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 рассмотренно, как определять функции в C++. Для начала, рассмотрим преимущества разбиения кода на функции:

- Программу, код которой разбит на функции, проще понять.
- Правильно подобранное 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 считает 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.

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

using namespace std;

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

int main() {
  cout << Contains({"air", "water", "fire"},
   ↪  "water") << endl; // 1
  return 0;
}
```

№	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.

```cpp
??? 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 << t << 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 << endl << t << endl;
```

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 << endl << t << endl;
```

```
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 строчки.

```
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, напишем 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 создался 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
```

Продемонстрируем, 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 gcc.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 gcc 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 : n) {
                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 масимума 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. Третим 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 условием

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». Непостредственно 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;
}
```

3.3. 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();
}
```

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 тяжела: 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 вызваются 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.

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 напишем 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, который 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
```

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, использовав 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 (≫) 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 прочитать 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;
```

```
    }
  }
}
```

Предыдущаяя программа примет вид:

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

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

ReadAll(path);
```

Следует отметить, что при каждом запустке 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 Задает 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 напишем 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);
```