1.4. Условный оператор и циклы

1.4.1. Условный оператор if

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

Синтаксис оператора if в простейшем случае имеет вид:

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

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

```
int a, b;

cin >> a >> b;

if (a == b)
  cout << "equal";
```

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

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

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

В результате, программу можно дополнить следующим образом.

```
int a, b;

cin >> a >> b;

if (a == b)
  cout << "equal" << endl;
else
  cout << "not equal" << endl;
```

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

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

```
if ( <условие> ) {
  ...
}
```

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

```
int a, b;

cin >> a >> b;

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

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

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

```
int a = -1;

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

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

Если явно расставить скобки, получится:

```
int a = -1;

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

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

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

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

1.5. Цикл while

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

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

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

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

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

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

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

```
int a = 5;
int b;

do {
  cout << "Guess the number: ";
  cin >> b;
} while (a != b);

cout << "You are right!";
```

1.6. Цикл for

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

vector

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

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

cout << sum;
```

map

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

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

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

string

```
string a = "asdfasdfasdf";

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

Простой цикл for позволяет создавать цикл с индексом:

```
string a = "asdfasdfasdf";

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

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

```
string a = "sdfasdfasdf";

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

cout << "Yes";
```

```
3
Yes
```

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.

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.

3.3. 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();
}
```