / 2 << endl;
// -647483648
```

Хоть их среднее вмещается в `int`, но программа сначала сложила `x` и `y` и получила число, не уместившееся в тип `int`, и только после этого поделила его на 2. Если в процессе случается

переполнение, то и с результатом будет не то, что мы ожидаем. Теперь попробуем поработать с беззнаковыми типами:

```
int x = 2'000'000000;
unsigned int y = x; // сохраняем в переменную беззнакового типа 2000000000
unsigned int z = -z;
cout << x << " " << y << " " << -x << " " << z << endl;
// 2000000000 2000000000 -2000000000 2294967296
```

Если значение уместится даже в `int`, то проблем не будет. Но если записать отрицательное число в `unsigned`, мы получим не то, что ожидали. Возвращаясь к задаче о средней температуре, посмотрим, в чём была проблема:

```
vector<int> t = {-8, -7, 3};
int sum = 0; // знаковое
for (int x : t){
    sum += x;
}
int avg = sum / t.size(); // sum / t.size(); уже беззнаковое, т.к. t.size() беззнаковое
cout << avg << endl;
```

### Правила вывода общего типа:

1. Перед сравнениями и арифметическими операциями числа приводятся к общему типу;
2. Все типы размера меньше `int` приводятся к `int`;
3. Из двух типов выбирается больший по размеру;
4. Если размер одинаковый, выбирается беззнаковый.

### Примеры:

Слева	Операция	Справа	Общий тип	Комментарий
<code>int</code>	<code>/</code>	<code>size_t</code>	<code>size_t</code>	большой размер
<code>int32_t</code>	<code>+</code>	<code>int8_t</code>	<code>int32_t (int)</code>	тоже больший размер
<code>int8_t</code>	<code>*</code>	<code>uint8_t</code>	<code>int</code>	все меньшие приводятся к <code>int</code>
<code>int32_t</code>	<code>&lt;</code>	<code>uint32_t</code>	<code>uint32_t</code>	знаковый к беззнаковому

Для определения типа в самой программе можно просто вызвать ошибку компиляции и посмотреть лог ошибки. Изменим одну строчку:

```
int avg = (sum / t.size()) + vector<int>{}; // прибавили пустой вектор
```

И получим ошибку, в логе которой указано, что наша переменная `avg` имеет тип `int`. Теперь попробуем сравнить знаковое и беззнаковое число:

```
int main () {
    int x = -1;
    unsigned y = 1;
    cout << (x < y) << " ";
    cout << (-1 < 1u) << endl; // суффикс u делает 1 типом unsigned по умолчанию
    return 0;
}
// 0 0
```

Как было сказано ранее, при операции между знаковым и беззнаковым типом обе переменные приводятся к беззнаковому. `-1`, приведённая к беззнаковому, становится очень большим числом, большим `1`. Суффикс `u` также приводит `1` к `unsigned`, а операция `<` теперь сравнивает `unsigned` `-1` и `1`. Причём в данном случае компилятор предупреждает нас о, возможно, неправильном сравнении.

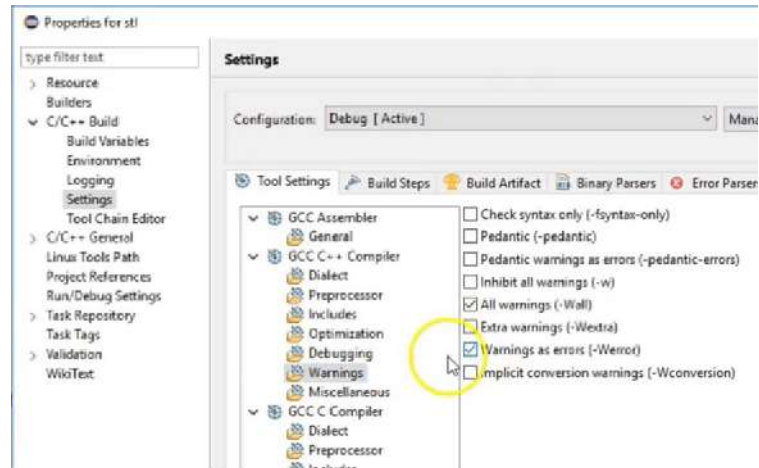
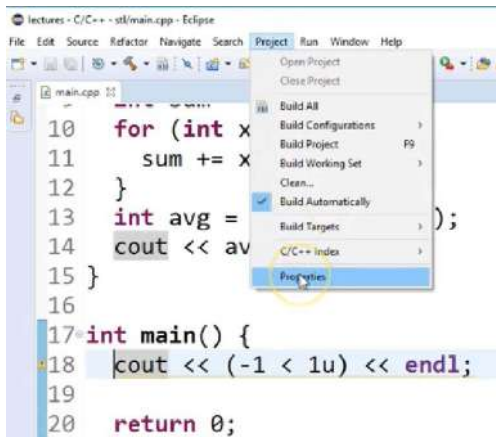
### 1.1.3. Безопасное использование целочисленных типов

**Настроим компилятор:**

Попросим компилятор считать каждый warning (предупреждение) ошибкой. Project → Properties → C/C++ → Build → Settings → GCC C++ Compiler → Warnings и отмечаем Warnings as errors.

После этого ещё раз компилируем код. И теперь каждое предупреждение считается ошибкой, которую надо исправить. Это одно из правил хорошего кода.





```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < x.size(); ++i) {
        cout << i << " " << x[i] << endl;
    }
    return 0;
}
// error: "i < x.size()"... comparison between signed and unsigned
```

Есть два способа это исправить: объявить `i` типом `size_t` или явно привести `x.size()` к типу `int` с помощью `static_cast<int>(x.size())`.

```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < static_cast<int>(x.size()); ++i) {
        cout << i << " " << x[i] << " ";
    }
    return 0;
}
// 0 4 1 5
```

## Исправляем задачу о температуре:

В задаче о температуре тоже приводим `t.size()` к знаковому с помощью оператора `static_cast`:

```
int main () {
    vector<int> t = {-8, -7, 3};    // сумма -4
```

```
...
int avg = sum / static_cast<int>(t.size()); // явно привели типы
cout << avg << endl;
return 0;
}
// -4
```

Предупреждений и ошибок не было. Всё, задача средней температуры для положительных и отрицательных значений решена! Таким образом если у нас где-то могут быть проблемы с беззнаковыми типами, мы либо следуем семантике и помним про опасности, либо приводим всё к знаковым с помощью `static_cast`.

### Ещё примеры опасностей с беззнаковыми типами:

Переберём в векторе все элементы кроме последнего:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i < v.size() - 1; ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

После запуска код падает. Вычтя из `v.size()` 1 мы получили максимальное значение типа `size_t` и вышли из своей памяти. Чтобы такого не произошло, мы перенесём единицу в другую часть сложения:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i + 1 < v.size(); ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

Теперь на пустом векторе у нас все компилируется и вывод пустой. А на непустом выводит все элементы, кроме последнего. Напишем программу вывода элементов вектора в обратном порядке:

```
int main() {
```

```
vector<int> v = {1, 4, 5};
for (size_t i = v.size() - 1; i >= 0; --i) {
    cout << v[i] << endl;
}
return 0;
}
```

На пустом векторе, очевидно, будет ошибка. Но даже на не пустом он сначала выводит 5, 4, 1, а затем очень много чисел и программа падает. Это произошло из-за того, что `i >= 0` выполняется всегда и мы входим в бесконечный цикл. От этой проблемы мы избавимся «заменой переменной» для итерации:

```
for (size_t k = v.size(); k > 0 ; --k) {
    size_t i = k - 1; // теперь
    cout << v[i] << endl;
}
```

Теперь всё работает нормально. В итоге, проблем с беззнаковыми типами помогают избежать:

- Предупреждения компилятора;
- Внимательность при вычитании из беззнаковых;
- Приведение к знаковым типам с помощью `static_cast`.

## 1.2. Кортежи и пары

### 1.2.1. Упрощаем оператор сравнения

Поговорим про новые типы данных – пары и кортежи. Начнём с проблемы, которая возникла в курсовом проекте курса «Белый пояс по C++»: мы должны были хранить даты в виде структур из полей «год», «месяц», «день» в ключах словарей. А поскольку словарь хранит ключи отсортированными, нам надо определить для этого типа данных оператор «меньше». Можно сделать это так:

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

```
using namespace std;

struct Date {
    int year;
    int month;
    int day;
};

bool operator < (const Date& lhs, const Date& rhs) {
    if (lhs.year != rhs.year) {
        return lhs.year < rhs.year;
    }
    if (lhs.month != rhs.month) {
        return lhs.month < rhs.month;
    }
    return lhs.day < rhs.day;
}

...
```

Сначала сравниваем года, потом месяцы и затем дни. Но здесь много мест для ошибок и код довольно длинный. В эталонном решении курсового проекта эта проблема решена так:

```
bool operator < (const Date& lhs, const Date& rhs) {
    return vector<int>{lhs.year, lhs.month, lhs.day} <
        vector<int>{rhs.year, rhs.month, rhs.day};
}

...
```

Выигрыш в том, что для векторов лексикографический оператор сравнения уже определён и этот код работает для каких-нибудь двух дат:

```
int main() {
    cout << (Date{2017, 6, 8} < Date {2017, 1, 26}) << endl;
    return 0;
}

// 0
```

Действительно, первая дата больше второй и выводит 0. Но тип `vector` слишком мощный: он позволяет делать `push_back` в себя, удалять из середины, что-то ещё. Нам этот тип не нужен в данном случае. Нам нужно всего лишь объединить для левой даты и для правой даты три

значения в одно и сравнить. Кроме того, вектор работает только при однотипных элементах. Если бы у нас месяц был строкой, вектор бы не подходил.

Как же объединить разные типы в один массив? Для этого нужно подключить библиотеку `tuple`. И вместо вектора вызывать функцию `tie` от тех значений, которые надо связать. В нашем случае год, месяц и день левой даты и год, месяц и день правой даты надо связать и сравнить.

```
#include <tuple>
...
bool operator <(const Date& lhs, const Date&rhs) {
    auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // сохраним левую дату
    auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // и правую
    return lhs_key < rhs_key
}
int main() {
    cout << (Date{2017, "June", 8} < Date{2017, "January", 26}) << endl;
    return 0;
}
// 0
```

И действительно, строка «June» лексикографически больше строки «January» и наша программа делает то, что нужно. Теперь узнаем, какого типа у нас `lhs_key` и `rhs_key`, породив ошибку компиляции.

```
...
auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // левая
auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // правая
lhs_key + rhs_key; // тут нарочно порождаем ошибку
return lhs_key < rhs_key
...
// operator+ не определен для std::tuple<const int&, const std::string&, const int&>...
```

То есть они имеют тип `tuple<const int&, const string&, const int&>`. Tuple – это *кортеж*, т. е. структура из ссылок на нужные нам данные (возможно, разнотипные).

## 1.2.2. Кортежи и пары

Создадим кортеж из трёх элементов:

```
#include <iostream>
#include <vector>
#include <tuple>
using namespace std;
int main() {
    tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
    auto t = tie(7, "C++", true); // пытаемся связать константные значения в tie
    return 0;
}
```

В первой строке всё хорошо – мы создали структуру из трёх полей. А во второй – ошибка компиляции, потому что `tie` создает кортеж из ссылок на объекты (которые хранятся в каких-то переменных). Используем функцию `make_tuple`, создающую кортеж из самих значений. И будем обращаться к полям кортежа:

```
...
// tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
auto t = make_tuple(7, "C++", true)
cout << get<1>(t) << endl;
return 0;
}
// C++
```

С помощью функции `get<1>(t)` мы получили 1-ый (нумерация с 0) элемент кортежа `t`. Использование кортежа `tuple` целесообразно, только если нам необходимо, чтобы в кортеже были разные типы данных.

*Замечание:* в C++ 17 разрешается не указывать шаблонные параметры `tuple` в `< ... >`, т. е. кортеж можно создавать так:

```
tuple t(7, "C++", true);
```

Но при компиляции компилятор попросит параметры. Оказывается надо явно сказать ему, чтобы он использовал стандарт C++ 17. Для этого снова идём в Project → Properties → C/C++ Build → Dialect → Other dialect flags и пишем `std = c++17`, т. е. версии C++ 17 (для этого компилятор должен быть обновлен до версии GCC7 или больше).

Если же мы хотим связать только два элемента, то используем структуру «пара» – `pair`. Пара – это частный случай кортежа, но отличие пары в том, что её поля называются `first` и `second` и к ним можно обращаться напрямую:

```
int main() {
    pair p(7, "C++"); // в новом стандарте можно и без <int, string>
    // auto p = make_pair(7, "C++"); // второй вариант
    cout << p.first << " " << p.second << endl;
    return 0;
}
// 7 C++
```

В результате нам вывело нашу пару. Эту структуру мы уже видели при итерировании по словарию. Так что нам удобно её использовать, не объявляя структуру явно.

Для примера создадим словарь (map):

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}};
    for (const auto& item : digits) {
        cout << item.first << " " << item.second << endl;
    }
    return 0;
}
// 1 one
```

Поскольку словарь – это пара «ключ-значение», то можно (как было объяснено в предыдущем курсе) распаковать значения item:

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}}
    for (const auto& [key, value] : digits) {
        cout << key << " " << value << endl;
    }
    return 0;
}
// 1 one
```

Всё скомпилировалось, значит, мы можем итерироваться по словарию, не создавая пар.

### 1.2.3. Возврат нескольких значений из функций

Возврат нескольких значений из функций – ещё одна область применения кортежей и пар. Будем хранить класс с информацией о городах и странах:

```
#include <iostream>
#include <vector>
#include <utility>
#include <map>
#include <set>
using namespace std;

class Cities { // класс городов и стран
public:
    tuple<bool, string> FindCountry(const string& city) const {
        if (city_to_country.count(city) == 1) {
            return {true, city_to_country.at(city)};
            // city_to_country[city] выдало бы ошибку, потому что могло нарушить const словаря
        } else if (ambiguous_cities.count(city) == 1) {
            return {false, "Ambiguous"};
        } else {
            return {false, "Not exists"}; // если нет
        }
        // выводит значение, нашлась ли единственная страна для города
    } // и сообщение - либо страна не нашлась, либо их несколько
private:
    // по названию города храним название страны:
    map<string, string> city_to_country;
    // множество городов, принадлежащих нескольким странам:
    set<string> ambiguous_cities;
}

int main() {
    Cities cities;
    bool success;
    string message; // свяжем кортежем ссылок
    tie(success, message) = cities.FindCountry("Volgograd");
    cout << " " << message << endl; // вывели результат
    return 0;
}
//0 Not exists
```



Всё работает. Таким образом, мы научились возвращать из метода несколько значений с помощью кортежа. А по новому стандарту можно получить кортеж и распаковать его в пару переменных:

```
int main() {
    Cities cities;
    auto [success, message] = cities.FindCountry("Volgograd"); // сразу распаковали
    cout << success << " " << message << endl;
    return 0;
}
//0 Not exists
```

Итак, если вы хотите вернуть несколько значений из функции или из метода, используйте кортеж. А если вы хотите сохранить этот кортеж в какой-то набор переменных, используйте `structured bindings` или функцию `tie`.

Кортежи и пары нужно использовать аккуратно. Они часто мешают читаемости кода, если вы начинаете обращаться к их полям. Например, представьте себе, что вы хотите сохранить словарь из названия города в его географические координаты. У вас будет словарь, у которого в значениях будут пары двух вещественных чисел. Назовем его `cities`. Как же потом вы будете, например, итерироваться по этому словарю?

```
int main() {
    map<string, pair<double, double>> cities;
    for (const auto& item : cities) {
        cout << item.second.first << endl; // абсолютно нечитаемый код
    }
    return 0;
}
```

*Заключение:* кортежи позволяют упростить написание оператора `<` или вернуть несколько значений из функции. Пары – это частный случай кортежей, у которых понятные названия полей, к которым удобно обращаться.

## 1.3. Шаблоны функций

### 1.3.1. Введение в шаблоны

Рассмотрим шаблоны функций на примере функции возведения числа в квадрат.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
using namespace std;

int Sqr(int x) { // функция возведения в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2) << endl; // результат выведем в поток
    return 0;
}
// 4
```

Функция работает. Теперь мы хотим возводить в квадрат дробные числа, например, 2.5. Получается снова 4. Это неправильно.

```
... cout << Sqr(2.5) << endl; // результат функции выведем в поток
// 4
```

Это происходит потому, что у нас нет аналогичной функции для работы с дробными числами. Заведём её:

```
int Sqr(int x) { // функция возведения целого числа в квадрат
    return x * x;
}

double Sqr(double x) { // функция возведения дробного числа в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2.5) << endl;
    return 0;
}
```

```
}
// 6.25
```

Всё работает, но у нас появилось две функции, которые делают одно и то же, но с разными типами. Гораздо удобнее написать функцию, работающую с каким-то типом T:

```
using namespace std;
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    cout << Sqr(2.5) << " " << Sqr(3) << endl;
    return 0;
}
// 6.25 9
```

Собираем наш код и видим, что всё работает. Тип T компилятор выведет сам, чтобы типы поддерживали умножение. Нужно было только его объявить ключевым словом `template <typename T>`. Теперь попробуем возвести в квадрат пару:

```
#include <utility> // добавляем нужную библиотеку
... // код функции оставляем таким же
int main() {
    auto p = make_pair(2, 3); // создаем пару
    cout << Sqr(p) << endl; // пытаемся возвести ее в квадрат
    return 0;
}
// no match for 'operator*' (operand types are std::pair<int,int> and std::pair<int,int>)
```

Видим ошибку, т. к. для оператора умножения не определены аргументы «пара и пара». Тогда напомним шаблонный оператор умножения для пар:

```
using namespace std;
template <typename First, typename Second>
// т.к. умножение для пар не определено, вручную определим оператор умножения для пар:
pair<First, Second> operator * (const pair<First, Second>& p1,
                               const pair<First, Second>& p2) {
    // мы можем создавать переменные шаблонного типа
    First f = p1.first * p2.first;
```

```

    Second s = p1.second * p2.second;
    return {f, s};
}
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    auto p = make_pair(2.5, 3); // создаем пару
    auto res = Sqr(p); // возводим пару в квадрат
    cout << res.first << " " << res.second << endl; // выводим получившееся
    return 0;
}
// 6.25 9

```

Код работает – пара возвелась в квадрат (и её дробная часть, и целая). Одним из важных плюсов языка C++ является возможность подобным образом избавляться от дублирований и сильно сокращать код.

### 1.3.2. Универсальные функции вывода контейнеров в поток

В курсах ранее мы часто печатали содержимое наших контейнеров, будь то `vector` или `map`, на экран. Для этого мы определяли специальную функцию либо перегружали оператор вывода в поток. Давайте это сделаем и сейчас:

```

#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
// напишем свой оператор вывода в поток вектора целых типов
ostream& operator<< (ostream& out, const vector<int>& vi) {
    for (const auto& i : vi) { // проитерируем по вектору
        out << i << ' '; // выведем все элементы в поток
    }
    return out;
}

```

```

}

int main() {
    vector<int> vi = {1, 2, 3};
    cout << vi << endl;
}
// 1 2 3

```

Всё выводится. Но если поменять тип вектора на `double`, то будет ошибка:

```

...
vector<double> vi = {1, 2, 3}; // теперь дробный вектор
...
// no match for 'operator <<' (operand types are std::ostream and std::vector<double>)

```

Для решения этой проблемы можно было бы каждый раз заново дублировать код. Но с помощью шаблонов функций мы меняем тип вектора с `int` на шаблонный `T`:

```

using namespace std;
template <typename T> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const vector<T>& vi) { // вектор на шаблон
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}

int main() {
    vector<double> vi = {1.4, 2, 3}; // дробные числа
    cout << vi << endl;
}
// 1.4 2 3

```

Ошибки пропали, вывелся наш вектор из дробных чисел. Мы научились универсальным способом решать задачу для вектора. Таким же универсальным способом научимся решать задачу для других контейнеров.

```

int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <' (operand types are std::ostream and std::map<int, int>)

```

Видим, что оператор вывода для `map` не определён. Определим его так же, как и для вектора:

```
...
template <typename Key, typename Value> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
...
int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <<' (operand types are std::ostream and std::pair<const int, int>)
```

Заметим, что ошибка для `map` имеет интересный вид: `pair<const int, int>`. Действительно, ведь `map` — это `pair`, в которой `key` нельзя модифицировать, а `value` можно. Получается, нам достаточно определить оператор вывода в поток для пары. Тогда мы сможем вывести и `map`.

```
// весь рабочий код
...
template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    out << p.first << ", " << p.second;
    return out;
}
template <typename T> // для vector
ostream& operator<< (ostream& out, const vector<T>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
template <typename Key, typename Value> // для пар
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
```

```
int main() {
    // vector<double> vi = 1.4, 2,3;
    // cout << vi << endl;
    map<int, int> m = {{1, 2}, {3, 4}}; // целые числа
    map<int, int> m2 = {{1.4, 2.1}, {3.4, 4}}; // дробные числа
    cout << m << ' ' << m2 << endl;
}
//1, 2 3, 4; 1.4, 2.1 3.4, 4
```

Код отработал корректно. Оба `map`'а вывелись, как нам и было нужно. Заметим, что код с вектором, если его добавить, до сих пор будет работать. Но этот код тоже можно доработать. Шаблоны для вектора и для `map`'а выглядят почти одинаково. Кроме того, для читаемости можно сделать улучшение: когда мы выводим `map`, обрамлять его в фигурные скобки, когда вектор – в квадратные, а когда мы выводим пару, обрамлять её круглыми скобками.

### 1.3.3. Рефакторим код и улучшаем читаемость вывода

Исправим предыдущую программу и сделаем её вывод более читаемым. Нужно создать шаблонную функцию, которая на вход будет принимать коллекцию. На вход этой функции будем передавать разделитель, через который надо вывести элементы нашей коллекции. Единственное, что мы требуем от входной коллекции: по ней можно итерироваться с помощью цикла `range-based for` и её элементы можно выводить в поток.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
template <typename Collection> // тип коллекции
string Join(const Collection& c, char d) { // передаем коллекцию и разделитель
    stringstream ss; // завели строковый поток
    bool first = true; // первый ли это элемент?
    for (const auto& i : c) {
        if (!first) {
            ss << d; // если вывели не первый элемент - кладем поток в разделитель
        }
    }
}
```

```

    first = false; // т.к. следующий элемент точно не будет первым
    ss << i; // кладем следующий элемент в поток
}
return ss.str();
}
template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    return out << '(' <<p.first << ',' << p.second << ')'; // тоже изменили
}
template <typename T> // для vector изменили код и добавили скобочки
ostream& operator<< (ostream& out, const vector<T>& vi) {
    return out << '[' << Join(vi, ',') << ']';
} // оператор вывода возвращает ссылку на поток
template <typename Key, typename Value> // для map убрали аналогично vector
ostream& operator<< (ostream& out, const map<Key, Value>& m) {
    return out << '{' << Join(m, ',') << '}'; // и добавили фигурные скобочки
}

int main() {
    vector<double> vi = {1.4, 2, 3};
    pair<int, int> m1 = {1, 2};
    map<double, double> m2 = {{1.4, 2.1}, {3.4, 4}};
    cout << vi << ' ' << m1 << ' ' << m2 << endl;
}
// [1.4, 2,3] (1, 2) {(1.4, 2.1), (3.4, 4)}

```

Всё работает. Наша программа вывела сначала вектор, потом пару и затем `map`. Для более сложных конструкций она тоже будет работать. Например, вектор векторов:

```

int main() {
    vector<vector<int>> vi = {{1, 2}, {3, 4}};
    cout << vi << endl;
}
// [[1, 2], [3, 4]]

```

В итоге всё работает и на сложных контейнерах. Таким образом, мы сильно упростили наш код и избежали ненужного дублирования с помощью шаблонов и универсальных функций.



### 1.3.4. Указание шаблонного параметра-типа

Рассмотрим случай, когда компилятор не знает, как на основе вызова шаблонной функции вывести тип `T` на примере задачи о выводе максимального из двух чисел.

```
#include <iostream>
using namespace std;
template <typename T>
T max(T a, T b) {
    if (b < a) {
        return a;
    }
    return b;
}

int main() {
    cout << Max(2, 3) << endl;
    return 0;
}
// 3
```

Если оба числа целые, то всё работает. Но если поменять одно число на вещественное, мы увидим ошибки:

```
...
cout << Max(2, 3.5) << endl;
// deduce conflicting types for parameter 'T' (int and double)
```

Т. е. вывод шаблонного параметра типа `T` не может состояться, потому что компилятор не знает, что поставить: `int` или `double`. В таких ситуациях мы либо приводим переменные к одному типу, либо подсказываем компилятору таким образом:

```
cout << Max<double>(2, 3.5) << endl; // явно показываем компилятору тип T
// 3.5
```

А если же мы попросим `int`, получим следующее:

```
cout << Max<int>(2, 3.5) << endl; // 3.5 приведётся к int и мы сравнили
// 3
```

Писать уже существующие функции плохо, поэтому вызовем стандартную функцию `max` из библиотеки `algorithms`:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    cout << max<int>(2, 3.5) << ' ' << max<double>(2, 3.5) << endl;
    return 0;
}
// 3 3.5
```

Функция `max` тоже шаблонная. Если явно указать тип, к которому приводить результат, у нас будет то же, что мы уже видели. Если же тип не указывать, произойдёт знакомая ошибка компиляции.

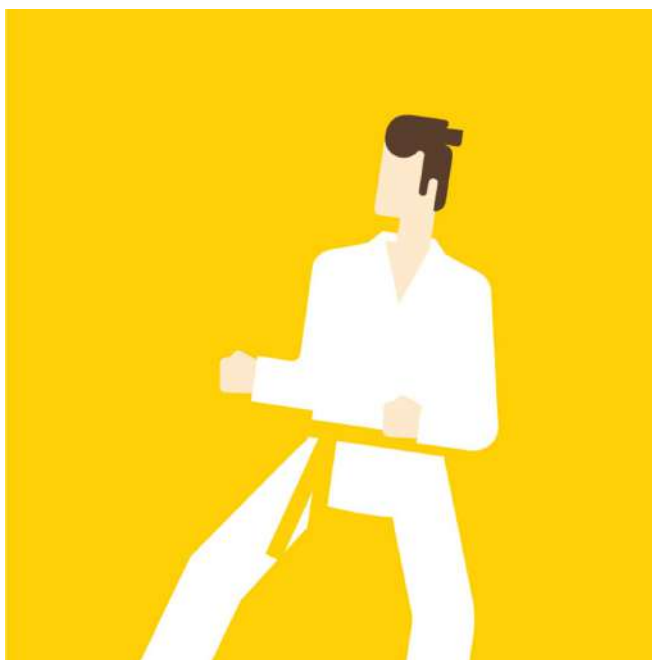
### Подведём итоги:

1. Шаблонные функции объявляются так: `template <typename T> T Foo(T var) { ... };`
2. Вместо слова `typename` можно использовать слово `class`, т. к. в данном контексте они эквивалентны;
3. Шаблонный тип может автоматом выводиться из контекста вызова функции;
4. После объявления используется, как и любой другой тип;
5. Выведение шаблонного типа может происходить либо автоматически, на основе аргументов, либо с помощью явного указания в угловых скобках (`std::max<double>(2, 3.5)`);
6. Цель шаблонных функций: сделать код короче (избавившись от дублирования) и универсальнее.

# Основы разработки на C++: жёлтый пояс

Неделя 2

Тестирование



# Оглавление

<b>Тестирование</b>	<b>2</b>
2.1 Тестирование и отладка . . . . .	2
2.1.1 Введение в юнит-тестирование . . . . .	2
2.1.2 Декомпозиция решения в задаче «Синонимы» . . . . .	3
2.1.3 Простейший способ создания юнит-тестов на C++ . . . . .	5
2.1.4 Отладка решения задачи «Синонимы» с помощью юнит-тестов . . . . .	7
2.1.5 Анализ недостатков фреймворка юнит-тестов . . . . .	10
2.1.6 Улучшаем <code>assert</code> . . . . .	11
2.1.7 Внедряем шаблон <code>AssertEqual</code> во все юнит-тесты . . . . .	13
2.1.8 Изолируем запуск отдельных тестов . . . . .	15
2.1.9 Избавляемся от смешения вывода тестов и основной программы . . . . .	17
2.1.10 Обеспечиваем регулярный запуск юнит-тестов . . . . .	18
2.1.11 Собственный фреймворк юнит-тестов. Итоги . . . . .	19
2.1.12 Общие рекомендации по декомпозиции программы и написанию юнит-тестов	20

# Тестирование

## 2.1. Тестирование и отладка

### 2.1.1. Введение в юнит-тестирование

Вспомним, что говорилось в курсе «C++: белый пояс». Если решение не принимается тестирующей системой, то нужно:

1. Внимательно перечитать условия задачи;
2. Убедиться, что программа корректно работает на примерах;
3. Составить план тестирования (проанализировать классы входных данных);
4. Тестировать программу, пока она не пройдет все тесты;
5. Если идеи тестов кончились, но программа не принимается, то выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами.

Как юнит-тесты помогают в отладке:

1. Позволяют протестировать каждый компонент изолированно;
2. Их проще придумывать;
3. Упавшие юнит-тесты указывают, в каком блоке программы ошибка.

### 2.1.2. Декомпозиция решения в задаче «Синонимы»

На примере задачи «Синонимы» из курса «C++: белый пояс» покажем применение юнит-тестов.

Условие задачи:

Два слова называются синонимами, если они имеют похожие значения. Надо реализовать словарь синонимов, обрабатывающий три вида запросов:

- `ADD word1 word2` – добавить в словарь пару синонимов (`word1`, `word2`);
- `COUNT word` – выводит текущее количество синонимов для слова `word`;
- `CHECK word1 word2` – проверяет, являются ли слова синонимами.

Ввод:	Вывод:
<code>ADD program code</code>	
<code>ADD code cipher</code>	
<code>COUNT cipher</code>	<code>1</code>
<code>CHECK code program</code>	<code>YES</code>
<code>CHECK program cipher</code>	<code>NO</code>

Посмотрим на решение, которое у нас уже есть и которое надо протестировать:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;

int main() {
    int q; // считываем количество запросов
    cin >> q;
    // храним словарь синонимов (для строки хранит множество всех её синонимов)
    map<string, set<string>> synonyms;
    for (int i = 0; i < q; ++i) { // обрабатываем запросы
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") { // считываем две строки и добавляем в словарь
```

```
    string first_word, second_word;
    cin >> first_word >> second_word;
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
} else if (operation_code == "COUNT") { // считываем одну строку и выводим
// размер
    string word;
    cin >> word;
    cout << synonyms[word].size() << endl;
} else if (operation_code == "CHECK") { // считываем два слова и проверяем
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (synonyms[first_word].count(second_word) == 1) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Отправляем в тестирующую систему и видим, что решение не принялось. Он говорит нам, что неправильный ответ на третьем тесте, и больше информации нет. Решение надо тестировать на различных входных данных. Но вроде всё работает и стоит переходить к юнит-тестированию. А для него надо сначала выполнить декомпозицию задачи «Синонимы». Давайте ввод и вывод оставим в `main`, а обработку запроса вынесем в отдельные функции:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;
void AddSynonyms(map<string, set<string>>& synonyms, // функция добавления
                 const string& first_word, const string& second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(map<string, set<string>>& synonyms, // количество синонимов
                      const string& first_word) {
    return synonyms[first_word].size();
}
```

```

}
bool AreSynonyms(map<string, set<string>>& synonyms, // проверка
                 const string& first_word, const string& second_word) {
    return synonyms[first_word].count(second_word) == 1;
}
int main() {
    int q;
    cin >> q;
    map<string, set<string>> synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word)
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << GetSynonymCount(synonyms, word) << endl;
        } else if (operation_code == "CHECK") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            if (AreSynonyms(synonyms, first_word, second_word)) {
                cout << "YES" << endl;
            } else {
                cout << "NO" << endl;
            }
        }
    }
}
return 0;
}

```

### 2.1.3. Простейший способ создания юнит-тестов на C++

Посмотрим, как писать юнит-тесты и как они должны себя вести на примере функции Sum, которая находит сумму двух чисел.

```
#include <iostream>
```



```
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y;
}
void TestSum() { // собираем набор тестов для функции Sum
    assert(Sum(2, 3) == 5); // мы ожидаем, что 2+3=5
    assert(Sum(-2, -3) == -5); // проверка отрицательных чисел
    assert(Sum(-2, 0) == -2); // проверка прибавления 0
    assert(Sum(-2, 2) == 0); // проверка, когда сумма = 0
    cout << "TestSum OK" << endl;
}
int main() {
    TestSum();
    return 0;
}
// TestSum OK
```

Тесты отработали и не нашли ошибок. Теперь посмотрим, что должно быть при наличии ошибок:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y - 1; // сделали заведомо неправильно
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Нам написали, в каком файле, в какой строке какой **Assert** не сработал. Это облегчает поиск ошибок. Мы можем добиться другой ошибки, например:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    if (x < 0) {
        x -= 1
    }
    return x + y;
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(-2, -3) == -5' failed ...
```

Видим, что первый тест прошёл, а на втором уже ошибка. Таким образом, мы можем проверять каждую функцию по отдельности на ожидаемых значениях.

#### 2.1.4. Отладка решения задачи «Синонимы» с помощью юнит-тестов

Для задачи «Синонимы» покроем каждую функцию юнит-тестами и сократим код заменой `map<string, set<string>>` на что-то более короткое:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <cassert>
using namespace std;
using Synonyms = map<string, set<string>>;
// сократили запись типа и везде изменили на Synonyms
void AddSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word); // тут должен не сработать AddSynonyms
}
size_t GetSynonymCount(Synonyms& synonyms, const string& word) {
    return synonyms[word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

void TestAddSynonyms() { // тестируем AddSynonyms
{
    Synonyms empty; // тест 1
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}}, // ожидаем, что при добавлении синонимов появятся две записи в
        // словаре
        {"b", {"a"}}
    };
    assert(empty == expected);
}
```

```

}
{ // заметим, что мы формируем корректный словарь и ожидаем, что он останется корректным

    Synonyms synonyms = { // если вдруг корректность нарушится, то assert скажет, где
        {"a", {"b"}}, // тест 2
        {"b", {"a", "c"}},
        {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
        {"a", {"b", "c"}},
        {"b", {"a", "c"}},
        {"c", {"a", "b"}}
    };
    assert(synonyms == expected);
}
cout << "TestAddSynonyms OK" << endl;
}

void TestCount() { // тестируем Count
{
    Synonyms empty;
    assert(GetSynonymCount(empty, "a") == 0);
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    assert(GetSynonymCount(synonyms, "a") == 2);
    assert(GetSynonymCount(synonyms, "b") == 1);
    assert(GetSynonymCount(synonyms, "z") == 0);
}
cout << "TestCount OK" << endl;
}

void TestAreSynonyms() { // тестируем AreSynonyms
{
    Synonyms empty; // пустой словарь для любых двух слов вернёт false
    assert(!AreSynonyms(empty, "a", "b"));
}
}

```

```
    assert(!AreSynonyms(empty, "b", "a"));
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    assert(AreSynonyms(synonyms, "a", "b"));
    assert(AreSynonyms(synonyms, "b", "a"));
    assert(AreSynonyms(synonyms, "a", "c"));
    assert(AreSynonyms(synonyms, "c", "a"));
    assert(!AreSynonyms(synonyms, "b", "c")); // false
    assert(!AreSynonyms(synonyms, "c", "b")); // false
}
cout << "TestAreSynonyms OK" << endl;
}
void TestAll() { // функция, вызывающая все тесты
    TestCount();
    TestAreSynonyms();
    TestAddSynonyms();
}
int main() {
    TestAll();
    return 0;
}
// TestCount OK
// TestAreSynonyms OK
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.
```

Видим, что Count и AreSynonyms работают нормально, а вот в AddSynonyms у нас ошибка. Смотрим, что не так. Идём в AddSynonyms и видим, что:

```
synonyms[first_word].insert(first_word); // мы должны к 1 слову добавить 2, а не 1
```

Теперь после исправления все наши тесты отработали успешно. Снова пробуем отправить в тестирующую систему наше решение, закомментировав вызов TestAll(); в main. Тестирование завершилось и решение принято тестирующей системой. Таким образом мы на простом примере продемонстрировали эффективность декомпозиции программы и юнит-тестов.

### 2.1.5. Анализ недостатков фреймворка юнит-тестов

По ходу разработки юнит-тестов во время решения задачи «Синонимы» мы смогли написать небольшой юнит-тест фреймворк. Посмотрим, что за фреймворк получился. Во-первых, он основан на функции `assert`. Его главный плюс – мы узнаём, какая именно проверка сработала неправильно. На предыдущей задаче мы видели:

```
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.
```

Основные недостатки:

1. При проверке равенства в консоль не выводятся значения сравниваемых переменных. И мы не знаем, чем была переменная `empty`;
2. После невыполненного `assert` код падает. Если в `TestAll` поставить `TestAddSynonyms` на первое место, то остальные два теста даже не начнутся;
3. Кроме того, у нас пока что результаты тестов выводят ОК в стандартный вывод и смешиваются с тем, что должен вывести код.

В C++ уже существует много фреймворков для работы с тестами, в которых этих недостатков нет.

C++ Unit Testing Frameworks:

1. Google Test
2. CxxTest
3. Boost Test Library

Далее мы свой Unit Testing Framework улучшим для того, чтобы показать, что текущих знаний C++ хватает для таких вещей. И вы будете понимать как он работает, и сможете его менять под свои нужды.

### 2.1.6. Улучшаем assert

Избавимся от первого недостатка `assert`: когда он срабатывает, мы не видим, чему равен каждый из операндов. Т. е. мы хотим видеть для кода такой вывод:

```
int x = Add(2, 3);
assert(x == 4);
// Assertion failed: 5 != 4
```

И работало оно для любых типов данных:

```
vector<int> sorted = Sort({1, 4, 3});
assert(sorted == vector<int> {1, 3, 4}));
// Assertion failed: [1, 4, 3] != [1, 3, 4]
```

Такие универсальные выводы помогут догадаться о возможной ошибке. Т. е. нам нужна функция сравнения двух переменных какого-то произвольного типа. Напишем шаблон `AssertEqual` перед `TestAddSynonyms()`.

```
#include <exception> // подключим исключения
#include <sstream>    // подключили строковые потоки
...
template <class T, class U>
void AssertEqual (class T& t, const U& u) {
    // значения двух разных типов для удобства
    if (t != u) { // если значения не равны, то мы даём знать, что этот assert не сработал
        ostringstream os;
        os << "Assertion failed: " << t << "!=" << u;
        throw runtime_error(os); // бросим исключение с сообщением со значениями t и u
    }
}
```

Встроим это в `TestCount()`. Заменим `assert` на наш `AssertEqual` внутри `TestCount()`.

```
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"), 0);
        AssertEqual(GetSynonymCount(empty, "b"), 0);
    }
    {
        Synonyms synonyms = {
```

```

    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
};
AssertEqual(GetSynonymCount(synonyms, "a"), 2);
AssertEqual(GetSynonymCount(synonyms, "b"), 1);
AssertEqual(GetSynonymCount(synonyms, "z"), 0);
}
}
// Warning ... comprison between signed and unsigned ...

```

Код скомпилировался, но мы получили Warning из-за сравнения между знаковым и беззнаковым типами в `AssertEqual`. Это происходит потому, что все константы (2, 1 и 0 в нашем случае) имеют тип `int` (как уже было сказано в неделе 1), который мы сравниваем с типом `size_t`. Исправляем это, дописав к ним `u` справа: `1 → 1u` и т. д.

Теперь, сделав нарочную ошибку где-нибудь в `GetSynonymCount`, мы получим предупреждение: `Assertion failed: 1 != 0`. Но пока мы не видим, какой именно `Assert` сработал. Исправим это, передавая в `Assert` строчку `hint` и также добавим каждому `Assert`'у строку идентификации, по которой мы сможем однозначно понять, какой именно `Assert` выдал ошибку:

```

void AssertEqual (class T& t, const U& u, const string& hint) {
    if (t != u) {
        ostreamstream os;
        os << "Assertion failed: " << t << "!=" << u << "Hint: " << hint;
        throw runtime_error(os);
    }
}
...
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"),
                    0u, "Synonym count for empty dict a");
        AssertEqual(GetSynonymCount(empty, "b"),
                    0u, "Synonym count for empty dict b");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},

```

```

    {"c", {"a"}}
};
AssertEqual(GetSynonymCount(synonyms, "a"), 2u, "Nonempty dict, count a");
AssertEqual(GetSynonymCount(synonyms, "b"), 1u, "Nonempty dict, count b");
AssertEqual(GetSynonymCount(synonyms, "z"), 0u, "Nonempty dict, count z");
}
}
// Asserting failed: 1!= 0 Hint: Synonym count for empty dict

```

### 2.1.7. Внедряем шаблон AssertEqual во все юнит-тесты

Добавим `Assert` в `AreSynonyms`. Только `AssertEqual` нам не подходит, потому что в данной функции у нас только два константных значения: `true` и `false`. Вместо этого напомним аналог классического `assert`, который назовём `Assert` (C++ чувствителен к регистру). И если мы испортим функцию `AreSynonyms`, то получим соответствующую ошибку с подсказкой.

```

void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}
... // модернизируем наш TestAreSynonyms
void TestAreSynonyms() {
    {
        Synonyms empty;
        Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
        Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},
            {"c", {"a"}}
        };
        Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
        Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
        Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
        Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
        Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
        Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms nonempty c b");
    }
}

```



```

    }
}
// Asserting failed: 0!= 1 Hint: AreSynonyms empty a b

```

Получили нужную ошибку, и по ней мы можем увидеть, где что-то не так. Осталась только функция `AddSynonyms`, и ловим ошибку:

```

void TestAddSynonyms() {
{
    Synonyms empty;
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}},
        {"b", {"a"}}},
    };
    AssertEqual(empty, expected, "Add to empty");
}
{
    Synonyms synonyms = {
        {"a", {"b"}},
        {"b", {"a", "c"}},
        {"c", {"b"}}},
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
        {"a", {"b", "c"}},
        {"b", {"a", "c"}},
        {"c", {"b", "a"}}},
    };
    AssertEqual(synonyms, expected, "Nonempty");
}
}
// no match for 'operator <<' (operand types are std::ostream<char> and std::map ...)

```

Ошибка произошла с `empty` и `synonyms`, которые являются `map<string, set<string>`. Мы пытаемся их вывести в стандартный поток вывода (см. неделя 1). Пишем перегрузку оператора вывода для `map` и для `set`, предварительно исправив ошибку в `AreSynonyms`, допустим ошибку в `AddSynonyms` и поймаем её:

```

template <class T> // учимся выводить в поток set
ostream& operator << (ostream& os, const set<T>& s) {

```

```
os << "{";
bool first = true;
for (const auto& x : s) {
    if (!first) {
        os << ", ";
    }
    first = false;
    os << x;
}
return os << "}";
}

template <class K, class V> // учимся выводить в поток map
ostream& operator << (ostream& os, const map<K, V>& m) {
    os << "{";
    bool first = true; // грамотная расстановка запятых
    for (const auto& kv : m) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << kv.first << ": " << kv.second;
    }
    return os << "}";
}

// Asserting failed: {a: {a}, b: {a} } != {a: {b}, b: {a}. Hint: Add to empty
```

Таким образом, мы внедрили шаблон `AssertEqual`, который позволяет найти ошибку, узнать, с чем она возникла и найти конкретное место в коде благодаря подсказке.

### 2.1.8. Изолируем запуск отдельных тестов

Теперь исправим следующий недостаток `Assert`: если он срабатывает, то код падает и другие тесты не выполняются. Аварийное завершение программы у нас возникало из-за вылета исключения в `AssertEqual`. Теперь будем ловить эти исключения в `main`:

```
int main() {
    try {
        TestAreSynonyms(); // ловим исключение
    }
```

```

} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAreSynonyms" << " fail: " << e.what() << endl;
} // если мы словили исключение, то работа всё равно продолжится
try {
    TestCount();
} catch (runtime_error& e) {
    ++fail_count;
    cout << " TestCount" << " fail: " << e.what() << endl;
}
try {
    TestAddSynonyms();
} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAddSynonyms" << " fail: " << e.what() << endl;
}
}

```

Но это неудобно, код дублируется. Нам бы хотелось, чтобы этот `try/catch` и вывод были написаны в одном месте, а мы туда могли бы передавать различные тестовые функции, и они бы там выполнялись, и исключения бы от них ловились, всё бы работало. В C++ это можно сделать, ведь функции имеют тип и их можно передавать в другие функции как аргумент. Создадим шаблон функции `RunTest`, который будет запускать тесты и ловить исключения.

```

...
template <class TestFunc>

void RunTest(TestFunc func, const string& test_name) { // передаём тест и его имя
    try {
        func();
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // ловим исключение
    }
}

int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
}

```

Заметим, что все тесты работают в любом порядке. Таким образом, мы с вами применили шаблон функций, для того чтобы передавать в качестве параметров функции другие функции, и смогли за счет этого написать универсальный такой шаблон, который для любого юнит-теста ловит исключения и позволяет нам все юнит-тесты, которые мы написали, выполнять при каждом запуске нашей программы.

### 2.1.9. Избавляемся от смешения вывода тестов и основной программы

Добавим в `RunTest` ещё одну удобную вещь: будем выводить OK снаружи каждого юнит-теста.

```
void RunTest(TestFunc func, const string& test_name) {
    try {
        func(); // заменим cout на cerr - стандартный поток ошибок
        cerr << test_name << " OK" << endl; // выводит OK, если всё работает
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // fail, если ошибка
    }
}
```

Всё это время сам алгоритм решения задачи «Синонимы» (который всё это время был закомментирован), всё ещё хорошо работает. Вот он сам:

```
int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
    int q;
    cin >> q;
    Synonyms synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word);
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
        }
    }
}
```

```

    cout << GetSynonymCount(synonyms, word) << endl;
} else if (operation_code == "CHECK") {
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
}
}

```

Вся проблема в том, что и юнит-тесты, и сама программа выводят в стандартный вывод. Пусть юнит-тесты выводят в `stderr` (стандартный поток ошибок). По окраске вывода в Eclipse можно отличать **стандартный вывод**, **стандартный ввод** и **стандартный поток ошибок**.

Вывод: **TestAreSynonyms OK, 5 TestAddSynonyms OK, TestCount OK, 1, COUNT a, 0**. Теперь не нужно окружать комментарием эти части кода перед отправкой, ведь они всё равно не повлияют на работу самой программы.

### 2.1.10. Обеспечиваем регулярный запуск юнит-тестов

Мы хотим, чтобы юнит-тесты были автоматическими, и их код находился где-то отдельно. Кроме того, стоит считать ошибки, чтобы если существует хоть одна, то программа не ждала получения данных от пользователя. Таким образом, мы хотим:

1. Запускаем тесты при старте программы. Если хоть один тест упал, программа завершается;
2. Если все тесты прошли, то должно работать решение самой задачи.

Для этого обернём наш шаблон `RunTest` в класс `TestRunner`:

```

class TestRunner { // класс тестирования
public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {

```

```

    try { // RunTest стал шаблонным методом класса
        func();
        cerr << test_name << " OK" << endl;
    } catch (runtime_error& e) {
        ++fail_count; // увеличиваем счётчик упавших тестов
        cerr << test_name << " fail: " << e.what() << endl;
    }
}

~TestRunner() { // деструктор класса TestRunner, в котором анализируем fail_count
    if (fail_count > 0) { //это как раз тот момент, когда
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1); // завершение программы с кодом возврата 1
    }
}

private:
    int fail_count = 0; // счётчик числа упавших тестов
};

void TestAll() { // переместили все тесты в одну функцию
    TestRunner tr;
    tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
    tr.RunTest(TestCount, "TestCount");
    tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}

int main() {
    TestAll(); // т.к. мы деструктор класса объявили в самом классе,
...} // выполняется он в конце TestAll

```

Теперь, если мы словили хоть одну ошибку, то программа перестанет выполняться с кодом возврата 1. Если же ничего плохого не произошло, то мы можем ввести число команд и сами команды.

### 2.1.11. Собственный фреймворк юнит-тестов. Итоги

Подведём итоги написания собственного фреймворка. Его основные свойства:

- Если срабатывает `assert`, в консоль выводятся его аргументы (работает для контейнеров);
- Вывод тестов не смешивается с выводом основной программы;

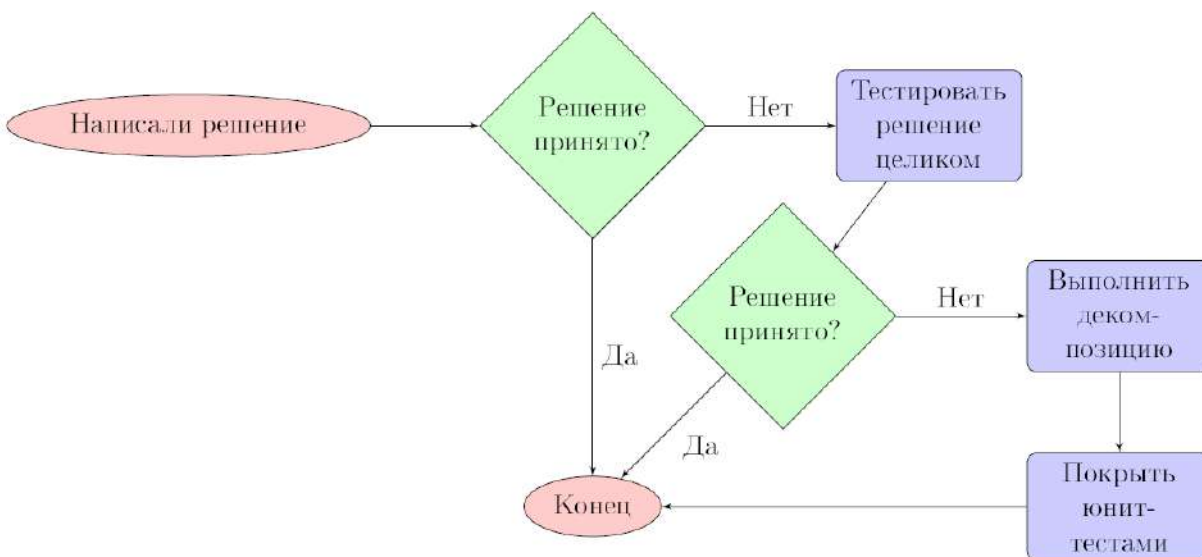
- При каждом запуске программы выполняются все юнит-тесты;
- Если хотя бы один тест упал, программа завершится с ненулевым кодом возврата.

Для того, чтобы пользоваться фреймворком, надо написать:

```
void TestSomething() { // функция, что-то тестирующая
    AssertEqual(..., ...);
    // выполняем какие-то проверки с помощью AssertEqual
}
void TestAll() {
    TestRunner tr;
    tr.RunTest(TestSomething, "TestSomething")
    // вызываем методом RunTest
}
int main() {
    TestAll(); // должна быть до самой программы
} // код фреймворка выложен рядом с видео
```

### 2.1.12. Общие рекомендации по декомпозиции программы и написанию юнит-тестов

Общий алгоритм решения задач с помощью декомпозиции и юнит-тестов:



Но кроме этой схемы, стоит выполнять декомпозицию задачи по ходу написания самого кода. Декомпозицию лучше делать сразу:

- Отдельные блоки проще реализовать и вероятность допустить ошибку ниже;
- Их проще тестировать, соответственно, выше вероятность найти ошибку или убедиться в её отсутствии;
- В больших проектах декомпозиция упрощает понимание и переиспользование кода;
- Уже реализованные функции можно брать и использовать в другом месте;
- Сама декомпозиция иногда защищает от ошибок.

Вспомним задачу «Уравнение» из курса «C++: белый пояс». Нужно было найти все различные действительные корни уравнения  $Ax^2 + By^2 + C = 0$ . Гарантируется, что  $A^2 + B^2 + C^2 > 0$ . Монолитное решение могло выглядеть так:

```

#include <cmath>
#include <iostream>
using namespace std;
int main() {

```



```
double a, b, c, D, x1, x2;
cin >> a >> b >> c;
D = b * b - 4 * a * c;
if ((a == 0 && c == 0) || (b == 0 && c == 0)) {
    cout << 0;
} else if (a == 0) {
    cout << -(c / b); // тут ошибка. Если b == 0, мы всё равно разделим на 0
} else if (b == 0) {
    cout << " ";
} else if (c == 0) {
    cout << 0 << " " << -(b / a);
} else if (D < 0) {
    cout << " ";
} else if (D == 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    cout << x1;
} else if (D > 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    x2 = ((-1 * b) - sqrt(D)) / (2 * a);
    cout << x1 << " " << x2;
}
return 0;
}
```

Быстро просмотрев этот код, сложно понять, работает он или нет. А в нём есть ошибка, которую из-за монолитности кода сложно заметить сразу. Теперь рассмотрим декомпозированное решение той же задачи:

```
#include <cmath>
#include <iostream>
using namespace std;

void SolveQuadraticEquation(double a, double b, double c) {
    // ... тут решение гарантированного квадратного уравнения
}

void SolveLinearEquation(double b, double c) {
    // b * x + c = 0
    if (b != 0) { // тут не забыли проверить деление на 0
        cout << -c / b;
    }
}
```

```
}

int main() {
    double a, b, c;
    cin >> a >> b >> c;
    if (a != 0) { // точно знаем, что уравнение квадратное
        SolveQuadraticEquation(a, b, c);
    } else { // просто решаем линейное
        SolveLinearEquation(b, c);
    }
    return 0;
}
```

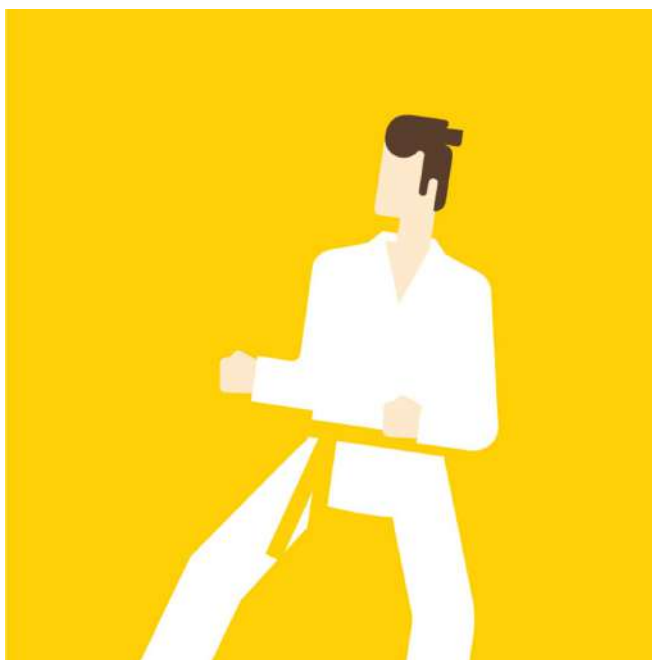
Юнит-тесты тоже лучше делать сразу. Причины:

- Разрабатывая тесты, вы сразу продумываете все варианты использования вашего кода и все крайние случаи входных данных;
- Тесты позволяют вам сразу проконтролировать корректность вашей реализации (особенно актуально для больших проектов);
- В больших проектах обширный набор тестов позволяет убедиться, что вы ничего не сломали во время дополнения кода.

# Основы разработки на C++: жёлтый пояс

Неделя 3

Разделение кода по файлам



# Оглавление

Разделение кода по файлам	2
2.1 Распределение кода по файлам . . . . .	2
2.1.1 Введение в разработку в нескольких файлах на примере задачи «Синонимы»	2
2.1.2 Механизм работы директивы <code>#include</code> . . . . .	9
2.1.3 Обеспечение независимости заголовочных файлов . . . . .	11
2.1.4 Проблема двойного включения . . . . .	12
2.1.5 Понятия объявления и определения . . . . .	14
2.1.6 Механизм сборки проектов, состоящих из нескольких файлов . . . . .	17
2.1.7 Правило одного определения . . . . .	27
2.1.8 Итоги . . . . .	29

# Разделение кода по файлам

## Распределение кода по файлам

### Введение в разработку в нескольких файлах на примере задачи «Синонимы»

До этого мы весь код хранили в одном файле. Но в общем случае это приводит к проблемам:

1. Для использования одного и того же кода в нескольких программах его приходится копировать;
2. Даже самое маленькое изменение программы приводит к её полной перекомпиляции;

Как говорит автор языка C++ Бьёрн Страуструп в своей книге «Язык программирования C++»: «Разбиение программы на модули помогает подчеркнуть ее логическую структуру и облегчает понимание». Рассмотрим это всё на примере кода нашей программы из прошлой недели. Здесь у нас есть логически не связанные друг с другом вещи. Первый кусок – само решение задачи «Синонимы»:

```
using Synonyms = map <string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms,
    const string& first_word) {
    return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
```

```
const string& second_word) {  
    return synonyms[first_word].count(second_word) == 1;  
}
```

Далее идёт наш юнит-тест фреймворк:

```
template <class T>  
ostream& operator<<(ostream& os, const set<T>& s) {  
    os << "{";  
    bool first = true;  
    for (const auto& x : s) {  
        if (!first) {  
            os << ", ";  
        }  
        first = false;  
        os << x;  
    }  
    return os << "}";  
}  
  
template <class K, class V>  
ostream& operator<<(ostream& os, const map<K, V>& m) {  
    os << "{";  
    bool first = true;  
    for (const auto & kv : m) {  
        if (!first) {  
            os << ", ";  
        }  
        first = false;  
        os << kv.first << ": " << kv.second;  
    }  
    return os << "}";  
}  
  
template <class T, class U>  
void AssertEqual(const T& t, const U& u, const string& hint) {  
    if (t != u) {  
        ostringstream os;  
        os << "Assertion failed: " << t << " != " << u <<  
            " hint: " << hint;  
        throw runtime_error(os.str());  
    }  
}
```

```
void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}

class TestRunner {
public:
    template<class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {
        try {
            func();
            cerr << test_name << " OK" << endl;
        } catch (runtime_error & e) {
            ++fail_count;
            cerr << test_name << " fail: " << e.what() << endl;
        }
    }

    ~TestRunner() {
        if (fail_count > 0) {
            cerr << fail_count << " unit tests failed. Terminate" << endl;
            exit(1);
        }
    }

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

Весь юнит-тест фреймворк логически не зависит от функций, которые решают нашу задачу.

Следующая логически независимая часть программы – это сами юнит-тесты:

```
void TestAddSynonyms() {
{
    Synonyms empty;
    AddSynonyms(empty, "a", "b");

    const Synonyms expected = {
        {"a", {"b"}},
        {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Empty");
}
{
```

```
Synonyms synonyms = {
    {"a", {"b"}},
    {"b", {"a", "c"}},
    {"c", {"b"}}
};
AddSynonyms(synonyms, "a", "c");

const Synonyms expected = {
    {"a", {"b", "c"}},
    {"b", {"a", "c"}},
    {"c", {"b", "a"}}
};
AssertEqual(synonyms, expected, "Nonempty");
}
}

void TestCount() {
{
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty, "a"), 0u, "Syn. count for empty dict");
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2u, "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms, "b"), 1u, "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms, "z"), 0u, "Nonempty dict, count z");
}
}

void TestAreSynonyms() {
{
    Synonyms empty;
    Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
    Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
}
{
    Synonyms synonyms = {
```



```
    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
};
Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
}
}
void TestAll() { // объединяем запуск всех юнит-тестов
    TestRunner tr;
    tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
    tr.RunTest(TestCount, "TestCount");
    tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}
```

Ещё у нас есть main:

```
int main() {
    TestAll();

    int q;
    cin >> q;
    Synonyms synonyms;

    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word);
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << GetSynonymCount(synonyms, word) << endl;
        } else if (operation_code == "CHECK") {
```

```
string first_word, second_word;
cin >> first_word >> second_word;
if (AreSynonyms(synonyms, first_word, second_word)) {
    cout << "YES" << endl;
} else {
    cout << "NO" << endl;
}
}
}
return 0;
}
```

Теперь рассмотрим вынесение в отдельные файлы в Eclipse. Итак, у нас в программе есть 4 логически обособленных компонента:

1. Функции решения нашей задачи;
2. Юнит-тест фреймворк;
3. Сами юнит-тесты;
4. Решение нашей задачи в main.

И довольно логично отделить эти части друг от друга, поместив их в отдельные файлы. Открываем наш проект Coursera: *Window* → *Show View* → *C/C++ Projects* (как это сделано на рис. 2.1). Нажимаем на него *\*project name\** → *New* → *Header File*. (см. 2.2) Вводим имя заголовочному файлу (test\_runner.h) и у нас создаётся пустой файл test\_runner.h.

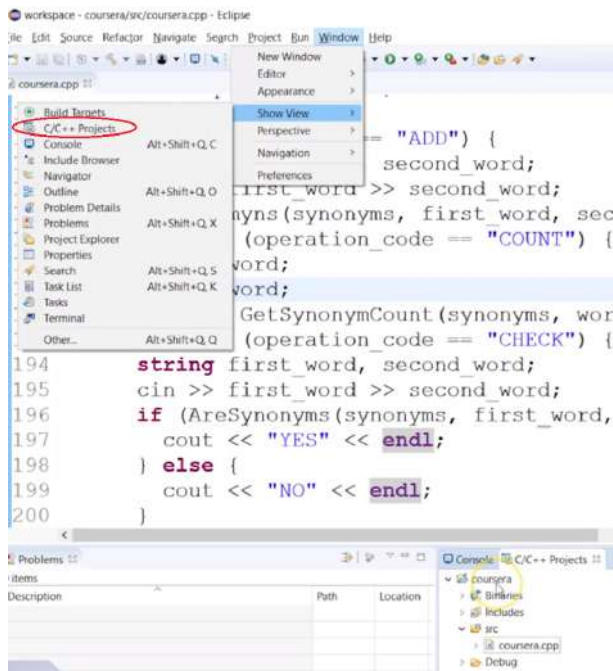


Рис. 2.1: Открытие C/C++ projects

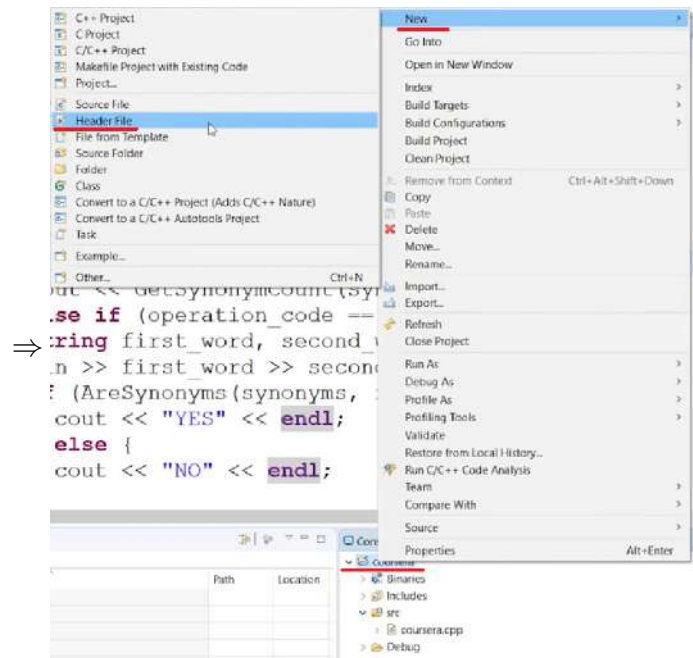


Рис. 2.2: New → Header File

Теперь из основного монолитного файла решения задачи «Синонимы» вырезаем сами юнит-тесты в `test_runner.h`. Теперь запустим нашу программу и она не скомпилируется, потому что мы, как минимум, не знаем, что такое **Assert**. Нам надо дописать в начало нашей программы

```
#include "test_runner.h" // подключаем файл с юнит-тестами
```

Теперь всё компилируется и работает. Аналогичным образом в файл `synonyms.h` вынесем функции самого решения задачи, а в файл `tests.h` вынесем все юнит-тесты и допишем:

```
#include "synonyms.h"
#include "tests.h"
```

Программа компилируется и тесты выполняются. Таким образом мы смогли разбить исходную программу на 4 файла, в каждом из которых лежат независимые блоки.

## Механизм работы директивы `#include`

Несмотря на кажущуюся корректность в выполнении этих операций, у нас есть немало проблем. И давайте посмотрим, какие это проблемы. Для примера закомментируем `#include <set>` в начале нашей программы:

```
#include <cassert>
#include <sstream>
#include <exception>
#include <iostream>
#include <string>
#include <map>
#include <vector>
// #include <set> // закомментировали
using namespace std;

#include "test_runner.h"
#include "synonyms.h"
#include "tests.h"

int main() {
    ...
}
// 'AddSynonyms' was not declared in this scope...
```

И ещё несколько ошибок. Компилятор пишет, что мы не объявили функцию `AddSynonyms`, хотя мы её объявляли в `test_runner.h`. Перед нами встаёт проблема: мы не можем понять, где именно возникает ошибка.

Теперь посмотрим на другую. Поменять инклюды из начала мы можем без проблем. А вот если сделать подключение `tests` не третьим, а вторым, программа снова выдаст нам ошибку.

Третья демонстрация: если мы перенесём подключения в начало программы (например, между подключениями `sstream` и `exception`), снова появится куча ошибок о необъявленных переменных.

Разберёмся, как работает `#include`:

- Директива `#include "file.h"` вставляет содержимое файла `file.h` в месте использования;
- Файл, полученный после всех включений, подаётся на вход компилятору.

Разберёмся на примере маленького проектика Sum. У нас есть два файла: `how_include_works.cpp` с самой программой...

```
#include "sum.h"
int main() {
    int k = Sum(3, 4);
    return 0;
}
```

...в которой подключается `sum.h` с функцией суммирования:

```
int Sum(int a, int b) {
    return a + b;
}
```

Переключимся в консоль операционной системы. Зайдём в нашу директорию и увидим там два файла: `how_include_works.cpp` и `sum.h`. (рис. 2.3)

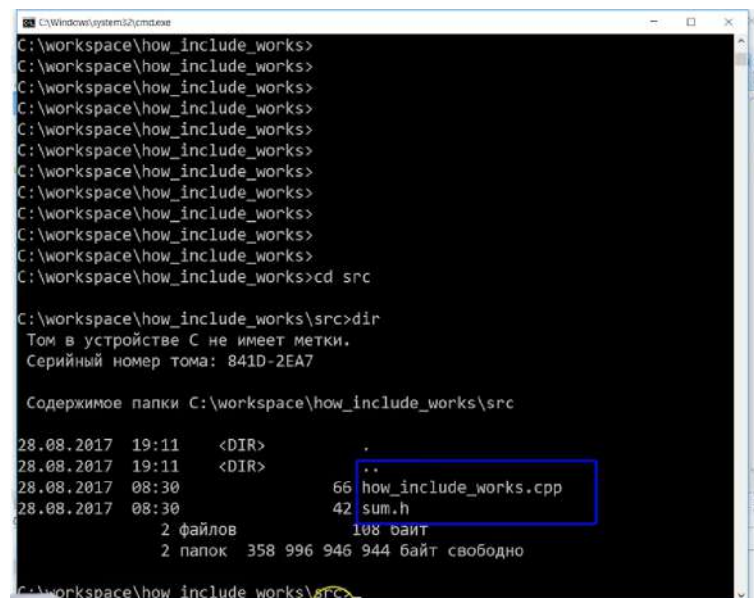


Рис. 2.3: Консоль cmd

Вызовем команду компилятора:

```
g++ -E how_include_works.cpp
```

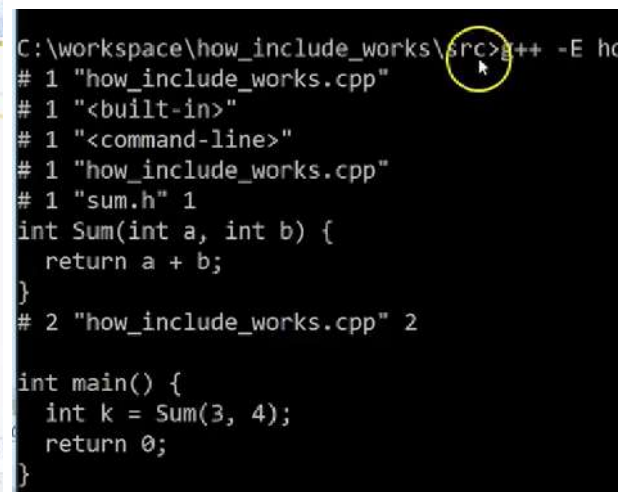


Рис. 2.4: Преппроессинг проекта

Вызов компилятора с флагом `-E` значит, что мы просим компилятор не выполнять полную

сборку проекта, а просто выполнить стадию препроцессинга (стадию выполнения директив `#include`). В итоге мы видим, что в файле есть функция `main`, а выше вставлен `sum.h` (рисунок 2.4). За символом `#` – уже служебные символы компилятора.

Теперь вернёмся к нашему большому проекту и посмотрим, как препроцессинг работает на нашем проекте тем же образом: в терминале `cmd.exe` переходим в директорию проекта и вводим:

```
g++ -E coursera.cpp > coursera.i
```

(чтобы результат препроцессинга вывелся в файл `coursera.i`)

Размер файла оказался 37980 строк после отрабатывания директив `include`. Содержимое каждого модуля было вставлено в файл с исходником. И само наше решение (`main` и все файлы, в которые мы до этого выносили части кода) начинается только с 37780 строки. А всё до этого – модули стандартных библиотек.

Отсюда и ответ на все те проблемы, которые мы получали: если мы убирали какую-то стандартную библиотеку, например `#include <set>`, нигде не было написано, что `set` – это множество, какие у него есть операции. И поэтому у нас возникала ошибка компиляции. При переносе тоже была ошибка, потому что в заголовочных файлах мы использовали функции и структуры, которые включались позже, и компилятор не мог их найти.

## Обеспечение независимости заголовочных файлов

Избавляемся от одной из проблем, описанных выше. Нам надо, чтобы наши файлы были независимыми и порядок включения не влиял на компилируемость программы.

Решение: включим в каждый файл проекта те заголовочные файлы, которые ему нужны. Начнём с `test_runner.h`. Ему нужны `set`, `map`, `ostream` и `string`. Просто перенесём эти включения из основного файла в `test_runner.h`:

```
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std; // попытаемся добавить, хотя так делать не стоит
```

Программа компилируется. Тогда попробуем поставить `#include "test_runner.h"` самым первым в нашем основном файле. Но программа не компилируется, потому что файлу `synonyms.h` нужно знать `map`, `string`, `set`, а он об этом сейчас не знает, ведь мы подключаем файлы в данном порядке:

```
#include "synonyms.h"
#include "test_runner.h"

#include <exception>
#include <iostream>
#include <vector>

using namespace std;

#include "tests.h"
```

Раньше `synonyms.h` стоял после `test_runner.h` и получал из него все нужные `include`'ы. Теперь поставим и его (`synonyms.h`) вперёд и добавим все необходимые `include`'ы:

```
#include <map>
#include <set>
#include <string>
using namespace std; // попытаемся добавить, хотя так делать не стоит
```

Теперь всё компилируется.

Рассмотрим функцию `main()`. Он состоит из функции `TestAll()` и кода, который решает задачу. В этом конкретном файле мы нигде не используем наш фреймворк. Значит, `test_runner.h` нам в этом файле не нужен. Он нужен в `tests.h`, потому что именно они используют тестовый фреймворк. Таким образом мы сделали `test_runner.h` и `synonyms.h` независимыми, и подключать их можно в любом порядке до функции `main()`.

## Проблема двойного включения

Функция `AddSynonyms()` в `tests.h` определена в `synonyms.h`, и если мы поставим `tests.h` перед `synonyms.h`, наш проект не скомпилируется. Тогда добавим в `tests.h` все зависимости, в частности, `synonyms.h` и скомпилируем:

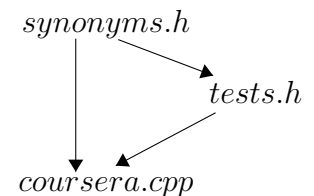
```
#include "test_runner.h"
```

```
#include "synonyms.h"
...
// redefinition of "bool AreSynonyms...."
```

Программа не компилируется, причём со странными ошибками о переопределении наших функций. Для того, чтобы понять, что произошло, вернёмся к маленькой задачке. Продублируем строчку `#include "sum.h"`.

```
#include "sum.h"
#include "sum.h"
int main() {
    int k = Sum(3, 4);
    return 0;
}
// redefinition of "int Sum...."
```

После компиляции увидим ту же ошибку. Теперь получим препроцессинг проекта, как на рисунках 2.3 и 2.4. Заметим, что в файле получилось две функции `sum`. Когда компилятор это видит, он выкидывает ошибку компиляции **Redefinition**, т.е. повторное определение. Точно та же ситуация у нас в большом проекте: в `coursera.cpp` включается `tests.h`, который подключает `synonyms.h`, который так же включается в `coursera.cpp`. Таким образом у нас получается переопределение всего `synonyms.h`.



Избежать двойного включения очень просто: добавляем в начало каждого заголовочного файла `#pragma once`. В нашем случае дописываем в `synonyms.h` (и уже сейчас всё заработает), `tests.h` и `test_runner.h`. Эта директива говорит компилятору игнорировать все повторные включения.

Также добавим в `sum.h` эту строчку и проверим, что всё работает. Выполним его препроцессинг и увидим, что функция `sum` там встречается только один раз. Здесь мог возникнуть вопрос: «Почему препроцессор не отслеживает, что заголовочные файлы включаются несколько раз, и почему препроцессор по умолчанию не выкидывает повторные включения?» Потому что C++ делался обратно совместимым с C, и вообще C++ развивается так, чтобы не терять обратную совместимость. Но мы можем забывать каждый раз прописывать эту строчку в каждом заголовочном файле, и тут нам на помощь приходит IDE. В Eclipse оно работает так: *Window* → *Preferences* → *C/C++* → *Code Style* → *Code Templates* → *Files* → *C++ Header File* → *Default C++ header template*, там нажимаем *Edit* и у нас открывается окно ввода шаблона, который будет вставляться во все заголовочные файлы, которые мы создаём. Сюда можно добавлять специальные макропеременные, которые вставляют ваше имя, дату создания файла, имя проекта и так далее. Но вот мы сюда прямо и напишем `#pragma once`, перевод строки. ОК, Apply,



Apply and Close. Снова создадим новый header-файл, как на рисунке 2.2. Как только мы его создали, он по умолчанию сразу идёт с вставленным шаблоном.

## Понятия объявления и определения

Когда у нас есть большой проект, в котором много файлов, то мы, естественно, не можем помнить досконально, в каком файле какие функции есть. И очень часто хочется, открыв файл, понять интерфейс этого файла, то есть понять, какие функции и классы в этом файле есть. Т. е. зайти в файл и сразу увидеть его интерфейс. Нам придётся пролистать весь файл, чтобы понять, что за функции в нём есть. Хотелось бы короткий список функций. В Eclipse можно нажать Ctrl+O и получить краткий список с названиями и типами функций.

Иногда хочется видеть только интерфейс – список функций и классов, которые там есть. Нас не будет интересовать, как это работает (допускаем, что оно работает). Нас интересует только, что мы с ним можем делать. Введём два новых определения:

- **Объявление функции (function declaration)** – сигнатура функции (возвращаемый тип, имя функции и список параметров с типами). Оно говорит, что где-то в программе есть функция с заданными параметрами;

```
int GreatestCommonDivisor(int a, int b);
```

- **Определение функции (function definition)** – сигнатура + реализация функции.

```
int GreatestCommonDivisor(int a, int b) {  
    while (a > 0 && b > 0) {  
        if (a > b) {  
            a %= b;  
        } else {  
            b %= a;  
        }  
    }  
    return a + b;  
}
```

**Функция может быть объявлена несколько раз, но определена должна быть только в одном месте.** Ещё важно, чтобы все объявления функции были одинаковыми.

На простом примере разберёмся, как это работает. Итак, у нас есть

```
void foo() {
    bar();
}
void bar() {
}
int main() {
    return 0;
}
// "bar" was not declared in this scope
```

Функция `bar` не объявлена и файл не компилируется. Теперь в самом начале файла объявим функцию `bar`:

```
void bar(); // добавили в самое начало программы
```

Теперь всё заработало. И даже если объявлений будет много, программа будет компилироваться. А вот если мы продублируем определение, то всё сломается и мы получим `redefinition error`. Это было насчёт определения и объявления функций. Аналогично у нас будет и для классов:

- **Объявление класса (class declaration)** – объявление класса, его поля и методы. Но методы не реализованы:

```
class Rectangle {
public:
    Rectangle(int width, int height);
    int Area() const;
    int Perimeter() const;

private:
    int width, height;
};
```

- **Определение методов класса (class methods definition)**

```
Rectangle::Rectangle(int w, int h) { // по принципу: имя класса::имя метода
    width = w;
    height = h;
}
int Rectangle::Area() const {
```

```
    return width * height;
}
int Rectangle::Perimeter() const{
    return 2 * (width + height);
}
```

Теперь вспомним, а зачем оно нам: мы хотели в начале файла видеть объявления всех функций и классов, которые есть в файле. Сделаем это для нашего большого проекта. Допишем в tests.h:

```
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();
```

Теперь аналогично сделаем для synonyms.h и test\_runner.h. Причём во второй у нас есть шаблоны функций, которые точно так же стоит объявить в начале:

```
void AddSynonyms(Synonyms& synonyms,
    const string& first_word, const string& second_word);
bool AreSynonyms(Synonyms& synonyms,
    const string& first_word, const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);
```

```
template <class T> // копируем объявление шаблонов
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name);
    ~TestRunner();

private:
```

```
int fail_count = 0;
};
```

Итоги:

- Объявление в начале файла сообщает компилятору, что функция/класс/шаблон где-то определены;
- Объявлений может быть несколько. Определение – только одно;
- Группировка объявлений в начале файла позволяет узнать, какие функции и классы в нём есть, не вникая в их реализацию.

## Механизм сборки проектов, состоящих из нескольких файлов

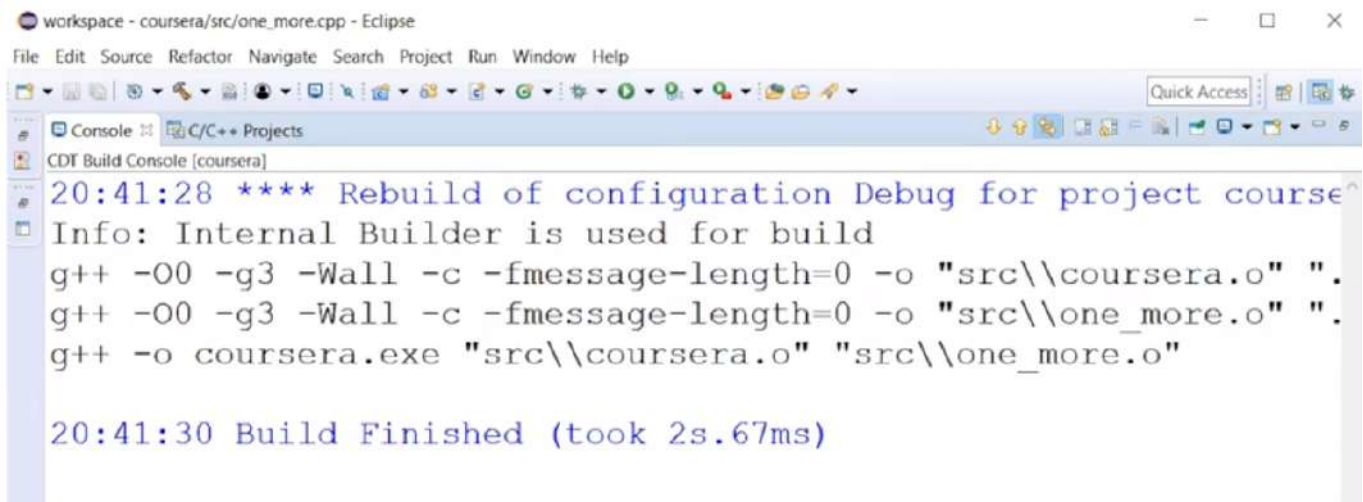
Когда мы начинали разговор о разделении кода на несколько файлов, то в качестве одного из недостатков хранения всего кода в одном файле мы называли то, что при минимальном изменении программы у нас она пересобирается вся, в случае, если весь код лежит в одном файле. Сейчас мы разделили код нашего проекта на целых четыре файла. Но при этом каждый раз, когда мы меняем что угодно в нашем проекте, он все равно пересобирается целиком. Почему это происходит? Потому что у нас есть файл `coursea.cpp`, в который так или иначе включаются с помощью директивы `#include` три других наших файла. Соответственно, если мы в них что-нибудь меняем, то они вставляются в наш `coursea.cpp`, и вся программа перекомпилируется целиком. Но давайте подумаем. Например, есть у нас функции в `"synonyms.h"`, которые умеют работать со словарём синонимов – добавлять в него, проверять количество синонимов. Есть эти функции и есть тесты на них. Если мы внесем изменения в тесты, например, добавим какой-нибудь ещё тестовый случай, например, в тест на `TestCount`, давайте там проверим, что при пустом словаре и для строки `b` у нас тоже вернётся ноль. Если мы поменяли тесты, то нам нет никакой необходимости перекомпилировать сами функции. Но мы все равно перекомпилируем всё.

Нам надо не пересобирать проект целиком при изменении в конкретном месте. Разберёмся в механике сборки проектов в C++. Посмотрим на расширения файлов в нашем проекте:

- `tests.h`
- `synonyms.h`

- test\_runner.h
- coursera.cpp

Пока у нас 3 файла **.h** и только один файл **.cpp**. Добавим ещё один, как на картинке 2.2: *project name* → *New* → *Source File* и назовём его `one_more.cpp`. Очистим результаты сборки (*project name* → *Clean Project*) и соберём проект с нуля. Запустим сборку и после её завершения посмотрим, какие команды выполнял Eclipse в процессе сборки проекта:



```

workspace - coursera/src/one_more.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
20:41:28 **** Rebuild of configuration Debug for project course
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\coursera.o" ".
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\one_more.o" ".
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"

20:41:30 Build Finished (took 2s.67ms)
  
```

Рис. 2.5: Команды по сборке проекта

Первый раз он запускался для файла `coursera.cpp`. Второй раз он запускался для нашего только что добавленного файла `one_more.cpp`. В результате было получено два файла с расширением `.o`. Вот этот параметр `-o` задает имя выходного файла, поэтому по значению параметра `-o` мы можем понимать, какие выходные файлы формировались в этой стадии. И потом была третья стадия, в которой на вход были поданы вот эти файлы с расширением `.o`, а на выходе получился исполняемый файл `coursera.exe`. Этот пример демонстрирует, каким образом выполняется сборка проектов на C++, состоящих из нескольких файлов.

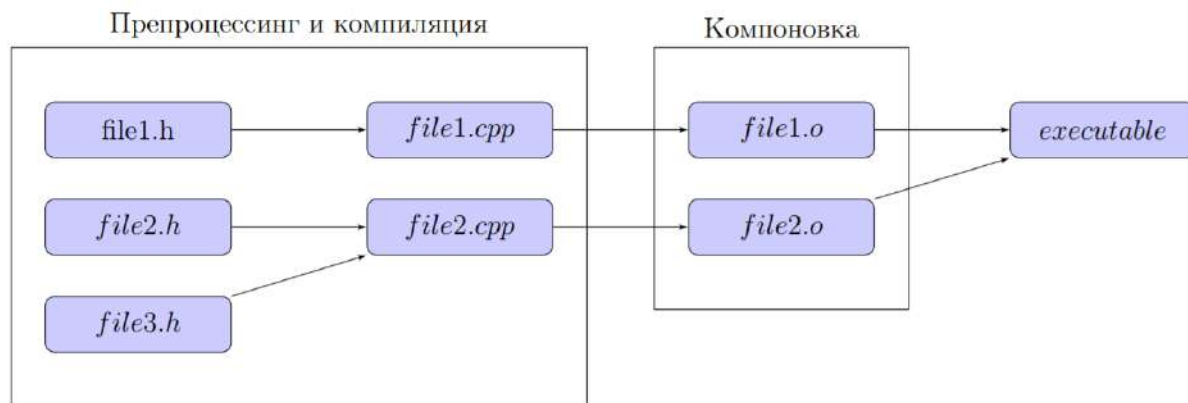


Рис. 2.6: Компиляция нескольких файлов

Как мы уже видели, первая стадия – это препроцессинг, когда выполняются все директивы `include`. Далее, после того как препроцессинг выполнен, берётся каждый отдельный `.cpp` файл и компилируется. В результате компиляции каждого `.cpp` файла получается так называемый объектный файл. Вот на схеме (рисунок 2.6) у нас объектные файлы изображены как файлы с расширением `.o`. И затем начинается третья стадия – это стадия компоновки, когда берутся все объектные файлы, которые у нас получились, и компонуются в один исполняемый файл.

Теперь, если мы в наш `one_more.cpp` добавим какое-нибудь изменение, например, комментарий, запустим сборку и посмотрим на команды консоли, видим, что теперь вместо всего проекта перекомпилировался только `one_more.cpp` и потом был собран исходный файл `coursera.exe`. Аналогично внесём изменения в `coursera.cpp` и запустим сборку. В консоли увидим, что `one_more.cpp` не был тронут, и перекомпилировался только `coursera.cpp`. Если мы изменим любой из `.h` файлов, подключаемых в `coursera.cpp`, увидим то же самое.

```

20:44:29 **** Incremental Build of configuration Debug for proj...
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\one_more.o" "src\one_more.cpp"
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"
20:44:29 Build Finished (took 590ms)

```

Рис. 2.7: Сообщения в консоли

Вывод:

1. При сборке проекта компилируются только изменённые **.cpp**-файлы;
2. Внесённые изменения в **.h** файл приводит к перекомпиляции всех **.cpp**-файлов, в которые он включён;
3. Если перенести определения функций и методов классов в **.cpp**-файлы, то они будут пересобираются только после изменений.

Теперь используем эти знания на нашем проекте, чтобы при небольшом изменении наш код реже пересобирался. Определения функций и методов классов переносим в .cpp файлы, а в заголовочных файлах оставляем только объявления. Мы логически не связанные друг с другом определения разнесём в разные файлы. Когда мы меняем, например, определения тестов, то определения функций, которые эти тесты покрывают, не меняются, и соответственно, они не будут перекомпилироваться. Таким образом мы минимизируем количество .cpp-файлов, которые нужно перекомпилировать при каждом изменении программы.

Давайте выполним такое преобразование с нашим проектом, то есть вынесем определение в .cpp-файл. И начнем вот, например, с test\_runner.h. Добавим в наш проект файл test\_runner.cpp. И вынесем в него определения. Здесь есть нюанс (далее в курсе мы это разберём) с шаблонами, так что пока их переносить в .cpp-файл мы не будем.

```
#include "test_runner.h"
void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}
TestRunner::~TestRunner() {
    if (fail_count > 0) {
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1);
    }
}
```

Таким же образом с synonyms.cpp и tests.cpp:

```
#include "synonyms.h"
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    synonyms[second_word].insert(first_word);
}
```

```

    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word) {
    return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

```

```

#include "tests.h"
void TestAddSynonyms() {
    {
        Synonyms empty;
        AddSynonyms(empty, "a", "b");
        const Synonyms expected = {
            {"a", {"b"}},
            {"b", {"a"}},
        };
        AssertEqual(empty, expected, "Empty");
    }
    {
        Synonyms synonyms = {
            {"a", {"b"}},
            {"b", {"a", "c"}},
            {"c", {"b"}}
        };
        AddSynonyms(synonyms, "a", "c");
        const Synonyms expected = {
            {"a", {"b", "c"}},
            {"b", {"a", "c"}},
            {"c", {"b", "a"}}
        };
        AssertEqual(synonyms, expected, "Nonempty");
    }
}

void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"), 0,

```



```
    "Syn. count for empty dict");
    AssertEqual(GetSynonymCount(empty, "b"), 0u,
        "Syn. count for empty dict b");
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2u,
        "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms, "b"), 1u,
        "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms, "z"), 0u,
        "Nonempty dict, count z");
}
}

void TestAreSynonyms() {
    {
        Synonyms empty;
        Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
        Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},
            {"c", {"a"}}
        };
        Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
        Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
        Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
        Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
        Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
        Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
    }
}

void TestAll() {
```

```

TestRunner tr;
tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
tr.RunTest(TestCount, "TestCount");
tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}

```

А в самих .h файлах у нас остаётся:

```

#pragma once
#include "test_runner.h"
#include "synonyms.h"
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();

```

```

#pragma once
#include <map>
#include <set>
#include <string>
using namespace std;
using Synonyms = map<string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word);
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);

```

```

#pragma once
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std;

template <class T>
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);

```

```
template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
public:
    template <class TestFunc>
    void RunTest (TestFunc func, const string & test_name);
    ~TestRunner();
private:
    int fail_count = 0;
};

template <class T>
ostream& operator << (ostream& os, const set<T>& s) {
    os << "{";
    bool first = true;
    for (const auto& x : s) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << x;
    }
    return os << "}";
}

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m) {
    os << "{";
    bool first = true;
    for (const auto & kv:m) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << kv.first << ": " << kv.second;
    }
    return os << "}";
}
```

```
}

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint) {
    if (t != u) {
        ostringstream os;
        os << "Assertion failed: " << t << " != " << u << " hint: " << hint;
        throw runtime_error(os.str());
    }
}

template <class TestFunc>
void TestRunner::RunTest (TestFunc func, const string& test_name) {
    try {
        func ();
        cerr << test_name << " OK" << endl;
    } catch (runtime_error & e) {
        ++fail_count;
        cerr << test_name << " fail: " << e.what () << endl;
    }
}
```

Вспомним, что у нас есть и основной файл с решением:

```
#include "tests.h"
#include "synonyms.h"
#include <exception>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    TestAll();
    int q;
    cin >> q;
    Synonyms synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") {
            string first_word, second_word;
```

```
    cin >> first_word >> second_word;
    AddSynonyms(synonyms, first_word, second_word);
} else if (operation_code == "COUNT") {
    string word;
    cin >> word;
    cout << GetSynonymCount(synonyms, word) << endl;
} else if (operation_code == "CHECK") {
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Таким образом, весь наш проект представляет собой 7 файлов:

1. **coursera.cpp** – главный файл с `main()`, в котором лежит решение нашей задачи;
2. **synonyms.h** – объявления функций, решающих нашу задачу и **synonyms.cpp** – определения этих самых функций;
3. **test\_runner.h** и **test\_runner.cpp** – определения и объявления функций и классов, связанных с юнит-тестированием;
4. **tests.h** – объявления тестирующих функций и **test.cpp** – их определение.

Если внести изменения в `test_runner.h`, то у нас пепесоберётся всё: `test_runner.cpp`, `tests.cpp`, `coursera.cpp`. Потому что `coursera.cpp` включает в себя `test.h`, который включает в себя `test_runner.h`. Таким образом:

1. Сборка проектов состоит из трёх стадий: препроцессинг, компиляция и компоновка;
2. При повторной сборке проекта компилируются только изменённые `.cpp`-файлы;

3. Внесение определений в .cpp-файлы позволяет при каждой сборке компилировать только изменённые файлы;
4. Это сильно ускоряет пересборку проекта.

## Правило одного определения

Мы ранее говорили, что объявлений может быть сколько угодно, а определение обязательно должно быть ровно одно. И давайте мы ещё раз это продемонстрируем: вот у нас есть функция, например, `GetSynonymCount`, и у неё есть определение в файле `synonyms.cpp`. Если мы просто возьмём и скопируем это определение, а потом запустим компиляцию, то мы получим знакомую ошибку `redefinition`. Однако в больших проектах бывают ситуации, когда в вашем проекте, вроде бы, есть всего одно определение функции, но при этом компилятор сообщает вам, что у вас одна и та же функция определена несколько раз. И давайте посмотрим, как это выглядит и по какой причине случается. Давайте, например, возьмём нашу функцию `GetSynonymCount` и перенесём её определение обратно в заголовочный файл, как было у нас несколькими видео ранее. И скомпилируем наш проект. Давайте мы его соберём. И что-то пошло не так. Нам компилятор написал `first defined here`. А в консоли увидим "multiple definition of `GetSynonymsCount`". Вроде определение функции одно, но ошибка возникает. Посмотрим на строки запуска компилятора:

```
21:09:19 **** Incremental Build of configuration Debug for proj
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\coursera.o" ".
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\tests.o" "..\
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\synonyms.o" ".
g++ -o coursera.exe "src\\coursera.o" "src\\synonyms.o" "src\\t
src\\synonyms.o: In function `std::_Rb_tree<std::_cxx11::basic_
c:/dev/mingw-w64/mingw64/lib/gcc/x86_64-w64-mingw32/7.1.0/inclu
src\\coursera.o:C:\workspace\coursera\Debug\../src/synonyms.h:18
src\\tests.o: In function `std::_Rb_tree<std::_cxx11::basic_str
C:\workspace\coursera\Debug\../src/synonyms.h:18: multiple defi
src\\coursera.o:C:\workspace\coursera\Debug\../src/synonyms.h:18
collect2.exe: error: ld returned 1 exit status

21:09:25 Build Finished (took 5s.944ms)
```

Рис. 2.8: Настройка компилятора

Каждый .cpp файл успешно скомпилировался. На этапе компоновки возникает ошибка.

# Multiple definition of GetSynonymCount

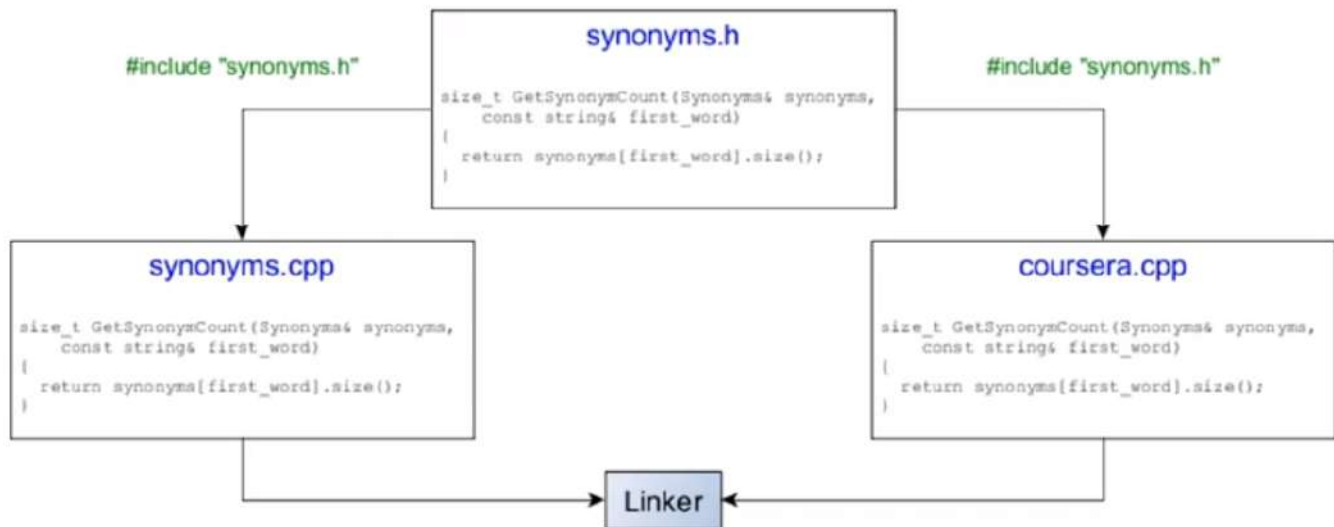


Рис. 2.9: Схема компиляции и сборки проекта

Ошибка происходит, когда компоновщик видит два определения одной и той же `GetSynonymCount` уже на этапе компоновки двух разных `.cpp`-файлов. Он видит, что одна и та же функция определена в двух объектных файлах и сообщает об ошибке. Вспомним, что основной причиной разделения на файлы было ускорение сборки. Теперь у нас есть ещё одна причина помещать определения в `.cpp`-файлы – это позволяет избежать ошибки `Multiple definitions`. Таким образом:

1. В C++ есть One Definition Rule (ODR);
2. Если функция определена в `.h`-файле, который включается в несколько `.cpp`-файлов, то нарушается ODR;
3. Чтобы не нарушать ODR, все определения надо помещать в `.cpp`-файлы.

## Итоги

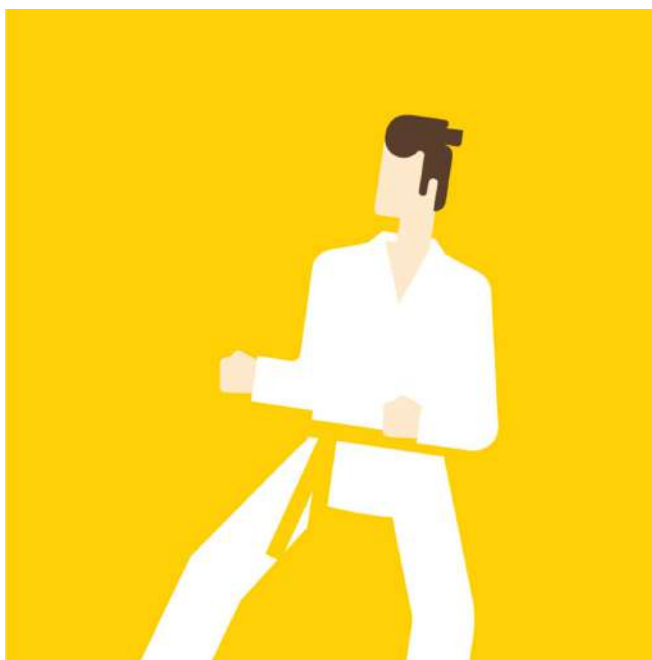
1. Разбиение программы на файлы упрощает её понимание и переиспользование кода, а также ускоряет перекомпиляцию;
2. В C++ есть два типа файлов: заголовочные (чаще .h) и файлы реализации .cpp;
3. Включение одного файла в другой осуществляется с помощью директивы `#include`;
4. Чтобы избежать двойного включения, надо добавлять `#pragma once`;
5. Знаем, что такое объявления и определения. Объявлений может быть сколько угодно, а определение только одно (ODR);
6. В .h-файлы обычно помещают объявления, а в .cpp – определения;
7. Если помещать определения в .h-файлы, то возможно нарушение ODR на этапе компоновки.



# Основы разработки на C++: жёлтый пояс

Неделя 4

Итераторы, алгоритмы, контейнеры



# Оглавление

<b>Итераторы, алгоритмы, контейнеры</b>	<b>3</b>
4.1 Введение в итераторы . . . . .	3
4.1.1 Введение в итераторы . . . . .	3
4.1.2 Концепция полуинтервалов итераторов . . . . .	5
4.1.3 Итераторы множеств и словарей . . . . .	8
4.1.4 Продвинутое итерирование по контейнерам . . . . .	9
4.2 Использование итераторов в алгоритмах и контейнерах . . . . .	11
4.2.1 Использование итераторов в методах контейнеров . . . . .	11
4.2.2 Использование итераторов в алгоритмах . . . . .	12
4.2.3 Обратные итераторы . . . . .	14
4.2.4 Алгоритмы, возвращающие набор элементов . . . . .	15
4.2.5 Итераторы <code>inserter</code> и <code>back_inserter</code> . . . . .	17
4.2.6 Отличия итераторов векторов и множеств . . . . .	18
4.2.7 Категории итераторов, документация . . . . .	19
4.3 Очередь, дек и алгоритмы поиска . . . . .	20

4.3.1	Стек, очередь и дек . . . . .	20
4.3.2	Алгоритмы поиска . . . . .	22
4.3.3	Анализ распространённых ошибок . . . . .	25

# Итераторы, алгоритмы, контейнеры

## 4.1. Введение в итераторы

### 4.1.1. Введение в итераторы

Рассмотрим задачу. Пусть у нас есть вектор строк с языками программирования:

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

using namespace std;

int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
    return 0;
}
```

Зададимся целью найти в нашем векторе язык, начинающийся с буквы 'C' или сказать, что такого языка нет. Можно написать цикл `for` и проверять первую букву каждого языка, но чтобы знать, нашёлся язык или нет, нам нужно хранить флажок типа `bool` и с каждым шагом всё больше шанс ошибиться. Но существует стандартный алгоритм `find_if()`, принимающий начало диапазона, конец диапазона и лямбда-функцию, которая ищет нужный язык. Дописываем строчку:

```
#include <algorithm> // подключаем модуль с алгоритмами
...
auto result = find_if(
    begin(langs), end(langs), [](const string& lang) { // лямбда-функция по языкам
        return lang[0] == 'C'; // возвращает true, если нулевая буква = 'C'
    });
cout << *result; // вызываем * от find_if, чтобы обратиться к найденному элементу
// C++
```

Видим, что нашли первый подходящий язык. Этот результат можно сохранить в какую-нибудь строку:

```
string& ref = *result; // сохраняем в строку
cout << ref << endl; // выводим
```

Причём, поскольку `*result` – неконстантная ссылка, мы по ней можем поменять элемент. Например:

```
string& ref = *result; // сохраняем в строку C++
ref = "D++";
cout << *result << endl;
// D++
```

А если у нас в векторе лежат не строки, а более сложные структуры:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang { // у каждого языка программирования есть название и возраст
    string name;
    int age;
};
int main() {
    vector<Lang> langs = // тоже поменяли на Lang
        {"Python", 26},
        {"C++", 34},
        {"C", 45},
        {"Java", 22},
        {"C#", 17}};

    auto result = find_if(
        begin(langs), end(langs),
        [](const Lang& lang) { // лямбда-функция немного меняется
            return lang.name[0] == 'C';
        });
    if (result == end(langs)) { // если результат выводит ссылку на конец диапазона
        cout << "Not found"; // значит, элемент не найден
    } else { // иначе мы нашли элемент
        cout << (*result).age << endl; // выводим возраст первого попавшегося языка на C
    }
}
```

```
    return 0;
}
// 34
```

Если результат не нашёлся, то `result` указывает на конец диапазона. Это значит, что мы можем спокойно проверить на наличие. Кроме того, обращаться можно короче:

```
cout << result->age << endl; // более удобная форма
```

А для языков на букву 'D' получим: `Not Found`. И `begin()`, `end()` и `result` указывают на какую-то позицию в контейнере. Все типы, указывающие на какую-то позицию контейнера, называются **итераторами**. А операция `*` над итератором называется **разыменованием итератора**. Выведем начало контейнера:

```
cout << begin(langs)->name << " " << langs.begin()->age << endl;
// Python 26
```

А вот `end(langs)` уже указывает не на последний элемент контейнера, а на конец. Заметим, что к началу можно обращаться методом `langs.begin()`.

#### 4.1.2. Концепция полуинтервалов итераторов

Итераторы есть не только у вектора, но и у любого контейнера. Например, у строки:

```
#include <string>
...
string lang = langs[1].name;
auto it = begin(lang);
cout << *it;
// C
```

`begin()` у строки указывает на её первый символ. Пока что мы с их помощью только получали элемент контейнера, но из названия следует, что с их помощью можно итерироваться по контейнеру. Допишем:

```
++it; // получим следующий символ
cout << *it
// C+
```

Теперь мы вывели сначала нулевой элемент, а затем первый. Давайте с помощью цикла `for` проитерируемся по вектору `langs`:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang {
    string name;
    int age;
};
int main() {
    vector<Lang> langs = // тоже поменяли на Lang
        {"Python", 26},
        {"C++", 34},
        {"C", 45},
        {"Java", 22},
        {"C#", 17}};
    for (auto it = begin(langs); it != end(langs); ++it) {
        cout << it-> name << " ";
    }
    return 0;
}
// Python C++ C Java C#
```

Таким образом мы вывели все языки. Когда итератор показывает на `begin(langs)`, выводится первый элемент. Итератор двигается вправо, проверяет, что он не достиг конца и т. д. Выводим последний элемент и сдвигаем итератор на `end`. Т. е. `end(langs)` – это итератор сразу за последним элементом. Получается полуинтервал  $[begin, end)$ .

Для 5 элементов получаем 5 элементов и `end`. А вот пустой диапазон представляется из равных итераторов `begin` и `end`. Попробуем обратиться к полю `end(langs)`. Код компилируется, но падает, потому что там либо чужая память, либо вообще ничего не лежит. Напишем универсальную функцию `PrintRange`, принимающую два итератора на `Lang`:

```
using LangIt = vector<string>::iterator; // укорачиваем запись
void PrintRange(LangIt range_begin, // начало
    LangIt range_end) { // конец
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " "; // мы не хотим переопределять вывод структур
    }
}
```

```

}
int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
    PrintRange(begin(langs), end(langs));
    return 0;
}
// Python C++ C Java C#

```

Всё снова работает. Вспомним, что итераторы есть для разных контейнеров, и хотелось бы написать универсальный итератор. Цикл `for` уже достаточно универсален. Напишем шаблонную функцию:

```

template <typename It> // написали шаблонную функцию
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора
                It range_end) { // и тут
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " ";
    }
}
...
PrintRange(begin(langs), end(langs)); // проходимся по контейнеру с языками
PrintRange(begin(langs[0]), end(langs[0])); // проходимся по контейнеру с python
// Python C++ C Java C# P y t h o n

```

Теперь, используя концепцию полуинтервалов, попробуем вывести половину векторов по отдельности, например, все языки строго до 'C' и отдельно все языки начиная с него:

```

auto border = find(begin(langs), end(langs), "C"); // раздел
PrintRange(begin(langs), border); // Python, C++
PrintRange(border, end(langs)); // C, Java, C#

```

Мы просто разбили по нужному языку наш полуинтервал на два диапазона.  
Заключение:

- Итератор – способ задать позицию в контейнере;
- `begin(c)` и `end(c)` – границы полуинтервала [*begin* , *end*);
- Алгоритмы часто принимают пару итераторов, образующую полуинтервал;
- С помощью шаблонов легко написать свой универсальный алгоритм, например, `PrintRange`.



### 4.1.3. Итераторы множеств и словарей

Давайте посмотрим, как работает итератор от других контейнеров. Увидим, что по сути они работают точно так же. Например, множество. Давайте возьмем наш вектор языков и заменим его на множество языков.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set> // подключили множества
using namespace std;
template <typename It> // написали шаблонную функцию
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора
               It range_end) { // и тут
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " ";
    }
}

int main() {
    set<string> langs = {"Python", "C++", "C", "Java", "C#"};
    PrintRange(begin(langs), end(langs));
    // тоже выводит названия, но в алфавитном порядке
    auto it = langs.find("C++");
    PrintRange(begin(langs), it);
    // выведем все языки до C++, в отсортированном порядке
    return 0;
}
// C C# C++ Java Python C C#
```

Получим вывод сначала всех строк, а потом строк, которые лексикографически меньше C++. Теперь рассмотрим словарь. Но для вызова `PrintRange` должен быть определён оператор `<<`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map> // подключили словари
using namespace std;
void PrintRange...
int main() {
    map<string, int> langs = // словарь Имя-возраст
```

```

    {"Python", 26},
    {"C++", 34},
    {"C", 45},
    {"Java", 22},
    {"C#", 17}];
return 0;
}

```

Какой тип у элементов словаря? Когда вы итерировались по нему `range based for`'ом, этот тип был парой. И здесь `*it` – это пара. Переопределять оператор вывода для словаря мы уже умеем. Создадим функцию `PrintMapRange` для вывода словаря:

```

void PrintMapRange(It range_begin, It range_end) {
    for (auto it = range_begin; it != range_end; ++it){
        cout << it->first << '/' << it->second << " ";
    }
}

...
auto it = langs.find("C++");
PrintMapRange(begin(langs), it); // вызовем наш
return 0;
}
// C/45 C#/17

```

Видим, что вывелись нужные нам языки и их возраст.

#### 4.1.4. Продвинутое итерирование по контейнерам

Продemonстрируем, что итераторы универсальнее, чем `range-based for`, что их можно использовать в гораздо более широком числе случаев. Вернёмся к множеству языков. Сделаем множество строк с языками. И с помощью итераторов выведем множество в обратном порядке с помощью `--it`.

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
}

```

```
auto it = end(langs);
while (it != begin(langs)) {
    --it;
    cout << *it << " ";
}
}
// C# Java C C++ Python
```

Всё работает. Итерирование в обратную сторону работает так: проверяем, что `it != begin`, двигаем его влево и уже можем вывести последний элемент. И т. д., пока не доходим до начала. Мы вывели вектор в обратном порядке. Кроме того, тот же код можно переписать в виде:

```
while (it != begin(langs)) { // заметим, что нельзя делать --it от it = begin
    --it;
    cout << *it << " ";
}
```

### Опасные операции над итераторами:

- `*end(c)` – обращаться к элементу после последнего;
- `auto it = end(c); ++it;` – смотреть на элемент после `end`;
- `auto it = begin(c); --it;` – смотреть на элемент до `begin`.

Итераторы очень похожи на ссылки, но есть разница.

### Отличия итераторов от ссылок:

- Итераторы могут указывать «в никуда» – на `end`. Ссылка всегда привязана к чему-то;
- Итераторы можно перемещать на другие элементы:

```
vector<int> numbers = {1, 3, 5};
auto it = numbers.begin();
++it; // it указывает на 3
int& ref = numbers.front();
++ref; // теперь numbers[0] == 2
```

В итоге мы узнали, что у всех контейнеров есть итераторы, и чем они отличаются от ссылок. И итераторы предоставляют универсальный способ обхода контейнеров.

## 4.2. Использование итераторов в алгоритмах и контейнерах

### 4.2.1. Использование итераторов в методах контейнеров

Где мы раньше встречали итераторы?

1. В алгоритмах, например `sort`, `count`, `count_if`, `reverse`, `find_if`;
2. Конструкторы вектора и множества;
3. Метод `find` у множества.

Рассмотрим методы контейнеров на примере вектора строк.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = find(begin(langs), end(langs), "C++"); // получаем позицию C++
langs.erase(it); // метод удаления элемента из контейнера по итератору it
langs.insert(begin(langs), "C++") // вставка перед итератором элемента C++
langs.erase(it, end(langs)); // удалили все с C++ и до конца
// C# Java C C++ Python
```

У вектора длины 5 есть 6 позиций для вставки от `v.insert(begin(), value)` до `v.insert(end(), value)`. Вставка в произвольное место вектора:

- Вставка диапазона

```
v.insert(it, range_begin, range_end);
// вставит диапазон [range_begin, range_end) в позицию it
```

- Вставка элемента несколько раз

```
v.insert(it, count, value);
// count раз вставляет элемент value в позицию it
```

- Вставка нескольких элементов

```
v.insert(it, {1, 2, 3});
// вставляет 1, 2, 3 в позицию it
```

### 4.2.2. Использование итераторов в алгоритмах

Рассмотрим алгоритмы, которые можно применять к векторам.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
remove_if(begin(langs), end(langs),
    [](const string& lang) { // лямбда-функция, по которой удаляем
        return lang[0] == 'C'; // хотим удалить все языки, начинающиеся на C
    });
PrintRange(begin(langs), end(langs));
// Java Python C C#
```

Но у нас не удалились C и C#. Оказывается даже у стандартных алгоритмов C++ итераторы – это настолько общая концепция, что они не позволяют обратиться к исходным контейнерам. Поэтому алгоритм `remove_if` может лишь подсказать вам, помочь вам удалить что-то из контейнера. Он помог. Как он помог? Всё, что должно в контейнере остаться, перебросил в начало этого контейнера. И он вернул новый конец вашего вектора, то, что должно стать его концом. И теперь ваша задача сделать то, что вернул `remove_if`, новым концом вектора. Как это можно сделать? Например, с помощью метода `erase`.

```
langs.erase(it, end(langs)); // удаляем начиная с итератора и до конца
// Java Python
```

Ещё раз: `remove_if` ничего не удаляет. Он лишь помогает нам удалить, потому что общие алгоритмы не могут влиять на контейнер (например, изменять его размер).

```
vector<string> langs = {"Python", "C++", "C++", "Java", "C++"};
// оставляем из подряд идущих повторов только один элемент
auto it = unique(begin(langs), end(langs));
langs.erase(it, end(langs)); // удаляем повторяющиеся, которые выкинуты в конец
PrintRange(begin(langs), end(langs));
// Python C++ Java C++
```

Таким образом мы удалили подряд идущие дубликаты элементов. Получается, что можно оставить в векторе только уникальные элементы, сначала их отсортировав, потом удалив повторы, то есть сначала `sort`, потом `unique`, и даже не нужно создавать их множество для этого, так получается экономнее.

Еще есть алгоритм нахождения минимума в массиве:

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
```

```
auto it = min_element(begin(langs), end(langs)); // кладем в итератор мин. элемент
cout << *it << endl; // min_element может вернуть end(langs), только если langs пуст
// C
```

```
auto it = max_element(begin(langs), end(langs)); // максимальный элемент
cout << *it << endl;
// Python
```

```
auto p =
    minmax_element(begin(langs), end(langs)); // пара - min и max в контейнере
cout << *p.first << ' ' << *p.second << endl;
// C Python
```

Таким образом, мы умеем получать минимальный и максимальный элементы в векторе. Но имеет ли смысл применять такие алгоритмы ко множеству? Ответ – нет. На множестве (`set`) нам достаточно взять `it = begin(langs)` для получения минимального элемента и `it = end(langs); --it`; для получения максимального.

Теперь попробуем вызвать `remove` для элементов множества:

```
set<string> langs = {"Python", "C++", "C", "Java", "C#"};
remove(begin(langs), end(langs), "C");
```

Но оно не скомпилировалось и выдало много ошибок. Вспомним, что делает `remove` – он переставляет элементы в конец для удаления. А множество не даёт переставлять элементы.

Но есть, например, алгоритм, проверяющий какое-то свойство для всего диапазона:

```
set<string> langs = {"Python", "C++", "C", "Java", "C#"};
cout << all_of(begin(langs), end(langs), [](const string& lang) {
    return lang[0] >= 'A' && lang[0] <= 'Z'; // все названия с большой буквы
}) << endl;
// 1
```

Все элементы начинаются с заглавной английской буквы и алгоритм выдал `true` (1). Если изменить первую букву какого-нибудь языка на малую, начнёт выводить 0. Что мы узнали?

1. Методы вектора, позволяющие вставить в конкретное место вектора или удалить: `insert`, `erase`;
2. Алгоритмы:

- `remove_if` – оставляет в начале вектора элементы, не удовлетворяющие условию;
- `unique` – оставляет в начале вектора по одному элементу из группы подряд идущих повторений;
- `min_element`, `max_element`, `minmax_element` – `min`, `max` элементы;
- `all_of` – проверка условия для всех элементов.

### 4.2.3. Обратные итераторы

Бывают не только обычные итераторы. Например, вывод в обратном порядке можно сделать так:

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
PrintRange(rbegin(langs), rend(langs)); // обратные итераторы
// C# Java C C++ Python
```

`rbegin()` и `rend()` от слова `reverse` – перевернутый. Это **обратные итераторы**.

Причём `*rbegin(langs) = "C#"`, а вот `*rend()` не скомпилируется.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = rbegin(langs); // итератор на последний элемент
cout << *it << " "; // выводим
++it;
cout << *it << " "; // сдвинули итератор и вывели предпоследний элемент
// C# Java
```

Если `begin` и `end` имеют тип `vector<string>::iterator`, то `rbegin` и `rend` в свою очередь имеют тип `vector<string>::reverse_iterator`.

Python	C++	C	Java	C#	
	Python	C++	C	Java	C#

`begin` указывает на Python, а `end` за C#;  
а вот `rbegin` – уже на C# и `rend` перед Python.

Передадим обратные итераторы в наши алгоритмы:

```
auto result = find_if(
    rbegin(langs), rend(langs),
    [](const string& lang) { // лямбда-функция по языкам
        return lang[0] == 'C'; // возвращает true, если нулевая буква = 'C'
    })
```

```
});  
// C#
```

Раньше `find_if` возвращал первый подходящий элемент, а теперь последний (первый для диапазона обратных итераторов). Вызовем `sort` от обратных итераторов и посмотрим на результат:

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <map> // подключили словари  
using namespace std;  
template <typename It> // написали шаблонную функцию  
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора  
               It range_end) { // и тут  
    for (auto it = range_begin; it != range_end; ++it) {  
        cout << *it << " ";  
    }  
}  
  
int main() {  
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};  
    sort(rbegin(langs), rend(langs)); // сортируем по убыванию  
    PrintRange(begin(langs), end(langs));  
    return 0;  
}  
// Python Java C++ C# C
```

`reverse` для обратных операторов работает как и для прямых (переворачивает).

В итоге обратные итераторы упрощают итерирование по контейнеру в обратную сторону и могут быть переданы в алгоритмы. А `sort(rbegin(langs), rend(langs));` – простой способ сортировки вектора по убыванию.

#### 4.2.4. Алгоритмы, возвращающие набор элементов

Мы рассматривали алгоритм `remove_if`, который позволяет удалить элемент по какому-то критерию в массиве, в диапазоне элементов в векторе. А что если мы хотим эти элементы не удалить, а аккуратно отложить, например, в конец вектора? Для этого есть алгоритм `partition`. Я вызвал алгоритм `partition` от диапазона, который я хочу разбить на две части, и передаю



критерий, по которому я хочу разбить.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = partition(begin(langs), end(langs), [](const string& lang) {
    return lang[0] == 'C'; // делим по принципу "начинается или не начинается на C"
});
PrintRange(begin(langs), end(langs));
// C# C++ C Java Python
```

Сначала идут все языки, которые начинаются с буквы C в каком-то порядке, как получилось: C#, C++ и C, а потом все остальные языки. То есть все элементы, которые удовлетворяют критерию, для которых лямбда-функция возвращает `true`, перемещаются в начало диапазона. Причём мы можем сохранить результат `partition` (правую границу полуинтервала тех элементов, которые удовлетворяют условию) в итератор `it`.

Если мы хотим переложить какие-то элементы из одного вектора в другой, используем алгоритм `copy_if`:

```
// исправим PrintRange, чтобы выводил через запятую
...
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
vector<string> c_langs(langs.size()); // вектор, куда мы копируем, должен быть
                                     // объявлен и иметь подходящий размер
auto it = copy_if(begin(langs), end(langs), begin(c_langs),
    [](const string& lang) {
        return lang[0] == 'C';
    });
PrintRange(begin(c_langs), end(c_langs));
// C++, C, C#, , ,
```

Заметим, что размер остался равным 5. Итераторы не могут менять размер векторов. Но `copy_if` копирует подходящие под условие элементы и возвращает в `it` итератор на новый конец вектора, в который копировали.

Теперь поговорим о `set`'ах. Что можно делать с множествами в математике? Объединять, пересекать, вычитать. Рассмотрим алгоритмы C++, которые помогают делать это же с множествами здесь.

```
// исправим PrintRange, чтобы выводил через запятую
set<int> a = {1, 8 ,3};
set<int> b = {3, 6 ,8};
```

```
vector<int> v(a.size()); // вектор для хранения результата размера как set a
auto it = set_intersection(begin(a), end(a), begin(b), end(b), begin(v));
// intersection принимает два полуинтервала и итератор, куда сохранять результат
PrintRange(begin(v), end(v));
// 3, 8, 0,
```

В итоге мы имеем пересечение множеств и 0, который дополняет до размера `a`. Сам `set_intersection` возвращает итератор за концом итогового пересечения. Т. е.:

```
PrintRange(begin(v), it);
// 3, 8,
```

Замечание: если мы не укажем явно размер вектора, в который помещаем результат, то код упадёт.

#### 4.2.5. Итераторы `inserter` и `back_inserter`

Рассмотрим более удобный способ.

```
#include <iterator>
...
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
vector<string> c_langs; // о размере уже не беспокоимся
copy_if(begin(langs), end(langs),
        back_inserter(c_langs), // специальный итератор, вставляющий в конец
        [](const string& lang) { // привычная лямбда-функция
            return lang[0] == 'C'; // снова выделяем только начинающиеся на C
        });
PrintRange(begin(c_langs), end(c_langs));
// C++, C, C#,
```

Вектор имеет нужный размер и не содержит лишних элементов. Итераторы умеют делать лишь ограниченный набор действий: `*` (ссылка на конкретный элемент), `++`, `--` и сравнение итераторов. А итератор `back_inserter` (если с ним происходят перечисленные операции) делает `push_back` в контейнер, к которому он относится.

Аналогично поступим с алгоритмом `set_intersection`. У множества есть просто `insert`:

```
set<int> a = {1, 8, 3}
```

```
set<int> b = {3, 6 ,8}
set<int> res;
auto it = set_intersection(begin(a), end(a), begin(b), end(b),
    inserter(res, end(res)) ); // итератор для вставки в множество
PrintRange(begin(res), end(res));
```

`inserter` возвращает специальный итератор, который вставляет элемент в множество.

#### 4.2.6. Отличия итераторов векторов и множеств

Рассмотрим задачу: найти в векторе строк язык, который начинается с буквы 'C', и найти позицию элемента в векторе (а не итератор).

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = find_if(begin(langs), end(langs),
    [](const string& lang) {
        return lang[0] == 'C';
    });
cout << it - begin(langs) << endl; // таким образом получаем номер элемента в векторе
// 1
```

Для получения номера элемента достаточно из полученного с помощью `find_if` вычесть итератор начала диапазона. Получим 1. Действительно, у 'C++' в векторе номер 1. Таким образом, **итераторы вектора можно вычитать друг из друга**. Но такое не сработает для итераторов множества, т. к. для них не определена операция «минус». Аналогично итераторы вектора можно сравнивать не только с помощью `==` и `!=`, но и с помощью `<` и `>`. А для множеств применимо только равно и не равно.

Попробуем для множества чисел вывести все элементы, строго большие его самого.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
++it;
PrintRange(it, end(s));
// 8, 9,
```

Но часто бывает, что `it` нельзя изменить и поэтому нельзя писать `it + 1`. Но этого можно избежать операцией `next(it)`. По сути она прибавляет 1 с помощью `++`.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
PrintRange(next(it), end(s));
// 8, 9,
```

Точно так же работает функция `prev`, только вычитает 1.

#### 4.2.7. Категории итераторов, документация

Научимся читать документацию по языку C++ с помощью сайта [ru.cppreference.com](http://ru.cppreference.com). Сейчас нас интересует документация по алгоритмам. Посмотрим на `unique_copy`.

```
template <class InputIt, class OutputIt>
ForwardIt unique_copy(InputIt first, InputIt last, OutputIt d_first);
```

Видим, что это шаблонная функция, которая принимает `InputIt first`, `InputIt last`, `OutputIt d_first`. Категории итераторов в документации:

- **Input:** итераторы, из которых можно читать (итераторы любых контейнеров). Но не подходят `inserter` и `back_inserter`;
- **Forward, Bidirectional:** обычные итераторы, из которых можно читать (кроме `set` и `map`, если по `forward`-итератору что-то меняется);
- **Random:** итераторы, к которым можно прибавлять число или вычитать друг из друга (итераторы векторов и строк);
- **Output:** итераторы, в которые можно писать (итераторы векторов и строк, `inserter` и `back_inserter`).

Например, `partial_sort` принимает `random`-итераторы, потому что он сортирует (переставляет элементы) и пользуется функцией.

## 4.3. Очередь, дек и алгоритмы поиска

### 4.3.1. Стек, очередь и дек

Рассмотрим новый контейнер: **очередь**. Очередь бывает из людей или запросов. Новые приходят в конец и удаляются из начала (или наоборот). Её можно реализовать с помощью вектора:

```
v.push_back(x);    // добавляем новый
v.erase(begin(v)); // удаляем из начала вектора, что очень долго
```

Элементы вектора хранятся подряд, и поэтому удаление из начала вектора будет работать за длину вектора, потому что он будет переставлять все элементы в начало, чтобы заполнить полученную пустоту: удалили нулевой, переместили первый на нулевое место, второй на первое и т.д.

Для работы с очередью есть специальный контейнер – **deque** (double-ended queue)

- Это двусторонняя очередь;
- `#include <deque>;`
- Быстрые операции:

```
d.push_back(x)    // добавление в конец
d.pop_back(x)     // удаление из конца
d.push_front(x)   // добавление в начало
d.pop_front(x)    // удаление из начала
d[i]              // обращение к элементу по индексу
```

На коде, представленном ниже, продемонстрируем скорость работы **deque**:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    int n = 80000;
    vector<int> v(n);
    while (!v.empty()) {
        v.erase(begin(v));
    }
}
```

```
}  
cout << "Empty!" << endl;  
return 0;  
}
```

Т.е. мы сделали примерно  $80000^2/2$  операций (т. к. каждый раз удаляли и перемещали).

```
#include <iostream>  
#include <deque>  
#include <algorithm>  
using namespace std;  
int main() {  
    int n = 80000;  
    deque<int> v(n);  
    while (!v.empty()) {  
        v.erase(begin(v));  
    }  
    cout << "Empty!" << endl;  
    return 0;  
}
```

Сработало моментально. И даже `pop_front` тоже сработает сразу. Дек умеет больше, но, как следствие, он менее эффективен. Если нужно работать с двумя концами, используйте дек. Но если хватает вектора, используйте вектор.

Разберём ещё одну структуру: **очередь** (`queue`).

- Если нужна только очередь, используйте `queue`;
- Основана на деке, но работает немного быстрее;
- `#include <queue>;`
- Умеет совсем немного:

```
q.push(x), q.pop(x)    // вставляем в начало и удаляем из конца  
q.front(), q.back()   // ссылки на первый и последний элементы очереди  
q.size(), q.empty()   // размер и проверка на пустоту
```

Кроме того, существует **стек** (`stack`).

- Позволяет лишь добавлять в конец и удалять из конца;
- `#include <stack>;`
- Как вектор, но умеет меньше:

```
st.push(x), st.pop(x)  // вставляем в конец и удаляем из конца
st.top()              // ссылка на последний элемент
st.size(), st.empty() // размер и проверка на пустоту
```

### 4.3.2. Алгоритмы поиска

Рассмотрим специальный класс методов контейнеров и алгоритмов – алгоритмы поиска. Мы с ними уже сталкивались при поиске по вектору и множеству;

- Подсчёт количества:

```
count(begin(v), end(v), x);
s.count(x);
```

- Поиск:

```
find(begin(v), end(v), x);
s.find(x);
```

Рассмотрим задачу поиска элементов в контейнерах:

#### 1. Где будем искать?

- Неотсортированный вектор (или строка);
- Отсортированный вектор;
- Множество (или словарь).

#### 2. Что будем искать и проверять?

- Проверить существование;
- Проверить существование и найти первое вхождение;
- Найти первый элемент, больший или равный данному;

- Найти первый элемент, больший данного;
- Подсчитать количество;
- Перебрать все.

Как осуществляется поиск в неотсортированном векторе?

- Поиск конкретного элемента:

```
find(begin(v), end(v), x)
```

- Элемент по какому-то условию (больше, меньше, больше или равен):

```
find_if(begin(v), end(v), [](int y) {...})
```

- Посчитать количество:

```
count(begin(v), end(v), x)
```

- Перебрать все можно с помощью цикла и `find`.

Например, выведем позиции всех пробелов в строке:

```
for (auto it = find(begin(s), end(s), ' ');  
     it != end(s);  
     it = find(next(it), end(s), ' ')) { // переходим в цикле к следующему пробелу  
    cout << it - begin(s) << " "; // next(it) эквивалентен it + 1  
}
```

В отсортированном векторе поиск можно осуществить быстрее с помощью [бинарного поиска](#). Количество операций равно  $\log_2(N)$  – двоичному логарифму числа элементов. Столько же работает поиск во множестве и словаре.

Отсюда следствие: если вы просто хотите быстро искать по набору элементов, но не хотите добавлять новые или удалять какие-то, вам достаточно отсортированного вектора. Это будет оптимальнее, чем если вы используете множество. В отсортированном векторе можно искать так:

- Проверка на существование:

```
binary_search(begin(v), end(v), x)
```

- Первый больший или равный данному:



```
lower_bound(begin(v), end(v), x)
```

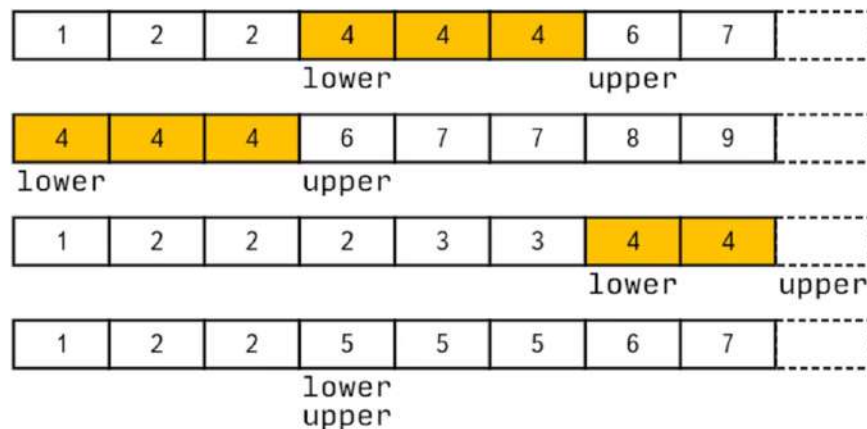
- Первый элемент, больший данного:

```
upper_bound(begin(v), end(v), x)
```

- Диапазон элементов, равных данному (аналог minmax):

```
equal_range(begin(v), end(v), x) ==  
make_pair(lower_bound(...), upper_bound(...))
```

## lower\_bound и upper\_bound



Ещё про `equal_range`.

- Если элемент есть, то `equal_range = [lower_bound, upper_bound)` – диапазон всех вхождений;
- Если же элемента нет, то `lower_bound == upper_bound` – позиция, куда можно вставить элемент без нарушения порядка сортировки;
- Количество вхождений `== upper_bound - lower_bound`;
- А перебрать все элементы, равные данному, можно просто проитерировавшись от `lower_bound` до `upper_bound`.

Поиск во множестве мы уже знаем:

- `s.count(x);`
- `s.find(x);`
- `s.lower_bound(x);`
- `s.upper_bound(x);`
- `s.equal_range(x).`

### 4.3.3. Анализ распространённых ошибок

Первый пример распространённых ошибок – вычитание итераторов множества:

```
int main() {
    set<int> s = {1, 2, 7};
    end(s) - begin(s);
    return 0;
}
// no match for 'operator-'
```

Это ошибка простая, а вот если мы возьмём алгоритм, принимающий **Random** итераторы (например, `partial_sort`) и передадим ему итераторы множества:

```
int main() {
    set<int> s = {1, 2, 7};
    partial_sort(begin(s), end(s), end(s))
    return 0;
}
// no match for 'operator-', 'operator+', 'operator<'
```

Всё по той же причине. Итератор множества – не **Random** итератор, по нему нельзя сравнивать или перемещать элементы.

Теперь попробуем вызвать `remove`:

```
int main() {
    set<int> s = {1, 2, 7};
    remove(begin(s), end(s), 0);
    return 0;
}
// assignment of read-only location ...
```

Т. е. присваивание в итератор, содержимое которого нельзя менять. Ссылка под итераторами константная.

Теперь одна из самых страшных ошибок (не ловится на этапе компиляции) – передача диапазона, у которого итераторы от разных контейнеров:

```
int main() {
    vector<int> s1 = {1, 2, 7};
    vector<int> s2 = {2, 7};
    sort(begin(s1), end(s2)); // диапазон от начала одного вектора до конца другого
    return 0;
}
```

Запускаем код – и программа упала. В обратном порядке она, скорее всего, заиклится. Проверить это можно, закомментировав код, и если он заработает, проблема в нём.

Если же мы передадим итераторы разных типов, то:

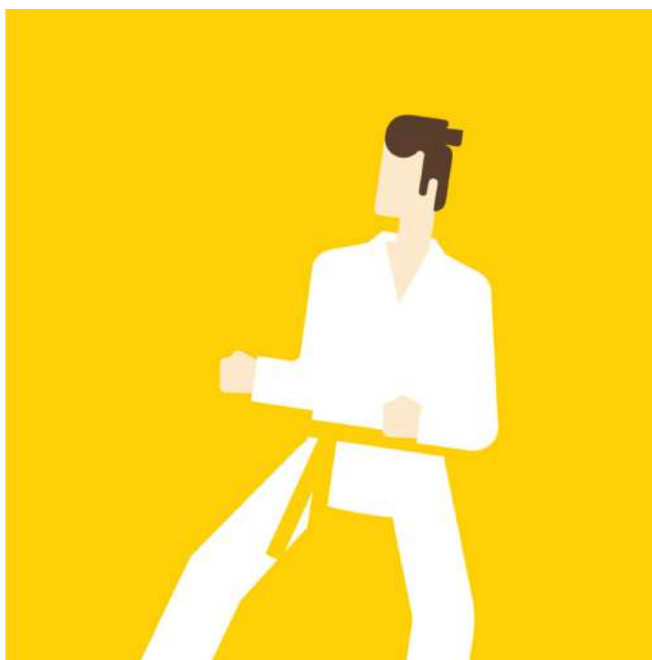
```
int main() {
    vector<int> s1 = {1, 2, 7};
    vector<int> s2 = {2, 7};
    sort(begin(s1), rend(s2));
    return 0;
}
// deduced conflicting types for parameter random access iterator
```

Итераторы должны иметь один тип. А у нас в данном случае тип разный и не получается вызвать функцию `sort`.

# Основы разработки на C++: жёлтый пояс

Неделя 5

Наследование и полиморфизм



# Оглавление

<b>Наследование и полиморфизм</b>	<b>2</b>
5.1 Наследование	2
5.1.1 Введение в наследование	2
5.1.2 Доступ к полям классов. Знакомство со списками инициализации	8
5.1.3 Порядок конструирования экземпляров классов	11
5.2 Полиморфизм	15
5.2.1 Унификация работы с классо-специфичным кодом. Постановка проблемы	15
5.2.2 Решение проблемы с помощью виртуальных методов	18
5.2.3 Свойства виртуальных методов. Абстрактные классы	21
5.2.4 Виртуальные методы и передача объектов по ссылке	23
5.2.5 Хранение объектов разных типов в контейнере с помощью <code>shared_ptr</code>	24
5.2.6 Задача о разборе арифметического выражения. Описание решения	25
5.2.7 Решение задачи частного примера	29
5.2.8 Описание и обзор общего решения задачи	30

# Наследование и полиморфизм

## 5.1. Наследование

### 5.1.1. Введение в наследование

Наследование в языке C++ – это способ выразить связи между классами. Например, сказать, что один класс является подтипом другого класса, реализуя часть его функциональности, и может использоваться в связке вместе с ним. Посмотрим, как всё это работает на практическом примере. Мы хотим написать некий волшебный мир, в котором есть животные (кошки и собаки), которые могут есть фрукты – яблоки и апельсины, – а у фруктов есть некоторое количество здоровья. Значит, мы тогда реализуем все эти действия с помощью наших классов, пусть у нас кошки и собаки будут иметь метод «покушать», и когда они едят, будем выводить в консоль, что произошло.

```
#include <iostream>
using namespace std;
struct Apple { // структура Яблоко
    int health = 10;
};
struct Orange { // структура Апельсин
    int health = 5;
};
class Cat { // создаём класс Кошка
public:
    void Meow() const { // кошка может мяукать
        cout << "meow! ";
    }
    void Eat(const Apple& a) { // кошка может есть яблоко
        cout << "Cat eats apple. " << a.health << "hp." << endl;
    }
};
```

```
int main() {
    Cat c; // создали кошку
    c.Meow(); // помяукали
    Apple a; // создали яблоко
    c.Eat(a); // съели яблоко кошкой
    return 0;
}
// meow!
// Cat eats apple. 10hp.
```

Всё, кошка съела яблоко, в котором было 10 единиц здоровья.

А теперь попробуем скормить кошке апельсин. По идее она должна уметь есть фрукты, и апельсины тоже.

```
...
    Orange o; // создали апельсин
    c.Eat(o); // съели апельсин кошкой
    return 0;
}
// error: no matching function for call to 'Cat::Eat(Orange&)'
```

Появляется ошибка: нет метода «поесть» для апельсина. Определим его, продублировав метод яблока:

```
...// сразу под Meow() {}
void Eat(const Apple& a) { // кошка может есть яблоко
    cout << "Cat eats apple. " << a.health << "hp." << endl;
}
void Eat(const Orange& o) { // кошка может есть апельсин
    cout << "Cat eats orange. " << o.health << "hp." << endl;
} // продублировали всё, заменив apple на orange
// meow!
// Cat eats orange. 5hp.
```

Всё, апельсин тоже съели. Всё бы ничего, но теперь добавим собаку, скопировав класс Cat:

```
class Dog { // создаём класс Собака дублированием класса Cat с исправлениями
public: // удалили мяуканье
    void Eat(const Apple& a) {
        cout << "Dog eats apple. " << a.health << "hp." << endl;
```

```
}
void Eat(const Orange& o) {
    cout << "Dog eats orange. " << o.health << "hp." << endl;
} // продублировали все, заменив apple на orange
};
...
int main() {
    Cat c; // создали кошку
    c.Meow();
    Apple a;
    Orange o;
    c.Eat(o);
    Dog d;
    d.Eat(a); // съели яблоко кошкой
    return 0;
}
// meow!
// Cat eats orange. 5hp.
// Dog eats apple. 10hp.
```

Всё работает. Но мы 4 раза продублировали функцию `Eat()`. Заметим, что объекты `Apple` и `Orange` отличаются только числом `hp`, но с ними одинаково работает функция `Eat()`. Если же нам придётся добавить ещё фрукты, то мы будем вынуждены копировать код для этого фрукта и в кошке, и в собаке. Это ужасно.

Создадим **отношение наследования** для классов яблоко и апельсин. Ведь они являются наследниками **базового класса** фрукт (`Fruit`). Яблоко и апельсин – наследники фрукта, и мы явно зададим публичное наследование:

```
struct Fruit { // структура фрукты
    int health = 0; // поле здоровья, пусть по умолчанию будет 0 hp
    string type = "fruit"; // добавляем тип того, что мы едим
};

struct Apple : public Fruit { // указываем, что яблоко наследуется от фруктов
    Apple() { // нам достаточно написать конструктор для яблока
        health = 10; // яблоко видит поле health у фрукта, поэтому присваиваем 10
        type = "apple"; // явно указываем, что яблоко имеет тип яблока
    } // т. к. наследование публичное, мы видим все поля Fruit в Apple
};
```



```
struct Orange : public Fruit { // без отношения наследования : public Fruit
    Orange() { // наш код не заработает
        health = 5;
        type = "orange";
    }
};
//
```

Таким образом с помощью публичного наследования мы получили доступ к полям, объявленным в нашем базовом классе. Теперь мы можем исправить наш код следующим образом:

```
class Cat { // создаём класс Кошка
public:
    void Meow() const {
        cout << "meow! ";
    }
    void Eat(const Fruit& f) { // кошка ест фрукты
        cout << "Cat eats " << f.type << ". " << f.health << "hp.";
    }
};
...
int main() {
    Cat c;
    c.Meow();
    Orange o;
    c.Eat(o);
    return 0;
}
// meow! Cat eats orange. 5 hp.
```

Теперь для кошки есть всего один метод `Eat()` для всех фруктов. Аналогично делаем для собаки:

```
class Dog {
public:
    void Eat(const Fruit& f) { // собака ест фрукты
        cout << "Dog eats " << f.type << ". " << f.health << "hp. ";
    }
};
...
int main() {
    Dog d;
```

```

Orange o;
d.Eat(o);
return 0;
}
// Dog eats orange. 5hp.

```

Вот, мы уменьшили дублирование для фруктов. Добавим, например, ананас:

```

struct PineApple : public Fruit {
    PineApple() {
        health = 15;
        type = "papple";
    }
};
...
int main() {
    Dog d;
    PineApple p;
    d.Eat(p); // теперь собака съест ананас
    return 0;
}
//Dog eats papple. 15hp.

```

Аналогичным образом уберём дублирование из собаки и кошки. На самом деле они должны наследоваться от базового класса Животные. Каждое животное умеет есть фрукты, но не каждое животное может мяукать (это умеет только кошка).

```

class Animal {
public:
    void Eat(const Fruit& f) { // животное типа type ест фрукты
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    string type = "animal"; // уберём дублирование в методе Eat
};
class Cat : public Animal { // создаём отношение наследования
public: // т. к. наследование публичное, Cat видит переменную type от Animal
    Cat() {
        type = "cat";
    }
    void Meow() const {
        cout << "meow! ";
    }
};

```

```
} // удалили метод Eat() у кошки. Она животное, а животные умеют есть
};
```

С помощью публичного наследования мы получили доступ к методам и полям класса «животное». И поэтому теперь кошка может вызывать метод `Eat`, хотя этот метод не определён внутри «кошки», а определён внутри класса, от которого отнаследован класс «кошка». И теперь точно так же мы уберем дублирование для класса «собака».

```
class Dog : public Animal {
public:
    Dog() {
        type = "dog";
    }
};
```

Всё продолжает работать. Но теперь и собака и кошка могут есть, хотя в их описании это явно не указано. Кроме того, кошка может мяукать, а собака – нет.

Чтобы показать всю силу наследования, давайте создадим функцию «покормить», `DoMeal`. В неё мы будем передавать некое животное, некоторый фрукт, при этом сама функция не будет знать, что это за конкретное животное и что это за конкретный фрукт. Она просто вызовет метод «поесть» для этого фрукта. И, по идее, вывод в консоль измениться не должен.

```
...
void DoMeal(Animal& a, Fruit& f) {
    a.Eat(f);
}
int main() {
    Dog d;
    Cat c;
    Orange o;
    Apple a;
    DoMeal(d, a); // эта функция ничего не знает ни о собаках, ни о кошках
    DoMeal(c, o); // она знает только о классах базового типа
    return 0;
}
// dog eats apple. 10hp.
// cat eats orange. 5hp.
```

Замечу, что сейчас в классе `Animal` у нас строчка типа `type` доступна всем. То есть снаружи эту строчку тоже можно менять, когда вы используете данный класс. И, на самом деле, в какой-

то момент по ошибке вы можете внести какие-то изменения, которые вносить не хотелось бы. Например, после того как мы покормим собаку, добавлять к ней звёздочку.

```
void DoMeal(Animal& a, Fruit& f) {
    a.Eat(f);
    a.type += "*";
}...
DoMeal(d, a);
DoMeal(d, o);
DoMeal(d, o);
// dog eats apple. 10hp.
// dog* eats apple. 10hp.
// dog** eats apple. 10hp.
```

О том, как этого избежать, в следующем видео.

### 5.1.2. Доступ к полям классов. Знакомство со списками инициализации

Зачем нужно наследование?

- Введение логической иерархии между классами ( $\text{Cat}, \text{Dog} \in \text{Animal}$ );
- Решение проблемы дублирования между однотипными структурами;
- Универсальные функции, работающие с базовыми классами-родителями (`DoMeal`).

Отношение публичного наследования объявляется так:

```
class TDerived : public TBase {...}
```

При таком публичном наследовании у класса-наследника появится в том числе доступ к полям и методам базового класса, от которого он унаследован.

Перейдём к проблеме непреднамеренной модификации полей классов в наших программах. Напомню проблему: у нас была функция «покормить», в неё мы добавляли звёздочку к типу, в результате двух кормёжек переменная `type` внутри объекта класса «собаки» менялась. Рассмотрим, как от этого можно защититься.

```
void DoMeal(const Animal& a, Fruit& f) {
    a.Eat(f); // передаём константный тип, но тогда мы не сможем его изменить
    a.type += "*";
} // этот способ не используем из-за его локальности
```

**Способ 1:** можно сделать переменную в базовом классе защищённой (написав ключевое слово `protected`). В данном случае, когда мы пишем ключевое слово `protected`, никто извне не может обратиться к переменной типа `type`, в данном случае, у класса `animal`, у объектов класса `animal` в рантайме. Но при этом к объекту типа `type` имеют доступ наследники класса `animal`, то есть, в данном случае, когда мы будем создавать экземпляр класса «кошки», когда мы будем создавать этот объект в его конструкторе, будет доступ в переменную `type`.

Скомпилируем:

```
class Animal {
public:
    void Eat(const Fruit& f) { // животное типа type ест фрукты
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
protected: // все переменные и методы ниже будут защищены (не видны снаружи)
    string type = "animal"; // уберём дублирование в методе Eat
};
```

И теперь компилятор будет жаловаться, пока мы не уберём изменение поля `type` в `DoMeal`. Заметим, что наследники могут обращаться к полям и менять их, но уже в себе.

**Способ 2:** объявление поля константным внутри базового класса. Это тоже хорошее, рабочее решение. Но далее мы видим, что где-то почему-то убирается константность. То есть читаем ошибку, видим, что в данном случае мы пытаемся записать новое значение в константную переменную. Но мы ведь сейчас находимся в конструкторе. Почему так происходит? Здесь надо сказать одну очень важную вещь: в языке C++ все объекты всегда имеют две стадии: либо объект сейчас создается, вот в данный текущий момент происходит его создание, он только-только появляется в памяти, его поля инициализируются, но он ещё до конца не создан. А вторая стадия – это момент, когда уже объект создан, то есть под него выделена память, там всё проинициализировано, как надо, и мы с ним работаем.

```
class Animal {
public:
    void Eat(const Fruit& f) {
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
}
```

```
    const string type = "animal";
};
class Cat : public Animal {
public:
    Cat() { // на этом этапе переменные внутри класса уже проинициализированы
        type = "cat"; // тут ошибка из-за константности поля
    }
    void Meow() const {
        cout << "meow! ";
    }
};
... // прокомментируем все последующие обращения к type и увидим:
// animal eats apple. 10hp.
```

Видим, что изначально `type` в `Cat` инициализируется значением `animal`, а мы потом пытаемся его переопределить на `Cat`. Создадим конструктор класса животное:

```
class Animal {
public:
    Animal(const string& t = "animal") {
        type = t; // опять пытаемся в константную строку что-то записывать
    }
    void Eat(const Fruit& f) {
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    const string type;
};
// passing 'const string'....
```

Видим, что `type` проинициализирована ещё до фигурных скобок. Воспользуемся синтаксисом списков инициализации:

```
class Animal {
public:
    Animal(const string& t = "animal")
    : type(t) { // хотим проинициализировать type значением t
    }
    void Eat(const Fruit& f) {
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    const string type;
};
```

```
};  
// animal eats apple. 10hp.
```

Мы смогли проинициализировать переменную `type` нужным нам значением `t` с помощью синтаксиса списков инициализации. Теперь нам надо из класса-наследника как-то передать свой собственный тип. Для этого нам надо вызвать конструктор базового класса, в данном случае `animal`, с другим аргументом.

```
class Cat : public Animal {  
public:  
    Cat() : Animal("cat") { // передаём в Animal нужное нам значение  
    } // которое запишется в константный type  
    void Meow() const {  
        cout << "meow! ";  
    }  
}; // далее при любой попытке модификации type, компилятор нам не позволит это сделать  
// cat eats apple. 10hp.
```

Таким образом, мы можем защитить поля классов двумя способами:

1. Объявить поле в секции `protected`. Тогда к нему доступ будут иметь только наследники;
2. Объявить поле константным. Его нужно будет проинициализировать при создании объекта и его нельзя будет менять на протяжении жизни.

Синтаксис списков инициализации выглядит так:

```
MyClass(int v1, int v2) // в конструкторе класса после объявления аргументов  
: Var1(v1) // пишем инициализируемые поля классов и значения, которые запишем  
 , Var2(v2)  
{  
    ...  
}
```

### 5.1.3. Порядок конструирования экземпляров классов

Будем работать с упрощённой версией кода:

```
#include <iostream>

using namespace std;

struct Log { // структура данных логер
    Log(string name) : n(name) {
        cout << "+ " << n << endl; // пишет + и свой id, когда создается
    };
    ~Log() { // деструктор
        cout << "- " << n << endl; // пишет - и свой id, когда удаляется
    }
    string n;
};

struct Fruit {
    string type = "fruit";
    Log l = Log("Fruit"); // заведём логер для фруктов
};

struct Apple : public Fruit {
    Apple() {
        type = "apple";
    }
    Log l = Log("Apple"); // теперь логер для яблока
};

int main() {
    Apple a;
    return 0;
}

// + Fruit
// + Apple
// - Apple
// - Fruit
```

Видим, что логер был создан сначала внутри `Fruit`, затем внутри яблока, затем удалён внутри яблока и только потом внутри `Fruit`. Это может говорить о том, что если у нас есть некоторая иерархия классов, то при создании объекта, принадлежащего данной иерархии, всегда будет сначала вызываться конструктор базового класса, затем класса-наследника и так далее по це-



почке. При этом, когда объекты данных классов будут удаляться, сначала всегда вызывается объект самого последнего класса наследника и так далее по цепочке вверх, пока мы не дойдем до деструктора самого базового класса.

Теперь зарефакторим наш код:

```
struct Fruit {
    Fruit(const string& t)
        : l(t + " (Fruit)") {} // инициализируем логер строкой + именем базового класса
    const Log l; // сделали логер константным
};
// т. к. яблоко унаследовано от Fruit, то перед Apple вызывается конструктор Fruit
struct Apple : public Fruit {
    Apple() : Fruit("apple") {
    }
    Log l = Log("Apple"); // теперь логер для яблока
};
// + apple (Fruit)
// + Apple
// - Apple
// - apple (Fruit)
```

Теперь сделаем яблокам уникальные идентификаторы:

```
struct Apple : public Fruit {
    Apple(const string& t) : Fruit(t), l(t) { // добавили идентификатор и логер
    }
    const Log l;
};

int main() {
    Apple a1 = Apple("a1");
    Apple a2 = Apple("a2");
    return 0;
}
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
```

Значит, сначала создали объект **a1**, потом объект **a2**. Когда для них начали вызываться деструкторы, произошло все ровно в обратном порядке. Значит, когда мы создаём переменные обычные, конструирование и удаление объектов работает точно так же. **Кто создаётся рань-**

ше, тот удаляется позже. Создадим более сложный класс Яблочного дерева, который будет содержать в себе много яблок:

```
class AppleTree {
public:
    AppleTree()
        : a1("a1")
        , a2("a2") {
    }
    Log l = Log("AppleTree");
    Apple a1;
    Apple a2;
}

int main() {
    AppleTree at;
    return 0;
}

// + AppleTree
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
// - AppleTrees
```

Вопрос: от чего зависит порядок инициализации? Даже поменяв строчки местами:

```
: a2("a2")
, a1("a1") { // выскочит пара warning-ов
```

Всё равно порядок создания не изменится. Список инициализации не меняет порядок создания. Он просто указывает значения, которыми мы их инициализируем. Реально же порядок зависит только от того, как они перечислены внутри класса.

Хотя это всего лишь предупреждение, инициализация не в том порядке может нам помешать:

```
class AppleTree {
public:
    AppleTree(const string& t)
        : type(t) // инициализируем переменную type первой
        , a1(type + "a1") // и тут её активно используем
        , a2(type + "a2") {
    }
}
```

```
Apple a1;
Apple a2;
string type; // а здесь эта переменная объявлена последней
};

int main() {
    AppleTree at("AppleTree");
    return 0;
}
// terminate called after throwing an instance of std::bad_alloc
```

Видим ошибки или какой-то бред. Поскольку мы инициализировали `type` последним и использовали его перед этим, мы обратились к неизвестной памяти. Если изменить порядок, то всё будет работать.

## 5.2. Полиморфизм

### 5.2.1. Унификация работы с классо-специфичным кодом. Постановка проблемы

Наследование нам помогало с дедупликацией повторяющегося кода в различных классах. Соответственно, дедупликацию мы исключали за счёт создания базового класса и вынесения общего кода в методы этого класса. Восстановим код из лекции:

```
#include <iostream>
using namespace std;

class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
private:
    const string type_;
};
```

```
class Cat : public Animal {
public:
    Cat() : Animal("cat") {}
    void Meow() const {
        cout << "Meow!" << endl;
    }
};

class Dog : public Animal {
public:
    Dog() : Animal("dog") {}
    void Bark() const {
        cout << "Whaf!" << endl;
    }
};

int main() {
    Cat c;
    Dog d;
    c.Eat("apple");
    d.Eat("orange");
    return 0;
}
// cat eats apple
// dog eats orange
```

Мы научились делать общий код для разных классов. Давайте теперь рассмотрим не общий код, а специфичный для каждого класса. Давайте напомним функции, которые позволяют нам единообразно просить животных издавать разные звуки. Соответственно, самый простой вариант – это написать отдельную функцию для кошки и для собаки:

```
void MakeSound(const Cat& c) {
    c.Meow();
}

void MakeSound(const Dog& d) {
    d.Bark();
}

int main() {
    ...
    MakeSound(c);
    MakeSound(d);
}
```

```
}  
// Meow!  
// Whaf!
```

Снова возникает проблема:

- Если мы имеем  $X$  различных классов: Cat, Dog, Horse, ...
- И у нас  $Y$  методов для каждого класса: голос, действие, ...
- Получим в итоге  $X * Y$  функций!

Что же можно сделать? Первая же идея, которая приходит нам в голову – это воспользоваться методикой из предыдущей лекции. Давайте мы объединим функцию `MakeSound` и унесём её в базовый класс.

```
class Animal {  
public:  
    Animal(const string& type) : type_(type) {}  
    void Eat(const string& fruit) {  
        cout << type_ << " eats " << fruit << endl;  
    }  
    void Voice() const { // приходится делать громоздкую конструкцию,  
        if (type_ == "cat") { // поскольку animal не знает про котов  
            cout << "Meow!" << endl;  
        } else if (type_ == "dog") {  
            cout << "Whaf!" << endl;  
        }  
    }  
private:  
    const string type_;  
}; // удаляем функции Bark и Meow  
...  
void MakeSound(const Animal& a) {  
    a.Voice();  
}  
  
int main() { ...  
    MakeSound(c);  
    MakeSound(d);  
}
```

Вроде исправили, но остались минусы, и основной заключается в том, что различное поведение классов-потомков необходимо переносить в базовый класс. Дополним класс животных, добавив попугая:

```
...
} else if (type_ == "dog") {
    cout << "Whaf!" << endl;
} else if (type == "parrot") { // попугай называет себя (по имени) хорошим
    cout << name_ << " is good!" << endl; // попытаемся сделать это тут
}
...
class Parrot : public Animal {
public:
    Parrot(const string& name) : Animal("parrot"), name_(name) {}
private:
    const string& name_; // имя попугая делаем приватным
};
```

Ловим ошибку: в базовом классе неизвестно приватное поле `name`. Таким образом мы не можем перенести функцию `Voice` из класса `Parrot` в родительский класс `Animal`.

### 5.2.2. Решение проблемы с помощью виртуальных методов

У нас не получилось объединить различные специфичные методы разных классов в рамках одного метода в базовом классе (мы не смогли использовать приватное поле одного из классов).

Вынесем логику определённых классов обратно в эти классы:

```
#include <iostream>
using namespace std;

class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
private:
    const string type_;
```

```
};

class Cat : public Animal {
public:
    Cat() : Animal("cat") {}
    void Voice() const {
        cout << "Meow!" << endl;
    }
};

class Dog : public Animal {
public:
    Dog() : Animal("dog") {}
    void Voice() const {
        cout << "Whaf!" << endl;
    }
};

class Parrot : public Animal {
public:
    Parrot(const string& name) : Animal("parrot"), name_(name) {}
    void Voice() const {
        cout << name_ << " is good!" << endl;
    }
private:
    const string name_;
};

void MakeSound(const Animal& a) {
    a.Voice();
}

int main() {
    Cat c;
    Dog d;
    MakeSound(c);
    MakeSound(d);
    return 0;
}
// error: 'const class Animal' has no member named 'Voice'
```

Для избавления от ошибки просто объявим этот метод в `animal`, но там он ничего не будет делать:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    void Voice() const {}; // вот тут
private:
    const string type_;
};
//
```

Всё корректно скомпилировалось, но никто не издаёт звуков. Нам нужно, чтобы базовый класс узнал о методах в классах-потомках. Это делается добавлением ключевого слова **virtual** к методу в базовом классе.

```
...
virtual void Voice() const {}; // в Animal
// Meow!
// Whaf!
```

То есть мы сказали базовому классу, что в классах-потомках может быть своя реализация метода `Voice` с той же сигнатурой (возвращаемый тип и константность должны совпадать). И теперь, когда мы вызываем метод `Voice()` у базового класса, то будет вызываться этот метод у нужного класса-потомка.

Теперь добавим попугая:

```
int main() {
    Parrot p("Kesha");
    MakeSound(p);
    return 0;
}
// Kesha is good
```

Всё работает и для попугая. Теперь допустим, что мы захотели изменить название `Voice()` в `Animal` на `Sound()`.

```
virtual void Sound() const {}; // в Animal
```



```
// error: 'const class Animal' has no member named 'Voice'
```

Ошибка возникает в функции `MakeSound()`. Поменяем название и здесь:

```
void MakeSound(const Animal& a) {  
    a.Sound();  
}  
//
```

Программа собирается, но ничего не выводит. Это происходит, потому что в потомках остался метод `Voice()`, но базовый класс ничего про него не знает.

Вернём обратно название `Voice()` везде. И в каждом потомке напишем:

```
void Voice() const override {  
    ...  
}  
...
```

Всё снова работает, как мы ожидаем. Если же мы сейчас повторим те же действия по переименованию в `Sound()` в базовом классе и в `MakeSound()`. Теперь нам выдаётся ошибка:

```
// ... marked 'override', but does not override
```

Компилятор подсказывает, что в базовом классе метод `Voice` неизвестен, и в этом случае мы не совершим ошибки по случайному переименованию функций.

Таким образом, мы узнали про два новых ключевых слова:

- **virtual** добавляется к методам в базовом классе. Позволяет вызывать методы производных классов через ссылку на базовый класс;
- **override** добавляется к методам в производных классах (классах-потомках). Требует объявления метода в базовом классе с такой же сигнатурой.

### 5.2.3. Свойства виртуальных методов. Абстрактные классы

Методы, помеченные словом **virtual**, называются **виртуальными**. Рассмотрим их свойства. Добавим к получившемуся после наших изменений коду класс Лошадь, в котором не будем

объявлять метод `Sound()`:

```
class Horse : public Animal {
public:
    Horse() : Animal("horse") {}
};
...
int main() { ...
    Horse h;
    MakeSound(h);
    return 0;
}
//
```

Для лошади ничего не вывелось. Посмотрим, какой метод `Sound()` вызывается для лошади. Изменим в `Animal` метод `Sound`:

```
virtual void Sound() const {
    cout << type_ << " is silent " << endl; // по умолчанию будет говорить так
}
// horse is silent
```

Таким образом, в данном отношении виртуальные методы не отличаются от обычных: если метод не объявлен в классе-потомке, то он будет вызван из базового класса.

Рассмотрим другое полезное свойство виртуальных методов: допустим, мы хотим, чтобы у каждого потомка был по-своему реализован метод `Sound()`. Пока под это условие не подходит `Horse`. Будем считать, что мы просто забыли его реализовать и хотим, чтобы компилятор подсказывал нам, что нам надо реализовать. Потребуем обязательной реализации в классах-потомках:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    virtual void Sound() const = 0; // это не присваивание, а формальный синтаксис
private:
    const string type_;
};
// cannot declare variable 'h' to be of abstract type 'Horse'
```

Мы сказали виртуальному методу `Sound()`, что он является абстрактным (чисто виртуальным), т.е. требует обязательной реализации в каждом классе-потомке. Класс `Horse` считается абстрактным, потому что в нем нет реализации абстрактного метода `Sound()`, и мы не можем создавать объекты абстрактных классов.

#### 5.2.4. Виртуальные методы и передача объектов по ссылке

Заметим, что мы передавали объекты классов-потомков по ссылке на базовый класс.

Попробуем передать по значению:

```
void MakeSound(const Animal a) { // теперь без &
    a.Voice();
}
```

Программа не собралась, потому что компилятор считает, что мы хотим преобразовать этот объект в тип базового класса. Но мы не можем создавать объекты базового класса, потому что он является абстрактным (из-за виртуального метода `Voice()` в базовом классе). Уберём абстрактность метода `Voice()`:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    virtual void Voice() const {
        cout << type_ << " is silent" << endl;
    }
    ...
// cat is silent
```

Видим, что при передаче по значению вызываются методы базового класса, а не класса-потомка. При передаче по значению мы теряем всю информацию о классе-потомке, и у нас сохраняется информация только о базовом классе, в котором `Voice()` печатает `is silent`. То есть при передаче по значению мы теряем преимущества виртуальных методов.

### 5.2.5. Хранение объектов разных типов в контейнере с помощью `shared_ptr`

Вернёмся к функции `main()`:

```
int main() {
    Cat c;
    Dog d;
    Parrot p("Kesha");
    Horse h;

    MakeSound(c);
    MakeSound(d);
    MakeSound(p); // 4 очень похожих вызова
    MakeSound(h);
    return 0;
}
```

Хочется сделать вызовы универсальными с помощью контейнеров. Мы могли бы сложить всех животных в контейнер и, например, в цикле `for` пробегаться по ним. Все объекты мы можем объединить ссылкой на базовый класс:

```
#include <memory> // заголовок с shared_ptr
...
int main() {
    // vector<Animals> animals; вектор из ссылок создать не получится
    // но есть другой тип из базовых библиотек, с помощью которого можно это сделать:
    shared_ptr<Animal> a;

    a = make_shared<Cat>(); // в угловых скобках указываем реальный тип
    a->Sound(); // похожим на итераторы образом вызываем метод Sound()
    return 0;
}
// Meow!
```

Видим, что вызвался метод `Sound` для `Cat`.

Самое полезное свойство `shared_ptr`, которое мы использовали сейчас: в качестве типа объекта, в который оборачивается `shared_ptr` можем указать просто `Animal`, и при этом `shared_ptr` будет вести себя как ссылка (мы сложим в него объекты производных классов и сможем обращаться к интерфейсу базового класса). Второе свойство: у `shared_ptr` интерфейс, похожий на итераторы (с ними просто работать).

```
int main() {
    shared_ptr<Animal> a;
    a = make_shared<Cat>();
    vector<shared_ptr<Animal>> animals; // векторы животных из shared_ptr
    for (auto a : animals) { // передаем по значению, а можно и const auto& a
        MakeSound(*a); // получаем объект, который обернут в shared_ptr
    }
    return 0;
}
// Meow!
```

Всё работает, теперь сделаем то же для вектора животных.

```
int main() {
    shared_ptr<Animal> a;
    a = make_shared<Cat>();
    vector<shared_ptr<Animal>> animals = {
        make_shared<Cat>(),
        make_shared<Dog>(),
        make_shared<Parrot>("Keshha"), // имя передаётся в конструктор попугая
    }
    for (auto a : animals) {
        MakeSound(*a);
    }
    return 0;
}
// Meow! Whaf! Keshha is good!
```

С помощью `shared_ptr` мы можем помещать различные типы объектов производных классов в контейнеры и обходить их с помощью циклов, как и обычные типы.

### 5.2.6. Задача о разборе арифметического выражения. Описание решения

Поставим задачу написать парсер арифметических выражений:

- Входные данные: Выражение, состоящее из операций `+`, `-`, `*`, цифр и переменной `x`. Её значение нам дано;

- Выходные данные: значение выражения при введённом значении переменной.

Пример работы:

```
// Enter expression:
// 1+2-3*x+x
// Enter x:
// 5
// Expression value -7. Enter x:
// 1
// Expression value 1...
```

Теперь попробуем написать наш разбиратель:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>

using namespace std;

int main() {
    string tokens;
    cout << "Enter expression: ";
    cin >> tokens; // запрашиваем у пользователя выражение
    int x = 0;
    auto expr = Parse(tokens, x); // функция разбора выражения для всех x
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << "Expression value: " << expr->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}
```

Теперь реализуем функции Parse и Evaluate. Мы пока не знаем ничего о типе переменной expr.

```
struct Node { // тип выводимых значений
public:
    int Evaluate() {
        return 0;
    }
};
```

```

    }
};
Node Parse(const string& tokens, int& x) // должна возвращать объект типа Node
    // у которого есть метод Evaluate, вычисляющий выражение

```

Чтобы понять, как разбирать арифметическое выражение, сделаем это вручную на примере:

```

int main() {
    string tokens;
    string tokens = "5+7-x*x+x"; //  $-x^2 + x + 12$ 
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << "Expression value: " << expr->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}

```

Для простоты разбора будем использовать только числа из одной цифры, а также не будет скобок, пробелов и ненужных символов. И будем считать, что выражение всегда корректно. Разделим выражение на типы ввода:

```

struct Node { // тип выводимых значений -- это базовый класс для переменных и цифр
public:
    virtual int Evaluate() {
        return 0;
    } // сделали метод виртуальным
}; // т. к. для каждого класса-потомка надо описывать, что делать в Evaluate

class Digit : public Node {
public: // класс цифра, который будет всегда возвращать значение самой цифры
    Digit(int d) : d_(d) {
    }
    int Evaluate() override {
        return d_;
    }
private:
    const int d_;
}

class Variable : public Node { // переменная x

```

```

public:
    Variable(const int& x) : x_(x) {
    } // сохраняем константную ссылку
    int Evaluate() override { // при вычислении x возвращаем его значение
        return x_;
    }
private:
    const int& x_;
} // остались операции. Заметим, что у них разный приоритет. Умножение раньше всего

class Operation { // для сохранения операции надо сохранять левый и правый операнды
public: // передаём символ операции, левое и правое слагаемое
    Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
        op_(op), left_(left), right_(right) {
    }
    int Evaluate() {
        // вычисление операций оставим на потом
    }
private:
    const char op_;
    shared_ptr<Node> left_, right_;
};

```

Заметим, что в выражении  $5 + 7 * 3$  для операции  $+$  левое слагаемое – это число 5, а правое – это результат умножения 7 и 3. При вычислении значения операции нужно сначала вычислить оба слагаемых, а потом произвести саму операцию и вернуть значение.

```

class Operation : public Node { // унаследуем от Node
public:
    Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
        op_(op), left_(left), right_(right) {
    }
    int Evaluate() override { // операция унаследована от Node
        if (_op == '*') {
            return _left->Evaluate() * _right->Evaluate();
        } else if (_op == '+') {
            return _left->Evaluate() + _right->Evaluate();
        } else if (_op == '-') {
            return _left->Evaluate() - _right->Evaluate();
        }
        return 0; // если операция не распознана, но для корректной строки не дойдём
    }
};

```



```
}
...
```

### 5.2.7. Решение задачи частного примера

Вспомним, как у нас вообще вычисляются арифметические выражения. Во-первых, они вычисляются слева направо для операций с равным приоритетом. Операции с большим приоритетом вычисляются в первую очередь. То есть для нашего выражения сначала мы должны вычислить операцию  $x * x$ . Затем у нас остаются три равноправные операции: это сложение, вычитание и сложение. (в выражении  $5 + 7 - x * x + x$ ). И мы должны их вычислить слева направо. Опишем множители:

```
int main() {
    string tokens = "5+7-x*x+x";
    shared_ptr<Node> var1 = make_shared<Variable>(x); // делаем операцию умножения
    shared_ptr<Node> var2 = make_shared<Variable>(x);
    shared_ptr<Node> mul1 = make_shared<Operation>('*', var1, var2);
    shared_ptr<Node> dig1 = make_shared<Digit>(5); // делаем операцию сложения
    shared_ptr<Node> plus1 = make_shared<Operation>('+', dig1, dig2);
    shared_ptr<Node> dig2 = make_shared<Digit>(7);
    shared_ptr<Node> var3 = make_shared<Variable>(x);
    shared_ptr<Node> minus1 = make_shared<Operation>('-', plus1, mul1);
    shared_ptr<Node> plus2 = make_shared<Operation>('+', minus1, var3);
    // итоговый результат будет в самом правом и самом неприоритетном операторе, plus2
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << plus2->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}
```

При значении  $x = 1$  выведет 12. Это правда. При  $x = 5$  получим: -8 и т. д.

### 5.2.8. Описание и обзор общего решения задачи

Решим исходную задачу: выражение мы должны, на самом деле, вводить также с консоли — оно должно быть произвольное — с точностью до тех условий, которые мы указали ранее. И мы должны его уметь преобразовывать в нужный нам формат не вручную, а с помощью алгоритма, который бы эффективно по нему проходил и сам составлял необходимый набор объектов, переменных, операций.

Возьмем заранее заготовленный код (находится в материалах к видеолекции) и разберём его:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>
using namespace std;
// структура классов у нас остаётся той же
struct Node { // базовый класс
    virtual int Evaluate() const = 0;
};

struct Value : public Node { // класс для значения. Ранее был Digit
    Value(char digit) : _value(digit - '0') {
    }
    int Evaluate() const override {
        return _value;
    }
private:
    const uint8_t _value;
};

struct Variable : public Node { // класс для переменной x
    Variable(const int &x) : _x(x) {
    }
    int Evaluate() const override {
        return _x;
    }
private:
    const int &_x;
};

// алгоритм называется shunting-yard, алгоритм сортировочных станций.
struct Op : public Node { // класс для операций
```

```

Op(char value)
: precedence([value] {
    if (value == '*') {
        return 2;
    } else {
        return 1;
    }
}()),
_op(value) {
}

const uint8_t precedence;
int Evaluate() const override {
    if (_op == '*') {
        return _left->Evaluate() * _right->Evaluate();
    } else if (_op == '+') {
        return _left->Evaluate() + _right->Evaluate();
    } else if (_op == '-') {
        return _left->Evaluate() - _right->Evaluate();
    }
    return 0;
}

// передаём левый и правый операнды отдельными методами. удобнее для произвольного
void SetLeft(shared_ptr<Node> node) {
    _left = node;
}

void SetRight(shared_ptr<Node> node) {
    _right = node;
}

private:
    const char _op;
    shared_ptr<const Node> _left, _right;
};

template <class Iterator>
shared_ptr<Node> Parse(Iterator token, Iterator end, const int &x) {
    // функцию Parse сделали шаблонной, т. е. для любых контейнеров из символов
    if (token == end) {
        return make_shared<Value>('0');
    }
    stack<shared_ptr<Node>> values;
    stack<shared_ptr<Op>> ops;

```

```

auto PopOps = [&](int precedence) { // реализация алгоритма сортировочных станций
    while (!ops.empty() && ops.top()->precedence >= precedence) {
        auto value1 = values.top();
        values.pop();
        auto value2 = values.top();
        values.pop();
        auto op = ops.top();
        ops.pop();
        op->SetRight(value1);
        op->SetLeft(value2);
        values.push(op);
    }
};

while (token != end) {
    const auto &value = *token;
    if (value >= '0' && value <= '9') {
        values.push(make_shared<Value>(value));
    } else if (value == 'x') {
        values.push(make_shared<Variable>(x));
    } else if (value == '*') {
        PopOps(2);
        ops.push(make_shared<Op>(value));
    } else if (value == '+' || value == '-') {
        PopOps(1);
        ops.push(make_shared<Op>(value));
    }
    ++token;
}

while (!ops.empty()) {
    PopOps(0);
}

return values.top();
}

int main() {
    string tokens;
    cout << "Enter expression: ";
    getline(cin, tokens); // считываем выражение
    int x = 0;
    auto node = Parse(tokens.begin(), tokens.end(), x); // парсим выражение
    // по итераторам начала и конца строки
}

```

```
cout << "Enter x: ";
while (cin >> x) {
    cout << "Expression value: " << node->Evaluate() << endl;
    cout << "Enter x: ";
}
return 0;
}
```

Введём различные переменные  $x$  для выражения  $5 + 7 - x * x + x$  и получим: для  $x = 0$  ответ 12, для 1 – 12, для 2 – 10. Всё верно. Введём другое выражение и убедимся в корректности. Кроме того,  $x$  может быть отрицательным.

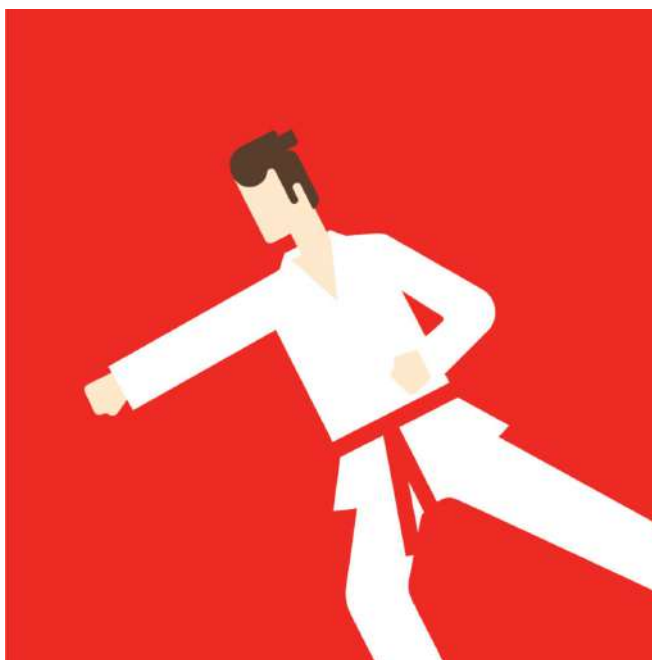
Таким образом, с помощью полученных нами знаний о типе `shared_ptr` и виртуальных методах мы смогли реализовать довольно непростой алгоритм и решить относительно сложную задачу по разбору выражений и вычислению их с использованием внешней переменной.



# Основы разработки на C++: красный пояс

Неделя 1

Макросы и шаблоны классов



# Оглавление

<b>Макросы и шаблоны классов</b>	<b>2</b>
1.1 Введение в макросы . . . . .	2
1.1.1 Введение в макросы . . . . .	2
1.1.2 Оператор # . . . . .	6
1.1.3 Макросы <code>__FILE__</code> и <code>__LINE__</code> . . . . .	7
1.1.4 Тёмная сторона макросов . . . . .	9
1.2 Шаблоны классов . . . . .	12
1.2.1 Введение в шаблоны классов . . . . .	12
1.2.2 Интеграция пользовательского класса в цикл <code>for</code> . . . . .	14
1.2.3 Разница между шаблоном и классом . . . . .	15
1.2.4 Вывод типов в шаблонах классов . . . . .	16
1.2.5 Автоматический вывод типа, возвращаемого функцией . . . . .	19

# Макросы и шаблоны классов

## 1.1. Введение в макросы

### 1.1.1. Введение в макросы

Первая тема нашего курса – введение в макросы. В курсе «Жёлтый пояс по C++» мы разработали unit test framework для создания юнит-тестов. Кроме того у нас была задача, которая называлась «Тестирование класса Rational», в которой нужно было написать набор юнит-тестов для класса Rational. Этот класс представлял собой рациональное число.

```
class Rational {
public:
    Rational() = default;
    Rational(int nn, int dd);

    int Numerator() const;
    int Denominator() const;

private:
    int n = 0;
    int d = 1;
};
```

Нам нужно было разработать набор юнит-тестов, которые проверяли, что этот класс реализован корректно. Рассмотрим программу, которая тестирует класс Rational с помощью unit test framework'a. В ней есть два теста: TestDefaultConstructor(), который проверяет, как работает конструктор по умолчанию в классе Rational, и TestConstruction(), в котором есть один тест, который проверяет, как ведет себя класс Rational, когда ему в конструктор передается числитель и знаменатель.

```
void TestDefaultConstructor() {
    const Rational defaultConstructed;
```



```

    AssertEqual(defaultConstrcuted.Numerator(), 0, "Default constructor
        denominator");
    AssertEqual(defaultConstrcuted.Denominator(), 1, "Default constructor
        denominator");
}

void TestConstruction() {
    const Rational r(3, 12);
    AssertEqual(r.Numerator(), 1, "3/12 numerator");
    AssertEqual(r.Denominator(), 4, "3/12 denominator");
}

```

В функцию `main` передаётся объект класса `TestRunner`, запускается два теста с помощью метода `RunTest`, куда передаём тест и текстовое сообщение.

```

int main() {
    TestRunner tr;
    tr.RunTest(TestDefaultConstructor, "TestDefaultConstructor");
    tr.RunTest(TestConstruction, "TestConstruction");
    return 0;
}

```

Строковые сообщения мы добавляли в `AssertEqual` для того, чтобы, когда наш `assert` срабатывает, понять, какой именно `assert` работает. Намеренно допустим в классе `Rational` ошибку (например, вместо знаменателя будем возвращать числитель). С помощью сообщения об ошибке мы можем найти в нашем коде `assert`, который сработал. Нам помогло то, что мы сделали эти сообщения уникальными.

```

int Rational::Denominator() const {
    return n;
}
// TestDefaultConstructor fail: Assertion failed: 0 != 1
// TestConstruction fail: Assertion failed: 1 != 4 hint: 2 unit tests failed. Terminate

```

Посмотрим, как устроен вызов метода `RunTest` в классе `TestRunner`. Он принимает функцию, которая выполняет тестирование, и строчку, которая совпадает с названием функции. Эта строчка нужна, чтобы формировать сообщения “`TestDefaultConstructor fail`, `TestConstruction fail`”, то есть чтобы в консоль выводить имя теста, который либо прошёл, либо не прошёл. Это неудобно, потому что это дублирование кода.

Что хотим получить:

- Если срабатывает `AssertEqual(x, y)`, на экран выводится `Assertion failed: 2 != 3 hint: x != y. main.cpp:35;`
- Запускать тесты кодом `tr.RunTest(TestConstruction);`
- Если тест проходит, на экран выводится `TestConstruction OK.`

Для того, чтобы решить эту задачу, вспомним, что сборка проекта на C++ состоит из трёх стадий: препроцессинг, компиляция, компоновка. На стадии препроцессинга выполняются директивы `#include`. Содержимое подключаемых файлов копируется в тело компилируемого файла, после этого наступает стадия компиляции. Также на стадии препроцессинга происходит разворачивание макросов. Они объявляются с помощью ключевого слова `#define`.

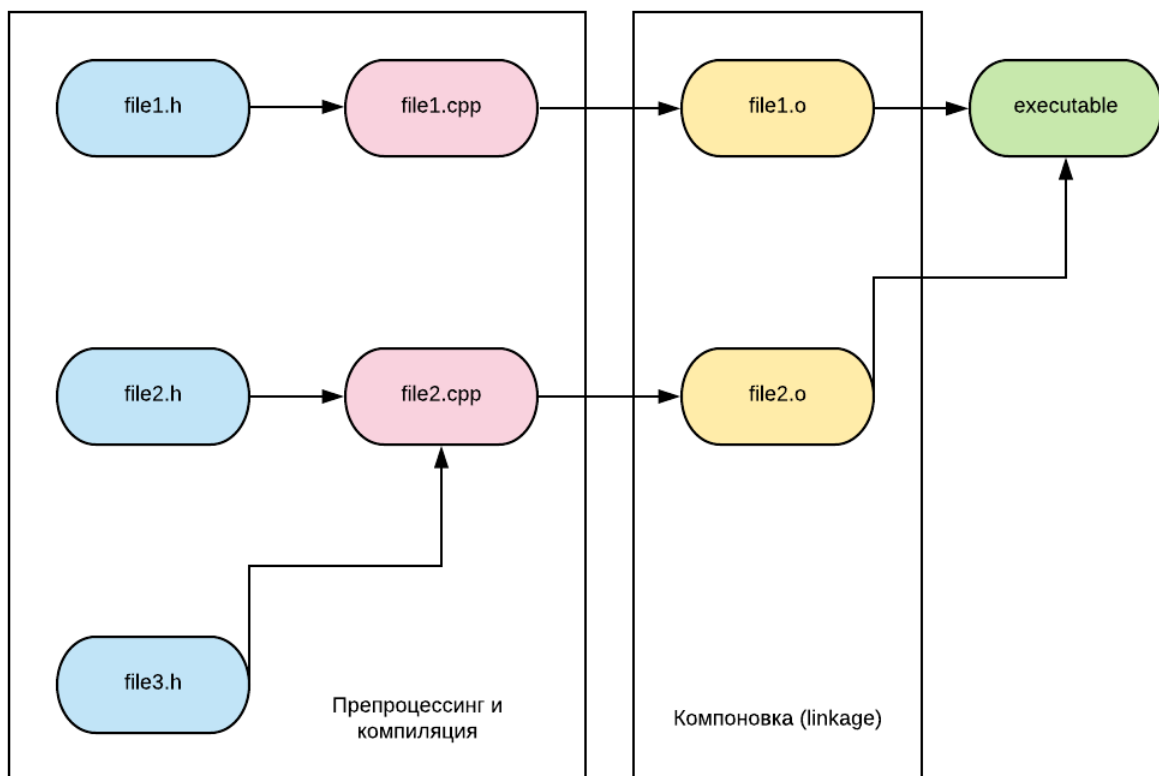


Рис. 1.1: Этапы сборки в C++

Для примера можно объявить макросы:

```
#define MY_MAIN int main()
#define FINISH return 0
```

Исправим функцию

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

на

```
MY_MAIN {
    FINISH;
}
```

Объявленные макросы на этапе препроцессинга будут развёрнуты в свои определения.

Объявим макрос с тремя параметрами `ASSERT_EQUAL`, который разворачивается в вызов шаблона `AssertEqual` с этими тремя параметрами:

```
#define ASSERT_EQUAL(x, y, m) \
    AssertEqual(x, y, m)
```

Можно заменить вызов шаблона на макрос, такая программа скомпилируется.

```
void TestDefaultConstructor() {
    const Rational defaultConstructed;
    ASSERT_EQUAL(defaultConstructed.Numerator(), 0, "Default constructor
        denominator");
    ASSERT_EQUAL(defaultConstructed.Denominator(), 1, "Default constructor
        denominator");
}
```

Мы с вами познакомились с макросами, узнали, что они создаются с помощью ключевого слова `#define`, и на этапе препроцессинга происходит текстовая замена имени макроса на его содержимое. При этом макросы могут не содержать параметров или содержать один или несколько параметров.

### 1.1.2. Оператор #

Нам хотелось бы избавиться от дублирования функции и её названия в коде (см. стр. 3).

Напишем макрос `RUN_TEST`, который принимает на вход параметр `tr` – объект класса `TestRunner`, `func` – функция, содержащая тесты. Этот тест будет у объекта `tr` вызывать метод `RunTest` и передавать туда `func` и ещё раз `func`:

```
#define RUN_TEST(tr, func) \  
    tr.RunTest(func, func)
```

Применим макрос в функции `main`.

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

Такой код не скомпилируется, потому что наш макрос получает в качестве второго параметра функцию, хотя этот параметр должен быть строкой.

```
#define RUN_TEST(tr, func) \  
    tr.RunTest(func, #func)
```

Поставим перед параметром `func` символ решётки. Такая программа работает. Теперь в качестве второго параметра передается имя функции, обернутое в кавычки, то есть строковый литерал. Таким образом, с помощью оператора `#` мы добились того, что когда мы вызываем юнит-тесты, мы можем не дублировать имя функции, а просто передавать эту функцию в макрос и препроцессор за нас в качестве второго параметра передаст имя этой функции.

Другой пример, когда оператор `#` в макросах бывает полезен. Иногда нам нужно что-то логировать, например, для отладки. Допустим, для отладки мы хотим выводить следующие значения в какой-нибудь поток:

```
int x = 4;  
string t = "hello";  
bool isTrue = false;
```

Выведем их:

```
cerr << x << " " << t << " " << isTrue << endl;
// 4 hello 0
```

Можно использовать макросы и оператор #, чтобы сделать вывод более понятным:

```
#define AS_KV(x) #x << " = " << x
```

Перепишем код:

```
cerr << boolsalph;
cerr << AS_KV(x) << endl
    << AS_KV(t) << endl
    << AS_KV(isTrue) << endl;
// x = 4
// t = hello
// isTrue = false
```

С помощью макроса и оператора # мы сделали более читаемый отладочный вывод.

Итоги:

- Оператор # вставляет в код строковое представление параметра макроса;
- Макрос RUN\_TEST упрощает запуск тестов и избавляет от дублирования.

### 1.1.3. Макросы \_\_FILE\_\_ и \_\_LINE\_\_

Упростим использование шаблонов Assert и AssertEqual.

Для того, чтобы понять, какой assert сработал, нам достаточно знать название файла и номер строки в этом файле. Эту информацию мы можем получить автоматически. Для этого есть специальные макросы.

Используем макросы \_\_FILE\_\_ и \_\_LINE\_\_.

```
const string file = __FILE__;
const int line = __LINE__;
```

На стадии препроцессинга `__FILE__` раскроется в название файла (в нашем случае это будет `macro_intro.cpp`). `__LINE__` раскроется в номер строки, в котором был объявлен (в нашем случае 14, поскольку макрос был объявлен на 14 строке).

Воспользуемся этими макросами, чтобы сформировать уникальное сообщение для `assert`'а. Чтобы создавать многострочные макросы, нужно каждую строчку кроме последней (с закрывающейся скобкой) завершать нисходящим слэшем.

```
#define ASSERT_EQUAL(x, y) { \
    ostringstream os; \
    os << __FILE__ << ":" << __LINE__ << " "; \
    AssertEqual(x, y, os.str()); \
}
```

Теперь макрос стал от двух параметров, потому что сообщение генерируется автоматически. Воспользуемся им:

```
void TestDefaultConstructor() {
    const Rational defaultConstructed;
    ASSERT_EQUAL(defaultConstructed.Numerator(), 0);
    ASSERT_EQUAL(defaultConstructed.Denominator(), 1);
}
// TestDefaultConstructor fail: Assertion failed: 0 != 1 hint: ..\src\macro_intro.cpp:19
```

В поле `hint` фреймворк выводит имя файла и строку, в котором вызван `assert`. Усовершенствуем выводимое сообщение:

```
os << #x << " != " << #y << ", "
    << __FILE__ << ":" << __LINE__;
```

В поле `hint` получаем сообщение:

```
// hint: defaultConstructed.Denominator() != 1, ..\src\macro_intro.cpp:20
```

Осталось перенести макрос в файл `test_runner.h`. Добавим туда также макросы `ASSERT` и `RUN_TEST`.

```
#define ASSERT(x) { \
    ostringstream os; \
    os << #x << " is false, " \
    << __FILE__ << ":" << __LINE__ << " "; \
    AssertEqual(x, x, os.str()); \
}
```

### 1.1.4. Тёмная сторона макросов

Макросы позволили сделать наш код короче и проще в использовании. Но вы могли слышать рекомендации, что макросы в C++ – это зло и что никогда нельзя использовать их в своих программах. Да, действительно, при чрезмерном их использовании могут возникать проблемы. Рассмотрим пример:

```
#define MAX (a, b) a > b ? a : b // находит максимум из двух своих аргументов

int main() {
    int x = 4;
    int y = 2;
    int z = MAX(x, y) + 5;
    cout << z;
}
// 4
```

Мы ожидаем, что на экран будет выведено 9, однако получаем 4. Посмотрим, во что раскрывается наш макрос.

```
int z = x > y ? x : y + 5;
```

Если  $x > y$ , то в переменную  $z$  записывается значение  $x$ , если это не так, то в  $z$  запишется  $y + 5$ . Чтобы макрос работал правильно, можно обернуть его в скобки.

```
#define MAX(a, b) (a > b ? a : b)
```

Однако в данном случае гораздо лучше использовать функцию `max` из библиотеки алгоритмов.

Рассмотрим более реальный пример. В стандартной библиотеке нет функции, которая возводит свой аргумент в квадрат. Реализуем макрос, учтём прошлые ошибки и сразу обернём его в скобки.

```
#define SQR(x) (x * x)
```

Реализуем следующий код:

```
int main() {
    int x = 3;
    int z = SQR(x + 1);
    cout << z;
}
```

```
// 7
```

В консоли мы ожидаем увидеть 16, но видим 7. Посмотрим вывод препроцессора, чтобы узнать, в какое выражение раскрылся макрос.

```
int z = (x + 1 * x + 1);
```

Ошибку можно быстро исправить, если обернуть `x` в скобки.

```
#define SQR(x) ((x) * (x))
```

Вместо этого макроса лучше написать шаблон.

```
template <typename T>
T Sqr(T x) {
    return x * x;
}
```

Такой шаблон прекрасно справляется с задачей возведения в квадрат, при этом мы можем не бояться забыть обернуть макрос и аргументы в скобки.

Напишем функцию `LogAndReturn` и передадим её в качестве параметра в макрос `SQR`. Мы ожидаем, что в консоли выведется `x = 3`, а потом выведется 9.

```
int LogAndReturn(int x) {
    cout << "x = " << x << endl;
    return x;
}

int main() {
    int z = SQR(LogAndReturn(3));
    cout << z;
}

// x = 3
// x = 3
// 9
```

Мы не ожидали, что функция `LogAndReturn` выполнится дважды. Посмотрим результаты препроцессорирования.

```
int main() {
    int z = ((LogAndReturn(3)) * (LogAndReturn(3)));
}
```



```
cout << z;  
}
```

Макрос выполнил прямую текстовую замену и вызов функции `LogAndReturn` добавился в код дважды. Это учебный пример. Если бы функция в реальном коде выполняла сложные, долгие вычисления, то мы на ровном месте могли бы получить просадку производительности из-за неудачного использования макроса.

Сохраним результат вызова функции в переменную `x` и передадим её в макрос:

```
int main() {  
    x = LogAndReturn(3);  
    int z = SQR(x);  
    cout << z;  
}  
// x = 3  
// 9
```

Всё будет работать нормально. Допустим, далее нам понадобится переменная `x`, увеличенная на 1. Ради экономии места увеличим её прямо на месте.

```
int main() {  
    x = LogAndReturn(3);  
    int z = SQR(x++);  
    cout << z;;  
}  
// x = 3  
// 12
```

Вместо ожидаемой 9 получили 12. В результатах препроцессирования видим:

```
int z = ((x++) * (x++));
```

Когда одну и ту же переменную мы изменяем несколько раз в одном и том же выражении, то результат не определён.

Если вместо макроса можно написать функцию или шаблон, то именно так и нужно сделать. Так вы защититесь от неожиданных ошибок. Следовательно, если макрос не использует `__FILE__`, `__LINE__` или оператор `#`, подумайте, можно ли обойтись без него. Если вы всё же пишете макрос, то старайтесь использовать каждый аргумент только один раз, максимально изолируйте аргументы с помощью скобок.

## 1.2. Шаблоны классов

В курсе «Жёлтый пояс по C++» мы изучили шаблоны функций. Они позволяют избежать дублирования кода в функциях, который отличаются типами своих аргументов или типом возвращаемого значения. В этом модуле мы изучим шаблоны классов. Они решают ту же самую задачу: позволяют избежать дублирования кода в классах, которые отличаются только типами своих полей, или типами параметров своих методов, или типами возвращаемых значений в методах.

### 1.2.1. Введение в шаблоны классов

Простейший пример шаблона класса – это пара. Забудем на время, что существует стандартная пара. Напишем соответствующую структуру.

```
struct PairOfStringAndInt {  
    string first;  
    int second;  
};
```

Воспользуемся этой парой.

```
int main() {  
    PairOfStringAndInt si;  
    si.first = "Hello";  
    si.second = 5;  
}
```

Потом у нас как-то проект развивается, мы пишем код дальше и понимаем, что нам нужна ещё пара, из логического значения и символа.

```
struct PairOfBoolAndChar {  
    bool first;  
    char second;  
};
```

Добавим в main:

```
int main() {  
    PairOfStringAndInt si;
```

```
si.first = "Hello";
si.second = 5;

PairOfBoolAndChar bc;
bc.first = true;
bc.second = 'z';
}
```

Понятно, что каждый раз, когда у нас возникает необходимость в новых сочетаниях типов, нам приходится объявлять новую структуру. Мы видим, что классы `PairOfStringAndInt` и `PairOfBoolAndChar` структурно одинаковы, но отличаются типами своих полей. Вместо них мы можем написать шаблон класса.

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};
```

Мы написали простейший шаблон класса «пара». Воспользуемся им.

```
int main() {
    Pair<string, int> si;
    si.first = "Hello";
    si.second = 5;

    Pair<bool, char> bc;
    bc.first = true;
    bc.second = 'z';
}
```

Важно отметить, что `Pair` – это шаблон класса, а `Pair<bool, char>` – это уже класс, самостоятельный тип. Таким образом, в этой программе мы создаем из шаблона класса два класса. Создание типа из шаблона класса называется **инстанцированием**.

### 1.2.2. Интеграция пользовательского класса в цикл `for`

Рассмотрим пример. У нас есть вектор целых чисел, по которому можно итерироваться, используя цикл `range-based for`. Если мы хотим проитерироваться по первым трем элементам вектора, то нам придется использовать обычный цикл со счетчиком. В этом нет универсальности, к тому же если размер вектора меньше 3, то цикл может выйти за пределы вектора и программа будет вести себя не так, как мы ожидаем. Давайте напишем функцию `Head`.

```
int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (int x : Head(v, 3)) {
        cout << x << " ";
    }
}
```

Эта запись означает, что с помощью удобного цикла `range-based for` мы хотим проитерироваться по первым трем элементам вектора. Кроме того, эта функция должна корректно работать, и когда в векторе меньше чем три элемента. Мы напишем шаблон функции в данном случае.

```
template <typename T>
vector<T> Head(vector<T>& v, size_t top) {
    return {
        v.begin(),
        next(v.begin(), min(top, v.size()))
    };
}
```

Функция принимает на вход вектор `v` по ссылке, параметр `top` задаёт размер префикса, по которому мы хотим проитерироваться. Функция возвращает два итератора: начало вектора и `begin`, который с помощью оператора `next` мы продвинули на минимум из значения параметра `top` и размера вектора.

Функция `Head` создает копию вектора. Это не практично. Более того, в текущей реализации функции мы не можем изменять элементы изначального вектора.

И нам нужен какой-то другой способ, который бы позволил нам из функции `Head` вернуть диапазон внутри исходного вектора, и с помощью этого диапазона обращаться к элементам самого вектора. Например, из функции `Head` вместо копии вектора возвращать лишь пару итераторов на вектор `v` и итерироваться в цикле по ним. Это возможно сделать как раз с помощью шаблонов классов, с которыми мы с вами познакомились в предыдущем уроке.

```
template <typename Iterator>
struct IteratorRange {
    Iterator first, last;
};
```

Применим этот шаблон класса внутри функции `Head`.

```
template <typename T>
IteratorRange<typename vector<T>::iterator> Head(vector<T>& v, size_t top) {
    return {
        v.begin(), next(v.begin(), min(top, v.size()))
    };
}
```

В текущем виде код не скомпилируется, поскольку у структуры `IteratorRange` нет методов `begin` и `end`. Давайте их добавим.

```
Iterator begin() const {
    return first;
}
Iterator end() const {
    return last;
}
```

Чтобы по объекту класса можно было проитерироваться с помощью цикла `for`, он должен иметь методы `begin()` и `end()`. Методы `begin()` и `end()` должны возвращать итераторы.

### 1.2.3. Разница между шаблоном и классом

Сам по себе `IteratorRange` не является классом, это шаблон класса. В него необходимо подставить конкретный тип, чтобы создать класс. `IteratorRange` у нас параметризован типом итератора, а чтобы создать из него класс, в него нужно подставить какой-то конкретный тип итератора, например, итератор вектора целых чисел.

```
IteratorRange<vector<int>::iterator>
```

Все стандартные контейнеры, которыми мы пользовались, например, `vector`, `map`, `set` являются шаблонами классов.

Допустим, мы хотим посчитать, сколько элементов у нас находится в диапазоне двух итераторов. Мы не можем оформить функцию вот так:

```
size_t RangeSize(IteratorRange r) {
    return r.end() - r.begin();
}
```

Дело в том, что параметр `r` должен иметь тип `IteratorRange` – не тип, это шаблон типа. Чтобы создать из этого шаблона тип, его нужно инстанцировать.

```
template <typename T>
size_t RangeSize(IteratorRange<T> r) {
    return r.end() - r.begin();
}
```

## 1.2.4. Вывод типов в шаблонах классов

Допустим, мы хотим обратиться к суффиксу вектора – его второй половине.

```
IteratorRange<vector<int>::iterator> second_half {
    v.begin() + v.size() / 2, v.end()
};
```

Чтобы объявить переменную `second_half`, нам пришлось написать достаточно громоздкую конструкцию. Напишем так называемую **порождающую функцию**.

```
template <typename Iterator>
IteratorRange<Iterator> MakeRange(Iterator begin, Iterator end) {
    return IteratorRange<Iterator>(begin, end);
}
```

Теперь мы можем лаконично объявить переменную `second_half`.

```
auto second_half = MakeRange {
    v.begin() + v.size() / 2, v.end()
};
```

Порождающие функции позволяют возложить на компилятор выводение шаблонных типов при инстанцировании шаблонных классов. Они сокращают код и избавляют от необходимости много

печатать. Однако для каждого шаблона класса порождающую функцию приходится писать самостоятельно. Кроме того, из записи `auto full = MakeRange(t.begin(), t.end())` неочевиден тип переменной `full`. Необходимо отдельно проверить, что возвращает функция `MakeRange`.

Порождающие функции – это не единственный способ возложить на компилятор вывод шаблонных типов при инстанцировании шаблонов класса. Другой способ появился в стандарте C++17. Чтобы им воспользоваться, необходимо настроить вашу среду разработки в соответствии с этим стандартом.

Необходимо убедиться, что у вас стоит компилятор GCC версии не младше седьмой. В самой IDE необходимо убедиться, что проект собирается с использованием самого свежего стандарта.

Если в классе есть конструктор, позволяющий определить тип шаблона, компилятор выводит его сам. Рассмотрим пример:

```
template <typename T> struct Widget {  
    Widget(T value);  
};  
  
Widget w_int(5);
```

У нас есть шаблон класса `Widget`, в котором есть конструктор, принимающий значение `value` типа `T`. Компилятор по этому конструктору может сам вывести тип. Мы можем объявить переменную `w_int`, проинициализировать её значением 5. При этом в качестве её типа мы просто пишем `Widget`. Компилятор берёт 5 и смотрит, какие конструкторы есть в шаблоне `Widget`. Видит конструктор, принимающий `value` типа `T`. Он понимает, что 5 имеет тип `int`, поэтому мы хотим инстанцировать шаблон типа `Widget` с помощью типа `int`, и он создаёт класс `Widget<int>`.

Рассмотрим другой пример:

```
pair<int, bool> p(t, true);
```

Из документации мы знаем, что у шаблона класса `pair` есть конструктор, который принимает параметры его шаблонных аргументов, поэтому компилятор может воспользоваться конструктором и вывести типа. Код мы можем переписать следующим образом:

```
pair p(5, true);
```

Код скомпилируется, поскольку по 5 и `true` компилятор поймёт, что нам нужно из шаблона `pair` создать класс `pair<int, bool>`.

Переделаем структуру `IteratorRange` в класс, добавим туда конструктор:

```
class IteratorRange {
private:
    Iterator first, last;

public:
    IteratorRange(Iterator f, Iterator l)
        : first(f)
        , last(l)
    {
    }
    Iterator begin() const {
        return first;
    }
    Iterator end() const {
        return last;
    }
}
```

Тогда пример с `second_half` мы можем переписать следующим образом:

```
IteratorRange second_half(
    v.begin() + v.size() / 2, v.end()
)
```

Компилятор видит, что мы создаём объект класса с помощью двух аргументов типа `vector<int>::iterator`, он понимает, благодаря конструктору, что мы должны инстанциировать шаблон `IteratorRange` с помощью типа `vector<int>::iterator`. Таким образом он создаёт объект класса `IteratorRange` от `vector<int>::iterator`.

Способ вывода типов с помощью конструктора обладает преимуществами:

- не всегда нужно писать дополнительный код;
- `IteratorRange full(t.begin(), t.end())` – проще понять, какой тип у `full`.

Из следующего кода может показаться, что `r_i` и `r_s` имеют один и тот же тип, потому что перед ними стоит `IteratorRange`.



```
vector<int> ints;
vector<string> strs;
IteratorRange r_i(begin(ints), end(ints));
IteratorRange r_s(begin(strs), end(strs));
```

По умолчанию при инстанцировании стоит явно указывать шаблонный тип. Если это не удобно, то используем способ вывода через конструктор, потому что в нём явно указано имя шаблона. Если по какой-то причине мы не можем использовать этот способ, то делаем порождающую функцию.

### 1.2.5. Автоматический вывод типа, возвращаемого функцией

Вернемся к функции `Head`. Сейчас она работает только для вектора. Перепишем её так, чтобы она позволяла итерироваться по префиксу произвольного контейнера.

```
template <typename Container>
IteratorRange<???> Head(Container v, size_t top) {
    return {
        v.begin(), next(v.begin(), min(top, v.size()))
    };
}
```

Возникает вопрос: что нам написать при инстанцировании шаблона `IteratorRange`? Можем написать `typename Container::iterator`. Воспользуемся функцией `Head` для вывода четырёх минимальных элементов множества:

```
set<int> nums = {5, 7, 12, 8, 10, 5, 6, 1};
for (int x : Head(nums, 4)) {
    cout << x << ' ';
}
// 1 5 6 7
```

Для `deque<int>` код также работает правильно, однако для `const deque<int>` код не скомпилируется. Дело в том, что у `const deque` метод `begin` возвращает `const_iterator`, который не разрешает изменять элементы вектора. Нам нужно уметь выбирать между константным итератором для константных объектов и неконстантным итератором для неконстантных объектов. Напишем `auto` в качестве типа возвращаемого значения функции `Head`. Таким образом мы укажем компилятору взять возвращаемый тип из команды `return`, то есть `IteratorRange`.

```
template <typename Container>
auto Head(Container v, size_t top) {
    return IteratorRange{
        v.begin(), next(v.begin(), min(top, v.size()))
    };
}
```

Такой код работает для константного `deque`. Кроме того, работает пример с модификацией вектора `v` с помощью функции `Head`. Теперь компилятор сам выводит тип итератора, с которым нужно инстанцировать `IteratorRange`.

По умолчанию следует явно указывать тип результата функции. Использовать `auto` в качестве типа результата функции стоит только если:

- тип результата громоздкий;
- тело функции очень короткое.

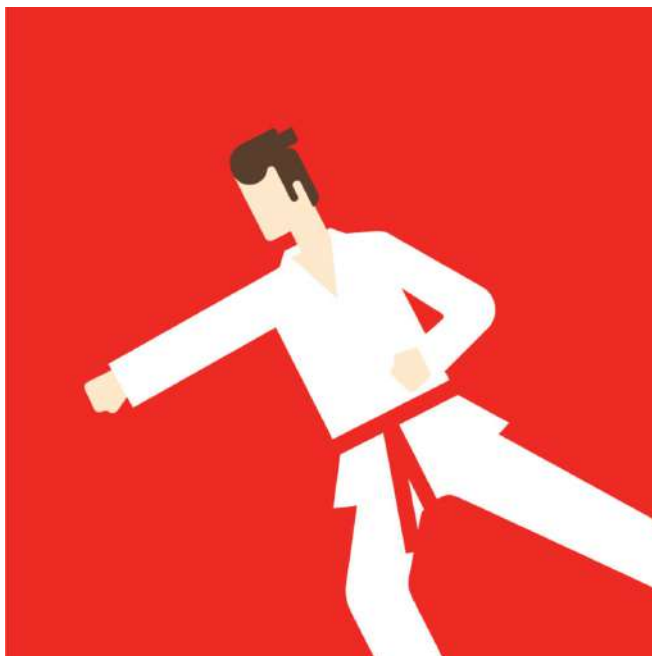
В противном случае может пострадать понятность кода.



# Основы разработки на C++: красный пояс

## Неделя 2

Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода



# Оглавление

<b>Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода</b>	<b>2</b>
2.1 Принципы оптимизации кода . . . . .	2
2.1.1 Первое правило оптимизации кода . . . . .	2
2.1.2 Второе правило оптимизации кода . . . . .	2
2.1.3 Разработка своего профайлера . . . . .	4
2.1.4 Совершенствование своего профайлера . . . . .	6
2.2 Эффективное использование потоков ввода/вывода . . . . .	8
2.2.1 Буферизация в выходных потоках . . . . .	8
2.2.2 Когда нужно использовать <code>endl</code> , а когда – <code>'\n'</code> . . . . .	9
2.2.3 Связанность потоков . . . . .	10
2.3 Сложность алгоритмов . . . . .	12
2.3.1 Введение . . . . .	13
2.3.2 Оценка сложности . . . . .	15
2.3.3 Практические применения . . . . .	16
2.3.4 Амортизированная сложность . . . . .	20

# Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода

## 2.1. Принципы оптимизации кода

### 2.1.1. Первое правило оптимизации кода

**Избегайте преждевременной оптимизации.** Преждевременная оптимизация – это упор на производительность в ущерб простоте, дизайну, понятности, поддерживаемости и так далее. В соответствии с принципом Парето 80% работы программы тратится на исполнение 20% кода. Эти 20% и нужно оптимизировать, но для начала их нужно найти. Делать максимально быстрым весь код тяжело, время на это будет уходить впустую. Преждевременная оптимизация приводит к необоснованному усложнению кода, затруднению поддержки и замедлению разработки. Как правило код надо стараться сделать правильным, простым, а только потом быстрым. Это не значит, что о производительности вообще не надо думать.

### 2.1.2. Второе правило оптимизации кода

Код не может быть просто быстрым или медленным. Он может быть достаточно или не достаточно быстрым для задачи, которую он решает. Рассмотрим пример.

```
vector<string> GenerateBigVector() {  
    vector<string> result;  
    for (int i = 0; i < 28000; ++i) {  
        result.insert(begin(result), to_string(i));  
    }  
    return result;  
}
```

```
}

int main() {
    cout << GenerateBigVector().size() << endl;
    return 0;
}
```

Функция `GenerateBigVector` генерирует вектор из 28000 элементов, программа выводит размер этого вектора. Замерим время работы программы. Программа работает 4.337 секунды. Понятно, что для данной задачи это медленно.

Мы можем бороться с проблемой интуитивно. Создадим версию `GenerateBigVector`, которая будет принимать вектор по ссылке, тогда не возникнет лишнего копирования и предположительно программа станет работать быстрее. Сделаем вектор из 32000 элементов.

```
void GenerateBigVector(vector<string>& result) {
    for (int i = 0; i < 32000; ++i) {
        result.insert(begin(result), to_string(i));
    }
}
```

Запустим программу с такой версией функции `GenerateBigVector` и замерим время работы. Программа отработала за 5.638 секунды. Это говорит о том, что интуиция нас подвела. Таким образом, мы приходим ко второму правилу оптимизации, которое звучит просто: **замеряйте!**

Интуиция часто даёт неверные результаты, а измерения объективны, они позволяют найти то место программы, из-за которой она работает медленно.

Замерим, сколько работает программа, а также замерим, сколько работает цикл по формированию вектора.

```
vector<string> GenerateBigVector() {
    vector<string> result;
    auto start = steady_clock::now();
    for (int i = 0; i < 28000; ++i) {
        result.insert(begin(result), to_string(i));
    }
    auto finish = steady_clock::now();
    auto duration = finish - start;
    cerr << "Cycle: "
         << duration_cast<milliseconds>(duration).count()
```

```
        << endl;
    return result;
}

int main() {
    auto start = steady_clock::now();
    cout << GenerateBigVector().size() << endl;
    auto finish = steady_clock::now();
    auto duration = finish - start;
    cerr << "Total: "
        << duration_cast<milliseconds>(duration).count()
        << endl;
    return 0;
}
```

Замерив время, мы понимаем, что общее время работы программы и время формирования вектора совпадает. Большая часть времени уходит на работу в цикле.

```
result.insert(begin(result), to_string(i));
```

Здесь мы вставляем каждый следующий элемент в начало вектора. Чтобы вставить элемент в начало, нужно сдвинуть текущее содержимое вектора вправо. Изменим программу и будем вставлять каждый новый элемент не в начало вектора, а в конец.

```
result.push_back(to_string(i));
```

Такая программа отработает всего за 6 миллисекунд.

Прежде чем ускорять код, замерьте, сколько он работает. Если это недостаточно быстро, то начинайте искать узкие места. Интуиция не работает – замеряйте!

### 2.1.3. Разработка своего профайлера

На практике для нахождения медленно работающего места в программе используются специальные инструменты, которые называются профайлерами. Некоторые из них:

- gperftools (Linux, Mac OS);

- perf (Linux);
- VTune (Windows, Linux).

Мы сделаем свой простой профайлер и будем им пользоваться.

Замерять время работы с помощью способа из предыдущего пункта громоздко. Напишем специальный класс.

```
class LogDuration {
public:
    LogDuration()
        : start(steady_clock::now())
    {
    }

    ~LogDuration() {
        auto finish = steady_clock::now();
        auto dur = finish - start;
        cerr << duration_cast<milliseconds>(dur).count()
              << " ms" << endl;
    }
private:
    steady_clock::time_point start;
};
```

`start` – это момент начала измерений. В конструкторе мы записываем в это поле `steady_clock::now()`. В деструкторе мы запоминаем момент окончания работы. Вычисляем длительность работы (разность). Выводим длительность в миллисекундах.

Теперь для измерения длительности работы блока кода мы пишем перед ним `LogDuration input;` и оборачиваем конструкцию в фигурные скобки:

```
{
    LogDuration input;
    for (int i = 0; i < element_count; ++i) {
        int x;
        cin >> x;
        elements.insert(x);
    }
}
```



Улучшим читаемость результатов работы программы.

```
class LogDuration {
public:
    explicit LogDuration(const string& msg = "")
        : message(msg + ": ")
        , start(steady_clock::now())
    {
    }

    ~LogDuration() {
        auto finish = steady_clock::now();
        auto dur = finish - start;
        cerr << message
            << duration_cast<milliseconds>(dur).count()
            << " ms" << endl;
    }
private:
    string message;
    steady_clock::time_point start;
};
```

Теперь мы можем добавлять сообщение, например:

```
{
    LogDuration input("Input");
    for (int i = 0; i < element_count; ++i) {
        int x;
        cin >> x;
        elements.insert(x);
    }
}
// Input: 113 ms
```

## 2.1.4. Совершенствование своего профайлера

Посмотрим, как мы пользуемся классом LogDuration.

```
{
    LogDuration input("Input");
    ...
}
```

Мы объявляем переменную с именем `input` и в качестве сообщения тоже используем `"Input"`. Имена переменных нам не нужны, мы не обращаемся к ним в коде. Нам нужен способ, который позволит объявлять объекты класса `LogDuration`, но избавит от необходимости придумывать имя для переменных. Тут нам помогут макросы.

```
#define LOG_DURATION(message) \
    LogDuration UNIQ_ID(__LINE__){message};
```

Макрос `UNIQ_ID` имеет следующее устройство (не будем останавливаться на том, как оно получилось):

```
#define UNIQ_ID_IMPL(lineno) _a_local_var_##lineno
#define UNIQ_ID(lineno) UNIQ_ID_IMPL(lineno)
```

Просто поверьте на слово: конструкция `UNIQ_ID(__LINE__)` позволяет объявить уникальный идентификатор в пределах заданного `.cpp`-файла.

Теперь мы можем замерять время с помощью конструкции

```
{
    LOG_DURATION("Input");
    ...
}
```

Наша программа работает. Ввод длится 102 миллисекунды, обработка запросов – 76, вся программа – 181. При этом мы добились желаемого удобства. Нам больше не нужно придумывать имя для объекта, мы просто пишем `LOG_DURATION` и сообщение и получаем в стандартный поток ошибок длительность работы соответствующего блока `count`. Это действительно удобно.

Допустим, мы захотим померить, сколько работает одна итерация нашего цикла. Мы просто пишем `LOG_DURATION("Iter " + to_string(i))`.

Остался один штрих, чтобы сделать класс `LogDuration` переиспользуемым. Давайте его вынесем в отдельный файл. Заведём заголовочный файл в нашем проекте. Назовем его `profile.h` и вынесем сюда класс `LogDuration` и необходимые `include`'ы. Это `#include <chrono>`, `using`

`namespace std;`, `using namespace std::chrono;`. И так как мы выводим в `cerr`, нам понадобится ещё `<iostream>`.

Подведём итоги. Мы смогли разработать класс `LogDuration`, который можно использовать для замера времени работы программы и отдельных её блоков, а также сделали специальный макрос `LOG_DURATION`, который нас избавляет от придумывания ненужных идентификаторов и упрощает работу с этим профайлером.

## 2.2. Эффективное использование потоков ввода/вывода

В этом модуле мы посмотрим, какие проблемы могут возникнуть при использовании потоков ввода/вывода и как эти проблемы решаются.

### 2.2.1. Буферизация в выходных потоках

Файловый поток `ofstream` не сразу пишет выводимые в него данные в файл, а накапливает их в промежуточном буфере и сбрасывает его в файл, когда он наполнился. Поток вывода `cout` ведет себя точно так же. Манипулятор `endl` не только выводит перевод строки, но и сбрасывает буфер потока в файл.

Сравним производительность потоков. Будем измерять, за какое время выведутся в файл 15000 строк при использовании `endl` и при использовании `'\n'`. Будем использовать профайлер `profile.h`.

```
int main() {
    {
        LOG_DURATION("endl");
        ofstream out("output.txt");
        for (int i = 0; i < 15000; ++i) {
            out << "London is the capital of Great Britain. "
                << "I am travelling down the river"
                << endl;
        }
    }
    {
        LOG_DURATION("'\\n'");
        ofstream out("output2.txt");
```

```
for (int i = 0; i < 15000; ++i) {
    out << "London is the capital of Great Britain. "
        << "I am travelling down the river"
        << '\n';
}
}
}
// endl: 137 ms
// '\n': 16 ms
```

Пусть теперь выводится в 10 раз больше строчек, снова выполним наш код, и увидим поразительную огромную разницу: `endl` – 530 миллисекунд, `'\n'` – 168 миллисекунд.

Тот факт, что `endl` сбрасывает буфер потока, имеет значительное влияние на производительность. Дело в том, что при использовании `endl` мы пишем в файл каждый раз, а при использовании `'\n'` – только когда буфер заполнился.

Таким образом, использование `endl` может приводить к снижению скорости вывода в файл или `cout`.

### 2.2.2. Когда нужно использовать `endl`, а когда – `'\n'`

Возникает вопрос: зачем использовать `endl`? Он нужен в ситуациях, когда нам не важна скорость вывода всей информации, а важно увидеть последнее выведенное сообщение, как только оно было выведено в выходной поток. Самый простой пример – отладка программы с использованием отладочного вывода в консоль. Использовать буферизованный вывод в данном случае неудобно, потому что вероятна ситуация, когда вы выводите отладочное сообщение, а на следующей команде программа падает. Таким образом, есть вероятность, что это сообщение не будет выведено в консоль, хотя оно может содержать важную информацию об ошибке.

Рассмотрим пример с классом `LogDuration`. Эта программа состоит из двух функций. Тела функций `CouldBeSlowOne` и `CouldBeSlowTwo` скрыты, потому что сейчас они не важны. Одна из них работает медленно, мы не знаем, какая. Чтобы узнать это, мы оборачиваем их в `LOG_DURATION`.

```
int main() {
    {
        LOG_DURATION("One");
```

```
    CouldBeSlowOne();  
}  
{  
    LOG_DURATION("Two");  
    CouldBeSlowTwo();  
}  
}  
// One: 9 ms  
// Two: 7093 ms
```

Если мы реализуем `LOG_DURATION` с помощью `'\n'`, то оба сообщения выведутся в конце работы программы, ждать придётся долго. Если же в реализации использовать `endl`, то первое сообщение выведется сразу после завершения работы первой функции. Это удобнее, мы можем не ждать выполнения второй функции.

Теперь давайте посмотрим на `TestRunner` из предыдущего курса (курс "[Основы разработки на C++: жёлтый пояс](#)"), давайте посмотрим на `class TestRunner` и его метод `RunTest`. `endl` здесь используется во всех сообщениях. Когда тест выполнен успешно, выводится `OK` и `endl`. И когда случилось падение теста, мы тоже выводим его имя, говорим, что он упал, сообщение из исключения и `endl`.

Если здесь заменить `endl` на символ перевода строки, то мы не будем видеть сразу результат выполнения тестов. Конечно, когда у нас простые юнит-тесты, они все отрабатывают очень быстро, и нам не важно, что эти сообщения буферизируются и потом выводятся целиком. Но если у нас есть юнит-тесты, которые занимают заметное время, то нам лучше сразу же получать информацию о том, прошли они или нет.

Перевод строки можно использовать, когда вам важна скорость вывода. Например, если вы пишете консольное приложение, которое превращает данные из стандартного ввода в какие-то другие данные и выводит их в стандартный вывод.

### 2.2.3. Связанность потоков

Итак, теперь мы с вами знаем, что замена `endl` на символ перевода строки может ускорять вывод программ на C++. Рассмотрим пример:

```
int main() {  
    for (int i = 0; i < 100000; ++i) {
```

```
int x;  
cin >> x;  
cout << x << endl;  
}  
}
```

В цикле 100000 раз из стандартного ввода считывается число и 100000 раз записывается в стандартный вывод. Логично ожидать, что при замене `endl` на `'\n'` программа заработает быстрее. Однако стоит проверить это и сделать измерения.

```
int main() {  
    {  
        LOG_DURATION("endl");  
        for (int i = 0; i < 100000; ++i) {  
            int x;  
            cin >> x;  
            cout << x << endl;  
        }  
    }  
    {  
        LOG_DURATION("'\\n'");  
        for (int i = 0; i < 100000; ++i) {  
            int x;  
            cin >> x;  
            cout << x << endl;  
        }  
    }  
}  
// endl: 387 ms  
// '\\n': 373 ms
```

Реализация с `'\n'` работает немного быстрее, однако мы ожидали, что она отработает в разы быстрее реализации с `endl`. Рассмотрим другой пример:

```
cout << "Enter two integers: ";  
for (;;) {  
}  
//
```

Данный код не выведет на экран ничего, потому что передаваемое сообщение попало в буфер и остаётся там на протяжении бесконечного цикла.

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
// Enter two integers:
```

Такой код выводит `Enter two integers: .` Получается, произошёл сброс внутреннего буфера `cout`. Этот пример нам показывает, что по умолчанию поток `cout` связан с потоком `cin`. Это специально сделано для интерактивных приложений вроде нашего калькулятора. Этим же объясняется неожиданно медленная работа реализации с `'\n'` из прошлого примера. Мы можем разорвать связь между `cout` и `cin`:

```
cin.tie(nullptr);
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
```

Что такое `nullptr`, будет рассказано далее в модуле «Модели памяти».

Ещё больше ускорить работу программ можно, если прописать строку:

```
ios_base::sync_with_stdio(false);
```

Что это за команда? Это выходит за рамки нашего курса. Важно помнить, что вызывать её надо до первой операции ввода/вывода.

## 2.3. Сложность алгоритмов

Мы научились замерять время работы кода. В частности, сравнивать два алгоритма по времени работы. Уже в предыдущем курсе возникали ситуации, когда один алгоритм заведомо эффективнее другого. Например, мы проходили контейнер `deque`. Чем он был лучше, чем вектор?

### 2.3.1. Введение

Рассмотрим пример. Попробуем вставлять элементы в начало вектора и в начало `deque`. Вспомним, что вставлять в начало вектора гораздо менее эффективно, чем вставлять в начало `deque`.

```
{
    LOG_DURATION("vector");
    vector<int> v;
    for (int i = 0; i < 100000; ++i) {
        v.insert(begin(v), i);
    }
}
{
    LOG_DURATION("deque");
    deque<int> v;
    for (int i = 0; i < 100000; ++i) {
        v.insert(begin(v), i);
    }
}
// vector: 1661 ms
// deque: 9 ms
```

Рассмотрим ещё один пример. Продемонстрируем, что метод `lower_bound` эффективнее, чем одноимённая функция на примере множества.

```
set<int> numbers;
for (int i = 0; i < 3000000; ++i) {
    numbers.insert(i);
}
const int x = 1000000;

{
    LOG_DURATION("global_lower_bound");
    cout << *lower_bound(begin(numbers), end(numbers), x);
}
{
    LOG_DURATION("lower_bound_method");
    cout << *numbers.lower_bound(x);
}
// global lower_bound: 944 ms
// lower_bound method: 0 ms
```



Рассмотрим третий пример. Сравним функции `lower_bound` и функции `find_if`. Функция `lower_bound` делает бинарный поиск, в то время как функция `find_if` запускает цикл `for`. `lower_bound` заведомо эффективнее, проверим это.

```
const int NUMBER_COUNT = 1000000;
const int NUMBER = 7654321;
const int QUERY_COUNT = 10;

int main() {
    vector<int> v;
    for (int i = 0; i < NUMBER_COUNT; ++i) {
        v.push_back(i * 10);
    }

    {
        LOG_DURATION("lower_bound");
        for (int i = 0; i < QUERY_COUNT; ++i) {
            lower_bound(begin(v), end(v), NUMBER);
        }
    }

    {
        LOG_DURATION("find_if");
        for (int i = 0; i < QUERY_COUNT; ++i) {
            find_if(begin(v), end(v)),
                [NUMBER](int y) { return y >= NUMBER});
        }
    }

    return 0;
}
// lower_bound: 0 ms
// find_if: 123 ms
```

`lower_bound` быстрее, об этом мы и так заранее знали.

Чтобы понять, будет ли эффективен алгоритм, не обязательно его программировать и замерять. Иногда можно заранее знать, какой алгоритм больше подходит.

### 2.3.2. Оценка сложности

Обозначим за  $N$  размер диапазона. Нам известно, что `lower_bound` работает за время порядка  $\log N$ , а `find_if` в худшем случае работает за  $N$ . Не будем учитывать множители при  $\log N$  и  $N$ , так как при достаточно большом  $N$ ,  $N$  будет больше  $\log N$  при любом константном множителе при них. В частности, будем считать, что логарифм двоичный.

Рассмотрим более сложный пример. В цикле пройдемся по контейнеру `numbers_to_find` и для каждого элемента этого контейнера вызовем `lower_bound`. Поищем это число в другом контейнере. Затем выполним простую операцию, например,  $x$  умножим на 2.

```
for (int& x : numbers_to_find) {
    lower_bound(begin(v), end(v), x);
    x *= 2;
}
```

Внешний цикл делает столько итераций, каков размер контейнера `numbers_to_find`. Дальше в каждой из итераций мы запускаем `lower_bound`, который работает  $\log N$  времени, потом еще одна операция тратится на умножение  $x$  на 2. Получаем выражение для грубой оценки количества операций.

```
numbers_to_find.size() * (log(v.size()) + 1)
```

Это оценка для худшего случая. В реальности она может не достигаться.

Выполним программу для `NUMBER = 1` и `QUERY_COUNT = 1000000`.

```
// lower_bound: 701 ms
// find_if: 81 ms
```

`lower_bound` выполнял логарифм операций, в то время как `find_if` сходилась практически сразу, выполнив всего несколько операций.

Обсудим сложность алгоритмов.

- **Константная** (1): арифметические операции, обращение к элементу вектора;
- **Логарифмическая** ( $\log N$ ): двоичный поиск (`lower_bound` и пр.), поиск в `set` или `map`;
- **Линейная** ( $N$ ): цикл `for` (с лёгкими итерациями), алгоритмы `find_if`, `min_element` и пр.;

- $N \log N$ : алгоритм `sort` (с лёгкими сравнениями).

Часто используют  $O$ -символику. Например,  $O(\log N)$  означает, что алгоритм выполняет не больше, чем  $\log N$  операций с точностью до константы. « $O$  большое» означает оценку сверху. В принципе она может не достигаться вообще никогда.

Как узнать сложность работы незнакомого алгоритма? Можно вычислить самостоятельно, если известны детали работы алгоритма. Также сложность можно посмотреть в документации.

### 2.3.3. Практические применения

Скорость алгоритмов стоит сравнивать по следующему несложному правилу:

$$1 < \log N < N < N \log N < N^2 < \dots$$

При сложении большее поглощает меньшее:

$$O(N) + O(\log N) = O(N)$$

$O(5N)$  и  $O(8N)$  надо сравнивать измерениями.

Как оценить, подходит ли алгоритм под заданные ограничения? Нужно рассматривать худший случай, брать сложность алгоритма и подставлять туда худшие значения входных данных. Так получается количество операций алгоритма. Далее нужно примерно прикинуть константу, умножить количество операций на неё. В результате получается грубая оценка количества элементарных операций. Далее можно прикинуть время работы алгоритма, если учесть, что на хороших процессорах за секунду можно успеть сделать миллиард простых операций.

Рассмотрим примеры.

**Задача «Синонимы»:**

Поступает три типа запросов.

- `ADD word1 word2`: добавить пару синонимов (`word1`, `word2`);

- COUNT word: узнать количество синонимов слова word;
- CHECK word1 word2: узнать, являются ли слова word1 и word2 синонимами.

В задаче приходит 70000 запросов, слова имеют длину не более 100 символов, решить задачу надо за 1 секунду.

```
int main() {
    int q;
    cin >> q;
    map<string, set<string>> synonyms; // для каждого слова запоминаем множество его
                                      // синонимов
    for (int i = 0; i < q; ++i) { // обрабатываем Q запросов (будет Q итераций)
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word; // если слова длины не более чем L, то
                                                // считываем их за L простейших операций
            synonyms[first_word].insert(second_word); // поиск в словаре конкретного
                                                        // ключа работает за логарифм от
                                                        // размера словаря. Он в худшем
                                                        // случае равен количеству слов,
                                                        // которые уже туда вставили.
                                                        // Сложность равна количеству
                                                        // запросов, умноженная на
                                                        // стоимость сравнения двух строк
                                                        // длины не более L.
                                                        // итоговая сложность: L logQ

            synonyms[second_word].insert(first_word);
            // O(L) + O(L logQ) = O(L logQ)
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << synonyms[word].size() << endl;
            // поиск в словаре занимает L logQ. Итоговая сложность: O(L logQ)
        } else if (operation_code == "CHECK") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            if (synonyms[first_word].count(second_word) == 1) {
                cout << "YES" << endl;
            }
        }
    }
}
```

```

    } else {
        cout << "NO" << endl;
    }
    // считываем два слова, ищем их в словаре, потом во множестве
    //  $O(L \log Q)$ 
}
}

// обрабатываем Q запросов за  $L \log Q$ 
// суммарная сложность программы:  $O(Q L \log Q)$ 
return 0;
}

```

### Задача «Имена и фамилии – 1»:

- `ChangeFirstName(year, first_name)`: добавить факт изменения имени на `first_name` в год `year`;
- `ChangeLastName(year, last_name)`: добавить факт изменения фамилии на `last_name` в год `year`;
- `GetFullName(year)`: получить имя и фамилию по состоянию на конец года `year`;

Были следующие ограничения: 100 запросов, слова длины 10, на выполнение дана одна секунда.

```

string FindNameByYear(const map<int, string>& names, int year) {
    string name; // изначально имя неизвестно
    for (const auto& item : names) { //  $O(Q)$  итераций
        if (item.first <= year) { // сравниваем два числа за  $O(1)$ 
            name = item.second; // присвоить одну строку в другую стоит столько же
                               // операций, как длина строки:  $O(L)$ 
        } else {
            // иначе пора остановиться, так как эта запись и все последующие относятся к
            // будущему
            break;
        }
    }
    return name;
    //  $O(QL)$  операций
}

```

```

class Person {
public:
    void ChangeFirstName(int year, const string& first_name) {
        first_names[year] = first_name; // ищем year как ключ в словаре и сохраняем
                                         // туда строчку
    } // O(logQ + L)
    void ChangeLastName(int year, const string& last_name) {
        last_names[year] = last_name;
    } // O(logQ + L)
    string GetFullName(int year) {
        const string first_name = FindNameByYear(first_names, year);
        const string last_name = FindNameByYear(last_names, year);

        if (first_name.empty() && last_name.empty()) {
            return "Incognito";
        } else if (first_name.empty()) {
            return last_name + " with unknown first name";
        } else if (last_name.empty()) {
            return first_name + " with unknown last name";
        } else {
            return first_name + " " + last_name;
        }
        // сложность O(L), которая тратится на сложение строк. Также тут вызывается
        // FindNameByYear. Итоговая сложность: O(L) + O(QL)
    }

private:
    map<int, string> first_names;
    map<int, string> last_names;
};
// итоговая сложность программы: O(Q * Q * L)

```

В задаче «Имена и фамилии – 4» оптимизирована функция `FindNameByYear`. В ней теперь вызывается метод `upper_bound`, которая работает за логарифм от размера контейнера, то есть за  $\log Q$ . Тогда суммарная сложность будет  $O(\log Q + L)$ . Метод `GetFullName` также отработает за  $O(\log Q + L)$ . Получаем суммарную сложность  $O(Q(L + \log Q))$ .

### 2.3.4. Амортизированная сложность

Рассмотрим задачу. К нам приходят события с временными метками. Нужно уметь добавлять событие с временной меткой и находить, сколько событий случилось за последние пять минут. Для решения задачи напомним класс:

```
class EventManager {  
public:  
    void Add(uint64_t time);  
    int Count(uint64_t time);  
}
```

Метод `Add` добавляет события, метод `Count` узнаёт количество событий за последние пять минут. Задачу можно решать с помощью очереди. Подключим `<queue>`.

```
private:  
    queue<uint64_t> events;
```

В очередь мы будем добавлять новые события и удалять старые:

```
void Add(uint64_t time) {  
    events.push(time);  
}  
int Count(uint64_t time) {  
    return events.size();  
}
```

Добавим в `private` метод `Adjust`, который принимает новые временные метки и удаляет старые.

```
void Adjust(uint64_t time) {  
    while (!events.empty() && events.front() <= time - 300) {  
        events.pop();  
    }  
}
```

Удалять старые события нужно и при добавлении в очередь, и при вычислении количества событий в ней.

```
void Add(uint64_t time) {  
    Adjust(time);  
    events.push(time);  
}
```

```
int Count(uint64_t time) {  
    Adjust(time);  
    return events.size();  
}
```

Попробуем оценить сложность каждого запроса.

```
class EventManager {  
public:  
    void Add(uint64_t time) { // O(Q)  
        Adjust(time);  
        events.push(time); // O(1)  
    }  
    int Count(uint64_t time) { // O(Q)  
        Adjust(time);  
        return events.size(); // O(1)  
    }  
private:  
    queue<uint64_t> events;  
    void Adjust(uint64_t time) { // O(Q)  
        while (!events.empty() && events.front() <= time - 300) { // в худшем  
                                                                    // случае O(Q)  
            events.pop(); // O(1)  
        }  
    }  
}
```

Получается, суммарная сложность будет квадратичной? Дело в том, что это верхняя оценка и она достигаться не будет. Каждый конкретный вызов `Adjust`, `Add` или `Count` может работать долго. Суммарно все эти события будут работать за  $O(Q)$ , а не за  $O(Q * Q)$ . В таком случае говорят, что сложность `Add`, `Count`, `Adjust` не  $O(Q)$ , а *amortized*  $O(1)$ , то есть в среднем каждый метод работает как  $O(1)$ .

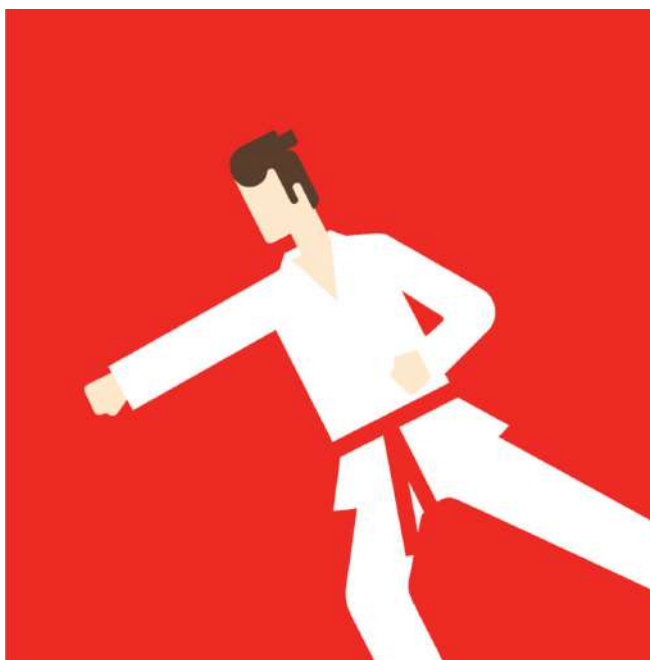




# Основы разработки на C++: красный пояс

Неделя 3

Модель памяти в C++



# Оглавление

Модель памяти в C++	2
3.1 Модель памяти	2
3.1.1 Введение в модель памяти: стек	2
3.1.2 Введение в модель памяти: куча	4
3.1.3 Оператор <code>new</code>	5
3.1.4 Оператор <code>delete</code>	6
3.1.5 <code>new</code> и <code>delete</code> для объектов классового типа	8
3.1.6 Операторы <code>new[]</code> и <code>delete[]</code>	9
3.1.7 Введение в арифметику указателей	10
3.1.8 Добавляем в вектор <code>begin</code> и <code>end</code>	13
3.1.9 Константный указатель и указатель на константу	15
3.1.10 Итоги раздела	17

# Модель памяти в C++

## 3.1. Модель памяти

В этом разделе мы изучим, как эффективно использовать основные типы языка C++. Для этого необходимо изучить, как они устроены внутри.

### 3.1.1. Введение в модель памяти: стек

Рассмотрим пример:

```
void second() {
    int s_a = 3;
    double s_d = 2.0;
}

void first() {
    int f_a = 2;
    char f_c = 'a';
    second();
}

int main() {
    int a = 1;
    char c = 'r';
    first();
    second();
    a = 2;
    c = 'q';
}
```

Стек	
main()	int a
	char c

У каждой функции есть локальные переменные. Они хранятся в специальной области оперативной памяти, которая называется стек.

Когда в программе запускается очередная функция, то на стеке резервируется блок памяти, достаточный для хранения локальных переменных. Кроме того, в этом блоке хранится служебная информация, такая как адрес возврата. Такая область памяти называется стековым фреймом функции. Когда функция `main` запускает функцию `first`, то на стеке ниже функции `main` резервируется фрейм для функции `first`. То же самое происходит, когда функция `first` вызывает функцию `second`.

Стек	
main()	int a
	char c
first()	int f_a
	char f_c
second()	int s_a
	double s_d

Стек	
main()	int a
	char c
second()	int s_a
	double s_d
second()	int s_a
	double s_d

Когда функция `second` завершает свою работу, то вершина стека перемещается на фрейм предыдущей функции. При этом фрейм отработавшей функции просто остаётся на стеке. Выйдем из функции `first` в функцию `main`, запустим функцию `second` из `main`.

Стековый фрейм перетирает данные, которые были в стеке от предыдущих вызовов. Выйдем из функции `second` и выйдем из функции `main`. Когда выходим из программы, то вершина стека поднимается до самого верха.

### 3.1.2. Введение в модель памяти: куча

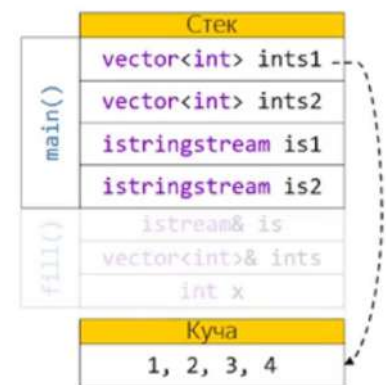
Рассмотрим пример

```
void fill(istream& is, vector<int>& ints) {
    int x;
    while (is >> x) {
        ints.push_back(x);
    }
}

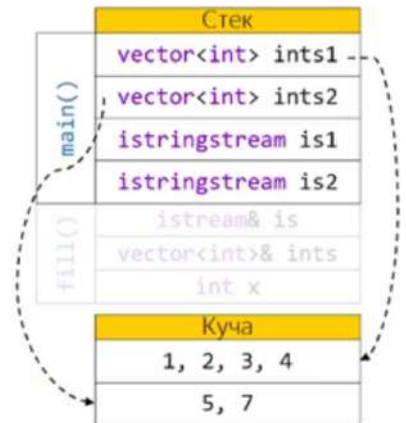
int main() {
    vector<int> ints1, ints2;
    istringstream is1("1 2 3 4");
    fill(is1, ints1);
    istringstream is2("5 7");
    fill(is2, ints2);
}
```

В функции `main` объявлены два вектора целых чисел, также объявлены два потока ввода из строк. Кроме того, есть функция `fill`, она читает из потока числа и помещает их в вектор. В результате работы программы в векторе `ints1` будут храниться цифры 1, 2, 3, 4; в векторе `ints2` будут храниться цифры 5, 7. Предположим, что числа, которые функция `fill` разместит в вектор, располагаются ниже её стекового фрейма. Это не так, потому что при повторном запуске `fill`, она перезапишет эти числа. Получается, в данном случае нам не подходит стек, потому что объекты, создаваемые функцией `fill`, должны жить дольше, чем функция `fill`. На стеке память постоянно будет перезаписываться вызовами новых функций.

Для хранения элементов вектора нам подходит источник памяти, который называется «куча». Во время работы программа может в любой момент обращаться в кучу и выделять в ней блок памяти произвольного размера или освободить ранее выделенный. Посмотрим, как программа из примера использует кучу для хранения элементов вектора.



После завершения работы функции `fill` в куче будет выделен блок памяти, достаточный для хранения четырех целых чисел. При этом вектор `ints1` будет ссылаться на эту область памяти в куче и будет иметь доступ к своим элементам.



Когда `fill` отработает во второй раз, то в куче будет выделен отдельный блок памяти, достаточный для хранения двух целых чисел.

Подведём итог. В модели памяти C++ есть два источника памяти: стек и куча. В стеке размещаются локальные переменные функций. В куче размещаются объекты, которые должны жить дольше, чем создавшая их функция.

### 3.1.3. Оператор `new`

Возможность выделения памяти в куче доступна любому программисту. Чтобы выделить в куче память для хранения одного значения типа `int` нужно объявить локальную переменную типа `int*`.

```
int* pInt = new int;
```

`new` – это специальный оператор, который выполняет выделение памяти из кучи. Оператор `int*` подсказывает, для хранения какого типа нам нужна память. При этом `pInt` – это указатель на значение типа `int`. Сама переменная `pInt` будет храниться в стеке, но она будет указывать на область памяти в куче.



Указатель – это адрес в памяти. Память в C++ представляется как линейный массив байтов. Указатель – это индекс в этом массиве.

Как обратиться к значению, на которое указывает указатель? Синтаксис при работе с указателями такой же, как при работе с итераторами.

Рассмотрим другой пример.

```
string* s = new string;
*s = "Hello";
cout << *s << ' ' << s->size() << endl;
// Hello 5
```

Оператор \*, примененный к указателю, возвращает ссылку на объект в куче.

```
string* s = new string;
*s = "Hello";
string& ref_to_s = *s;
ref_to_s += ", world";
cout << *s << endl;
// Hello, world
```

Инициализация объектов, выделенных на куче, выполняется аналогично инициализации объектов, создаваемых на стеке при объявлении локальных переменных.

### 3.1.4. Оператор delete

Рассмотрим пример. У нас есть вектор целых чисел `v`, в отдельной области видимости объявлен итератор `iter`, в который записан результат поиска пятерки в нашем векторе `v`.

```
vector<int> v = {1, 2, 3, 4, 5};
{
    auto iter = find(begin(v), end(v), 5);
}
```

Что станет с пятёркой после выхода `iter` из области видимости? Ничего не произойдет, потому что итератор просто указывает позицию в контейнере и никак не управляет значением, на которое он указывает. Сам итератор разрушился, а значение в векторе никуда не делось, оно продолжает там храниться и мы можем им пользоваться.

Рассмотрим другой пример. Мы в отдельной области видимости выделяем в куче память для хранения целого числа и инициализируем её пятеркой, указатель на эту память мы записываем

в переменную `pFive`. Что станет с пятёркой в куче после выхода `pFive` из области видимости? Ничего – пятёрка останется в куче, и произойдёт утечка памяти, то есть память, выделенная на куче, будет числиться за нашим процессом и у нас не будет возможности обратиться к этой памяти и освободить её, потому что мы потеряли указатель на эту память, он вышел за пределы области видимости.

Напишем программу, которая будет считывать со входа число `n` и будет считать сумму `n` случайных 64-битных чисел.

```
int main() {
    int n;
    cin >> n;

    mt19937_64 random_gen; // генератор случайных чисел
    uint64_t sum = 0;
    for (int i = 0; i < n; ++i) {
        uint64_t x = random_gen();
        sum += x;
    }
    cout << sum;
}
// на вход подаём 10
// на выходе 4762160605604824475
```

Программа работает нормально. Теперь будем выделять в куче память для хранения очередного случайного числа.

```
for (int i = 0; i < n; ++i) {
    auto x = new uint64_t;
    *x = random_gen();
    sum += *x;
}
```

Такая программа также работает, но происходит утечка памяти. Это может стать проблемой при запуске программы на больших `n`. Бороться с этим можно при помощи оператора `delete`. Он освобождает память, выделенную оператором `new`.

```
for (int i = 0; i < n; ++i) {
    auto x = new uint64_t;
    *x = random_gen();
    sum += *x;
```



```
delete x;  
}
```

Программа работает, при этом количество выделенной памяти не растёт.

### 3.1.5. new и delete для объектов классового типа

Продemonстрируем, что при вызове оператора `new` для объектов классового типа оператор `new` не только выделяет необходимую память в куче, но и вызывает конструктор.

```
struct Widget {  
    Widget() {  
        cout << "constructor" << endl;  
    }  
};  
  
int main() {  
    new Widget;  
}  
// constructor
```

Когда для этого объекта вызывается деструктор? Напишем его.

```
~Widget() {  
    cout << "destructor" << endl;  
}
```

При выполнении программы в консоль вывелось только `constructor`. Деструктор не был вызван. Деструктор вызывает оператор `delete`.

```
int main() {  
    Widget* w = new Widget;  
    delete w;  
}  
// constructor  
// destructor
```

Мы убедились, что оператор `delete` не только освобождает место в памяти, но и вызывает деструктор для соответствующего объекта в куче.

### 3.1.6. Операторы `new[]` и `delete[]`

Напишем свой собственный `vector`. Начнём с простого интерфейса:

```
template <typename T>
class SimpleVector {
public:
    explicit SimpleVector (size_t size);
    ~SimpleVector();

private:
    T* data;
};

int main() {
    SimpleVector<int> sv(5);
}
```

Объявим шаблон класса `SimpleVector`. В интерфейсе нашего класса пока будут только конструктор и деструктор. Конструктор принимает количество элементов в нашем векторе. Также у нас будет приватное поле `data`, которая будет указателем на тип `T`.

Реализуем конструктор. Нам в куче нужно выделить `size` объектов. Пока мы умеем создавать только один объект. Несколько объектов можно создать с помощью оператора `new[]`. Он создает блок памяти для хранения необходимого количества объектов.

```
explicit SimpleVector (size_t size){
    data = new_T[size];
}
```

Сейчас в нашей программе есть утечка памяти. Освободить её следует в деструкторе.

```
~SimpleVector() {
    delete[] data;
}
```

Если вместо `delete[]` пропишем просто `delete`, то программа будет работать некорректно.

### 3.1.7. Введение в арифметику указателей

Добавим в наш вектор возможность обращаться к отдельным элементам. У нас есть указатель `data` на первый элемент вектора. Указатели на другие элементы получаются очень просто: нужно добавить целое число, например, `data + 1` указывает на второй элемент вектора.



Можем написать оператор доступа по индексу.

```
T& operator[] (size_t index) {
    return *(data + index);
}
```

Теперь можно заполнять вектор числами и выводить их на экран.

```
int main() {
    SimpleVector<int> sv(5);
    for (int i = 0; i < 5; ++i) {
        sv[i] = 5 - i;
    }
    for (int i = 0; i < 5; ++i) {
        cout << sv[i] << ' ';
    }
}

// 5 4 3 2 1
```

В нашей программе мы выделили память на 5 элементов. Но если, например, попытаться вывести на экран `sv[12]`, то код скомпилируется и может даже отработать:

```
SimpleVector<int> sv(5);
cout << sv[12] << endl;
// 7827296
```

Язык C++ не контролирует доступ к данным, которые мы осуществляем через указатель. Сейчас мы прочитали элемент, который лежит за границами нашего вектора.

```
SimpleVector<string> sv(5);
cout << sv[12] << endl;
```

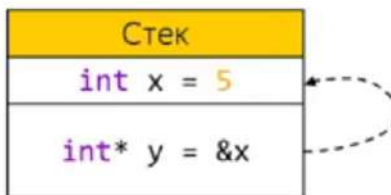
Такая программа упадёт, потому что мы обращаемся к случайному участку памяти, в котором не выполняются инварианты класса `string`.

К указателям можно не только прибавлять целые числа, но и вычитать их.



Локальные переменные хранятся на стеке. Стек – это область оперативной памяти, поэтому у его элементов тоже есть адрес. Его можно получить с помощью оператора `&`:

```
int main() {
    int x = 5;
    int* y = &x;
}
```



Переменная `y` хранит в себе адрес переменной `x`.

```
int main() {
    int x = 5;
    int* y = &x;
    *y = 7;
    cout << x << endl;
```

```

}
// 7

int main() {
    int a = 43;
    int b = 71;
    int c = 89;
    cout << *(&b - 1) << ' ' << *(&b + 1);
}
// 43 89

void f() {
    int a = 43;
    int b = 71;
}

int main() {
    int c = 89;
    for (int i = 0; i < 20; ++i){
        f();
        int x = *(&c - i);
        cout << i << ' ' << x << endl;
    }
}
// 0 89
// ...
// 14 43
// 15 71
// ...

```

Программа выводит 20 строк. В строках 14 и 15 записаны числа 43 и 71. Они совпадают с переменными `a` и `b` нашей программы. Если мы возьмем, например, `a = 434` и `b = 711`, то в строках 14 и 15 окажутся уже 434 и 711 соответственно.

В функции `main` мы запустили функцию `f`. Она запустилась, выделила на стеке фрейм и разместила в нем свои переменные. Мы не знаем точно, где на стеке эти переменные лежат, пробуем разные смещения относительно своей переменной `c`, пытаемся найти стековый фрейм функции `f`. Мы установили, что в четырнадцати `int`'ах от переменной `c` лежит переменная `a`. Мы убедились, что после завершения функции с её стековым фреймом ничего не происходит.

```
int main() {
```

```
int c = 89;
int* d = &c;
int* e = d + 1;
cout << d << endl
      << e << endl;
}
// 0x62fe3c
// 0x62fe40
```

В консоль вывелись два шестнадцатеричных числа. Если мы вычтем одно из другого, то получим 4. Если запустить программу с

```
int* e = d + 3;
```

то получаем 12. Этот пример иллюстрирует то, что при прибавлении к указателю числа целочисленное значение, которое хранится в указателе, изменяется на число, умноженное на размер типа, на который указывает наш указатель.

Перепишем оператор [], используя менее громоздкий синтаксис.

```
T& operator[] (size_t index) {
    return data[index];
}
```

### 3.1.8. Добавляем в вектор `begin` и `end`

Добавим в наш вектор методы `begin` и `end`, чтобы по нему можно было итерироваться в цикле. Цикл `range-based for` разворачивается в следующую конструкцию:

```
SimpleVector<int> v(5);
for (auto i = v.begin(); i != v.end(); ++i) {
}
```

Требования к итераторам в `range-based for`:

- `begin()` указывает на первый элемент;

- `end()` указывает на последний элемент;
- увеличение итератора на один переводит его на следующий элемент.

Всем этим требованиям удовлетворяют указатели на элементы нашего вектора. В качестве `begin` можно использовать указатель на первый элемент нашего вектора, в качестве `end` – указатель на последний элемент. Соответственно, в качестве типа возвращаемого значения для операторов `begin()` и `end()` можем использовать указатель на элемент типа `T`.

В результате класс `SimpleVector` будет выглядеть следующим образом.

```
class SimpleVector {
public:

    explicit SimpleVector (size_t size) {
        data = new T[size];
        end_ = data + size;
    }

    ~SimpleVector() {
        delete[] data;
    }

    T& operator[] (size_t index) {
        return data[index];
    }

    T* begin() { return data; }
    T* end()   { return end_; }

private:
    T* data;
    T* end_;
};
```

Теперь напомним функцию `print` для нашего вектора, которая будет печатать его на консоль.

```
template <typename T>
void Print(const SimpleVector<T>& v) {
    for (const auto& x : v) {
        cout << x << ' ';
    }
}
```

```
}  
}
```

Такая функция работать не будет из-за проблем с константностью. Вектор `v` передается по константной ссылке, поэтому и вызывать мы можем только константные методы объекта `v`. Методы `begin` и `end` константными не являются. Сделаем их константными:

```
T* begin() const { return data; }  
T* end()    const { return end_; }
```

В текущей функции `Print` мы можем случайно поменять наш вектор, например, прописав строку `*i = 42;`

Теперь вызвав функцию `Print` два раза, мы увидим на экране:

```
// 5 3 4 -1  
// 42 42 42 42
```

Чтобы избежать этой проблемы, в методах `begin()` и `end()` нам нужно возвращать указатели на константы:

```
const T* begin() const { return data; }  
const T* end()    const { return end_; }
```

Теперь компилятор будет запрещать изменять вектор, принимаемый по константной ссылке.

Есть еще одна проблема. Теперь мы не можем через возвращаемые итераторы менять наш вектор, например, его сортировать. Нам нужно завести две пары `begin` и `end`: константную и неконстантную.

```
T* begin() { return data; }  
T* end()   { return end_; }  
  
const T* begin() const { return data; }  
const T* end()    const { return end_; }
```

### 3.1.9. Константный указатель и указатель на константу

У константности указателей есть двойственность.





С одной стороны у нас есть указатель, с другой есть объект, на который он указывает. Соответственно мы можем менять сам указатель, а можем этот объект. Когда переменная объявлена как `T* ptr`, то это просто указатель и мы можем менять как объект, так и сам указатель.



Если же перед типом указателя мы пишем ключевое слово `const`, то получаем указатель на константу: мы можем изменять сам этот указатель, но объект мы менять не можем.

Если мы хотим менять объект, но не хотим менять сам указатель, мы можем объявить константный указатель.



Ключевое слово `const` у него идет после `*`. Тогда мы можем менять объект, на который указывает указатель, но не можем менять сам указатель. Если же мы хотим защититься от любых изменений данных, мы можем объявить константный указатель на константу:



### 3.1.10. Итоги раздела

#### Сравнение стека и кучи

	Стек	Куча
Создание объекта	<code>string s;</code>	<code>string* s = new string;</code>
Уничтожение объекта	Автоматически при выходе из области видимости	Вручную: <code>delete s;</code>
Применяется	Для локальных переменных	Для объектов, которые живут дольше, чем создавшая их функция

#### Недостатки использования `new/delete`

- можно забыть вызывать `delete`;
- можно перепутать `delete` и `delete[]`.

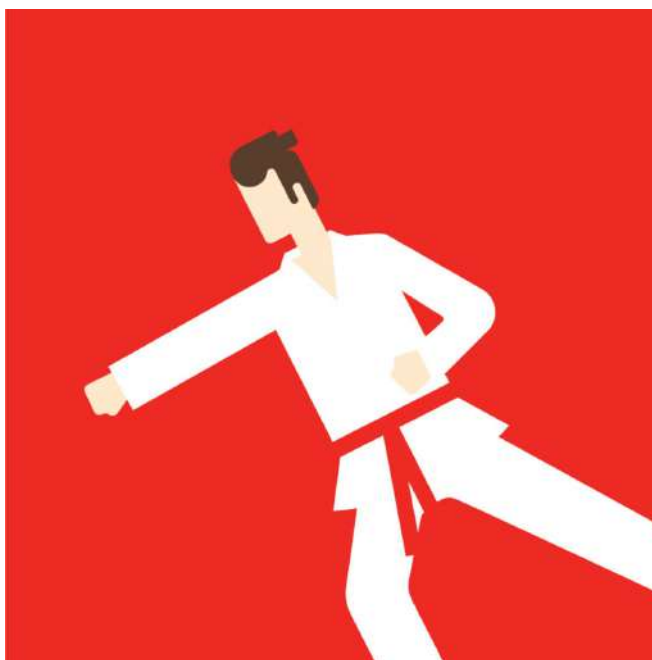
Всё, что связано с операторами `new`, `delete` и арифметикой указателей было рассказано, чтобы потом рассказать про внутреннее устройство контейнеров языка C++. В современном языке C++ нет ни одной причины применять `new` и `delete` в прикладном коде.



# Основы разработки на C++: красный пояс

Неделя 4

Эффективное использование линейных контейнеров



# Оглавление

<b>Эффективное использование линейных контейнеров</b>	<b>2</b>
4.1 Эффективное использование линейных контейнеров . . . . .	2
4.1.1 Эффективное использование вектора . . . . .	2
4.1.2 Инвалидация ссылок . . . . .	5
4.1.3 Эффективное использование дека . . . . .	7
4.1.4 Инвалидация итераторов . . . . .	9
4.1.5 Контейнер <code>list</code> . . . . .	10
4.1.6 Контейнер <code>array</code> . . . . .	14
4.1.7 Класс <code>string_view</code> . . . . .	16

# Эффективное использование линейных контейнеров

## 4.1. Эффективное использование линейных контейнеров

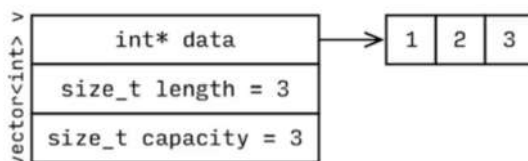
Мы приступаем к исследованию последовательных или линейных контейнеров. Так называют контейнеры, которые сохраняют порядок вставляемых в них элементов. Типичный пример – вектор. Пример непоследовательного контейнера – множество.

### 4.1.1. Эффективное использование вектора

Создадим вектор из трёх чисел:

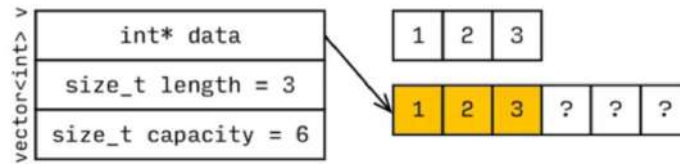
```
vector<int> v = {1, 2, 3};
```

На стеке у этого вектора будет указатель на кучу, также на стеке будет храниться длина этого вектора, а также `capacity` – количество памяти, доступное данному вектору в куче.



Что будет, если мы захотим добавить четвёрку в вектор? Нам нужен блок памяти, в который мы сможем положить 1, 2, 3 и 4. Соответственно, как и в задаче `SimpleVector`, стандартный вектор выделяет в два раза больше памяти, чем у него было до этого.

Он выделяет блок под 6 `int`'ов, чтобы туда можно было положить четыре числа. Теперь вектор присваивает себе блок памяти из шести элементов, про старый забывает, в новый копирует



три `int`'а, которые у него были, и теперь туда же может положить четверку, предварительно освободив старый блок памяти. Теперь длина – 4, `capacity` – 6.

Напишем функцию `LogVectorParams`, которая будет для вектора выводить его параметры: длину и его `capacity`.

```
void LogVectorParams (const vector<int>& v) {
    cout << "Length = " << v.size() << ", " <<
        "capacity = " << v.capacity() << "\n";
}
```

Вызовем функцию после создания вектора и после добавления четвёрки.

```
int main() {
    vector<int> v = {1, 2, 3};
    LogVectorParams(v);
    v.push_back(4);
    LogVectorParams(v);

    return 0;
}
// Length = 3, capacity = 3
// Length = 4, capacity = 6
```

Всё, как мы указали. Мы утверждали, что элементы вектора хранятся подряд. Более того, в `SimpleVector`'е мы оперировали указателями и там был указатель на данные в куче. Оказывается, стандартный вектор тоже может отдать нам этот указатель.

У стандартного вектора есть метод `data`, который возвращает указатель на те данные, которые у него лежат в куче. Вызовем этот метод, сохраним его в указателе. Мы хотим посмотреть на данные, которые лежат по указателю `data` и в следующих ячейках. Вектор константный, поэтому `data` возвращает константный указатель, поэтому переменная `data` должна иметь тип `const int*`.

```
const int* data = v.data();
```

```
for (size_t i = 0; i < v.capacity(); ++i) {
    cout << *(data + i) << " ";
}
cout << "\n";

// Length = 3, capacity = 3
// 1 2 3
// Length = 6, capacity = 6
// 1 2 3 4 78063648 0
```

С помощью указателей мы посмотрели на содержимое вектора: там лежат 1, 2, 3, 4 и два числа, которые когда-то лежали в куче. Вектор их никак не инициализировал, в эту память вектор потом сможет добавить новые числа.

Если мы захотим очистить память от этих двух чисел, то можем использовать `shrink_to_fit()`. Добавим его в программу и посмотрим, что получится.

```
v.shrink_to_fit();
LogVectorParams(v);
// Length = 4, capacity = 4
// 1 2 3 4
```

Рассмотрим пример. Допустим, мы хотим сложить в вектор какие-то числа, заранее зная, сколько этих чисел будет. Подадим на вход число 10000000.

```
int main() {
    int size;
    cin >> size;

    { LOG_DURATION("push_back");
      vector<int> v;
      for (int i = 0; i < size; ++i) {
          v.push_back(i);
      }
    }

    return 0;
}
// push_back: 427 ms
```

Казалось бы, за секунды мы должны успевать больше операций. Такая серия `push_back`'ов неэф-

эффективна за счёт того, что при `push_back`'е вектор при необходимости перевыделяет память и копирует данные из старой памяти в новую. В данном случае мы знаем, что вектор будет иметь размер `size`. Пусть вектор заранее выделит себе блок из `size` элементов.

```
{ LOG_DURATION("push_back");
    vector<int> v;
    v.reserve(size);
    for (int i = 0; i < size; ++i) {
        v.push_back(i);
    }
}
// push_back: 288 ms
```

#### 4.1.2. Инвалидация ссылок

У нас есть вектор из трех элементов, что будет, если перед добавлением четвертого элемента сохранить ссылку на первый элемент?

```
int main() {
    vector<int> v = {1, 2, 3};
    int& first = v[0];
    cout << first << "\n";
    v.push_back(4);
    cout << first << "\n";

    return 0;
}
// 1
// 45628208
```

До `push_back`'а там лежала 1, после `push_back`'а мы знаем, что вектор переместил данные в другое место, а старые данные освободил. Теперь по ссылке лежит мусорное число. Ссылка перестала быть валидной – инвалидировалась.

В материале «Класс `StringSet`» представлена реализация контейнера, в который можно добавлять строки с заданным приоритетом с помощью метода `Add`, а также находить последнюю добавленную строку и строку с наибольшим приоритетом.

В предложенной реализации строки складываются в вектор, также они складываются в `set` с



помощью структур, которые называются `StringItem`. В каждой структуре лежит сама строка вместе с приоритетом. Также есть оператор сравнения, который сравнивает `StringItem`'ы по приоритету. При добавлении строки в контейнер мы кладем её в вектор и в `set`. В методе `FindLast()` мы берем последний элемент вектора, в методе `FindBest()` мы берём последний элемент `set`'а.

Создадим контейнер `strings`, кладем туда строку `"upper"` с высоким приоритетом, затем строку `"lower"` с низким приоритетом. Метод `FindLast()` должен вывести последнюю добавленную строку, то есть `"lower"`, а метод `FindBest()` должен вывести строку с наивысшим приоритетом, то есть `"upper"`.

```
int main() {
    StringSet strings;
    strings.Add("upper", 10);
    strings.Add("lower", 0);
    cout << strings.FindLast() << "\n";
    cout << strings.FindBest() << "\n";
    return 0;
}
// lower
// upper
```

Является ли эффективным складывать одну и ту же строку и в вектор и в множество? Хочется этого избежать и складывать строки в один контейнер. Можно в один контейнер сложить строки, а в другой складывать ссылки на строки в том контейнере. Во множество теперь будем класть не саму строку, а ссылку на неё. В методе `Add` в `sorted_data` вставляем теперь ссылку на строку, которую мы добавили в вектор.

```
sorted_data.Insert(StringItem{data.back(), priority});
```

В результате работы программы на экран выведется только `lower`. `upper` не выведется, потому что произошла инвалидация.

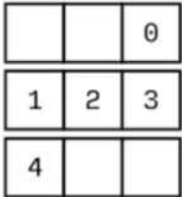
Чтобы решить проблему можно использовать контейнер, который не обладает таким недостатком, как вектор, например, `deque`.

Что делать, если важно, чтобы оставался вектор, а не `deque`? Тогда придется отказаться от ссылок. Например, вместо ссылок можно хранить индексы элементов в векторе.

### 4.1.3. Эффективное использование дека

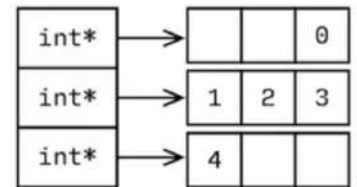
Дек можно получить, отказавшись от требования хранить все элементы подряд в едином куске памяти. Выделим кусок памяти и положим туда три `int`'а.

```
deque<int> d = {1, 2, 3};
```

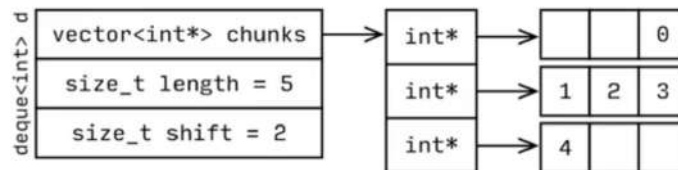


Если мы хотим добавить туда четверку, то не можем добавить её туда сразу после тройки, потому что выделили блок памяти под три `int`'а. Но мы можем выделить ещё один блок памяти на три `int`'а и положить в первую ячейку четверку. Аналогично, если мы хотим добавить в начало ноль, то нужно выделить блок памяти на три `int`'а и в последний элемент, чтобы соблюсти некий разумный порядок, положить ноль.

Нам нужно уметь обращаться к элементу по его номеру. Необходимо эти блоки памяти – они называются чанками – как-то проиндексировать. Сохраним вектор указателей на эти чанки, на эти блоки памяти.



В самом деке нужно хранить вектор указателей на чанки, количество элементов, также понадобится хранить так называемый сдвиг. Сдвиг того самого нуля от начала его чанка, то есть, сдвиг первого элемента дека от начала его блока памяти.



Как теперь найти второй элемент? Прибавляем сдвиг, с учетом сдвига необходим четвертый элемент. Размер блока памяти – 3. Делим 4 на 3, получаем 1, идем в первый чанк памяти и там идем в элемент  $4 \% 3$ , то есть берем остаток от деления и получаем, что в первом блоке памяти нам нужен первый элемент.

В итоге мы получили быструю вставку в начало и неинвалидацию ссылок при вставке в начало или в конец. Однако теперь мы тяжелее получаем элемент по индексу. Итерироваться тоже непросто, потому что нужно перепрыгивать между чанками.

Пусть нам нужно вставить набор чисел в наш контейнер, при этом мы заранее не знаем, сколько будет чисел – то есть не можем вызывать `reserve`.

```
int main() {
    const int SIZE = 1000000;

    vector<int> v;
    { LOG_DURATION("vector");
      for (int i = 0; i < SIZE; ++i) {
          v.push_back(i);
      }
    }

    deque<int> d;
    { LOG_DURATION("deque");
      for (int i = 0; i < SIZE; ++i) {
          d.push_back(i);
      }
    }

    return 0;
}
// vector: 61 ms
// deque: 52 ms
```

Особой разницы нет. Что, если элементов 5000000?

```
// vector: 398 ms
// deque: 168 ms
```

Это говорит о том, что нельзя заранее сказать, что будет полезнее: вектор или дек. Нужно измерять.

Теперь попробуем отсортировать полученные вектор и дек.

```
{ LOG_DURATION("sort vector");
  sort(rbegin(v), rend(v));
}
{ LOG_DURATION("sort deque");
  sort(rbegin(d), rend(d));
}
// sort vector: 5468 ms
```

```
// sort deque: 10851 ms
```

Мы быстрее заполнили дек, но сортировали его гораздо дольше, чем вектор.

Вектор хорош быстрыми обращениями к элементам. Итерирование по вектору тоже быстрое. Дек хорош тем, что можно быстро сделать серию `push_back`'ов, если заранее не известен размер. Кроме того, можно быстро вставлять в начало, при этом не инвалидируя ссылки на элементы дека.

#### 4.1.4. Инвалидация итераторов

Создадим вектор из одного элемента и сохраним итератор на это число. Затем вставим 2000 чисел в вектор. Посмотрим, что будет лежать по этому итератору.

```
int main() {
    vector<int> numbers = {1};
    auto it = begin(numbers);
    cout << *it << "\n";

    for (int i = 0; i < 2000; ++i) {
        numbers.push_back(i);
    }

    cout << *it << "\n";

    return 0;
}
// 1
// 78067936
```

Итераторы вектора по сути являются указателями. Соответственно при вставке в вектор они инвалидируются. Что будет для дека?

```
// 1
// 1
```

Итератор не пострадал. Попробуем с помощью этого итератора посмотреть на последний элемент дека.

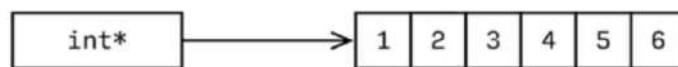
```
cout << *it << " " << *(it + numbers.size() - 1) << "\n";
```

Такая программа падает. Дело в том, что итератор был получен до добавления в дек элементов. Этот итератор знает только про устройство старого дека. Итераторы содержат дополнительную логику, позволяющую итерироваться по контейнеру. Дек не может сохранять итераторы валидными.

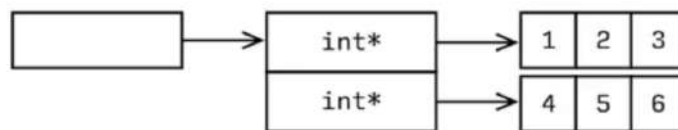
#### 4.1.5. Контейнер list

Мы узнали, что есть различные стратегии выделения памяти под линейные контейнеры: вектор выделяет единый блок памяти под все свои элементы, а дек выделяет память чанками.

```
vector<int> v = {1, 2, 3, 4, 5, 6};
```

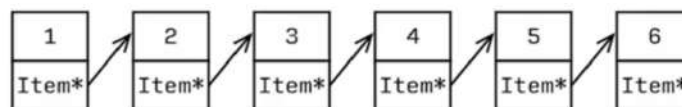


```
deque<int> d = {1, 2, 3, 4, 5, 6};
```

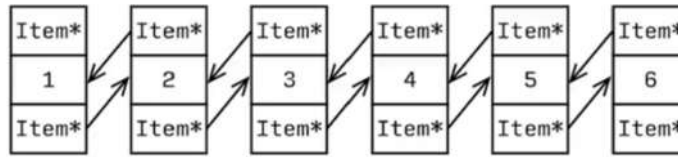


Что, если каждому элементу выделять свой кусок памяти? Такая стратегия используется в контейнере list.

Если каждому элементу контейнера соответствует свой блок памяти, то по такому контейнеру даже проитерироваться нельзя. Можно научить каждый элемент ходить к следующему, то есть рядом с ним положить указатель на следующий элемент.



Теперь мы можем проитерировать по списку из начала в конец. Иногда удобнее проитерироваться в обратную сторону, поэтому стоит положить указатель на обратный элемент.



Теперь мы можем быстро удалять элементы из середины списка. Если мы хотим удалить элемент 4, то мы находим его, удаляем и перевешиваем указатели.

```
list<int> numbers = {1, 2, 3, 4, 5, 6};
auto it = find(begin(numbers),
               end(numbers), 4); // O(N)
numbers.erase(it); // O(1)
```

Все указатели и итераторы, кроме указателя на 4, останутся валидными. Однако при таком устройстве списка мы не можем быстро обратиться к элементу по его номеру.

Рассмотрим пример. Будем складывать их в контейнер с помощью метода `Add`. Удалять элементы будем с помощью метода `Remove`, который будет принимать условие, по которому нужно удалять элементы. Он будет шаблонным и будет принимать функциональный объект.

```
class NumbersOnVector {
public:
    void Add(int x) {
        data.push_back(x);
    };

    template <typename Predicate>
    void Remove(Predicate predicate){
        data.erase(
            remove_if(begin(data), end(data), predicate),
            end(data));
    }
private:
    vector<int> data;
};
```

Давайте реализуем этот контейнер с помощью списка.

```
class NumbersOnList {
public:
    void Add(int x) {
```

```
    data.push_back(x);
};

template <typename Predicate>
void Remove(Predicate predicate){
    data.remove_if(predicate);
}
private:
    list<int> data;
};
```

Напишем бенчмарк, чтобы сравнить два контейнера. Заполним контейнеры миллионом элементов, затем будем удалять их с помощью метода `Remove`: в начале элементы, остаток от деления на 10 у которых равен нулю, затем единице и так далее. В итоге за 10 вызовов метода `Remove` мы удалим все числа.

```
const int SIZE = 1000000;
const int REMOVAL_COUNT = 10;

int main() {
    { LOG_DURATION("vector");
      NumbersOnVector number;
      for (int i = 0; i < SIZE; ++i) {
          numbers.Add(i);
      }
      for (int i = 0; i < REMOVAL_COUNT; ++i) {
          numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
      }
    }

    { LOG_DURATION("list");
      NumbersOnList number;
      for (int i = 0; i < SIZE; ++i) {
          numbers.Add(i);
      }
      for (int i = 0; i < REMOVAL_COUNT; ++i) {
          numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
      }
    }
    return 0;
}
```

```
// vector: 224 ms
// list: 433 ms
```

Мы удаляли по много элементов за раз. Попробуем удалять понемногу. Пусть будет `SIZE = 10000` элементов и мы их попробуем удалить за `REMOVAL_COUNT = 1000` шагов.

```
// vector: 154 ms
// list: 109 ms
```

Теперь большую роль играет тот факт, что из списка можно быстро удалить из середины. Получается, от конфигурации входных параметров зависит, что эффективнее: вектор или список.

На том же примере продемонстрируем, что при удалении элементов из списка у нас неинвалидируются ссылки, указатели и даже итераторы. Дополним класс `NumbersOnList`, который будет находить последний элемент, удовлетворяющий некоторому условию.

```
auto FindLast(Predicate predicate) {
    return find_if(rbegin(data), rend(data), predicate);
}
```

Перед серией удалений из списка мы хотим найти последний элемент, который делится на `REMOVAL_COUNT`. Оставим в контейнере те элементы, которые делятся на `REMOVAL_COUNT`, при удалении мы будем итерироваться, начиная с единицы. В конце посмотрим на элемент, который мы нашли перед серией удалений.

```
{ LOG_DURATION("list");
  NumbersOnList number;
  for (int i = 0; i < SIZE; ++i) {
      numbers.Add(i);
  }
  auto it = numbers.FindLast(
      [](int x) { return x % REMOVAL_COUNT == 0; });
  for (int i = 1; i < REMOVAL_COUNT; ++i) {
      numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
  }
  cout << *it << "\n";
}
// list: 133 ms
// 9000
```

Итератор указывает на верный элемент. Проитерируемся по нему до начала списка.



```
while() (*it != 0) {
    cout << *it << " ";
    ++it;
}
// 9000 8000 7000 6000 5000 4000 3000 2000 1000
```

Проитерироваться получилось. Мы вывели весь список целиком с помощью итератора, который создали еще тогда, когда мы из списка не поудаляли много элементов. Это доказывает, что списки страхуют нас от инвалидации.

#### 4.1.6. Контейнер array

Пусть нам необходимо вызывать функцию COUNT раз. Функция всегда возвращает пять чисел.

```
vector<int> BuildVector(int i) {
    return {i, i + 1, i + 2, i + 3, i + 4};
}

const int COUNT = 1000000;

int main() {
    { LOG_DURATION("vector");
      for (int i = 0; i < COUNT; ++i) {
          auto numbers = BuildVector()
      }
    }

    return 0;
}
// vector: 260 ms
```

Программа будет работать быстрее, если каждый раз выделять память не в куче, а в стеке. Будем использовать кортеж, он хранит свои данные на стеке.

```
tuple<int, int, int, int, int> BuildTuple(int i) {
    return make_tuple(i, i + 1, i + 2, i + 3, i + 4);
}

int main() {
```

```
{ LOG_DURATION("tuple");  
  for (int i = 0; i < COUNT; ++i) {  
    auto numbers = BuildTuple()  
  }  
}  
return 0;  
}  
// tuple: 118 ms
```

Программа работает быстрее, но использование `tuple` влечет неудобства: по нему нельзя нормально проитерироваться, нельзя обратиться к элементу с использованием квадратных скобок. Необходим аналог контейнера `vector`, но на стеке. Можем попросить выделить на стеке пять `int`'ов, используя тип `array`.

```
array<int, 5> BuildArray(int i) {  
  return {i, i + 1, i + 2, i + 3, i + 4};  
}
```

Фрейм функции должен занимать фиксированное количество байт, поэтому прямо в массиве нужно указать, что необходимо 5 `int`'ов. 5 в данном случае – это шаблонный параметр класса. До этого мы сталкивались с классами, у которых шаблонный параметр это тип.

```
{ LOG_DURATION("array");  
  for (int i = 0; i < COUNT; ++i) {  
    auto numbers = BuildArray()  
  }  
}  
// array: 9 ms
```

Почему массив настолько эффективнее, чем кортеж? Дело в том, что мы компилируем без оптимизаций. Если компиляция происходит без оптимизации, код может быть очень не эффективен. Если скомпилировать код, используя оптимизации, то результат работы программы для разных контейнеров будет таким:

```
// vector: 127 ms  
// tuple: 0 ms  
// array: 0 ms
```

Между `array` и `tuple` в данном случае нет разницы.

Сложность алгоритма с использованием каждого контейнера одинакова. Разница во временах

работы программ вытекает из разницы в константе, в накладных расходах, требуемых для каждого контейнера.

Продemonстрируем, что данные массива хранятся на стеке.

```
int main() {
    int x = 111111;
    array<int, 10> numbers;
    numbers.fill(8);
    int y = 222222;

    for (int* p = &y; p <- &x; ++p) {
        cout << *p << " ";
    }
    cout << "\n";

    return 0;
}
// 222222 8 8 8 8 8 8 8 8 8 8 123 0 111111
```

В начале идет переменная `y`, затем идёт десять восьмёрок, затем некоторая служебная информация, и переменная `x`. Действительно, всё лежит на стеке.

#### 4.1.7. Класс `string_view`

Класс `string` похож на `vector`: он позволяет делать `push_back` и `reserve`. Однако семантика строки иногда отличается от семантики вектора.

Рассмотрим как пример задачу из второго курса, где надо было по пробелам строку разбивать на слова.

```
vector<string> SplitIntoWords(const string& str) {
    vector<string> result;

    auto str_begin = begin(str);
    const auto str_end = end(str);

    while (true) {
        auto it = find(str_begin, str_end, ' ');
```

```
result.push_back(string(str_begin, it));

if (it == str_end) {
    break;
} else {
    str_begin = it + 1;
}

return result;
}
```

Функция итерируется по строке, на каждом шаге ищет пробел, создает строчку от текущего итератора до этого пробела, добавляет её в наш набор строк, двигается после этого на позицию, следующую за пробелом. Получается набор строк, разделённых пробелами, то есть слов. У этого кода есть очевидный недостаток. Можно было бы просто сказать, где в этой строке находятся слова, например, с первого до третьего элемента, с пятого до седьмого и так далее. Вместо этого мы стали создавать новые строки, выделять под них память, складывать эти строки в вектор.

Эта функция может возвращать не вектор, а некоторую ссылку на диапазон символов в строке. Для этого в стандарте C++17 доступен класс `string_view` – это ссылка на где-то хранящийся диапазон символов.

Новая функция будет возвращать `string_view`, принимать строчку `s`, внутри себя будет сохранять её в `string_view`. Далее появляется проблема: `string_view` не дружит с итераторами. `string_view` ссылается на подряд идущий диапазон символов. Он работает с позициями в строках, а не с итераторами. Мы начинаем не с итератора, а с нулевой позиции.

```
size_t pos = 0;
```

Аналогом итератора, указывающего за конец, является позиция за концом, её можно проинициализировать константой `str.npos` – это большое число, по сути означает несуществующую позицию.

```
const size_t pos_end = str.npos;
```

Будем искать пробел в `string_view str`, начиная с позиции `pos`.

```
size_t space = str.find(' ', pos);
```

Теперь нужно положить в результат подстроку текущего `string_view`, начиная от позиции `pos` и до пробела. Есть метод `substr`, который возвращает `string_view` на подстроку. Нужно начать подстроку с позиции `pos`, второй элемент – длина этой подстроки. Также нужно учесть случай, когда пробел не нашёлся и `space` равно `npos`.

```
result.push_back(
    space == pos_end
    ? str.substr(pos)
    : str.substr(pos, space - pos));
```

Добавив очередное слово, нужно проверить, если пробел уже не нашёлся, то все слова найдены.

```
if (space == pos_end) {
    break;
} else {
    pos = space + 1;
}
```

Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
    string_view str = s;

    vector<string_view> result;

    size_t pos = 0;
    const size_t pos_end = str.npos;

    while (true) {
        size_t space = str.find(' ', pos);
        result.push_back(
            space == pos_end
            ? str.substr(pos)
            : str.substr(pos, space - pos));

        if (space == str_end) {
            break;
        } else {
            pos = space + 1;
        }
    }
}
```

```
    return result;
}
```

Проверим этот код, сравнив его эффективность со старой функцией `SplitIntoWords`. Для этого заготовлена функция `GenerateText`, генерирующая текст из 10000000 букв "a", разбивающая его пробелами на слова через каждые сто символов.

```
int main() {
    const string text = GenerateText;
    { LOG_DURATION("string");
      const auto words = SplitIntoWords(text);
      cout << words[0] << "\n";
    }

    { LOG_DURATION("string_view");
      const auto words = SplitIntoWordsView(text);
      cout << words[0] << "\n";
    }
}
// *слово, состоящее из ста букв "a"*
// string: 188 ms
// *слово, состоящее из ста букв "a"*
// string_view: 22 ms
```

Сделаем код более компактным. Не будем держать единый `string_view` и ходить по нему позицией, а будем двигать начало `string_view`, когда обрабатываем сколько-то слов. В каждый момент `string_view` будет указывать на диапазон ещё необработанных символов. Если `string_view` указывает на ещё необработанные символы, то можно просто искать первый пробел:

```
size_t space = str.find(" ");
```

Подстроку будем брать не от `pos`, а от нулевой позиции. Если хотим взять подстроку от нулевой позиции до какой-то другой, возможно несуществующей, то можно брать подстроку от нуля до `space`.

```
result.push_back(str.substr(0, space));
```

Если пробел найден, то `space` — это его позиция, а также расстояние до него. Если пробел не найден, то вызываем `substr` от нуля до `pos` и это будет вся строка.

Теперь заканчиваем цикл. Если пробел не найден, то есть его позиция `pos`, то надо закончить

цикл, иначе надо откусить от начала `string_view` уже обработанный кусок длины `space + 1` с помощью специального метода `remove_prefix`.

```
if (space == str.npos) {
    break;
} else {
    str.remove_prefix(space + 1);
}
```

Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
    string_view str = s;

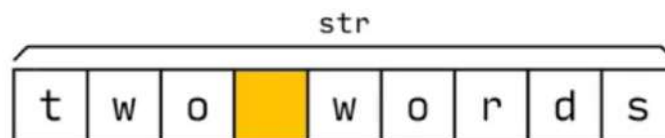
    vector<string_view> result;

    while(true) {
        size_t space = str.find(' ');
        result.push_back(str.substr(0, space));

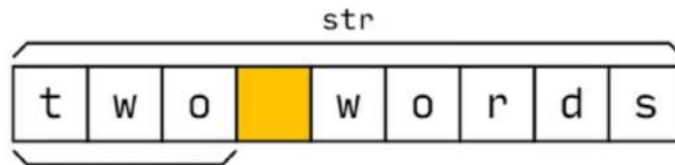
        if (space == str.npos) {
            break;
        } else {
            str.remove_prefix(space + 1);
        }
    }

    return result;
}
```

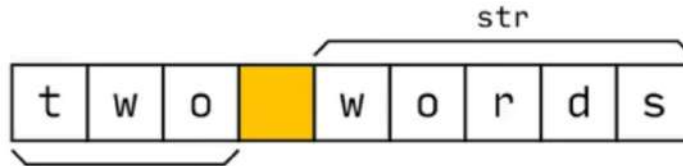
Разберём, как этот код работает. Пусть на вход подается строка "two words".



Мы заходим в цикл, ищем пробел, он находится, его позиция равна трём. Далее вызывается метод `substr` от параметров 0, 3. Он выдаёт строку, которая начинается с нулевой позиции и имеет длину три. Это как раз первое слово.



После этого идёт проверка, не пора ли закончить цикл, откусывается префикс длины 4.



Далее переходим во вторую итерацию цикла, ищем пробел, он не находится. Пробела нет, `space` равно `npos`. Вызывается `substr` от нуля и `npos`, тем самым к ответу добавляется весь текущий `string_view`. Условие `space == str.npos` выполняется, цикл можно закончить.

Вернёмся к первому варианту `SplitIntoWordsView`. Что, если бы мы не создавали `string_view` по строке, а работали бы со строкой, назвав её `str`, вызывая методы `find` и `substr`? Такой код отработает, но не всё однозначно. Если разбить на слова более короткую строчку, например "a b", то в результате получим:

```
// a
// ?
```

Первый вариант функции, который мы использовали ещё во втором курсе, выводит букву `a`. А второй вариант, в котором мы работали со строкой, а не `string_view`, возвращает теперь какой-то символ. Проблема в том, что метод `substr` для строки создаёт новую строчку. Созданные временные объекты могут уничтожиться, если их никуда не сохранить. Получается, мы создали новую строку, сохранили в вектор `string_view` указатель на данные этой строки и тут же эта строка уничтожилась, потому что была временным объектом, то есть указатель `string_view` указывает в никуда. Если `string_view` ссылается на какой-то диапазон символов, следите, чтобы он не уничтожился.

Как сделать правильно и более компактно? Можно сразу в функции принять `string_view` по значению:

```
vector<string_view> SplitIntoWordsView(string_view str) {
    ...
}
```



```
}
```

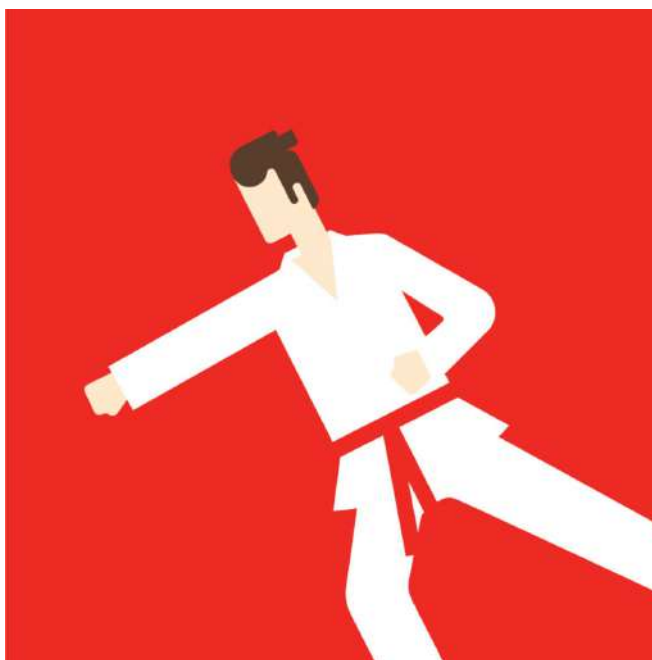
Если вы хотите в функции работать со `string_view`, удобно принять в неё `string_view`, потому что вы не получите проблем с методом `substr`, более того, в функцию можно передавать как строку, так и `string_view`. Этот вариант универсальнее.



# Основы разработки на C++: красный пояс

Неделя 5

Move-семантика и базовая многопоточность



# Оглавление

<b>Move-семантика и базовая многопоточность</b>	<b>3</b>
5.1 Move-семантика . . . . .	3
5.1.1 Перемещение временных объектов. . . . .	3
5.1.2 Перемещение в других ситуациях . . . . .	6
5.1.3 Функция <code>move</code> . . . . .	8
5.1.4 Когда перемещение не помогает . . . . .	11
5.1.5 Конструктор копирования и оператор присваивания . . . . .	13
5.1.6 Конструктор перемещения и перемещающий оператор присваивания . . . . .	15
5.1.7 Передача параметра по значению . . . . .	18
5.1.8 Move-итераторы . . . . .	18
5.1.9 Некопируемые типы . . . . .	20
5.1.10 NRVO и copy elision . . . . .	21
5.1.11 Опасности <code>return</code> . . . . .	22
5.2 Базовая многопоточность . . . . .	23
5.2.1 <code>async</code> и <code>future</code> . . . . .	23

5.2.2	Задача генерации и суммирования матрицы . . . . .	24
5.2.3	Особенности шаблона <code>future</code> . . . . .	26
5.2.4	Состояние гонки . . . . .	27
5.2.5	<code>mutex</code> и <code>lock_guard</code> . . . . .	29
5.2.6	<code>&lt;execution&gt;</code> , которого нет . . . . .	31

# Move-семантика и базовая МНОГОПОТОЧНОСТЬ

## 5.1. Move-семантика

### 5.1.1. Перемещение временных объектов.

Рассмотрим задачу. Пусть есть функция, которая возвращает тяжёлую строку:

```
string MakeString() {  
    return string(100000000, 'a');  
}
```

Пусть необходимо добавить эту строчку в некоторый вектор строк.

```
int main() {  
    vector<string> strings;  
    string heavy_string = MakeString();  
    strings.push_back(heavy_string);  
  
    return 0;  
}
```

Мы хотим положить результат вызова функции `MakeString()` в вектор. Это можно сделать, не используя переменную `heavy_string`.

```
int main() {  
    vector<string> strings;  
    strings.push_back(MakeString());  
  
    return 0;  
}
```

Замерим время работы каждой реализации, используя макрос LOG\_DURATION.

```
// with variable: 299 ms
// without variable: 168 ms
```

Второй вариант работает быстрее, потому что временный объект – результат вызова функции – не сохраняется в переменную.

Исследуем скорость работы алгоритмов, в которых в вектор добавляется временный объект, созданный с помощью конструктора строки.

```
vector<string> strings;
string heavy_string = string(100000000, 'a');
strings.push_back();

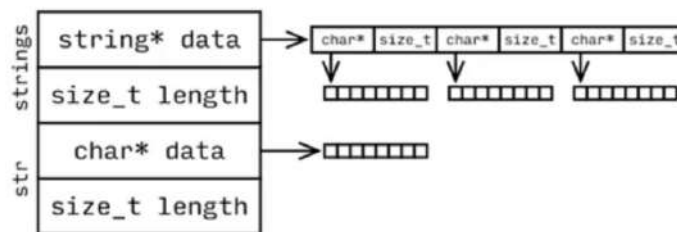
vector<string> strings;
strings.push_back(string(100000000, 'a'));

// ctor: with variable: 201 ms
// ctor: without variable: 122 ms
```

Без использования промежуточной переменной код работает быстрее. Во втором варианте в `push_back` передаётся временный объект, он забирает данные этого объекта, не копируя. Подробнее разберёмся, как это работает.

Рассмотрим процесс добавления в вектор переменной. Вспомним, как хранятся данные вектора и строки в памяти.

```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```

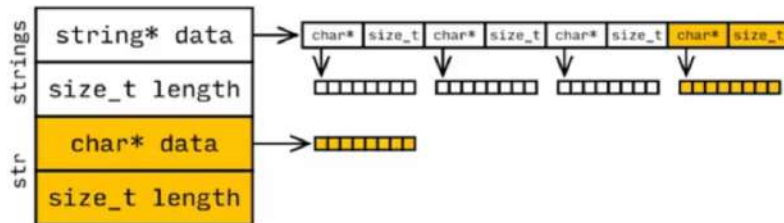


Локальная переменная `strings`, которая является вектором, и локальная переменная `str` со строкой хранятся на стеке. Вектор представляется указателем на свои данные в куче и длиной.

Кроме того, есть локальная переменная со строкой, которая представляет собой указатель на данные в куче и длину. У вектора в куче хранятся строки подряд, где каждая строка – указатель на свои данные и длина.

`push_back`, вызванный от локальной переменной, должен скопировать данные этой строки.

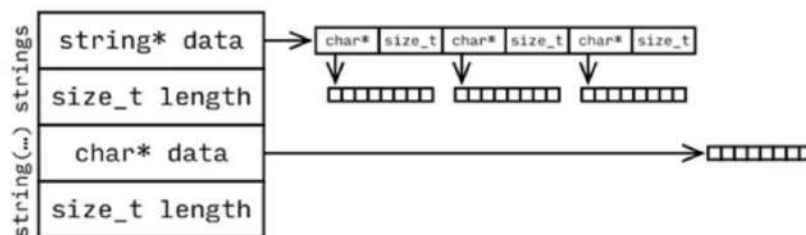
```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```



Выделяем память под восемь символов, в вектор кладём указатель на них.

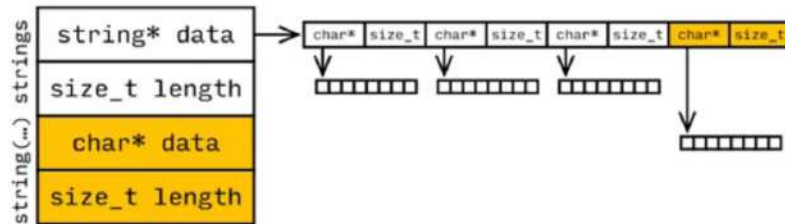
Если мы хотим добавить в вектор временный объект, то его данные в куче, на которые он ссылается, никому не будут нужны. Если их никто не заберёт, то они просто уничтожатся, потому что объект временный.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



`push_back` имеет право эти данные не копировать, а просто свой указатель, который добавится в данные вектора, направить на эти данные, а у той самой локальной временной строки эти данные отобразить.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



### 5.1.2. Перемещение в других ситуациях

Разберём ещё один класс ситуаций, когда у нас не происходит копирование данных временного объекта. В языке C++ везде, где может происходить копирование, имеет место и перемещение. Вспомним, где может происходить копирование объектов. Самый простой случай – это присваивание.

```
string target_string = "old_value";
string source_string = MakeString();
target_string = source_string;
```

В таком варианте здесь будет происходить копирование. Здесь тоже можно убрать промежуточную переменную `source_string` и сэкономить ресурсы на копировании.

```
string target_string = "old_value";
target_string = MakeString();
```

Замерим время работы программы с присваиванием в промежуточную переменную и без промежуточной переменной.

```
// assignment, with variable: 243 ms
// assignment, without variable: 119 ms
```

При использовании метода `set::insert` тоже можно обойтись без промежуточной переменной.

```
// set::insert, with variable
set<string> strings;
string heavy_string = MakeString();
strings.insert(heavy_string);
```



```
// set::insert, without variable
set<string> strings;
strings.insert(MakeString());

// set::insert, with variable: 217 ms
// set::insert, without variable: 100 ms
```

Теперь рассмотрим пример со словарём. Промежуточную переменную создаём и для ключа, и для значения.

```
map<string, string> strings;
string key = MakeString();
strings value = MakeString();
strings[key] = value;
// map::operator[], with variables: 474 ms
```

Избавимся от промежуточной переменной для значения.

```
map<string, string> strings;
string key = MakeString();
strings[key] = MakeString();
// map::operator[], with variable for key: 305 ms
```

Теперь избавимся и от промежуточной переменной для ключа.

```
map<string, string> strings;
strings[MakeString()] = MakeString();
// map::operator[], without variables: 210 ms
```

Напишем функцию MakeVector(), создающую вектор, и попробуем этот вектор куда-нибудь положить.

```
vector<int> MakeVector() {
    return vector<int>(100000000, 0);
}
```

Попробуем положить такой вектор во множество.

```
// set::insert for vector, with variable
set<vector<int>> vectors;
vector<int> heavy_vector = MakeVector();
vectors.insert(heavy_vector);
```

```
// set::insert for vector, without variable
set<vector<int>> vectors;
vectors.insert(MakeVector());

// set::insert for vector, with variable: 1062 ms
// set::insert for vector, without variable: 386 ms
```

Если у вас есть некоторый временный объект, старайтесь не потерять это важное свойство – временность. Так компилятор сэкономит на копировании.

### 5.1.3. Функция move

Есть ситуации, когда есть невременный объект, но про него известно, что там, где он создан, он больше не понадобится.

Есть задача – считать набор строк из потока ввода и эти строки положить в вектор. Это делает функция `ReadStrings()`.

```
vector<string> ReadStrings(istream& stream) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        strings.push_back(s);
    }
    return strings;
}
```

После того как мы кладем строчку `s` в вектор, она нам больше не понадобится. Хочется, чтобы метод `push_back` вел бы себя в этом случае как с временным объектом. Мы хотим изменить семантику объекта.

Есть функция `move`, которая может изменить семантику даже постоянного объекта. Чтобы использовать функцию `move`, нужно подключить модуль `utility`. Цикл теперь выглядит так:

```
while (stream >> s) {
    strings.push_back(move(s));
}
```

Проверим работу функции:

```
int main() {
    for (const string& s : ReadStrings(cin)) {
        cout << s << "\n";
    }
    return 0;
}
// ввод: Red belt C++
// вывод:
// Red
// belt
// C++
```

Разницу между реализациями с `move` и без можно определить не только по времени работы кода, но и по содержанию переменной `s`. Посмотрим, что лежит в переменной `s` до `push_back`'а и после него. Также посмотрим, что лежит в конце вектора.

```
while (stream >> s) {
    cout << "s = " << s << "\n";
    strings.push_back(s);
    cout << "s = " << s <<
        ", strings.back() = " << strings.back() << "\n";
}
// ввод: a b c
// вывод:
// s = a
// s = a, strings.back() = a
// s = b
```

Буква `a` скопировалась. Если добавить вызов `move` в метод `push_back`:

```
// ввод: a b c
// вывод:
// s = a
// s = , strings.back() = a
// s = b
```

Метод `push_back` забрал данные из строки `s`.

Покажем, что код ускоряется. Для этого напомним функцию, которая умеет как перемещать, так и не перемещать.

```
vector<string> ReadStrings(istream& stream, bool use_move) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        if (use_move) {
            strings.push_back(move(s));
        } else {
            strings.push_back(s);
        }
    }
    return strings;
}
```

Воспользуемся функцией `GenerateText()`. Она генерирует текст из миллиарда символов и разбивает его пробелами через каждые десять миллионов символов.

```
int main() {
    const string text = GenerateText();
    { LOG_DURATION("without move");
      istringstream stream(text);
      ReadStrings(stream, false);
    }
    { LOG_DURATION("with move");
      istringstream stream(text);
      ReadStrings(stream, true);
    }
}
// without move: 26359 ms
// with move: 17586 ms
```

Получили существенное ускорение за счет избавления от лишнего копирования.

Этим примером область применения функции `move` не ограничивается. Бывают ситуации, когда результат вызова функции разбиения строки на слова должен жить дольше чем исходная строка. Рассмотрим следующую реализацию функции `SplitIntoWords`:

```
vector<string> SplitIntoWords(const string& text) {
    vector<string> words;
    string current_word;
    for (const char c : text) {
        if (c == ' ') {
```

```
    words.push_back(current_word); // вызываем push_back от переменной, которая нам
    // дальше не нужна
    // логично будет обернуть её в move
    current_word.clear();
} else {
    current_word.push_back(c);
}
}
words.push_back(current_word);
return words;
}
```

#### 5.1.4. Когда перемещение не помогает

Если у объекта нет данных в куче, а основные данные на стеке, то данные придётся копировать. Массив хранит свои данные на стеке. Продемонстрируем на его примере, что перемещение не помогает ускорить работу с ним.

```
const int SIZE = 10000;

array<int, SIZE> MakeArray() {
    array<int, SIZE> a;
    a.fill(8);
    return a;
}
```

Создадим вектор массивов и положим массив в вектор десять тысяч раз с использованием промежуточной переменной и без использования промежуточной переменной.

```
int main() {
    { LOG_DURATION("with variable");
      vector<array<int, SIZE>> arrays;
      for (int i = 0; i < 10000; ++i) {
          auto heavy_array = MakeArray();
          arrays.push_back(heavy_array);
      }
    }
    { LOG_DURATION("without variable");
      vector<array<int, SIZE>> arrays;
```

```
    for (int i = 0; i < 10000; ++i) {
        arrays.push_back(MakeArray());
    }
}

return 0;
}
// with variable: 1191 ms
// without variable: 1148 ms
```

Ускорения не произошло. Рассмотрим другой пример.

```
string MakeString() {
    return "C++";
}

int main() {
    vector<string> strings;
    string s = MakeString();
    cout << s << "\n";
    strings.push_back(s);
    cout << s << "\n";

    return 0;
}
// C++
// C++
```

Теперь сделаем переменную `s` константной:

```
const string s = MakeString();

// C++
// C++
```

Обернём строчку `s` в `move`:

```
strings.push_back(move(s));

// C++
// C++
```

Перемещения строки не произошло. Строка константная, перемещение из неё не работает.

Вызов `move` для константного объекта бесполезен. Следите за константностью перемещаемого объекта.

### 5.1.5. Конструктор копирования и оператор присваивания

Обсудим, что происходит в следующих ситуациях:

```
vector<int> source = /* ... */;  
vector<int> target = source;  
  
vector<int> source2 = /* ... */;  
target = source2;
```

Зачем это необходимо:

- Научиться перемещать собственные классы;
- Разговаривать с разработчиками на одном языке;
- Читать документацию и понимать, где может происходить перемещение;
- Развить интуицию относительно `move`-семантики.

В следующих случаях вызывается конструктор копирования (copy constructor):

```
vector<int> source = /* ... */;  
vector<int> target = source;  
vector<int> target2(source);
```

В следующих случаях вызывается оператор копирующего присваивания (copy assignment operator):

```
vector<int> source = /* ... */;  
vector<int> target = /* ... */;  
target = source;  
target.operator=(source);
```

Познакомимся поближе с этими методами на примере класса `Logger`, который будет логировать вызовы этих методов.

```
class Logger {
public:
    Logger() { cout << "Default ctor\n"; } // конструктор по умолчанию
    Logger(const Logger&) { cout << "Copy ctor"; } // конструктор копирования
    void operator=(const Logger&) { cout << "Copy assignment\n"; } // оператор
    // присваивания
};

int main() {
    Logger source; // вызывается конструктор по умолчанию
    Logger target = source; // вызывается конструктор копирования

    return 0;
}
// Default ctor
// Copy ctor

vector<Logger> loggers;
loggers.push_back(target);
// Copy ctor

source = target; // вызывается оператор присваивания
// Copy assignment
```

Для собственных типов при необходимости компилятор сам генерирует конструктор копирования и оператор присваивания, которые просто копируют все поля.

Для типов, которые самостоятельно управляют памятью, нужно самостоятельно реализовывать копирование и присваивание.

Напишем конструктор копирования для класса `SimpleVector`.

Добавим конструктор от константной ссылки на `SimpleVector`.

```
SimpleVector(const SimpleVector& other);
```

Реализуем этот конструктор:

```
template <typename T>
```



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

### 5.1.6. Конструктор перемещения и перемещающий оператор присваивания

В следующих случаях вызывается конструктор перемещения (move constructor):

```
vector<int> source = /* ... */;
vector<int> target = move(source);
vector<int> target2(move(target));

vector<vector<int>> vectors;
vectors.push_back(vector<int>(5));
```

В следующем случае инициализация временным объектом оптимизируется без вызова конструктора перемещения. Этот случай особый, его обсудим позже.

```
vector<int> MakeVector();

vector<int> target = MakeVector();
```

В следующих случаях вызывается оператор перемещающего присваивания (move assignment operator):

```
vector<int> source = /* ... */;
vector<int> target = /* ... */;
target = move(source);
target = vector<int>(5);
```

Если у вас есть собственный класс с конструктором копирования, но без конструктора перемещения, то компилятор делает перемещение эквивалентным копированию. Рассмотрим пример с классом `SimpleVector`. Скажем компилятору самостоятельно сгенерировать конструктор перемещения.

```
// rvalue reference
SimpleVector(const SimpleVector&& other) = default;
```

`rvalue reference` ведёт себя как обыкновенная ссылка, но позволяет принимать временные объекты.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 1 1
```

Такой код падает в конце, на деструкторах. Придётся самостоятельно реализовывать конструктор перемещения.

```
template <typename T>
SimpleVector<T>::SimpleVector(SimpleVector<T>&& other)
    : data(other.data),
      size(other.size),
      capacity(other.capacity)
{
    other.data = nullptr;
    other.size = other.capacity = 0;
}
// 0 1
```

Теперь у объекта, из которого мы перемещали, размер 0, а у объекта, в который мы перемещали – размер 1.

Реализуем оператор перемещающего присваивания.

```
void operator=(SimpleVector&& other);
```

Он будет реализован примерно как конструктор перемещения.

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

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

Проверим конструктор перемещения.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target(1);
    target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 0 1
```

Перегрузка по rvalue-ссылке – способ отличить временный объект от постоянного.

```
target = source;
// ↑ вызывается operator=(const vector<int>&)
target = vector<int>(5);
// ↑ вызывается operator=(vector<int>&&)
target = move(source);
// ↑ вызывается operator=(vector<int>&&)
```

При необходимости компилятор сам генерирует конструктор перемещения и оператор перемещающего присваивания, которые просто перемещают все поля. Если класс не управляет памятью самостоятельно, то перемещение для него будет просто работать.

### 5.1.7. Передача параметра по значению

Оптимизируем методы `ChangeFirstName` и `ChangeLastName` из класса `Person`, который реализовывался в задаче «Имена и фамилии-4»

```
void ChangeFirstName(int year, const string& first_name) {
    first_names[year] = first_name;
}
void ChangeLastName(int year, const string& last_name) {
    last_names[year] = last_name;
}
```

Необходимо принимать строки по значению, а не по ссылке. Внутри функции будем перемещать строку внутрь словаря.

```
void ChangeFirstName(int year, string first_name) {
    first_names[year] = move(first_name);
}
void ChangeLastName(int year, string last_name) {
    last_names[year] = move(last_name);
}
```

Возможны два случая. Если мы вызываем `ChangeFirstName` от временного объекта, то он проинициализирует переменную `first_name`, поскольку объект временный, то для него вызовется конструктор перемещения. Затем мы снова вызовем перемещение, переместим `first_name` внутрь контейнера. Будет два перемещения.

Если мы вызываем этот метод не от временного объекта, тогда этот объект скопируется в аргумент функции, затем случится перемещение этого объекта внутрь контейнера.

Итого, приняв параметр функции по значению, мы можем сделать его универсальным, вызывать как от временных объектов, так и от постоянных.

### 5.1.8. Move-итераторы

Рассмотрим конструкцию, позволяющую сделать использование `move`-семантики более простым.

Рассмотрим задачу `SplitIntoSentences`, в которой нужно было написать функцию, принимающую набор токенов, разбивающую их на предложения. Рассмотрим саму функцию `SplitIntoSentences`:

```

template <typename Token>
vector<Sentence<Token>> SplitIntoSentences(
    vector<Token>> tokens) {
    vector<Sentence<Token>> sentences;
    auto tokens_begin = begin(tokens);
    while (tokens_begin != tokens_end) {
        const auto sentence_end =
            FindSentenceEnd(tokens_begin, tokens_end);
        Sentence<Token> sentence;
        for (; tokens_begin != sentence_end; ++tokens_begin) {
            sentence.push_back(move(*tokens_begin));
        }
        sentences.push_back(move(sentence));
    }
    return sentences;
}

```

В ней мы заводим итератор `tokens_begin` и идём этим итератором по токенам, находим конец очередного предложения. Берем текущий токен, идем итератором от текущего токена до конца предложения, все эти токены вставляем в текущее предложение. Затем это предложение вставляем в вектор предложений.

Рассмотрим работу цикла `for`. Он перебирает токены в их диапазоне, каждый из них вставляет в вектор.

Подключим модуль `iterator`. После этого станет доступна специальная функция, которую мы вызовем от итераторов. Она называется `make_move_iterator`. Такая функция возвращает обёртку над итератором, которая, если мы хотим скопировать объект, перемещает его. Функция `make_move_iterator` меняет семантику итератора, чтобы при обращении к нему данные перемещались бы, а не копировались.

Используя `make_move_iterator`, можем заменить этот кусок...

```

Sentence<Token> sentence;
for (; tokens_begin != sentence_end; ++tokens_begin) {
    sentence.push_back(move(*tokens_begin));
}
sentences.push_back(move(sentence));

```

...на такой:

```
sentences.push_back(Sentence<Token>(
    make_move_iterator(tokens_begin),
    make_move_iterator(sentence_end)
));
```

Осталось в конце менять итератор `tokens_begin`:

```
tokens_begin = sentence_end;
```

Рассмотрим задачу «Считалка Иосифа». Рассмотрим следующий цикл:

```
for (uint32_t i = 0; i < range_size; ++i, ++range_begin) {
    *range_begin = move(permutation[i]);
}
```

Без цикла это можно записать так:

```
copy(
    make_move_iterator(begin(permutation)), make_move_iterator(end(permutation)),
    range_begin);
```

Алгоритм `move` будет перемещать данные, а не копировать. Это избавит от необходимости вызывать `make_move_iterator`.

```
move(begin(permutation), end(permutation), range_begin);
```

### 5.1.9. Некопируемые типы

Продemonстрируем, как можно сделать тип, который не будет уметь копировать на примере `Logger`'а. Нужно написать `delete` вместо тела конструктора копирования:

```
Logger(const Logger&) = delete;
```

В языке есть такие типы, которые копировать бессмысленно, например, потоки ввода и вывода. Посмотрим, как с ними обращаться на примере вектора потоков.

```
vector<ofstream> streams;
streams.reserve(5);
```

```
for (int i = 0; i < 5; ++i) {
    ofstream stream(to_string(i) + ".txt");
    stream << "File #" << i << "\n";
    streams.push_back(move(stream));
}
for (auto& stream : streams) {
    stream << "Vector is ready!" << endl;
}
```

Откроем, например, файл «0.txt». Там написано:

```
// File #0
// Vector is ready!
```

Получилось работать с файловыми потоками, несмотря на то, что это не копируемые объекты.

### 5.1.10. NRVO и copy elision

Можно оптимизировать копирование объектов, у которых данные только на стеке. Продемонстрируем это на примере Logger'a. Пусть у нас есть функция MakeLogger():

```
Logger MakeLogger() {
    return Logger(); // temporary -> returned temporary
}
```

Временным объектом Logger() мы инициализируем тот промежуточный объект, который должен вернуться из функции. В функции main мы из временного объекта, который вернулся из функции, инициализируем переменную:

```
int main() {
    Logger logger = MakeLogger(); // temporary -> variable

    return 0;
}
// Default ctor
```

Поскольку в обоих случаях новый объект инициализируется временным объектом, происходит перемещение.

В результате работы кода вызывался только конструктор по умолчанию, конструкторы копирования и перемещения не вызвались. Дело в том, что в некоторых случаях компилятор умеет оптимизировать перемещение. Например:

- при возвращении из функции временного объекта;
- при инициализации нового объекта временным объектом.

Такая оптимизация называется *copy elision*.

При возвращении локальной переменной из функции перемещение и копирование опускаются. Такая оптимизация называется *named return value optimization*.

### 5.1.11. Опасности `return`

Рассмотрим два случая, когда `return` оптимизируется не так хорошо, как в случаях из предыдущего пункта.

```
pair<ifstream, ofstream> MakeStreams(const string& prefix) {
    ifstream input(prefix + ".in");
    ofstream output(prefix + ".out");
    return {input, output};
}
```

Код не компилируется, компилятор не может составить пару потоков. `input` и `output` передаются в конструктор пары, затем созданная с помощью этого конструктора пара должна проинициализировать возвращаемый временный объект. Поскольку эта пара тоже временная, то инициализация временного объекта, возвращаемого из функции этой парой, происходит безболезненно, но передача переменных `input` и `output` в конструктор пары происходит по обычным правилам языка C++. Чтобы решить проблему, следует обернуть `input` и `output` в `move`.

Если функция должна вернуть некоторый объект, например поток ввода:

```
ifstream MakeInputStream(const string& prefix) {
    ...
}
```

Внутри функции получили пару объектов, вернуть ходим только один её элемент.



```
ifstream MakeInputStream(const string& prefix) {  
    auto streams = MakeStreams(prefix);  
    return streams.first;  
}
```

`streams.first` не является временным объектом. Также это выражение не является названием локальной переменной – это поле локальной переменной.

Проблема решается, если обернуть `streams` в `move`.

## 5.2. Базовая многопоточность

### 5.2.1. `async` и `future`

Напишем функцию, которая будет суммировать элементы двух векторов. В однопоточной синхронной версии наша функция будет выглядеть так:

```
int SumToVectors(const vector<int>& one,  
    const vector<int>& two) {  
    return accumulate(begin(one), end(one), 0)  
        + accumulate(begin(two), end(two), 0);  
}
```

В начале мы находим сумму элементов одного вектора, потом сумму элементов другого вектора. Мы могли бы один вектор суммировать асинхронно, другой вектор суммировать одновременно с первым, потом сложить результаты. Понадобится заголовочный файл `future`:

```
#include <future>
```

Вызываем функцию `async`, она запускает асинхронную операцию. В данном случае возвращается результат суммирования вектора `one`.

```
future<int> f = async([] {  
    return accumulate(begin(one), end(one), 0);  
});
```

Далее в переменную `result` присваиваем сумму элементов второго вектора.

```
int result = accumulate(begin(two), end(two), 0);
```

Далее возвращаем `result` плюс то, что возвращает `async`. `async` возвращает `future`.

```
return result + f.get();
```

Такой код не компилируется, потому что не захвачена переменная `one`.

```
future<int> f = async([&one] {  
    return accumulate(begin(one), end(one), 0);  
});
```

Когда мы пишем `one` в квадратных скобках лямбды, то происходит копирование внутрь лямбда-функции. Чтобы он не копировался, его следует передавать по ссылке.

### 5.2.2. Задача генерации и суммирования матрицы

У нас есть матрица:

```
vector<vector<int>> matrix;
```

Она генерируется с помощью функции `GenerateSingleThread`:

```
vector<vector<int>> GenerateSingleThread(size_t size) {  
    vector<vector<int>> result(size);  
    GenerateSingleThread(result, 0, size);  
    return result;  
}
```

Функция вызывает другую функцию `GenerateSingleThread`, которая является шаблоном.

```
template <typename ContainerOfVectors>  
void GenerateSingleThread(  
    ContainerOfVectors& result;  
    size_t first_row,  
    size_t column_size  
) {  
    for (auto& row : result) {  
        row.reserve(column_size);
```

```
    for (size_t column = 0; column < column_size; ++column) {  
        row.push_back(first_row ^ column);  
    }  
    ++first_row;  
}  
}
```

Далее матрица обрабатывается с помощью функции `SumSingleThread`.

Запустим программу:

```
int main() {  
    LOG_DURATION("Total");  
    const size_t matrix_size = 7000;  
  
    vector<vector<int>> matrix;  
    {  
        LOG_DURATION("Single thread generation");  
        matrix = GenerateSingleThread(matrix_size);  
    }  
    {  
        LOG_DURATION("Single thread sum");  
        cout << SumSingleThread(matrix) << endl;  
    }  
}  
  
// Single thread generation: 1254 ms  
// Single thread sum: 375 ms  
// 195928050144  
// Total: 1651 ms
```

Мы хотим ускорить программу. Попробуем генерировать матрицу многопоточно. Напишем функцию `GenerateMultiThread`. Она будет принимать `page_size` 0 – желаемый размер страницы, который будет передаваться потоку.

```
vector<vector<int>> GenerateMultiThread(  
    size_t size, size_t page_size  
) {  
    vector<vector<int>> result(size);  
    return result;  
}
```

Мы хотим разбивать вектор `result` на несколько частей. Здесь подходит шаблон `Paginator`.

```
vector<vector<int>> GenerateMultiThread(
    size_t size, size_t page_size
) {
    vector<vector<int>> result(size);
    vector<future<void>> futures;
    size_t first_row = 0;
    for (auto page : Paginator(result, page_size)) {
        futures.push_back(
            async([page, first_row, size] {
                GenerateSingleThread(page, first_row, size);
            })
        );
        first_row += page_size;
    }

    return result;
}
```

Теперь в `main()` будем вызывать многопоточный генератор.

```
{
    LOG_DURATION("Multi thread generation");
    matrix = GenerateMultiThread(matrix_size, 2000);
}

// Single thread generation: 1229 ms
// Multi thread generation: 611 ms
// Single thread sum: 365 ms
// 195928050144
// Total: 2345 ms
```

Многопоточная генерация матрицы оказалась в два раза быстрее, чем однопоточная.

### 5.2.3. Особенности шаблона `future`

Обратим внимание на вектор `futures` из функции `GenerateMultiThread`. Мы его объявили, сложили в него результаты вызова `async` и больше его не вызывали. Сохраняя результаты в вектор,

мы откладываем вызов их деструктора, в котором вызывается `get()`. Если результат вызова функции `async` не сохранить в переменную, то программа может выполняться последовательно.

### 5.2.4. Состояние гонки

Разработаем класс `Account`:

- Он представляет собой банковский счёт;
- Не должен позволять тратить больше денег, чем есть на счету;
- Не должен допускать, чтобы баланс счёта стал отрицательным.

```
struct Account {  
    int balance = 0;  
  
    bool Spend(int value) {  
        if (value <= balance) {  
            balance -= value;  
            return true;  
        }  
        return false;  
    };  
};
```

Напишем функцию, которая будет пытаться 100000 раз потратить один рубль. Она возвращает количество потраченных денег.

```
int SpendMoney (Account& account) {  
    int total_spent = 0;  
    for (int i = 0; i < 100000; ++i) {  
        if (account.Spend(1)) {  
            ++total_spent;  
        }  
    }  
    return total_spent;  
}
```

```
int main() {
    Account my_account(100000);

    cout << "Total spent: " << SpendMoney(my_account)
          << "Balance: " << my_account.balance << endl;
}
// Total spent: 100000
// Balance: 0
```

Пусть теперь несколько людей тратят деньги с семейного счёта асинхронно.

```
int main() {
    Account family_account(100000);

    auto husband = async(SpendMoney, ref(family_account));
    auto wife     = async(SpendMoney, ref(family_account));
    auto son      = async(SpendMoney, ref(family_account));
    auto daughter = async(SpendMoney, ref(family_account));

    int spent = husband.get() + wife.get() + son.get()
               + daughter.get();

    cout << "Total spent: " << spent << endl
          << "Balance: " << family_account.balance << endl;
}
// Total spent: 140573
// Balance: 0
```

Семья потратила больше денег, чем изначально лежало на счёте.

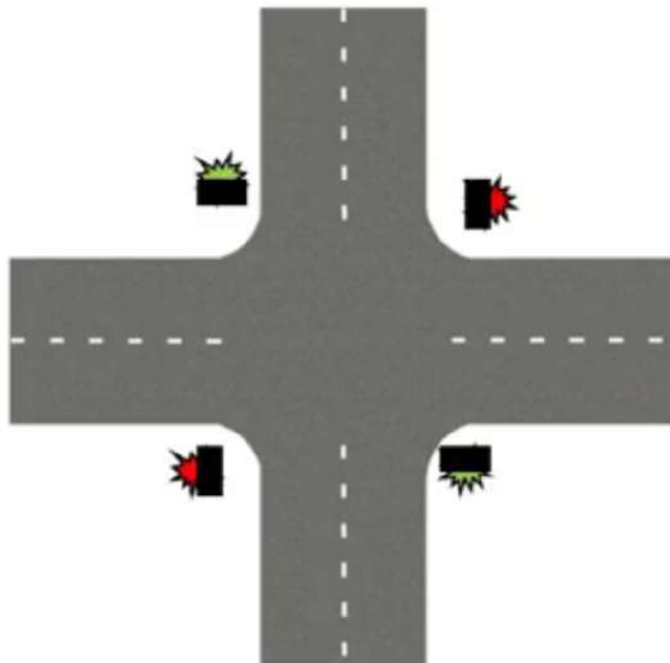
Если несколько потоков обращаются к одной и той же переменной, целостность данных может быть нарушена. Класс `Account` поддерживал инвариант: баланс никогда не становится отрицательным, мы не можем потратить больше, чем есть на счету. Этот инвариант был нарушен при одновременном обращении к данным. В этом заключается гонка данных. Чтобы её избежать, необходимо выполнять синхронизацию доступа к данным из нескольких потоков.

### 5.2.5. mutex и lock\_guard

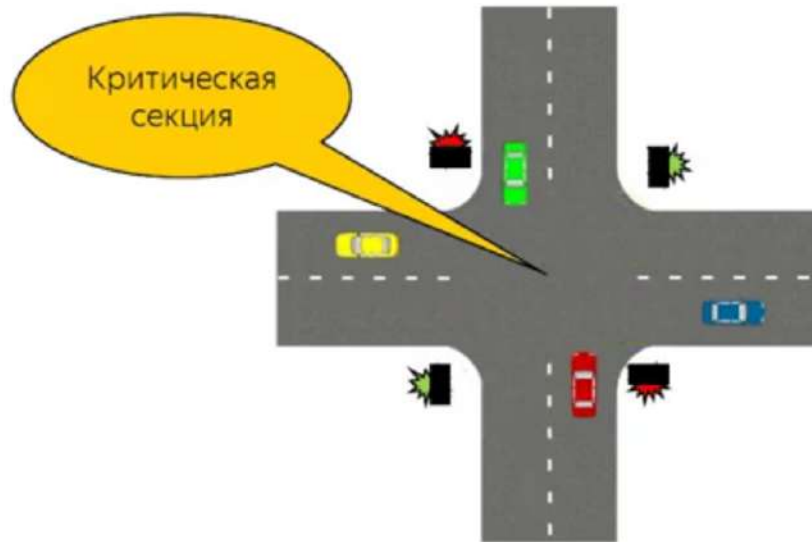
Добавим в наш класс Account `vector<int> history`, который будет сохранять все транзакции.

```
struct Account {  
    int balance = 0;  
    vector<int> history;  
  
    bool Spend(int value) {  
        if (value <= balance) {  
            balance -= value;  
            history.push_back(value);  
            return true;  
        }  
        return false;  
    };  
};
```

Текущий код нужно адаптировать для работы нескольких потоков, используя Mutex (MUTual EXclusion). Простым примером мьютекса является перекрёсток.



Мьютекс защищает критическую секцию. В программировании это тот участок кода, который в любой момент времени может выполнять не более одного потока.



Применим мьютекс в нашей программе. Подключим заголовочный файл `mutex`. В классе `Account` объявим поле

```
mutex m;
```

Критической секцией в нашем коде является метод `Spend`. Его нужно защитить мьютексом.

```
bool Spend(int value) {  
    lock_guard<mutex> g(m);  
    if (value <= balance) {  
        balance -= value;  
        history.push_back(value);  
        return true;  
    }  
    return false;  
};
```

Теперь программа работает правильно:

```
// Total spent: 100000  
// Balance: 0
```



### 5.2.6. <execution>, которого нет

Вернёмся к примеру с генерацией и суммированием элементов матрицы. Изменим реализацию функции `GenerateSingleThread`. Заменим цикл `for` на алгоритм `for_each`. По сути он делает то же самое.

```
void GenerateSingleThread(
    ContainerOfVectors& result;
    size_t first_row,
    size_t column_size
) {
    for_each (
        begin(result),
        end(result),
        [&first_row, column_size] (vector<int>& row) {
            row.reverse(column_size);
            for (size_t column = 0; column < column_size; ++column) {
                row.push_back(first_row ^ column);
            }
            ++first_row;
        }
    );
}
```

В стандарте C++17 были введены параллельные версии стандартных алгоритмов. Чтобы получить параллельную версию алгоритма `for_each` достаточно подключить заголовочный файл `execution` и в качестве первого параметра в функции указать `execution::par`.

```
for_each (execution::par, ...)
```

Ни один из ведущих компиляторов на момент записи видео (апрель 2018) не реализовал поддержку параллельных версий алгоритмов. Сейчас этим воспользоваться нельзя.

# Основы разработки на C++. Эффективное использование ассоциативных контейнеров

# Оглавление

<b>Эффективное использование ассоциативных контейнеров</b>	<b>2</b>
1.1 Введение в ассоциативные контейнеры . . . . .	2
1.2 Размен отсортированности на производительность . . . . .	4
1.3 Внутреннее устройство ассоциативных контейнеров . . . . .	6
1.4 Внутреннее устройство <code>unordered_map</code> и <code>unordered_set</code> . . . . .	8
1.5 Внутреннее устройство <code>map</code> и <code>set</code> . . . . .	9
1.6 Итераторы в <code>map</code> . Почему лучше использовать собственные методы для поиска . .	10
1.7 Итераторы в <code>unordered_map</code> . Инвалидация итераторов в ассоциативных контейнерах	14
1.8 Использование пользовательских типов в ассоциативных контейнерах . . . . .	15
1.9 Зависимость производительности от хеш-функции . . . . .	18
1.10 Рекомендации по выбору хеш-функции . . . . .	21
1.11 <code>map::extract</code> и <code>map::merge</code> . . . . .	22
1.12 Коротко о главном . . . . .	26

# 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 работала 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 в 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 удлиняться и удлиняться 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, попадают в его левое поддерево, а все значения 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 наткнется 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 поддерева 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  $h \sim \log_2 N$
- $\text{find} \sim \log_2 N$ ;  $\text{insert} \sim \log_2 N$ ;  $\text{delete} \sim \log_2 N$ ;

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 А 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 корзинам, 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 `srreference` 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 сравнения, 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 используя 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 А, 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 корзи́н, 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 А, сейчас начинается на а, 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 не копируемых 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

<b>Пространства имён</b>	<b>2</b>
2.1 Знакомство с учебным примером . . . . .	2
2.2 Проблема пересечения имён двух разных библиотек . . . . .	4
2.3 Знакомство с пространствами имён . . . . .	6
2.4 Особенности синтаксиса пространств имён . . . . .	8
2.5 Using-декларация . . . . .	10
2.6 Директива using namespace . . . . .	11
2.7 Глобальное пространство имён . . . . .	11
2.8 Using namespace в заголовочных файлах . . . . .	12
2.9 Пространство имён std . . . . .	15
2.10 Структурирование кода с использованием пространств имён . . . . .	17
2.11 Рекомендации по использованию пространств имён . . . . .	19
<b>Указатель this</b>	<b>21</b>
3.1 Присваивание объекта самому себе . . . . .	21
3.2 Знакомство с this . . . . .	22
3.3 Ссылка на себя . . . . .	24
3.4 this как неявный параметр методов класса . . . . .	26

# 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 `CalculateTotalSpending`s, 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;
    };
    return 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.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 " << CalculateTotalSpending(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 напомним `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.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.

Продemonстрируем 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.cpp`. 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 имён». 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 изменении 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 строку 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 condition_str;
    getline(cin, condition_str);

    istringstream in(condition_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 файле 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 эта 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 упростить 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 `сpp`-файлах, при определенных условиях, это директива отлично работает и упрощает написание кода, поэтому ее можно использовать в `сpp`-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 напомним

```
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 указателям 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

<b>Константность как элемент проектирования программ</b>	<b>2</b>
1.1 Введение	2
1.2 <code>const</code> защищает от случайного изменения	5
1.3 Использование <code>const</code> для поддержания инвариантов в классах и объектах	7
1.4 Идиома <code>immediately invoked lambda expression (IILE)</code>	10
1.5 Константные объекты в многопоточных программах	12
1.6 Логическая константность и <code>mutable</code>	13
1.7 Ещё раз о константности в многопоточной среде	14
1.8 Рекомендации по использованию <code>const</code>	17
<b>Умные указатели. Часть 1</b>	<b>20</b>
2.1 Введение	20
2.2 Обнаружение утечки памяти в <code>ObjectPool</code>	20
2.3 Откуда берётся утечка памяти?	24
2.4 Умный указатель <code>unique_ptr</code>	27
2.5 <code>unique_ptr</code> для исправления утечки	31
<b>Разбор задачи «Дерево выражения»</b>	<b>36</b>
3.1 Разбор задачи «Дерево выражения»	36

# Константность как элемент проектирования программ

## 1.1 Введение

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
if (!cache || cache->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.

```
cache.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 неконстантную 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 константном 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 пройти по всем функциям, которые принимали нашу базу данных по константной ссылке и вызывали метод `RangeByKarma` — и у них тоже убрать константность.

А я напомним, 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 константность 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(dst), 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 встречается 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 отсортированный 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 отсортированными 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 напомним, 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 констатный 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 констатная 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 констатности в C++. **Констатность гарантирует потокобезопасность.** Констатному объекту нет необходимости осуществлять синхронизацию доступа.

## 1.6 Логическая констатность и mutable

Итак, мы с вами увидели, что констатность 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 констатность.

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 констатность, что привело к «веерному» удалению констатности из кода, который использовал наш класс `Database`, и мы задались вопросом: а есть ли смысл вообще использовать констатность C++, когда она приводит к таким «веерным» изменениям. Теперь мы с вами знаем, что констатность крайне важна и крайне полезна, и поэтому, если у вас есть констатный метод, и вы вдруг понимаете, что вы хотите убрать у него констатность, 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 констатность сохранить.

И для этого давайте поглубже разберемся с тем, а какие же гарантии дают нам констатные методы? Что вообще это с точки зрения языка означает, констатный метод? В «белом поясе по C++?» мы говорили, что констатный метод не имеет право менять текущий объект. На самом деле это формулировка не совсем правильная, и с точки зрения C++ констатность 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 `GetById`
- 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 словом можно изменять в контактных методах.

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

И теперь мы можем вернуть константность методу `RangeByKarma`. Кроме того, мы в начале этого модуля удаляли константность из функций, 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 sарatov = "Saratov";
cout << sарatov << ' ' << city_population.Get().at("Saratov");
```

Давайте посмотрим внутри 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, хотя бы `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 **ШЕ** 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 необходимости 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 напомним некоторую те-

стирующую программу.

```
#include "ObjectPool.h"

#include <iostream>

using namespace std;

void run() {
    ObjectPool<char> pool;
    for (int i = 0; i < 1000; ++i) {
        cout << "Allocating object #" << i << endl;
        pool.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 запускаться 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;
            pool.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 создан. 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 создан, 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` указывали 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 экономит 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`. Заменяем наш `set` на `unordered_map` и также подключаем 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 динамический 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»

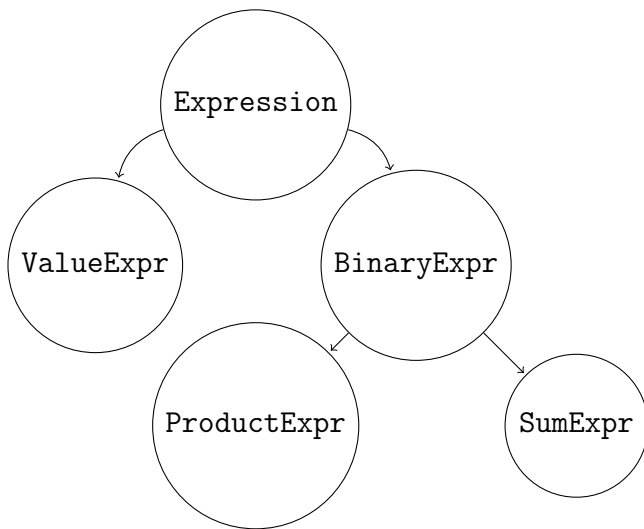
## 3.1 Разбор задачи «Дерево выражения»

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.



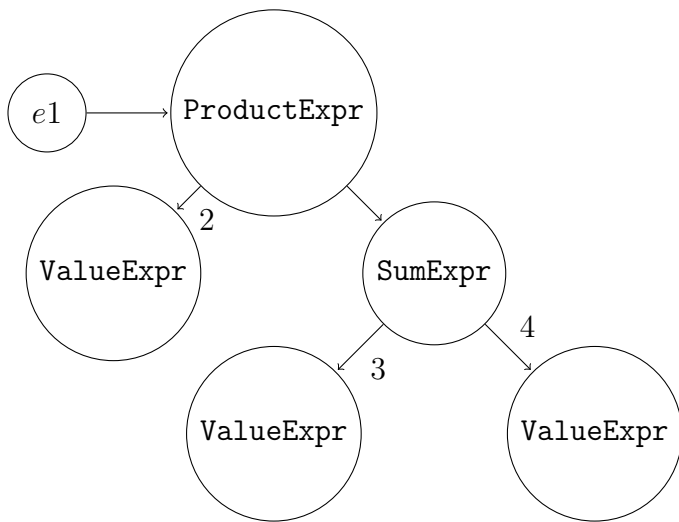
Давайте теперь посмотрим подробнее, как у нас происходит создание дерева выражения, например, вот для такого примера кода

```
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. Далее `Value(4)` аналогичным образом создаст `ValueExpr`, в котором сохраняется четверка, и вернет временный `unique_ptr` на этот объект.
2. Далее 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. Далее `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.

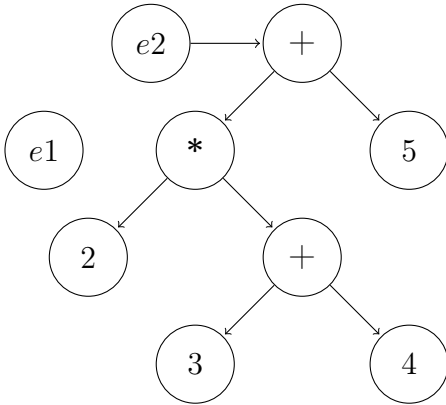
1. Разрушение дерева выражения начинается с того, что локальная переменная `e1` выходит из `scope` и начинает разрушаться. Начинает работать деструктор `unique_ptr`, он смотрит: так, есть ли какой-то объект, на который я указываю? Да, есть объект, `ProductExpr`. Замечательно. Вызывается `delete` для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



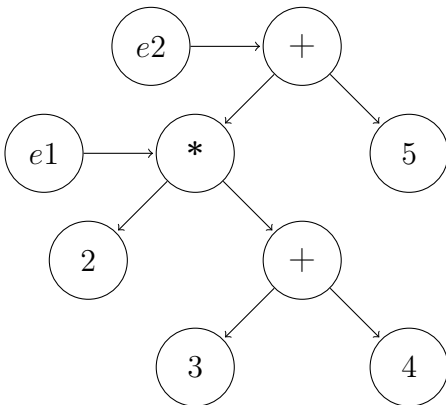
Давайте теперь на сокращенной записи посмотрим, как у нас будет создаваться дерево выражения для `e2`, которое не просто конструирует выражение с нуля, а берет уже существующее.

```
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

<b>Умные указатели. Часть 2</b>	<b>2</b>
1.1 Умный указатель <code>shared_ptr</code>	2
1.2 <code>shared_ptr</code> в дереве выражения	3
1.3 Внутреннее устройство умных указателей	6
1.4 Владение	9
1.5 Присваивание умных указателей	15
1.6 <code>shared_ptr</code> и многопоточность	18
1.7 Умный указатель <code>weak_ptr</code>	21
1.8 Пользовательский <code>deleter</code>	23
<b>Идиома RAII</b>	<b>26</b>
2.1 Знакомство с редактором <code>vim</code> и консольным компилятором	26
2.2 Жизненный цикл объекта	26
2.3 Идея RAII	29
2.4 RAII-обёртка над файлом	30
2.5 Копирование и перемещение RAII-обёрток	32
2.6 RAII вокруг нас	33
<b>Разбор задачи</b>	<b>36</b>
3.1 Разбор задачи с использованием идиомы RAII	36

# 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 отработывают, 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 решении 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.

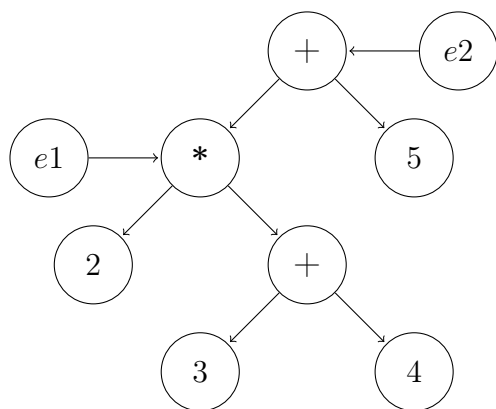
Но вот уберем этот блок. А теперь смотрите, на самом деле мы можем делать даже более интересные вещи. Например, давайте заведем такой `ExpressionPtr e3`, который будет равен сумме `e1` и `e2`, и напечатаем уже вот такой `e3`.

```
...
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.



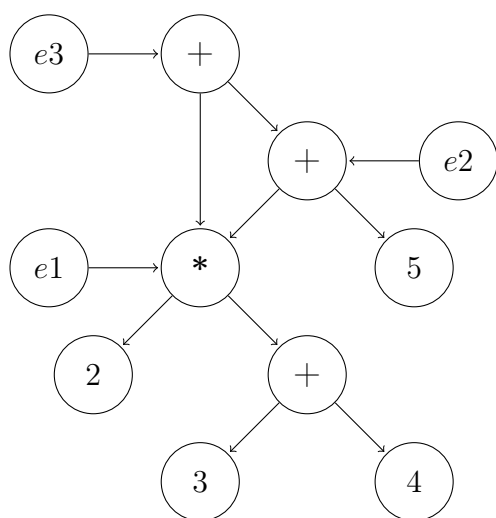
Теперь давайте посмотрим на то, как у нас происходит удаление `e2`.

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 создан. 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 создан 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 переменными 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 автоматическими 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 `unique_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 достаём из него сырой указатель и подаём 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 примере, 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 напомним

```
...
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 напомним 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 распарен 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 работать.

```
...
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()` устареваает 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 `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 участвовать 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_ptr<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 котором 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 напомним 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 скомпилированный результат. Пусть это будет файл `hello.out`. Выполняем команду, и если вы не увидели на экране никаких сообщений — это значит, что программа скомпилировалась 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 <iostream>

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 выключаться. 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 unsuccessfully, 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 напомним, 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 функции 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 симитируем 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. Поэтому



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

<b>Зачем нужны функции?</b>	<b>2</b>
1.1 Зачем нужны функции? . . . . .	2
1.2 Функции или методы классов? . . . . .	4
1.3 Какими должны быть функции? . . . . .	6
1.4 Философия понятного кода . . . . .	9
<b>Детали проектирования функций</b>	<b>10</b>
2.1 Как передать объект в функцию . . . . .	10
2.2 Как передать в функцию набор объектов . . . . .	11
2.3 Как вернуть объект из функции . . . . .	13
2.4 Как вернуть несколько объектов из функции . . . . .	15
2.5 Возврат данных через исключения . . . . .	16
<b>Вызовы конструкторов</b>	<b>18</b>
3.1 Понятность вызовов конструкторов . . . . .	18
3.2 Как рефакторить конструкторы с непонятными сигнатурами . . . . .	19

# 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 класс 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 сигатурой — 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 неконстантной 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 констатная 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 глобальные 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 считала 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 строку.

```
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 ссылке 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_assign(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 даруют 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 elision`, 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 отсеаем 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::PerfromPersonTasks(...) {
    ...
    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 бойтесь 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 ситуаций 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 вызывать 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.