

1.4. Условный оператор и циклы

1.4.1. Условный оператор if

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

Синтаксис оператора `if` в простейшем случае имеет вид:

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

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

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

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

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

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

В результате, программу можно дополнить следующим образом.

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

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

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

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

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

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

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

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

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

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

Если явно расставить скобки, получится:

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

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

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

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

1.5. Цикл while

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

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

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

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

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

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

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

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

1.6. Цикл for

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

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

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

```
    cout << sum;
```

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

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

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

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

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

Простой цикл for позволяет создавать цикл с индексом:

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

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

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

```
string a = "sdfasdfasdf";

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

cout << "Yes";
```

```
3
Yes
```

Неделя 2

Функции и контейнеры

2.1. Функции

2.1.1. Объявление функции. Возвращаемое значение.

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

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

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

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

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

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

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

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

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

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

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

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

Результат работы функции для нескольких случаев:

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

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

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

```

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

```

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

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

Ключевое слово `void`

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

```

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

```

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

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


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

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

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

2.1.2. Передача параметров по значению

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

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

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

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

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

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

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

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

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

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

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

2.1.3. Передача параметров по ссылке

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

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

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

Определим две переменные, `a` и `b`:

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

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

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

Выведем на экран значения переменных:

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

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

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

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

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

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

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

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

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

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

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

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

Без использования ссылок такая функция выглядела бы примерно так:

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

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

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

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

Перепишем функцию, используя передачу параметра по ссылке:

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

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

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

Именно это и хотелось получить.

2.1.4. Передача параметров по константной ссылке

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

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

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

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

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

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

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

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

Воспользуемся этой функцией:

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

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

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

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

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

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

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

В результате получаем:

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

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

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

Избежать копирования можно с помощью передачи параметров по ссылке:

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

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

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

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

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

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

- В случае, если промежуточная переменная не создается:

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

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

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

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

Определение функции принимает вид:

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

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

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

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

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

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

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

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

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

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

2.2. Модификатор `const` как защита от случайных изменений

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

Например, рассмотрим следующий код:

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

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

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

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


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

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

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

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

Рассмотрим пример со строковой переменной `s`:

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

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

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

Теперь добавляем модификатор `const`.

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

При компиляции только в одном месте выводится ошибка:

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

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

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

```
5
hello
hello!
```

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

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

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

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

В итоге компиляция завершается ошибкой:

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

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

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

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

2.3. Контейнеры

2.3.1. Контейнер vector

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

Создание вектора требуемой длины. Ввод и вывод с консоли

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

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

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

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

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

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

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

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

```
PrintVector(v);
```

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

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

Запустим программу и проверим, что она работает:

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

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

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

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

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

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

```
int i = 0;
```

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

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

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

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

```
PrintVector(v);
```

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

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

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

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

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

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

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

Задание элементов вектора при его создании

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

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

Такой вектор можно распечатать:

```
PrintVector(days_in_months);
```

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

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

Запустим программу, убеждаемся, что она работает как надо.

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

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

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

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

Создание вектора, заполненного значением по умолчанию

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

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

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

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

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

Вывести вектор в консоль можно с помощью функции PrintVector:

```
PrintVector(is_holiday);
```

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

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

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

Изменение длины вектора

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

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

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

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

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

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

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

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

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

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

```
PrintVector(is_holiday);
```

Очистить вектор можно с помощью метода `clear`:

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

2.3.2. Контейнер `map`

Создание словаря. Добавление элементов

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

```
#include <map>
```

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

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

Напишем функцию, которая позволяет вывести словарь на экран:

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

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

Итерирование по элементам словаря

Выведем получившийся словарь на экран с помощью написанной функции:

```
PrintMap(events);
```

На экран будут выведены три элемента:

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

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

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

Обращение по ключу к элементам словаря

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

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

```
Bjarne Stroustrup's birth
```

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

Удаление по ключу элементов словаря

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

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

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

Построение «обратного» словаря

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

Для построения такого словаря, напомним функцию `BuildReversedMap`:

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

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

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

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

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

Теперь можно запустить следующий код:

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

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

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

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

```
1950
```

Создание словаря по заранее известным данным

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

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

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

```
PrintMap(m);
```

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

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

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

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

В таком случае получается избежать лишнего копирования элементов словаря.

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

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

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

Подсчет количества различных элементов последовательности

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

Допустим, дана последовательность слов:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Группировка слов по первой букве

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

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

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

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

```
o
one
t
two three
```

Стандарт C++17

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

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

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

```
#include <map>

using namespace std;

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

    return 0;
}
```

В данном коде имеются следующие проблемы:

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

В новом стандарте появляется возможность писать такой код более понятно:

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

2.3.3. Контейнер set

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

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

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

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

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

Создание множества. Добавление элементов.

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

```
#include <set>
```

Теперь можно создать множество известных людей:

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

Добавить в это множество элементы можно с помощью метода insert:

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

Печать элементов множества

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

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

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

Теперь можно вывести на экран элементы множества известных людей:

```
PrintSet(famous_persons);
```

```
Size = 2  
Ritchie  
Stroustrup
```

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

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

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

```
Size = 2  
Ritchie  
Stroustrup
```

Удаление элемента

Удаление из множества производится с помощью метода erase:

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



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

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

```
Size = 2
Ritchie
Stroustrup
```

Создание множества с известными значениями

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

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

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

Сравнение множеств

Как и другие контейнеры, множества можно сравнивать:

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

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

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

Проверка принадлежности элемента множеству

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

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

Создание множества по вектору

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

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

Size = 2

a

b

С помощью аналогичного синтаксиса можно создать и вектор по множеству.

Неделя 3

Переменные в C++. Пользовательские типы данных

3.1. Алгоритмы. Лямбда-выражения

3.1.1. Вычисление минимума и максимума

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

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

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

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

Проверим, как эти функции работают:

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

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

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

Для работы со стандартными алгоритмами нужно подключить заголовочный файл:

```
#include <algorithm>
```

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

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

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

Точно также можно искать минимум и максимум двух строк:

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

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

3.1.2. Сортировка

Пусть необходимо отсортировать вектор целых чисел:

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

Для удобства определим функцию, выводящую значения вектора в консоль:

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

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

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

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

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

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

После этого распечатаем вектор с меткой «sort»:

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

Результат работы программы:

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

Программа работает так, как и ожидалось.

3.1.3. Подсчет количества вхождений конкретного элемента

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

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

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

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

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

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

3.1.4. Подсчет количества элементов, которые удовлетворяют некоторому условию

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

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

Теперь эту функцию можно передать в `count_if`:

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

По аналогии можно определить функцию «меньше двух»:

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

Которую также можно использовать в `count_if`:

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

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

3.1.5. Лямбда-выражения

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

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

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

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

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

Если попытаться воспользоваться этой переменной в лямбда-выражении:

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

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

```

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

```

3.1.6. Mutable range-based for

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

```

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

```

Вывод вектора на экран производится в функции Print:

```

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

```

Для этого можно воспользоваться обычным циклом for:

```

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

```

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

```

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

```

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


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

3.2. Видимость и инициализация переменных

3.2.1. Видимость переменной

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

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

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

Другой пример:

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

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

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

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

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

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

То же самое имеет место для цикла `while`:

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

И для цикла `for`:

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

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

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

Оказывается, что она также не видна.

Еще один пример:

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

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

```
world  
hello
```

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

3.2.2. Инициализация переменной

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

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

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

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

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

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

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

```
0  
3.14
```

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

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

В результате работы программы:

```
0  
3.14  
1074339512
```

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

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

¹«Не платить за то, что не используется.»

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

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

3.2.3. Инициализация переменной при объявлении: примеры

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

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

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

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

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

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

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

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

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

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

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

Для этого можно вынести часть кода в отдельную функцию:

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

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

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

3.3. Структуры. Классы

Структуры

3.3.1. Зачем нужны структуры?

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

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

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

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

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

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

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

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

```

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

```

Функцию PrintCourse также нужно модифицировать:

```

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

```

Основные недостатки представленного подхода:

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

3.3.2. Структуры

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

```

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

```


Синтаксис объявления полей похож на синтаксис объявления переменных.

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

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

Здесь лекция передается по ссылке, чтобы избежать копирования.

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

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

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

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

3.3.3. Создание структур

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

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

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

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

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

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

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

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

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

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

3.3.4. Вложенные структуры

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

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

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

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

Создать вложенную структуру можно используя уже известный синтаксис:

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

Этот код можно записать короче и без использования временной переменной:

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

Обращаться к внутренним полям можно ожидаемым образом:

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

3.3.5. Область видимости типа

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

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

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

Классы

3.3.6. Приватная секция

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

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

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

Кроме того, пусть дана функция для расчета длины пути.

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

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

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

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

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

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

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

Таким образом можно объявить приватную секцию:

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

Теперь к данным полям нет доступа снаружи класса:

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

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

3.3.7. Методы

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

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

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

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

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

Теперь созданные методы можно использовать следующим образом:

```
Route route;

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

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

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

Отличия методов от функций:

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

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

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

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

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

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

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

3.3.8. Контроль консистентности

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

SetSource — позволяет изменить начало маршрута.

SetDestination — позволяет изменить пункт назначения.

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

В итоге код класса будет выглядеть следующим образом:

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

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

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

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

Итак, смысловая связь между полями класса контролируется в методах.

3.3.9. Константные методы

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

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

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

Создадим маршрут и вызовем эту функцию:

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

При попытке запуска кода появляется ошибка.

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

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

```
class Route {
public:
```



```

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

```

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

```

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

```

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

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

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

```

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

```

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

```

ReverseRoute(route);
PrintRoute(route);

```

3.3.10. Параметризованные конструкторы

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

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

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

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

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

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

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

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

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

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

Чтобы такой код работал, нужно написать конструктор.

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

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

```

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

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

```

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

```

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

```

3.3.11. Конструкторы по умолчанию, использование конструкторов

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

```

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

```

```

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

```

```
Route route; // Теперь не компилируется
```

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

```

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

```

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

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

```

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

```

```
Route route; // Маршрут от Москвы до СПб
```

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

```

Route route1;
    // По умолчанию: Москва - Петербург

```

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

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

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

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

то в качестве объекта можно передать пустые фигурные скобки:

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

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

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

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

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

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

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

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

Аналогично можно делать и в случае встроенных типов:

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

3.3.12. Значения по умолчанию для полей структур

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

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

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

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

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

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

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

При этом все еще доступен синтаксис с фигурными скобками:

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

Также можно не указывать несколько последних полей:

```
Lecture lecture3 = {"ООР"};
```

В этом случае для них будут использоваться значения по умолчанию.

3.3.13. Деструкторы

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

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

Рассмотрим созданный ранее класс Route:

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

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

```

    int length;
};

```

Для демонстрационных целей в качестве ComputeDistance можно использовать простую заглушку:

```

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

```

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

Создадим лог вызовов функции ComputeDistance:

```

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

```

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

```

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

```



```

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

```

Теперь посмотрим, что выведет такой код:

```

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

```

```

Moscow — Saint Petersburg
Vyborg — Saint Petersburg
Vyborg — Vologda

```

3.3.14. Время жизни объекта

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

```

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

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

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

    string GetSource() const {
        return source;
    }
}

```

```

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

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

```

Выполним следующий код:

```

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

```

Результат его выполнения:

```

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

```

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

```

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

```

```

1
Default constructed
3
Default constructed
4
Destructed
Destructed

```

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

Теперь отправим на выполнение такой код:

```

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

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

```

Результат будет следующий:

```

1
Default constructed
2
Destructed
3

```

```

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

```

```

    return {};
}

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

```

```

1
Default constructed
2
Destructed

```

Если результат вызова функции не сохраняется, результат получается иной:

```

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

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

```

```

1
Default constructed
Destructed
2

```

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

Неделя 4

Потоки. Исключения. Перегрузка операторов

4.1. ООП: Примеры

4.1.1. Практический пример: класс «Дата»

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

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

Эту структуру можно использовать следующим образом:

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

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

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

Вывести созданную выше дату можно следующим образом:

```
PrintDate(date);
```

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

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

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

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

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

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

После этого прежняя запись перестает работать:

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

Это вынуждает записать такой код:

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

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

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

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

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

При этом он продолжает компилироваться.

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

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

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

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

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

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

Написав такие конструкторы, мы разрешили компилятору неявно преобразовывать целые числа к типам Day, Month, Year. Чтобы избежать неявного преобразования типов, нужно указать компилятору, что так делать не надо, 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 (\gg) или функции `getline`, которая позволяет читать данные из потока построчно.

Пусть заранее создан файл со следующим содержимым:

```
hello world!  
second line
```

Для работы с файлами нужно подключить библиотеку `fstream`:

```
#include <fstream>
```

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

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

В качестве аргумента конструктора указывается путь до желаемого файла.

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

```
string line;
```

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

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

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

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

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

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

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

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

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

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

4.2.3. Обработка случая, когда указанного файла не существует

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

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

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

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

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

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

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

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

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

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


4.2.4. Чтение из потока до разделителя

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

2017–01–25

Для этого создадим:

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

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

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

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

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

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

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

4.2.5. Оператор чтения из потока

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

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

```

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

```

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

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

4.2.6. Оператор записи в поток. Дозапись в файл.

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

```

const string path = "output.txt";

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

```

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

```

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

```

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

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

```

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

```

```

    }
  }
}

```

Предыдущая программа примет вид:

```

const string path = "output.txt";

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

ReadAll(path);

```

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

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

```

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

```

4.2.7. Форматирование вывода. Файловые манипуляторы.

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

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

```

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

```

Выведем их на экран:

```

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

```

При этом читать значения очень неудобно.

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

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

```
cout << fixed;
```

setprecision Задаёт количество знаков после запятой.

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

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

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

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

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

Здесь колонки были выведены в таком же формате.

setfill Указывает, каким символом заполнять расстояние между колонками.

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

left Выравнивание по левому краю поля.

```
cout << left;
```

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

```

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

```

Покажем как пользоваться манипуляторами setfill и left:

```

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

```

3.4. Перегрузка операторов для пользовательских типов

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

3.4.1. Тип Duration (Интервал)

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

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

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

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

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

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

Воспользуемся функциями, сперва заведя и инициализируя строковый поток:

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

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

3.4.2. Перегрузка оператора вывода в поток

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

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

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

Мы сделали класс гораздо удобнее для работы:

```
cout << dur1;
```

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

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

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

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

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

Можно вызвать по цепочке два оператора вывода:

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

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

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

После этого код начинает работать.

3.4.3. Перегрузка оператора ввода из потока

Аналогичным образом определим оператор ввода из потока:

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

Теперь считывать интервалы можно прямо из потока:

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

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

3.4.4. Конструктор по умолчанию

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

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

3.4.5. Перегрузка арифметических операций

Реализуем возможность складывать интервалы естественным образом:

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


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

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

Сокращения lhs и rhs обозначают Left/Right Hand Side.

После этого код начинает работать.

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

Однако, запустим этот код для другой пары интервалов:

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

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

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

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

После такого определения конструктора:

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

3.4.6. Сортировка. Перегрузка операторов сравнения.

Допустим, необходимо для вектора интервалов

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

расположить элементы этого вектора по возрастанию.

Для удобства напомним функцию PrintVector:

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

Использовать эту функцию можно следующим образом:

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

Попробуем отсортировать вектор:

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

При компиляции возникают ошибки, 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

```

3.3. Исключения в C++ (введение)

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

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

3.3.1. Практический пример: парсинг даты в заданном формате

Допустим необходимо парсить даты

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

из входного потока.

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

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

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

Проверим работоспособность этой функции:

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

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

2017a01b25

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

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

3.3.2. Выброс исключений в C++

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

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

Если формат даты правильный, такой код отработает без ошибок:

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

Если сделать строчку невалидной, программа упадет:

```

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

```

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

```

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

```

Функция ParseDate примет вид:

```

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

```

3.3.3. Обработка исключений. Блок try/catch

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

```

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

```

Проверим это на практике:

```

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

```

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

```

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

```

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

```

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

```




Основы разработки на C++: жёлтый пояс

Неделя 1

Целочисленные типы, кортежи, шаблонные функции



Оглавление

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

Целочисленные типы, кортежи, шаблонные функции

Целочисленные типы

Введение в целочисленные типы

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

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

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

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

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

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

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

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

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

Особенности целочисленных типов языка C++:

1. В языке C++ память для целочисленных типов ограничена. Если вам не нужны целые числа размером больше 2 миллиардов, язык C++ для вас выделит вот ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.
2. Возможны так же проблемы с беззнаковыми типами. Если бы в задаче 1 допускались отрицательные значения температуры, это то, что некоторые целочисленные типы языка C++ беззнаковые. Тем самым вы, сможете хранить больше положительных значений, но не сможете хранить отрицательные.

Виды целочисленных типов:

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

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

Узнаём размеры и ограничения типов:

Как узнать размеры типа? Очень просто:

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

Узнаём ограничения типов:

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

Преобразования целочисленных типов

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

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

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

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

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

переполнение, то и с результатом будет не то, что мы ожидаем. Теперь попробуем поработать с беззнаковыми типами:

```
int x = 2'000'000000;
unsigned int y = x; // сохраняем в переменную беззнакового типа 2000000000
unsigned int z = -z;
cout << x << " " << y << " " << -x << " " << z << endl;
// 2000000000 2000000000 -2000000000 2294967296
```

Если значение уместается даже в `int`, то проблем не будет. Но если записать отрицательное число в `unsigned`, мы получим не то, что ожидали. Возвращаясь к задаче о средней температуре, посмотрим, в чём была проблема:

```
vector<int> t = {-8, -7, 3};
int sum = 0; // знаковое
for (int x : t){
    sum += x;
}
int avg = sum / t.size(); // sum / t.size(); уже беззнаковое, т.к. t.size() беззнаковое
cout << avg << endl;
```

Правила вывода общего типа:

1. Перед сравнениями и арифметическими операциями числа приводятся к общему типу;
2. Все типы размера меньше `int` приводятся к `int`;
3. Из двух типов выбирается больший по размеру;
4. Если размер одинаковый, выбирается беззнаковый.

Примеры:

Слева	Операция	Справа	Общий тип	Комментарий
<code>int</code>	<code>/</code>	<code>size_t</code>	<code>size_t</code>	большой размер
<code>int32_t</code>	<code>+</code>	<code>int8_t</code>	<code>int32_t (int)</code>	тоже большой размер
<code>int8_t</code>	<code>*</code>	<code>uint8_t</code>	<code>int</code>	все меньшие приводятся к <code>int</code>
<code>int32_t</code>	<code><</code>	<code>uint32_t</code>	<code>uint32_t</code>	знаковый к беззнаковому

Для определения типа в самой программе можно просто вызвать ошибку компиляции и посмотреть лог ошибки. Изменим одну строчку:

```
int avg = (sum / t.size()) + vector<int>{}; // прибавили пустой вектор
```

И получим ошибку, в логе которой указано, что наша переменная `avg` имеет тип `int`. Теперь попробуем сравнить знаковое и беззнаковое число:

```
int main () {
    int x = -1;
    unsigned y = 1;
    cout << (x < y) << " ";
    cout << (-1 < 1u) << endl; // суффикс u делает 1 типом unsigned по умолчанию
    return 0;
}
// 0 0
```

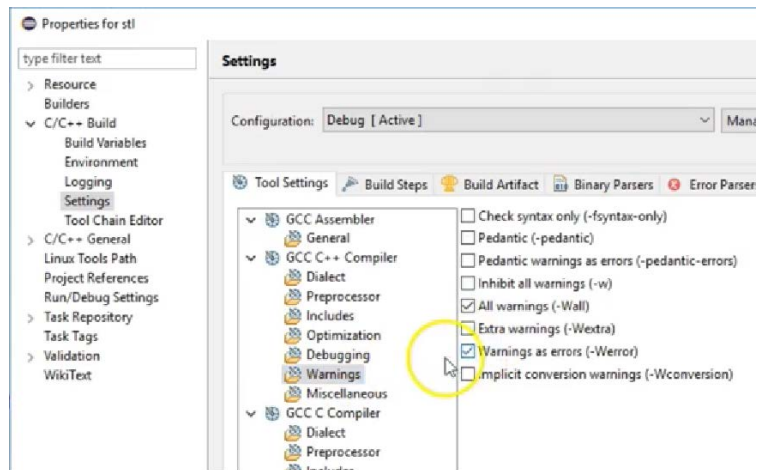
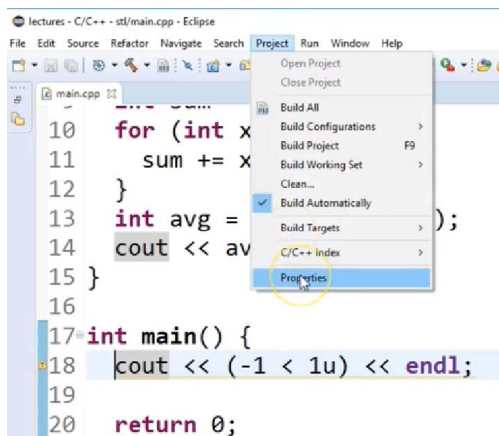
Как было сказано ранее, при операции между знаковым и беззнаковым типом обе переменные приводятся к беззнаковому. `-1`, приведённая к беззнаковому, становится очень большим числом, большим `1`. Суффикс `u` также приводит `1` к `unsigned`, а операция `<` теперь сравнивает `unsigned` `-1` и `1`. Причём в данном случае компилятор предупреждает нас о, возможно, неправильном сравнении.

Безопасное использование целочисленных типов

Настроим компилятор:

Попросим компилятор считать каждый warning (предупреждение) ошибкой. Project → Properties → C/C++ → Build → Settings → GCC C++ Compiler → Warnings и отмечаем Warnings as errors.

После этого ещё раз компилируем код. И теперь каждое предупреждение считается ошибкой, которую надо исправить. Это одно из правил хорошего кода.



```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < x.size(); ++i) {
        cout << i << " " << x[i] << endl;
    }
    return 0;
}
// error: "i < x.size()"... comparison between signed and unsigned
```

Есть два способа это исправить: объявить `i` типом `size_t` или явно привести `x.size()` к типу `int` с помощью `static_cast<int>(x.size())`.

```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < static_cast<int>(x.size()); ++i) {
        cout << i << " " << x[i] << " ";
    }
    return 0;
}
// 0 4 1 5
```

Исправляем задачу о температуре:

В задаче о температуре тоже приводим `t.size()` к знаковому с помощью оператора `static_cast`:

```
int main () {
    vector<int> t = {-8, -7, 3};    // сумма -4
```

```
...
int avg = sum / static_cast<int>(t.size()); // явно привели типы
cout << avg << endl;
return 0;
}
// -4
```

Предупреждений и ошибок не было. Всё, задача средней температуры для положительных и отрицательных значений решена! Таким образом если у нас где-то могут быть проблемы с беззнаковыми типами, мы либо следуем семантике и помним про опасности, либо приводим всё к знаковым с помощью `static_cast`.

Ещё примеры опасностей с беззнаковыми типами:

Переберём в векторе все элементы кроме последнего:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i < v.size() - 1; ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

После запуска код падает. Вычтя из `v.size()` 1 мы получили максимальное значение типа `size_t` и вышли из своей памяти. Чтобы такого не произошло, мы перенесём единицу в другую часть сложения:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i + 1 < v.size(); ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

Теперь на пустом векторе у нас все компилируется и вывод пустой. А на непустом выводит все элементы, кроме последнего. Напишем программу вывода элементов вектора в обратном порядке:

```
int main() {
```

```
vector<int> v = {1, 4, 5};
for (size_t i = v.size() - 1; i >= 0; --i) {
    cout << v[i] << endl;
}
return 0;
}
```

На пустом векторе, очевидно, будет ошибка. Но даже на не пустом он сначала выводит 5, 4, 1, а затем очень много чисел и программа падает. Это произошло из-за того, что `i >= 0` выполняется всегда и мы входим в бесконечный цикл. От этой проблемы мы избавимся «заменой переменной» для итерации:

```
for (size_t k = v.size(); k > 0 ; --k) {
    size_t i = k - 1; // теперь
    cout << v[i] << endl;
}
```

Теперь всё работает нормально. В итоге, проблем с беззнаковыми типами помогают избежать:

- Предупреждения компилятора;
- Внимательность при вычитании из беззнаковых;
- Приведение к знаковым типам с помощью `static_cast`.

Кортежи и пары

Упрощаем оператор сравнения

Поговорим про новые типы данных – пары и кортежи. Начнём с проблемы, которая возникла в курсовом проекте курса «Белый пояс по C++»: мы должны были хранить даты в виде структур из полей «год», «месяц», «день» в ключах словарей. А поскольку словарь хранит ключи отсортированными, нам надо определить для этого типа данных оператор «меньше». Можно сделать это так:

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

```
using namespace std;

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

bool operator <(const Date& lhs, const Date& rhs) {
    if (lhs.year != rhs.year) {
        return lhs.year < rhs.year;
    }
    if (lhs.month != rhs.month) {
        return lhs.month < rhs.month;
    }
    return lhs.day < rhs.day;
}
...
```

Сначала сравниваем года, потом месяцы и затем дни. Но здесь много мест для ошибок и код довольно длинный. В эталонном решении курсового проекта эта проблема решена так:

```
bool operator <(const Date& lhs, const Date& rhs) {
    return vector<int>{lhs.year, lhs.month, lhs.day} <
        vector<int>{rhs.year, rhs.month, rhs.day};
}
...
```

Выигрыш в том, что для векторов лексикографический оператор сравнения уже определён и этот код работает для каких-нибудь двух дат:

```
int main() {
    cout << (Date{2017, 6, 8} < Date {2017, 1, 26}) << endl;
    return 0;
}
// 0
```

Действительно, первая дата больше второй и выводит 0. Но тип `vector` слишком мощный: он позволяет делать `push_back` в себя, удалять из середины, что-то ещё. Нам этот тип не нужен в данном случае. Нам нужно всего лишь объединить для левой даты и для правой даты три

значения в одно и сравнить. Кроме того, вектор работает только при однотипных элементах. Если бы у нас месяц был строкой, вектор бы не подходил.

Как же объединить разные типы в один массив? Для этого нужно подключить библиотеку `tuple`. И вместо вектора вызывать функцию `tie` от тех значений, которые надо связать. В нашем случае год, месяц и день левой даты и год, месяц и день правой даты надо связать и сравнить.

```
#include <tuple>
...
bool operator <(const Date& lhs, const Date&rhs) {
    auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // сохраним левую дату
    auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // и правую
    return lhs_key < rhs_key
}
int main() {
    cout << (Date{2017, "June", 8} < Date{2017, "January", 26}) << endl;
    return 0;
}
// 0
```

И действительно, строка «June» лексикографически больше строки «January» и наша программа делает то, что нужно. Теперь узнаем, какого типа у нас `lhs_key` и `rhs_key`, породив ошибку компиляции.

```
...
    auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // левая
    auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // правая
    lhs_key + rhs_key; // тут нарочно порождаем ошибку
    return lhs_key < rhs_key
...
// operator+ не определен для std::tuple<const int&, const std::string&, const int&>...
```

То есть они имеют тип `tuple<const int&, const string&, const int&>`. Tuple – это *кортеж*, т. е. структура из ссылок на нужные нам данные (возможно, разнотипные).

Кортежи и пары

Создадим кортеж из трёх элементов:

```
#include <iostream>
#include <vector>
#include <tuple>
using namespace std;
int main() {
    tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
    auto t = tie(7, "C++", true); // пытаемся связать константные значения в tie
    return 0;
}
```

В первой строке всё хорошо – мы создали структуру из трёх полей. А во второй – ошибка компиляции, потому что `tie` создает кортеж из ссылок на объекты (которые хранятся в каких-то переменных). Используем функцию `make_tuple`, создающую кортеж из самих значений. И будем обращаться к полям кортежа:

```
...
// tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
auto t = make_tuple(7, "C++", true)
cout << get<1>(t) << endl;
return 0;
}
// C++
```

С помощью функции `get<1>(t)` мы получили 1-ый (нумерация с 0) элемент кортежа `t`. Использование кортежа `tuple` целесообразно, только если нам необходимо, чтобы в кортеже были разные типы данных.

Замечание: в C++ 17 разрешается не указывать шаблонные параметры `tuple` в `< ... >`, т. е. кортеж можно создавать так:

```
tuple t(7, "C++", true);
```

Но при компиляции компилятор попросит параметры. Оказывается надо явно сказать ему, чтобы он использовал стандарт C++ 17. Для этого снова идём в Project → Properties → C/C++ Build → Dialect → Other dialect flags и пишем `std = c++17`, т. е. версии C++ 17 (для этого компилятор должен быть обновлен до версии GCC7 или больше).

Если же мы хотим связать только два элемента, то используем структуру «пара» – `pair`. Пара – это частный случай кортежа, но отличие пары в том, что её поля называются `first` и `second` и к ним можно обращаться напрямую:

```
int main() {
    pair p(7, "C++"); // в новом стандарте можно и без <int, string>
    // auto p = make_pair(7, "C++"); // второй вариант
    cout << p.first << " " << p.second << endl;
    return 0;
}
// 7 C++
```

В результате нам вывело нашу пару. Эту структуру мы уже видели при итерировании по словарию. Так что нам удобно её использовать, не объявляя структуру явно.

Для примера создадим словарь (map):

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}};
    for (const auto& item : digits) {
        cout << item.first << " " << item.second << endl;
    }
    return 0;
}
// 1 one
```

Поскольку словарь – это пара «ключ-значение», то можно (как было объяснено в предыдущем курсе) распаковать значения `item`:

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}}
    for (const auto& [key, value] : digits) {
        cout << key << " " << value << endl;
    }
    return 0;
}
// 1 one
```

Всё скомпилировалось, значит, мы можем итерироваться по словарию, не создавая пар.

Возврат нескольких значений из функций

Возврат нескольких значений из функций – ещё одна область применения кортежей и пар. Будем хранить класс с информацией о городах и странах:

```
#include <iostream>
#include <vector>
#include <utility>
#include <map>
#include <set>
using namespace std;

class Cities { // класс городов и стран
public:
    tuple<bool, string> FindCountry(const string& city) const {
        if (city_to_country.count(city) == 1) {
            return {true, city_to_country.at(city)};
            // city_to_country[city] выдало бы ошибку, потому что могло нарушить const словаря
        } else if (ambiguous_cities.count(city) == 1) {
            return {false, "Ambigious"};
        } else {
            return {false, "Not exists"}; // если нет
        }
        // выводит значение, нашлась ли единственная страна для города
    } // и сообщение - либо страна не нашлась, либо их несколько
private:
    // по названию города храним название страны:
    map<string, string> city_to_country;
    // множество городов, принадлежащих нескольким странам:
    set<string> ambiguous_cities;
}

int main() {
    Cities cities;
    bool success;
    string message; // свяжем кортежем ссылок
    tie(success, message) = cities.FindCountry("Volgograd");
    cout << " " << message << endl; // вывели результат
    return 0;
}
//0 Not exists
```


Всё работает. Таким образом, мы научились возвращать из метода несколько значений с помощью кортежа. А по новому стандарту можно получить кортеж и распаковать его в пару переменных:

```
int main() {
    Cities cities;
    auto [success, message] = cities.FindCountry("Volgograd"); // сразу распаковали
    cout << " " << message << endl;
    return 0;
}
//0 Not exists
```

Итак, если вы хотите вернуть несколько значений из функции или из метода, используйте кортеж. А если вы хотите сохранить этот кортеж в какой-то набор переменных, используйте **structured bindings** или функцию **tie**.

Кортежи и пары нужно использовать аккуратно. Они часто мешают читаемости кода, если вы начинаете обращаться к их полям. Например, представьте себе, что вы хотите сохранить словарь из названия города в его географические координаты. У вас будет словарь, у которого в значениях будут пары двух вещественных чисел. Назовем его **cities**. Как же потом вы будете, например, итерироваться по этому словарю?

```
int main() {
    map<string, pair<double, double>> cities;
    for (const auto& item : cities) {
        cout << item.second.first << endl; // абсолютно нечитаемый код
    }
    return 0;
}
```

Заключение: кортежи позволяют упростить написание оператора **<** или вернуть несколько значений из функции. Пары – это частный случай кортежей, у которых понятные названия полей, к которым удобно обращаться.

Шаблоны функций

Введение в шаблоны

Рассмотрим шаблоны функций на примере функции возведения числа в квадрат.

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

int Sqr(int x) { // функция возведения в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2) << endl; // результат выведем в поток
    return 0;
}
// 4
```

Функция работает. Теперь мы хотим возводить в квадрат дробные числа, например, 2.5. Получается снова 4. Это неправильно.

```
... cout << Sqr(2.5) << endl; // результат функции выведем в поток
// 4
```

Это происходит потому, что у нас нет аналогичной функции для работы с дробными числами. Заведём её:

```
int Sqr(int x) { // функция возведения целого числа в квадрат
    return x * x;
}

double Sqr(double x) { // функция возведения дробного числа в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2.5) << endl;
    return 0;
}
```

```

}
// 6.25

```

Всё работает, но у нас появилось две функции, которые делают одно и то же, но с разными типами. Гораздо удобнее написать функцию, работающую с каким-то типом T:

```

using namespace std;
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    cout << Sqr(2.5) << " " << Sqr(3) << endl;
    return 0;
}
// 6.25 9

```

Собираем наш код и видим, что всё работает. Тип T компилятор выведет сам, чтобы типы поддерживали умножение. Нужно было только его объявить ключевым словом `template <typename T>`. Теперь попробуем возвести в квадрат пару:

```

#include <utility> // добавляем нужную библиотеку
... // код функции оставляем таким же
int main() {
    auto p = make_pair(2, 3); // создаем пару
    cout << Sqr(p) << endl; // пытаемся возвести ее в квадрат
    return 0;
}
// no match for 'operator*' (operand types are std::pair<int,int> and std::pair<int,int>)

```

Видим ошибку, т. к. для оператора умножения не определены аргументы «пара и пара». Тогда напишем шаблонный оператор умножения для пар:

```

using namespace std;
template <typename First, typename Second>
// т.к. умножение для пар не определено, вручную определим оператор умножения для пар:
pair<First, Second> operator * (const pair<First, Second>& p1,
                                const pair<First, Second>& p2) {
    // мы можем создавать переменные шаблонного типа
    First f = p1.first * p2.first;

```

```
Second s = p1.second * p2.second;
return {f, s};
}
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    auto p = make_pair(2.5, 3); // создаем пару
    auto res = Sqr(p); // возводим пару в квадрат
    cout << res.first << " " << res.second << endl; // выводим получившееся
    return 0;
}
// 6.25 9
```

Код работает – пара возвелась в квадрат (и её дробная часть, и целая). Одним из важных плюсов языка C++ является возможность подобным образом избавляться от дублирований и сильно сокращать код.

Универсальные функции вывода контейнеров в поток

В курсах ранее мы часто печатали содержимое наших контейнеров, будь то `vector` или `map`, на экран. Для этого мы определяли специальную функцию либо перегружали оператор вывода в поток. Давайте это сделаем и сейчас:

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
// напишем свой оператор вывода в поток вектора целых типов
ostream& operator<< (ostream& out, const vector<int>& vi) {
    for (const auto& i : vi) { // проитерируемся по вектору
        out << i << ' '; // выведем все элементы в поток
    }
    return out;
}
```

```
}

int main() {
    vector<int> vi = {1, 2, 3};
    cout << vi << endl;
}
// 1 2 3
```

Всё выводится. Но если поменять тип вектора на `double`, то будет ошибка:

```
...
vector<double> vi = {1, 2, 3}; // теперь дробный вектор
...
// no match for 'operator <<' (operand types are std::ostream and std::vector<double>)
```

Для решения этой проблемы можно было бы каждый раз заново дублировать код. Но с помощью шаблонов функций мы меняем тип вектора с `int` на шаблонный `T`:

```
using namespace std;
template <typename T> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const vector<T>& vi) { // вектор на шаблон
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}

int main() {
    vector<double> vi = {1.4, 2, 3}; // дробные числа
    cout << vi << endl;
}
// 1.4 2 3
```

Ошибки пропали, вывелся наш вектор из дробных чисел. Мы научились универсальным способом решать задачу для вектора. Таким же универсальным способом научимся решать задачу для других контейнеров.

```
int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <' (operand types are std::ostream and std::map<int, int>)
```

Видим, что оператор вывода для `map` не определён. Определим его так же, как и для вектора:

```
...
template <typename Key, typename Value> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
...
int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <<' (operand types are std::ostream and std::pair<const int, int>)
```

Заметим, что ошибка для `map` имеет интересный вид: `pair<const int, int>`. Действительно, ведь `map` – это `pair`, в которой `key` нельзя модифицировать, а `value` можно. Получается, нам достаточно определить оператор вывода в поток для пары. Тогда мы сможем вывести и `map`.

```
// весь рабочий код
...
template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    out << p.first << ", " << p.second;
    return out;
}
template <typename T> // для vector
ostream& operator<< (ostream& out, const vector<T>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
template <typename Key, typename Value> // для пар
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
```

```
int main() {
    // vector<double> vi = 1.4, 2,3;
    // cout << vi << endl;
    map<int, int> m = {{1, 2}, {3, 4}}; // целые числа
    map<int, int> m2 = {{1.4, 2.1}, {3.4, 4}}; // дробные числа
    cout << m << ' '; << m2 << endl;
}
//1, 2 3, 4; 1.4, 2.1 3.4, 4
```

Код отработал корректно. Оба `map`'а вывелись, как нам и было нужно. Заметим, что код с вектором, если его добавить, до сих пор будет работать. Но этот код тоже можно доработать. Шаблоны для вектора и для `map`'а выглядят почти одинаково. Кроме того, для читаемости можно сделать улучшение: когда мы выводим `map`, обрамлять его в фигурные скобки, когда вектор – в квадратные, а когда мы выводим пару, обрамлять её круглыми скобками.

Рефакторим код и улучшаем читаемость вывода

Исправим предыдущую программу и сделаем её вывод более читаемым. Нужно создать шаблонную функцию, которая на вход будет принимать коллекцию. На вход этой функции будем передавать разделитель, через который надо вывести элементы нашей коллекции. Единственное, что мы требуем от входной коллекции: по ней можно итерироваться с помощью цикла `range-based for` и её элементы можно вывести в поток.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
template <typename Collection> // тип коллекции
string Join(const Collection& c, char d) { // передаем коллекцию и разделитель
    stringstream ss; // завели строковый поток
    bool first = true; // первый ли это элемент?
    for (const auto& i : c) {
        if (!first) {
            ss << d; // если вывели не первый элемент - кладем поток в разделитель
        }
    }
}
```

```

    first = false; // т.к. следующий элемент точно не будет первым
    ss << i; // кладем следующий элемент в поток
}
return ss.str();
}

template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    return out << '(' <<p.first << ',' << p.second << ')'; // тоже изменили
}

template <typename T> // для vector изменили код и добавили скобочки
ostream& operator<< (ostream& out, const vector<T>& vi) {
    return out << '[' << Join(vi, ',') << ']';
} // оператор вывода возвращает ссылку на поток

template <typename Key, typename Value> // для map убрали аналогично vector
ostream& operator<< (ostream& out, const map<Key, Value>& m) {
    return out << '{' << Join(m, ',') << '}'; // и добавили фигурные скобочки
}

int main() {
    vector<double> vi = {1.4, 2, 3};
    pair<int, int> m1 = {1, 2};
    map<double, double> m2 = {{1.4, 2.1} , {3.4, 4}};
    cout << vi << ' ' << m1 << ' ' << m2 << endl;
}
// [1.4, 2,3] (1, 2) {(1.4, 2.1), (3.4, 4)}

```

Всё работает. Наша программа вывела сначала вектор, потом пару и затем `map`. Для более сложных конструкций она тоже будет работать. Например, вектор векторов:

```

int main() {
    vector<vector<int>> vi = {{1, 2}, {3, 4}};
    cout << vi << endl;
}
// [[1, 2], [3, 4]]

```

В итоге всё работает и на сложных контейнерах. Таким образом, мы сильно упростили наш код и избежали ненужного дублирования с помощью шаблонов и универсальных функций.

Указание шаблонного параметра-типа

Рассмотрим случай, когда компилятор не знает, как на основе вызова шаблонной функции вывести тип `T` на примере задачи о выводе максимального из двух чисел.

```
#include <iostream>
using namespace std;
template <typename T>
T max(T a, T b) {
    if (b < a) {
        return a;
    }
    return b;
}

int main() {
    cout << Max(2, 3) << endl;
    return 0;
}
// 3
```

Если оба числа целые, то всё работает. Но если поменять одно число на вещественное, мы увидим ошибки:

```
...
cout << Max(2, 3.5) << endl;
// deduce conflicting types for parameter 'T' (int and double)
```

Т. е. вывод шаблонного параметра типа `T` не может состояться, потому что компилятор не знает, что поставить: `int` или `double`. В таких ситуациях мы либо приводим переменные к одному типу, либо подсказываем компилятору таким образом:

```
cout << Max<double>(2, 3.5) << endl; // явно показываем компилятору тип T
// 3.5
```

А если же мы попросим `int`, получим следующее:

```
cout << Max<int>(2, 3.5) << endl; // 3.5 приведётся к int и мы сравнили
// 3
```

Писать уже существующие функции плохо, поэтому вызовем стандартную функцию `max` из библиотеки `algorithms`:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    cout << max<int>(2, 3.5) << ' ' << max<double>(2, 3.5) << endl;
    return 0;
}
// 3 3.5
```

Функция `max` тоже шаблонная. Если явно указать тип, к которому приводить результат, у нас будет то же, что мы уже видели. Если же тип не указывать, произойдёт знакомая ошибка компиляции.

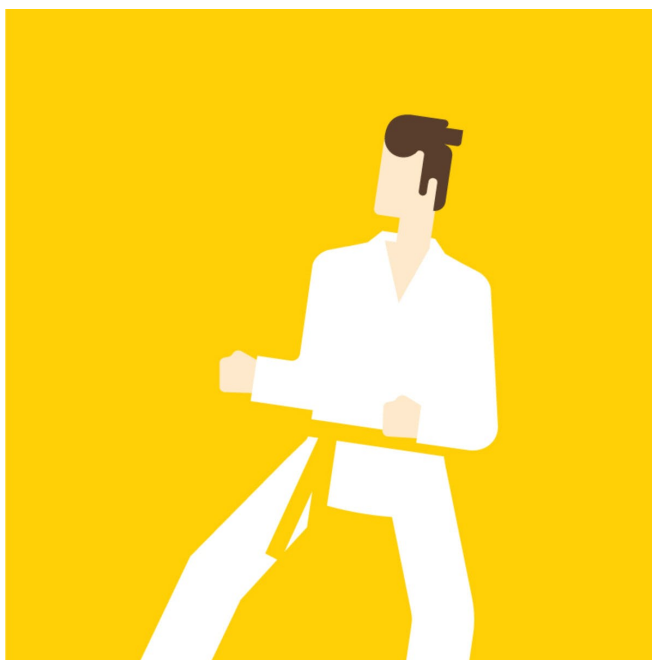
Подведём итоги:

1. Шаблонные функции объявляются так: `template <typename T> T Foo(T var) { ... };`
2. Вместо слова `typename` можно использовать слово `class`, т. к. в данном контексте они эквивалентны;
3. Шаблонный тип может автоматом выводиться из контекста вызова функции;
4. После объявления используется, как и любой другой тип;
5. Выведение шаблонного типа может происходить либо автоматически, на основе аргументов, либо с помощью явного указания в угловых скобках (`std::max<double>(2, 3.5)`);
6. Цель шаблонных функций: сделать код короче (избавившись от дублирования) и универсальнее.

Основы разработки на C++: жёлтый пояс

Неделя 2

Тестирование



Оглавление

Тестирование	2
2.1 Тестирование и отладка	2
2.1.1 Введение в юнит-тестирование	2
2.1.2 Декомпозиция решения в задаче «Синонимы»	3
2.1.3 Простейший способ создания юнит-тестов на C++	5
2.1.4 Отладка решения задачи «Синонимы» с помощью юнит-тестов	7
2.1.5 Анализ недостатков фреймворка юнит-тестов	10
2.1.6 Улучшаем <code>assert</code>	11
2.1.7 Внедряем шаблон <code>AssertEqual</code> во все юнит-тесты	13
2.1.8 Изолируем запуск отдельных тестов	15
2.1.9 Избавляемся от смешения вывода тестов и основной программы	17
2.1.10 Обеспечиваем регулярный запуск юнит-тестов	18
2.1.11 Собственный фреймворк юнит-тестов. Итоги	19
2.1.12 Общие рекомендации по декомпозиции программы и написанию юнит-тестов	20

Тестирование

Тестирование и отладка

Введение в юнит-тестирование

Вспомним, что говорилось в курсе «C++: белый пояс». Если решение не принимается тестирующей системой, то нужно:

1. Внимательно перечитать условия задачи;
2. Убедиться, что программа корректно работает на примерах;
3. Составить план тестирования (проанализировать классы входных данных);
4. Тестировать программу, пока она не пройдет все тесты;
5. Если идеи тестов кончились, но программа не принимается, то выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами.

Как юнит-тесты помогают в отладке:

1. Позволяют протестировать каждый компонент изолированно;
2. Их проще придумывать;
3. Упавшие юнит-тесты указывают, в каком блоке программы ошибка.

Декомпозиция решения в задаче «Синонимы»

На примере задачи «Синонимы» из курса «C++: белый пояс» покажем применение юнит-тестов.

Условие задачи:

Два слова называются синонимами, если они имеют похожие значения. Надо реализовать словарь синонимов, обрабатывающий три вида запросов:

- `ADD word1 word2` – добавить в словарь пару синонимов (`word1`, `word2`);
- `COUNT word` – выводит текущее количество синонимов для слова `word`;
- `CHECK word1 word2` – проверяет, являются ли слова синонимами.

Ввод:	Вывод:
<code>ADD program code</code>	
<code>ADD code cipher</code>	
<code>COUNT cipher</code>	<code>1</code>
<code>CHECK code program</code>	<code>YES</code>
<code>CHECK program cipher</code>	<code>NO</code>

Посмотрим на решение, которое у нас уже есть и которое надо протестировать:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;

int main() {
    int q; // считываем количество запросов
    cin >> q;
    // храним словарь синонимов (для строки хранит множество всех её синонимов)
    map<string, set<string>> synonyms;
    for (int i = 0; i < q; ++i) { // обрабатываем запросы
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") { // считываем две строки и добавляем в словарь
```

```
    string first_word, second_word;
    cin >> first_word >> second_word;
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
} else if (operation_code == "COUNT") { // считываем одну строку и выводим
// размер
    string word;
    cin >> word;
    cout << synonyms[word].size() << endl;
} else if (operation_code == "CHECK") { // считываем два слова и проверяем
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (synonyms[first_word].count(second_word) == 1) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Отправляем в тестирующую систему и видим, что решение не принялось. Он говорит нам, что неправильный ответ на третьем тесте, и больше информации нет. Решение надо тестировать на различных входных данных. Но вроде всё работает и стоит переходить к юнит-тестированию. А для него надо сначала выполнить декомпозицию задачи «Синонимы». Давайте ввод и вывод оставим в `main`, а обработку запроса вынесем в отдельные функции:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;
void AddSynonyms(map<string, set<string>>& synonyms, // функция добавления
                 const string& first_word, const string& second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(map<string, set<string>>& synonyms, // количество синонимов
                      const string& first_word) {
    return synonyms[first_word].size();
}
```

```
}
bool AreSynonyms(map<string, set<string>>& synonyms, // проверка
                 const string& first_word, const string& second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

int main() {
    int q;
    cin >> q;
    map<string, set<string>> synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word)
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << GetSynonymCount(synonyms, word) << endl;
        } else if (operation_code == "CHECK") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            if (AreSynonyms(synonyms, first_word, second_word)) {
                cout << "YES" << endl;
            } else {
                cout << "NO" << endl;
            }
        }
    }
}
return 0;
}
```

Простейший способ создания юнит-тестов на C++

Посмотрим, как писать юнит-тесты и как они должны себя вести на примере функции Sum, которая находит сумму двух чисел.

```
#include <iostream>
```



```
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y;
}
void TestSum() { // собираем набор тестов для функции Sum
    assert(Sum(2, 3) == 5); // мы ожидаем, что 2+3=5
    assert(Sum(-2, -3) == -5); // проверка отрицательных чисел
    assert(Sum(-2, 0) == -2); // проверка прибавления 0
    assert(Sum(-2, 2) == 0); // проверка, когда сумма = 0
    cout << "TestSum OK" << endl;
}
int main() {
    TestSum();
    return 0;
}
// TestSum OK
```

Тесты отработали и не нашли ошибок. Теперь посмотрим, что должно быть при наличии ошибок:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y - 1; // сделали заведомо неправильно
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Нам написали, в каком файле, в какой строке какой **Assert** не сработал. Это облегчает поиск ошибок. Мы можем добиться другой ошибки, например:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    if (x < 0) {
        x -= 1
    }
    return x + y;
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Видим, что первый тест прошёл, а на втором уже ошибка. Таким образом, мы можем проверять каждую функцию по отдельности на ожидаемых значениях.

Отладка решения задачи «Синонимы» с помощью юнит-тестов

Для задачи «Синонимы» покроем каждую функцию юнит-тестами и сократим код заменой `map<string, set<string>>` на что-то более короткое:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <cassert>
using namespace std;
using Synonyms = map<string, set<string>>;
// сократили запись типа и везде изменили на Synonyms
void AddSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word); // тут должен не сработать AddSynonyms
}
size_t GetSynonymCount(Synonyms& synonyms, const string& word) {
    return synonyms[word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

void TestAddSynonyms() { // тестируем AddSynonyms
{
    Synonyms empty; // тест 1
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}}, // ожидаем, что при добавлении синонимов появятся две записи в
        // словаре
        {"b", {"a"}}
    };
    assert(empty == expected);
}
```

```
}
{ // заметим, что мы формируем корректный словарь и ожидаем, что он останется корректным
  Synonyms synonyms = { // если вдруг корректность нарушится, то assert скажет, где
    {"a", {"b"}}, // тест 2
    {"b", {"a", "c"}},
    {"c", {"b"}}
  };
  AddSynonyms(synonyms, "a", "c");
  const Synonyms expected = {
    {"a", {"b", "c"}},
    {"b", {"a", "c"}},
    {"c", {"a", "b"}}
  };
  assert(synonyms == expected);
}
cout << "TestAddSynonyms OK" << endl;
}

void TestCount() { // тестируем Count
{
  Synonyms empty;
  assert(GetSynonymCount(empty, "a") == 0);
}
{
  Synonyms synonyms = {
    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
  };
  assert(GetSynonymCount(synonyms, "a") == 2);
  assert(GetSynonymCount(synonyms, "b") == 1);
  assert(GetSynonymCount(synonyms, "z") == 0);
}
cout << "TestCount OK" << endl;
}

void TestAreSynonyms() { // тестируем AreSynonyms
{
  Synonyms empty; // пустой словарь для любых двух слов вернёт false
  assert(!AreSynonyms(empty, "a", "b"));
  assert(!AreSynonyms(empty, "b", "a"));
}
```

```

}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    assert(AreSynonyms(synonyms, "a", "b"));
    assert(AreSynonyms(synonyms, "b", "a"));
    assert(AreSynonyms(synonyms, "a", "c"));
    assert(AreSynonyms(synonyms, "c", "a"));
    assert(!AreSynonyms(synonyms, "b", "c")); // false
    assert(!AreSynonyms(synonyms, "c", "b")); // false
}
cout << "TestAreSynonyms OK" << endl;
}
void TestAll() { // функция, вызывающая все тесты
    TestCount();
    TestAreSynonyms();
    TestAddSynonyms();
}
int main() {
    TestAll();
    return 0;
}
// TestCount OK
// TestAreSynonyms OK
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.

```

Видим, что `Count` и `AreSynonyms` работают нормально, а вот в `AddSynonyms` у нас ошибка. Смотрим, что не так. Идём в `AddSynonyms` и видим, что:

```
synonyms[first_word].insert(first_word); // мы должны к 1 слову добавить 2, а не 1
```

Теперь после исправления все наши тесты отработали успешно. Снова пробуем отправить в тестирующую систему наше решение, закомментировав вызов `TestAll()`; в `main`. Тестирование завершилось и решение принято тестирующей системой. Таким образом мы на простом примере продемонстрировали эффективность декомпозиции программы и юнит-тестов.

Анализ недостатков фреймворка юнит-тестов

По ходу разработки юнит-тестов во время решения задачи «Синонимы» мы смогли написать небольшой юнит-тест фреймворк. Посмотрим, что за фреймворк получился. Во-первых, он основан на функции `assert`. Его главный плюс – мы узнаём, какая именно проверка сработала неправильно. На предыдущей задаче мы видели:

```
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.
```

Основные недостатки:

1. При проверке равенства в консоль не выводятся значения сравниваемых переменных. И мы не знаем, чем была переменная `empty`;
2. После невыполненного `assert` код падает. Если в `TestAll` поставить `TestAddSynonyms` на первое место, то остальные два теста даже не начнутся;
3. Кроме того, у нас пока что результаты тестов выводят ОК в стандартный вывод и смешиваются с тем, что должен выводить код.

В C++ уже существует много фреймворков для работы с тестами, в которых этих недостатков нет.

C++ Unit Testing Frameworks:

1. Google Test
2. CxxTest
3. Boost Test Library

Далее мы свой Unit Testing Framework улучшим для того, чтобы показать, что текущих знаний C++ хватает для таких вещей. И вы будете понимать как он работает, и сможете его менять под свои нужды.

Улучшаем assert

Избавимся от первого недостатка `assert`: когда он срабатывает, мы не видим, чему равен каждый из операндов. Т. е. мы хотим видеть для кода такой вывод:

```
int x = Add(2, 3);
assert(x == 4);
// Assertion failed: 5 != 4
```

И работало оно для любых типов данных:

```
vector<int> sorted = Sort({1, 4, 3});
assert(sorted == vector<int> {1, 3, 4});
// Assertion failed: [1, 4, 3] != [1, 3, 4]
```

Такие универсальные выводы помогут догадаться о возможной ошибке. Т. е. нам нужна функция сравнения двух переменных какого-то произвольного типа. Напишем шаблон `AssertEqual` перед `TestAddSynonyms()`.

```
#include <exception> // подключим исключения
#include <sstream>    // подключили строковые потоки
...
template <class T, class U>
void AssertEqual (class T& t, const U& u) {
    // значения двух разных типов для удобства
    if (t != u) { // если значения не равны, то мы даём знать, что этот assert не сработал
        ostringstream os;
        os << "Assertion failed: " << t << "!=" << u;
        throw runtime_error(os); // бросим исключение с сообщением со значениями t и u
    }
}
```

Встроим это в `TestCount()`. Заменим `assert` на наш `AssertEqual` внутри `TestCount()`.

```
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"), 0);
        AssertEqual(GetSynonymCount(empty, "b"), 0);
    }
    {
        Synonyms synonyms = {
```

```

    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
};
AssertEqual(GetSynonymCount(synonyms, "a"), 2);
AssertEqual(GetSynonymCount(synonyms, "b"), 1);
AssertEqual(GetSynonymCount(synonyms, "z"), 0);
}
}
// Warning ... comprison between signed and unsigned ...

```

Код скомпилировался, но мы получили Warning из-за сравнения между знаковым и беззнаковым типами в `AssertEqual`. Это происходит потому, что все константы (2, 1 и 0 в нашем случае) имеют тип `int` (как уже было сказано в неделе 1), который мы сравниваем с типом `size_t`. Исправляем это, дописав к ним `u` справа: `1 → 1u` и т. д.

Теперь, сделав нарочную ошибку где-нибудь в `GetSynonymCount`, мы получим предупреждение: `Assertion failed: 1 != 0`. Но пока мы не видим, какой именно `Assert` сработал. Исправим это, передавая в `Assert` строчку `hint` и также добавим каждому `Assert`'у строку идентификации, по которой мы сможем однозначно понять, какой именно `Assert` выдал ошибку:

```

void AssertEqual (class T& t, const U& u, const string& hint) {
    if (t != u) {
        ostringstream os;
        os << "Assertion failed: " << t << "!=" << u << "Hint: " << hint;
        throw runtime_error(os);
    }
}
...
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"),
            0u, "Synonym count for empty dict a");
        AssertEqual(GetSynonymCount(empty, "b"),
            0u, "Synonym count for empty dict b");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},

```

```

    {"c", {"a"}}
};
AssertEqual(GetSynonymCount(synonyms, "a"), 2u, "Nonempty dict, count a");
AssertEqual(GetSynonymCount(synonyms, "b"), 1u, "Nonempty dict, count b");
AssertEqual(GetSynonymCount(synonyms, "z"), 0u, "Nonempty dict, count z");
}
}
// Asserting failed: 1!= 0 Hint: Synonym count for empty dict

```

Внедряем шаблон AssertEqual во все юнит-тесты

Добавим Assert в AreSynonyms. Только AssertEqual нам не подходит, потому что в данной функции у нас только два константных значения: true и false. Вместо этого напомним аналог классического assert, который назовём Assert (C++ чувствителен к регистру). И если мы испортим функцию AreSynonyms, то получим соответствующую ошибку с подсказкой.

```

void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}
... // модернизируем наш TestAreSynonyms
void TestAreSynonyms() {
    {
        Synonyms empty;
        Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
        Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},
            {"c", {"a"}}
        };
        Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
        Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
        Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
        Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
        Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
        Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms nonempty c b");
    }
}

```



```

}
}
// Asserting failed: 0!= 1 Hint: AreSynonyms empty a b

```

Получили нужную ошибку, и по ней мы можем увидеть, где что-то не так. Осталась только функция `AddSynonyms`, и ловим ошибку:

```

void TestAddSynonyms() {
{
    Synonyms empty;
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}},
        {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Add to empty");
}
{
    Synonyms synonyms = {
        {"a", {"b"}},
        {"b", {"a", "c"}},
        {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
        {"a", {"b", "c"}},
        {"b", {"a", "c"}},
        {"c", {"b", "a"}}
    };
    AssertEqual(synonyms, expected, "Nonempty");
}
}
// no match for 'operator <<' (operand types are std::ostream<char> and std::map ...)

```

Ошибка произошла с `empty` и `synonyms`, которые являются `map<string, set<string>`. Мы пытаемся их вывести в стандартный поток вывода (см. неделя 1). Пишем перегрузку оператора вывода для `map` и для `set`, предварительно исправив ошибку в `AreSynonyms`, допустим ошибку в `AddSynonyms` и поймаем её:

```

template <class T> // учимся выводить в поток set
ostream& operator << (ostream& os, const set<T>& s) {

```

```
os << "{";
bool first = true;
for (const auto& x : s) {
    if (!first) {
        os << ", ";
    }
    first = false;
    os << x;
}
return os << "}";
}

template <class K, class V> // учимся выводить в поток map
ostream& operator << (ostream& os, const map<K, V>& m) {
    os << "{";
    bool first = true; // грамотная расстановка запятых
    for (const auto& kv : m) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << kv.first << ": " << kv.second;
    }
    return os << "}";
}

// Asserting failed: {a: {a}, b: {a}} != {a: {b}, b: {a}}. Hint: Add to empty
```

Таким образом, мы внедрили шаблон `AssertEqual`, который позволяет найти ошибку, узнать, с чем она возникла и найти конкретное место в коде благодаря подсказке.

Изолируем запуск отдельных тестов

Теперь исправим следующий недостаток `Assert`: если он срабатывает, то код падает и другие тесты не выполняются. Аварийное завершение программы у нас возникало из-за вылета исключения в `AssertEqual`. Теперь будем ловить эти исключения в `main`:

```
int main() {
    try {
        TestAreSynonyms(); // ловим исключение
    }
```

```

} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAreSynonyms" << " fail: " << e.what() << endl;
} // если мы словили исключение, то работа всё равно продолжится
try {
    TestCount();
} catch (runtime_error& e) {
    ++fail_count;
    cout << " TestCount" << " fail: " << e.what() << endl;
}
try {
    TestAddSynonyms();
} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAddSynonyms" << " fail: " << e.what() << endl;
}
}
}

```

Но это неудобно, код дублируется. Нам бы хотелось, чтобы этот `try/catch` и вывод были написаны в одном месте, а мы туда могли бы передавать различные тестовые функции, и они бы там выполнялись, и исключения бы от них ловились, всё бы работало. В C++ это можно сделать, ведь функции имеют тип и их можно передавать в другие функции как аргумент. Создадим шаблон функции `RunTest`, который будет запускать тесты и ловить исключения.

```

...
template <class TestFunc>

void RunTest(TestFunc func, const string& test_name) { // передаём тест и его имя
    try {
        func();
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // ловим исключение
    }
}

int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
}

```

Заметим, что все тесты работают в любом порядке. Таким образом, мы с вами применили шаблон функций, для того чтобы передавать в качестве параметров функции другие функции, и смогли за счет этого написать универсальный такой шаблон, который для любого юнит-теста ловит исключения и позволяет нам все юнит-тесты, которые мы написали, выполнять при каждом запуске нашей программы.

Избавляемся от смешения вывода тестов и основной программы

Добавим в `RunTest` ещё одну удобную вещь: будем выводить OK снаружи каждого юнит-теста.

```
void RunTest(TestFunc func, const string& test_name) {
    try {
        func(); // заменим cout на cerr - стандартный поток ошибок
        cerr << test_name << " OK" << endl; // выводит OK, если всё работает
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // fail, если ошибка
    }
}
```

Всё это время сам алгоритм решения задачи «Синонимы» (который всё это время был закомментирован), всё ещё хорошо работает. Вот он сам:

```
int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
    int q;
    cin >> q;
    Synonyms synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word);
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
        }
    }
}
```

```

    cout << GetSynonymCount(synonyms, word) << endl;
} else if (operation_code == "CHECK") {
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
}
}

```

Вся проблема в том, что и юнит-тесты, и сама программа выводят в стандартный вывод. Пусть юнит-тесты выводят в `stderr` (стандартный поток ошибок). По окраске вывода в Eclipse можно отличать **стандартный вывод**, **стандартный ввод** и **стандартный поток ошибок**.

Вывод: **TestAreSynonyms OK**, **5 TestAddSynonyms OK**, **TestCount OK**, **1**, **COUNT a**, **0**. Теперь не нужно окружать комментарием эти части кода перед отправкой, ведь они всё равно не повлияют на работу самой программы.

Обеспечиваем регулярный запуск юнит-тестов

Мы хотим, чтобы юнит-тесты были автоматическими, и их код находился где-то отдельно. Кроме того, стоит считать ошибки, чтобы если существует хоть одна, то программа не ждала получения данных от пользователя. Таким образом, мы хотим:

1. Запускаем тесты при старте программы. Если хоть один тест упал, программа завершается;
2. Если все тесты прошли, то должно работать решение самой задачи.

Для этого обернём наш шаблон `RunTest` в класс `TestRunner`:

```

class TestRunner { // класс тестирования
public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {

```

```

    try { // RunTest стал шаблонным методом класса
        func();
        cerr << test_name << " OK" << endl;
    } catch (runtime_error& e) {
        ++fail_count; // увеличиваем счётчик упавших тестов
        cerr << test_name << " fail: " << e.what() << endl;
    }
}

~TestRunner() { // деструктор класса TestRunner, в котором анализируем fail_count
    if (fail_count > 0) { //это как раз тот момент, когда
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1); // завершение программы с кодом возврата 1
    }
}

private:
    int fail_count = 0; // счётчик числа упавших тестов
};

void TestAll() { // переместили все тесты в одну функцию
    TestRunner tr;
    tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
    tr.RunTest(TestCount, "TestCount");
    tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}

int main() {
    TestAll(); // т.к. мы деструктор класса объявили в самом классе,
    ...} // выполняется он в конце TestAll

```

Теперь, если мы словили хоть одну ошибку, то программа перестанет выполняться с кодом возврата 1. Если же ничего плохого не произошло, то мы можем ввести число команд и сами команды.

Собственный фреймворк юнит-тестов. Итоги

Подведём итоги написания собственного фреймворка. Его основные свойства:

- Если срабатывает `assert`, в консоль выводятся его аргументы (работает для контейнеров);
- Вывод тестов не смешивается с выводом основной программы;

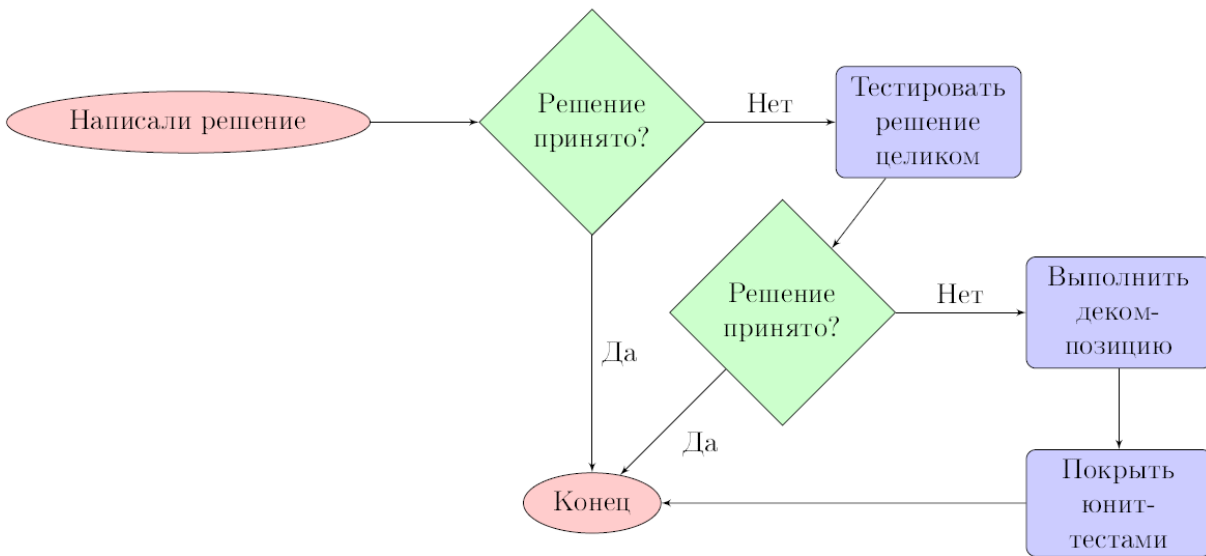
- При каждом запуске программы выполняются все юнит-тесты;
- Если хотя бы один тест упал, программа завершится с ненулевым кодом возврата.

Для того, чтобы пользоваться фреймворком, надо написать:

```
void TestSomething() { // функция, что-то тестирующая
    AssertEqual(..., ...);
    // выполняем какие-то проверки с помощью AssertEqual
}
void TestAll() {
    TestRunner tr;
    tr.RunTest(TestSomething, "TestSomething")
    // вызываем методом RunTest
}
int main() {
    TestAll(); // должна быть до самой программы
} // код фреймворка выложен рядом с видео
```

Общие рекомендации по декомпозиции программы и написанию юнит-тестов

Общий алгоритм решения задач с помощью декомпозиции и юнит-тестов:



Но кроме этой схемы, стоит выполнять декомпозицию задачи по ходу написания самого кода. Декомпозицию лучше делать сразу:

- Отдельные блоки проще реализовать и вероятность допустить ошибку ниже;
- Их проще тестировать, соответственно, выше вероятность найти ошибку или убедиться в её отсутствии;
- В больших проектах декомпозиция упрощает понимание и переиспользование кода;
- Уже реализованные функции можно брать и использовать в другом месте;
- Сама декомпозиция иногда защищает от ошибок.

Вспомним задачу «Уравнение» из курса «C++: белый пояс». Нужно было найти все различные действительные корни уравнения $Ax^2 + By^2 + C = 0$. Гарантируется, что $A^2 + B^2 + C^2 > 0$. Монолитное решение могло выглядеть так:

```

#include <cmath>
#include <iostream>
using namespace std;
int main() {

```



```
double a, b, c, D, x1, x2;
cin >> a >> b >> c;
D = b * b - 4 * a * c;
if ((a == 0 && c == 0) || (b == 0 && c == 0)) {
    cout << 0;
} else if (a == 0) {
    cout << -(c / b); // тут ошибка. Если b == 0, мы всё равно разделим на 0
} else if (b == 0) {
    cout << " ";
} else if (c == 0) {
    cout << 0 << " " << -(b / a);
} else if (D < 0) {
    cout << " ";
} else if (D == 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    cout << x1;
} else if (D > 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    x2 = ((-1 * b) - sqrt(D)) / (2 * a);
    cout << x1 << " " << x2;
}
return 0;
}
```

Быстро просмотрев этот код, сложно понять, работает он или нет. А в нём есть ошибка, которую из-за монолитности кода сложно заметить сразу. Теперь рассмотрим декомпозированное решение той же задачи:

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

void SolveQuadraticEquation(double a, double b, double c) {
    // ... тут решение гарантированного квадратного уравнения
}

void SolveLinearEquation(double b, double c) {
    // b * x + c = 0
    if (b != 0) { // тут не забыли проверить деление на 0
        cout << -c / b;
    }
}
```

```
}

int main() {
    double a, b, c;
    cin >> a >> b >> c;
    if (a != 0) { // точно знаем, что уравнение квадратное
        SolveQuadraticEquation(a, b, c);
    } else { // просто решаем линейное
        SolveLinearEquation(b, c);
    }
    return 0;
}
```

Юнит-тесты тоже лучше делать сразу. Причины:

- Разрабатывая тесты, вы сразу продумываете все варианты использования вашего кода и все крайние случаи входных данных;
- Тесты позволяют вам сразу проконтролировать корректность вашей реализации (особенно актуально для больших проектов);
- В больших проектах обширный набор тестов позволяет убедиться, что вы ничего не сломали во время дополнения кода.



Основы разработки на C++: жёлтый пояс

Неделя 3

Разделение кода по файлам



Оглавление

Разделение кода по файлам	2
2.1 Распределение кода по файлам	2
2.1.1 Введение в разработку в нескольких файлах на примере задачи «Синонимы»	2
2.1.2 Механизм работы директивы <code>#include</code>	9
2.1.3 Обеспечение независимости заголовочных файлов	11
2.1.4 Проблема двойного включения	12
2.1.5 Понятия объявления и определения	14
2.1.6 Механизм сборки проектов, состоящих из нескольких файлов	17
2.1.7 Правило одного определения	27
2.1.8 Итоги	29

Разделение кода по файлам

Распределение кода по файлам

Введение в разработку в нескольких файлах на примере задачи «Синонимы»

До этого мы весь код хранили в одном файле. Но в общем случае это приводит к проблемам:

1. Для использования одного и того же кода в нескольких программах его приходится копировать;
2. Даже самое маленькое изменение программы приводит к её полной перекомпиляции;

Как говорит автор языка C++ Бьёрн Страуструп в своей книге «Язык программирования C++»: «Разбиение программы на модули помогает подчеркнуть ее логическую структуру и облегчает понимание». Рассмотрим это всё на примере кода нашей программы из прошлой недели. Здесь у нас есть логически не связанные друг с другом вещи. Первый кусок – само решение задачи «Синонимы»:

```
using Synonyms = map <string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms,
    const string& first_word) {
    return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
```

```
const string& second_word) {  
    return synonyms[first_word].count(second_word) == 1;  
}
```

Далее идёт наш юнит-тест фреймворк:

```
template <class T>  
ostream& operator<<(ostream& os, const set<T>& s) {  
    os << "{";  
    bool first = true;  
    for (const auto& x : s) {  
        if (!first) {  
            os << ", ";  
        }  
        first = false;  
        os << x;  
    }  
    return os << "}";  
}  
  
template <class K, class V>  
ostream& operator<<(ostream& os, const map<K, V>& m) {  
    os << "{";  
    bool first = true;  
    for (const auto & kv : m) {  
        if (!first) {  
            os << ", ";  
        }  
        first = false;  
        os << kv.first << ": " << kv.second;  
    }  
    return os << "}";  
}  
  
template <class T, class U>  
void AssertEqual(const T& t, const U& u, const string& hint) {  
    if (t != u) {  
        ostringstream os;  
        os << "Assertion failed: " << t << " != " << u <<  
            " hint: " << hint;  
        throw runtime_error(os.str());  
    }  
}
```

```
void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}

class TestRunner {
public:
    template<class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {
        try {
            func();
            cerr << test_name << " OK" << endl;
        } catch (runtime_error & e) {
            ++fail_count;
            cerr << test_name << " fail: " << e.what() << endl;
        }
    }

    ~TestRunner() {
        if (fail_count > 0) {
            cerr << fail_count << " unit tests failed. Terminate" << endl;
            exit(1);
        }
    }

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

Весь юнит-тест фреймворк логически не зависит от функций, которые решают нашу задачу.

Следующая логически независимая часть программы – это сами юнит-тесты:

```
void TestAddSynonyms() {
{
    Synonyms empty;
    AddSynonyms(empty, "a", "b");

    const Synonyms expected = {
        {"a", {"b"}},
        {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Empty");
}
{
```

```
Synonyms synonyms = {
    {"a", {"b"}},
    {"b", {"a", "c"}},
    {"c", {"b"}}
};
AddSynonyms(synonyms, "a", "c");

const Synonyms expected = {
    {"a", {"b", "c"}},
    {"b", {"a", "c"}},
    {"c", {"b", "a"}}
};
AssertEqual(synonyms, expected, "Nonempty");
}
}

void TestCount() {
{
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty, "a"), 0u, "Syn. count for empty dict");
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2u, "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms, "b"), 1u, "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms, "z"), 0u, "Nonempty dict, count z");
}
}

void TestAreSynonyms() {
{
    Synonyms empty;
    Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
    Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
}
{
    Synonyms synonyms = {
```



```
    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
};
Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
}
}
void TestAll() { // объединяем запуск всех юнит-тестов
    TestRunner tr;
    tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
    tr.RunTest(TestCount, "TestCount");
    tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}
```

Ещё у нас есть main:

```
int main() {
    TestAll();

    int q;
    cin >> q;
    Synonyms synonyms;

    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            AddSynonyms(synonyms, first_word, second_word);
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << GetSynonymCount(synonyms, word) << endl;
        } else if (operation_code == "CHECK") {
```

```
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Теперь рассмотрим вынесение в отдельные файлы в Eclipse. Итак, у нас в программе есть 4 логически обособленных компонента:

1. Функции решения нашей задачи;
2. Юнит-тест фреймворк;
3. Сами юнит-тесты;
4. Решение нашей задачи в main.

И довольно логично отделить эти части друг от друга, поместив их в отдельные файлы. Открываем наш проект Coursera: *Window* → *Show View* → *C/C++ Projects* (как это сделано на рис. 2.1). Нажимаем на него **project name** → *New* → *Header File*. (см. 2.2) Вводим имя заголовочному файлу (test_runner.h) и у нас создаётся пустой файл test_runner.h.

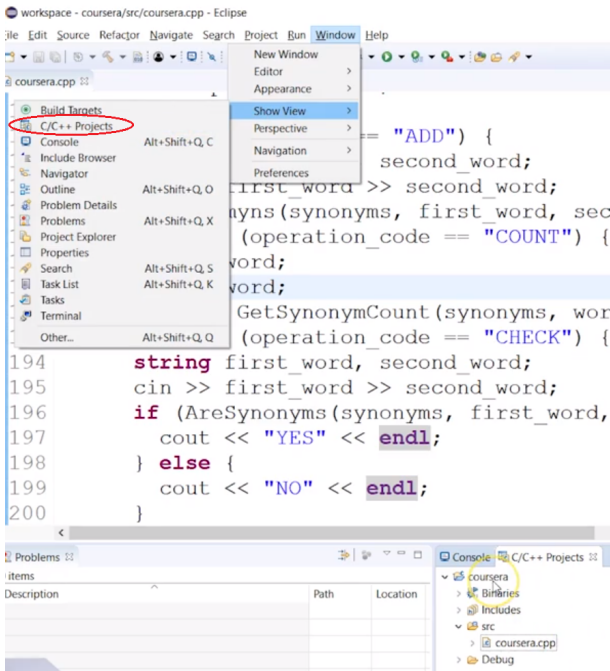


Рис. 2.1: Открытие C/C++ projects

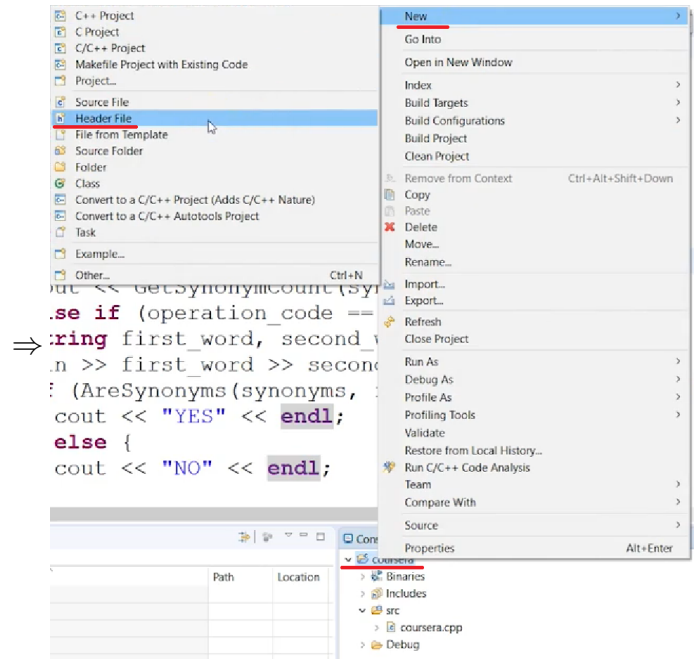


Рис. 2.2: New → Header File

Теперь из основного монолитного файла решения задачи «Синонимы» вырезаем сами юнит-тесты в `test_runner.h`. Теперь запустим нашу программу и она не скомпилируется, потому что мы, как минимум, не знаем, что такое **Assert**. Нам надо дописать в начало нашей программы

```
#include "test_runner.h" // подключаем файл с юнит-тестами
```

Теперь всё компилируется и работает. Аналогичным образом в файл `synonyms.h` вынесем функции самого решения задачи, а в файл `tests.h` вынесем все юнит-тесты и допишем:

```
#include "synonyms.h"
#include "tests.h"
```

Программа компилируется и тесты выполняются. Таким образом мы смогли разбить исходную программу на 4 файла, в каждом из которых лежат независимые блоки.

Механизм работы директивы `#include`

Несмотря на кажущуюся корректность в выполнении этих операций, у нас есть немало проблем. И давайте посмотрим, какие это проблемы. Для примера закомментируем `#include <set>` в начале нашей программы:

```
#include <cassert>
#include <sstream>
#include <exception>
#include <iostream>
#include <string>
#include <map>
#include <vector>
// #include <set> // закомментировали
using namespace std;

#include "test_runner.h"
#include "synonyms.h"
#include "tests.h"

int main() {
    ...
}
// 'AddSynonyms' was not declared in this scope...
```

И ещё несколько ошибок. Компилятор пишет, что мы не объявили функцию `AddSynonyms`, хотя мы её объявляли в `test_runner.h`. Перед нами встаёт проблема: мы не можем понять, где именно возникает ошибка.

Теперь посмотрим на другую. Поменять инклюды из начала мы можем без проблем. А вот если сделать подключение `tests` не третьим, а вторым, программа снова выдаст нам ошибку.

Третья демонстрация: если мы перенесём подключения в начало программы (например, между подключениями `sstream` и `exception`), снова появится куча ошибок о необъявленных переменных.

Разберёмся, как работает `#include`:

- Директива `#include "file.h"` вставляет содержимое файла `file.h` в месте использования;
- Файл, полученный после всех включений, подаётся на вход компилятору.

Разберёмся на примере маленького проекта Sum. У нас есть два файла: `how_include_works.cpp` с самой программой...

```
#include "sum.h"
int main() {
    int k = Sum(3, 4);
    return 0;
}
```

...в которой подключается `sum.h` с функцией суммирования:

```
int Sum(int a, int b) {
    return a + b;
}
```

Переключимся в консоль операционной системы. Зайдём в нашу директорию и увидим там два файла: `how_include_works.cpp` и `sum.h`. (рис. 2.3)

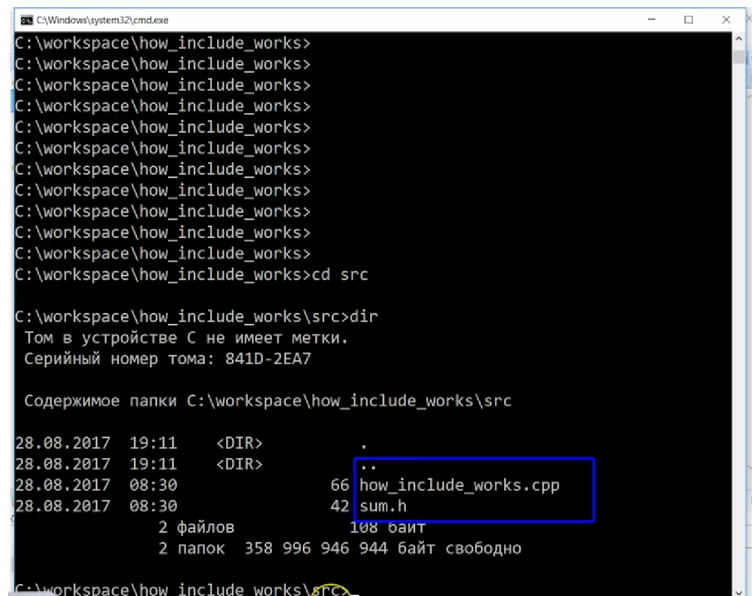


Рис. 2.3: Консоль cmd

Вызовем команду компилятора:

```
g++ -E how_include_works.cpp
```

Вызов компилятора с флагом `-E` значит, что мы просим компилятор не выполнять полную

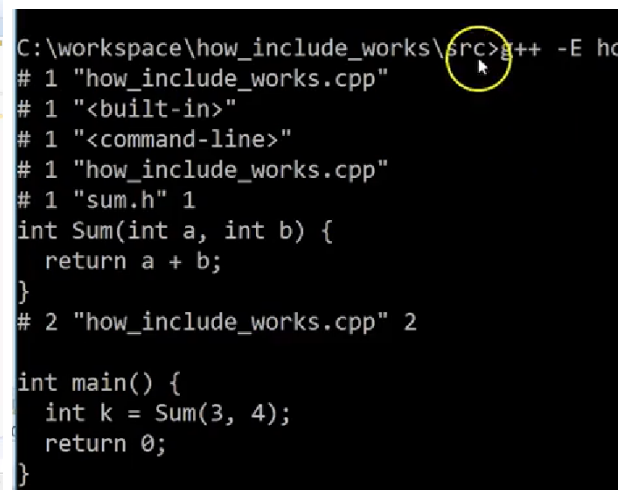


Рис. 2.4: Преппроессинг проекта

сборку проекта, а просто выполнить стадию препроцессинга (стадию выполнения директив `#include`). В итоге мы видим, что в файле есть функция `main`, а выше вставлен `sum.h` (рисунок 2.4). За символом `#` – уже служебные символы компилятора.

Теперь вернёмся к нашему большому проекту и посмотрим, как препроцессинг работает на нашем проекте тем же образом: в терминале `cmd.exe` переходим в директорию проекта и вводим:

```
g++ -E coursera.cpp > coursera.i
```

(чтобы результат препроцессинга вывелся в файл `coursera.i`)

Размер файла оказался 37980 строк после отрабатывания директив `include`. Содержимое каждого модуля было вставлено в файл с исходником. И само наше решение (`main` и все файлы, в которые мы до этого выносили части кода) начинается только с 37780 строки. А всё до этого – модули стандартных библиотек.

Отсюда и ответ на все те проблемы, которые мы получали: если мы убирали какую-то стандартную библиотеку, например `#include <set>`, нигде не было написано, что `set` – это множество, какие у него есть операции. И поэтому у нас возникала ошибка компиляции. При переносе тоже была ошибка, потому что в заголовочных файлах мы использовали функции и структуры, которые включались позже, и компилятор не мог их найти.

Обеспечение независимости заголовочных файлов

Избавляемся от одной из проблем, описанных выше. Нам надо, чтобы наши файлы были независимыми и порядок включения не влиял на компилируемость программы.

Решение: включим в каждый файл проекта те заголовочные файлы, которые ему нужны. Начнём с `test_runner.h`. Ему нужны `set`, `map`, `ostream` и `string`. Просто перенесём эти включения из основного файла в `test_runner.h`:

```
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std; // попытаемся добавить, хотя так делать не стоит
```

Программа компилируется. Тогда попробуем поставить `#include "test_runner.h"` самым первым в нашем основном файле. Но программа не компилируется, потому что файлу `synonyms.h` нужно знать `map`, `string`, `set`, а он об этом сейчас не знает, ведь мы подключаем файлы в данном порядке:

```
#include "synonyms.h"
#include "test_runner.h"

#include <exception>
#include <iostream>
#include <vector>

using namespace std;

#include "tests.h"
```

Раньше `synonyms.h` стоял после `test_runner.h` и получал из него все нужные `include`'ы. Теперь поставим и его (`synonyms.h`) вперёд и добавим все необходимые `include`'ы:

```
#include <map>
#include <set>
#include <string>
using namespace std; // попытаемся добавить, хотя так делать не стоит
```

Теперь всё компилируется.

Рассмотрим функцию `main()`. Он состоит из функции `TestAll()` и кода, который решает задачу. В этом конкретном файле мы нигде не используем наш фреймворк. Значит, `test_runner.h` нам в этом файле не нужен. Он нужен в `tests.h`, потому что именно они используют тестовый фреймворк. Таким образом мы сделали `test_runner.h` и `synonyms.h` независимыми, и подключать их можно в любом порядке до функции `main()`.

Проблема двойного включения

Функция `AddSynonyms()` в `tests.h` определена в `synonyms.h`, и если мы поставим `tests.h` перед `synonyms.h`, наш проект не скомпилируется. Тогда добавим в `tests.h` все зависимости, в частности, `synonyms.h` и скомпилируем:

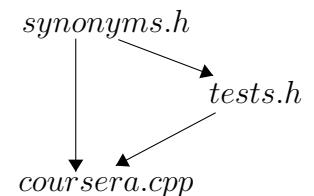
```
#include "test_runner.h"
```

```
#include "synonyms.h"
...
// redefinition of "bool AreSynonyms...."
```

Программа не компилируется, причём со странными ошибками о переопределении наших функций. Для того, чтобы понять, что произошло, вернёмся к маленькой задачке. Продублируем строчку `#include "sum.h"`.

```
#include "sum.h"
#include "sum.h"
int main() {
    int k = Sum(3, 4);
    return 0;
}
// redefinition of "int Sum...."
```

После компиляции увидим ту же ошибку. Теперь получим препроцессинг проекта, как на рисунках 2.3 и 2.4. Заметим, что в файле получилось две функции `sum`. Когда компилятор это видит, он выкидывает ошибку компиляции **Redefinition**, т.е. повторное определение. Точно та же ситуация у нас в большом проекте: в `coursera.cpp` включается `tests.h`, который подключает `synonyms.h`, который так же включается в `coursera.cpp`. Таким образом у нас получается переопределение всего `synonyms.h`.



Избежать двойного включения очень просто: добавляем в начало каждого заголовочного файла `#pragma once`. В нашем случае дописываем в `synonyms.h` (и уже сейчас всё заработает), `tests.h` и `test_runner.h`. Эта директива говорит компилятору игнорировать все повторные включения.

Также добавим в `sum.h` эту строчку и проверим, что всё работает. Выполним его препроцессинг и увидим, что функция `sum` там встречается только один раз. Здесь мог возникнуть вопрос: «Почему препроцессор не отслеживает, что заголовочные файлы включаются несколько раз, и почему препроцессор по умолчанию не выкидывает повторные включения?» Потому что C++ делался обратно совместимым с C, и вообще C++ развивается так, чтобы не терять обратную совместимость. Но мы можем забывать каждый раз прописывать эту строчку в каждом заголовочном файле, и тут нам на помощь приходит IDE. В Eclipse оно работает так: *Window* → *Preferences* → *C/C++* → *Code Style* → *Code Templates* → *Files* → *C++ Header File* → *Default C++ header template*, там нажимаем *Edit* и у нас открывается окно ввода шаблона, который будет вставляться во все заголовочные файлы, которые мы создаём. Сюда можно добавлять специальные макропеременные, которые вставляют ваше имя, дату создания файла, имя проекта и так далее. Но вот мы сюда прямо и напишем `#pragma once`, перевод строки. ОК, Apply,

Apply and Close. Снова создадим новый header-файл, как на рисунке 2.2. Как только мы его создали, он по умолчанию сразу идёт с вставленным шаблоном.

Понятия объявления и определения

Когда у нас есть большой проект, в котором много файлов, то мы, естественно, не можем помнить досконально, в каком файле какие функции есть. И очень часто хочется, открыв файл, понять интерфейс этого файла, то есть понять, какие функции и классы в этом файле есть. Т. е. зайти в файл и сразу увидеть его интерфейс. Нам придётся пролистать весь файл, чтобы понять, что за функции в нём есть. Хотелось бы короткий список функций. В Eclipse можно нажать Ctrl+O и получить краткий список с названиями и типами функций.

Иногда хочется видеть только интерфейс – список функций и классов, которые там есть. Нас не будет интересовать, как это работает (допускаем, что оно работает). Нас интересует только, что мы с ним можем делать. Введём два новых определения:

- **Объявление функции (function declaration)** – сигнатура функции (возвращаемый тип, имя функции и список параметров с типами). Оно говорит, что где-то в программе есть функция с заданными параметрами;

```
int GreatestCommonDivisor(int a, int b);
```

- **Определение функции (function definition)** – сигнатура + реализация функции.

```
int GreatestCommonDivisor(int a, int b) {  
    while (a > 0 && b > 0) {  
        if (a > b) {  
            a %= b;  
        } else {  
            b %= a;  
        }  
    }  
    return a + b;  
}
```

Функция может быть объявлена несколько раз, но определена должна быть только в одном месте. Ещё важно, чтобы все объявления функции были одинаковыми.

На простом примере разберёмся, как это работает. Итак, у нас есть

```
void foo() {
    bar();
}
void bar() {
}
int main() {
    return 0;
}
// "bar" was not declared in this scope
```

Функция `bar` не объявлена и файл не компилируется. Теперь в самом начале файла объявим функцию `bar`:

```
void bar(); // добавили в самое начало программы
```

Теперь всё заработало. И даже если объявлений будет много, программа будет компилироваться. А вот если мы продублируем определение, то всё сломается и мы получим `redefinition error`. Это было насчёт определения и объявления функций. Аналогично у нас будет и для классов:

- **Объявление класса (class declaration)** – объявление класса, его поля и методы. Но методы не реализованы:

```
class Rectangle {
public:
    Rectangle(int width, int height);
    int Area() const;
    int Perimeter() const;

private:
    int width, height;
};
```

- **Определение методов класса (class methods definition)**

```
Rectangle::Rectangle(int w, int h) { // по принципу: имя класса::имя метода
    width = w;
    height = h;
}
int Rectangle::Area() const {
```

```
    return width * height;
}
int Rectangle::Perimeter() const{
    return 2 * (width + height);
}
```

Теперь вспомним, а зачем оно нам: мы хотели в начале файла видеть объявления всех функций и классов, которые есть в файле. Сделаем это для нашего большого проекта. Допишем в tests.h:

```
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();
```

Теперь аналогично сделаем для synonyms.h и test_runner.h. Причём во второй у нас есть шаблоны функций, которые точно так же стоит объявить в начале:

```
void AddSynonyms(Synonyms& synonyms,
    const string& first_word, const string& second_word);
bool AreSynonyms(Synonyms& synonyms,
    const string& first_word, const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);
```

```
template <class T> // копируем объявление шаблонов
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name);
    ~TestRunner();

private:
```

```
int fail_count = 0;
};
```

Итоги:

- Объявление в начале файла сообщает компилятору, что функция/класс/шаблон где-то определены;
- Объявлений может быть несколько. Определение – только одно;
- Группировка объявлений в начале файла позволяет узнать, какие функции и классы в нём есть, не вникая в их реализацию.

Механизм сборки проектов, состоящих из нескольких файлов

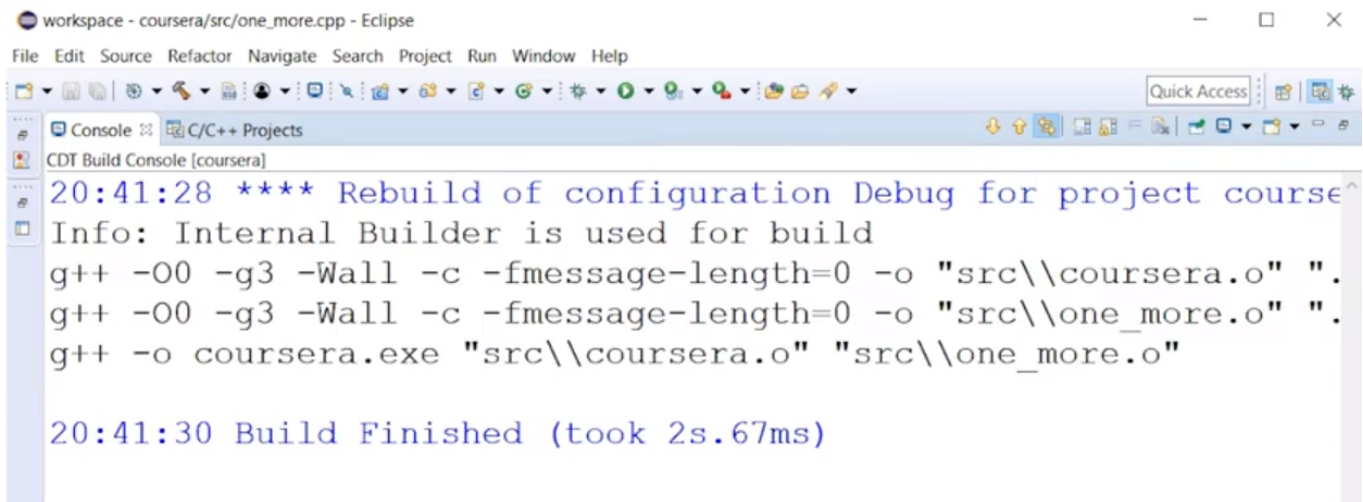
Когда мы начинали разговор о разделении кода на несколько файлов, то в качестве одного из недостатков хранения всего кода в одном файле мы называли то, что при минимальном изменении программы у нас она пересобирается вся, в случае, если весь код лежит в одном файле. Сейчас мы разделили код нашего проекта на целых четыре файла. Но при этом каждый раз, когда мы меняем что угодно в нашем проекте, он все равно пересобирается целиком. Почему это происходит? Потому что у нас есть файл `coursea.cpp`, в который так или иначе включаются с помощью директивы `#include` три других наших файла. Соответственно, если мы в них что-нибудь меняем, то они вставляются в наш `coursea.cpp`, и вся программа перекомпилируется целиком. Но давайте подумаем. Например, есть у нас функции в `"synonyms.h"`, которые умеют работать со словарём синонимов – добавлять в него, проверять количество синонимов. Есть эти функции и есть тесты на них. Если мы внесем изменения в тесты, например, добавим какой-нибудь ещё тестовый случай, например, в тест на `TestCount`, давайте там проверим, что при пустом словаре и для строки `b` у нас тоже вернётся ноль. Если мы поменяли тесты, то нам нет никакой необходимости перекомпилировать сами функции. Но мы все равно перекомпилируем всё.

Нам надо не пересобирать проект целиком при изменении в конкретном месте. Разберёмся в механике сборки проектов в C++. Посмотрим на расширения файлов в нашем проекте:

- `tests.h`
- `synonyms.h`

- `test_runner.h`
- `coursera.cpp`

Пока у нас 3 файла `.h` и только один файл `.cpp`. Добавим ещё один, как на картинке 2.2: *project name* → *New* → *Source File* и назовём его `one_more.cpp`. Очистим результаты сборки (*project name* → *Clean Project*) и соберём проект с нуля. Запустим сборку и после её завершения посмотрим, какие команды выполнял Eclipse в процессе сборки проекта:



```
workspace - coursera/src/one_more.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
CDT Build Console [coursera]
20:41:28 **** Rebuild of configuration Debug for project course
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\coursera.o" ".
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\one_more.o" ".
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"
20:41:30 Build Finished (took 2s.67ms)
```

Рис. 2.5: Команды по сборке проекта

Первый раз он запускался для файла `coursera.cpp`. Второй раз он запускался для нашего только что добавленного файла `one_more.cpp`. В результате было получено два файла с расширением `.o`. Вот этот параметр `-o` задает имя выходного файла, поэтому по значению параметра `-o` мы можем понимать, какие выходные файлы формировались в этой стадии. И потом была третья стадия, в которой на вход были поданы вот эти файлы с расширением `.o`, а на выходе получился исполняемый файл `coursera.exe`. Этот пример демонстрирует, каким образом выполняется сборка проектов на C++, состоящих из нескольких файлов.

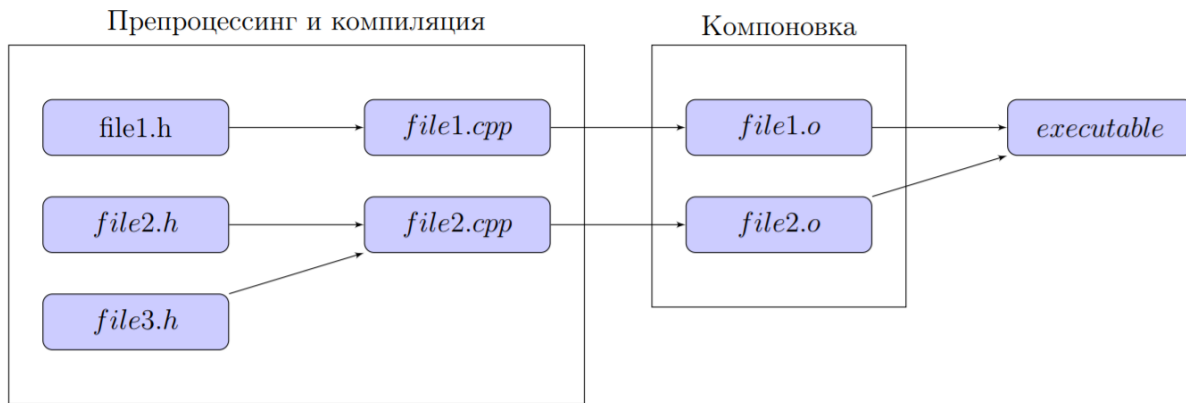


Рис. 2.6: Компиляция нескольких файлов

Как мы уже видели, первая стадия – это препроцессинг, когда выполняются все директивы `include`. Далее, после того как препроцессинг выполнен, берётся каждый отдельный `.cpp` файл и компилируется. В результате компиляции каждого `.cpp` файла получается так называемый объектный файл. Вот на схеме (рисунок 2.6) у нас объектные файлы изображены как файлы с расширением `.o`. И затем начинается третья стадия – это стадия компоновки, когда берутся все объектные файлы, которые у нас получились, и компонуются в один исполняемый файл.

Теперь, если мы в наш `one_more.cpp` добавим какое-нибудь изменение, например, комментарий, запустим сборку и посмотрим на команды консоли, видим, что теперь вместо всего проекта перекомпилировался только `one_more.cpp` и потом был собран исходный файл `coursera.exe`. Аналогично внесём изменения в `coursera.cpp` и запустим сборку. В консоли увидим, что `one_more.cpp` не был тронут, и перекомпилировался только `coursera.cpp`. Если мы изменим любой из `.h` файлов, подключаемых в `coursera.cpp`, увидим то же самое.

```

20:44:29 **** Incremental Build of configuration Debug for proj^
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\one_more.o" ".
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"

20:44:29 Build Finished (took 590ms)

```

Рис. 2.7: Сообщения в консоли

Вывод:

1. При сборке проекта компилируются только изменённые **.cpp**-файлы;
2. Внесённые изменения в **.h** файл приводит к перекомпиляции всех **.cpp**-файлов, в которые он включён;
3. Если перенести определения функций и методов классов в **.cpp**-файлы, то они будут пересобираются только после изменений.

Теперь используем эти знания на нашем проекте, чтобы при небольшом изменении наш код реже пересобирался. Определения функций и методов классов переносим в .cpp файлы, а в заголовочных файлах оставляем только объявления. Мы логически не связанные друг с другом определения разнесём в разные файлы. Когда мы меняем, например, определения тестов, то определения функций, которые эти тесты покрывают, не меняются, и соответственно, они не будут перекомпилироваться. Таким образом мы минимизируем количество .cpp-файлов, которые нужно перекомпилировать при каждом изменении программы.

Давайте выполним такое преобразование с нашим проектом, то есть вынесем определение в .cpp-файл. И начнем вот, например, с test_runner.h. Добавим в наш проект файл test_runner.cpp. И вынесем в него определения. Здесь есть нюанс (далее в курсе мы это разберём) с шаблонами, так что пока их переносить в .cpp-файл мы не будем.

```
#include "test_runner.h"
void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}
TestRunner::~TestRunner() {
    if (fail_count > 0) {
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1);
    }
}
```

Таким же образом с synonyms.cpp и tests.cpp:

```
#include "synonyms.h"
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    synonyms[second_word].insert(first_word);
}
```

```

    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word) {
    return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

```

```

#include "tests.h"
void TestAddSynonyms() {
    {
        Synonyms empty;
        AddSynonyms(empty, "a", "b");
        const Synonyms expected = {
            {"a", {"b"}},
            {"b", {"a"}},
        };
        AssertEqual(empty, expected, "Empty");
    }
    {
        Synonyms synonyms = {
            {"a", {"b"}},
            {"b", {"a", "c"}},
            {"c", {"b"}}
        };
        AddSynonyms(synonyms, "a", "c");
        const Synonyms expected = {
            {"a", {"b", "c"}},
            {"b", {"a", "c"}},
            {"c", {"b", "a"}}
        };
        AssertEqual(synonyms, expected, "Nonempty");
    }
}

void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"), 0,

```



```
    "Syn. count for empty dict");
    AssertEqual(GetSynonymCount(empty, "b"), 0u,
        "Syn. count for empty dict b");
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2u,
        "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms, "b"), 1u,
        "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms, "z"), 0u,
        "Nonempty dict, count z");
}
}

void TestAreSynonyms() {
    {
        Synonyms empty;
        Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
        Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
    }
    {
        Synonyms synonyms = {
            {"a", {"b", "c"}},
            {"b", {"a"}},
            {"c", {"a"}}
        };
        Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
        Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
        Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
        Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
        Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
        Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
    }
}

void TestAll() {
```

```

TestRunner tr;
tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
tr.RunTest(TestCount, "TestCount");
tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}

```

А в самих .h файлах у нас остаётся:

```

#pragma once
#include "test_runner.h"
#include "synonyms.h"
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();

```

```

#pragma once
#include <map>
#include <set>
#include <string>
using namespace std;
using Synonyms = map<string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word);
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
    const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);

```

```

#pragma once
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std;

template <class T>
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);

```

```
template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
public:
    template <class TestFunc>
    void RunTest (TestFunc func, const string & test_name);
    ~TestRunner();
private:
    int fail_count = 0;
};

template <class T>
ostream& operator << (ostream& os, const set<T>& s) {
    os << "{";
    bool first = true;
    for (const auto& x : s) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << x;
    }
    return os << "}";
}

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m) {
    os << "{";
    bool first = true;
    for (const auto & kv:m) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << kv.first << ": " << kv.second;
    }
    return os << "}";
}
```

```
}

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint) {
    if (t != u) {
        ostringstream os;
        os << "Assertion failed: " << t << " != " << u << " hint: " << hint;
        throw runtime_error(os.str());
    }
}

template <class TestFunc>
void TestRunner::RunTest (TestFunc func, const string& test_name) {
    try {
        func ();
        cerr << test_name << " OK" << endl;
    } catch (runtime_error & e) {
        ++fail_count;
        cerr << test_name << " fail: " << e.what () << endl;
    }
}
```

Вспомним, что у нас есть и основной файл с решением:

```
#include "tests.h"
#include "synonyms.h"
#include <exception>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    TestAll();
    int q;
    cin >> q;
    Synonyms synonyms;
    for (int i = 0; i < q; ++i) {
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") {
            string first_word, second_word;
```

```
    cin >> first_word >> second_word;
    AddSynonyms(synonyms, first_word, second_word);
} else if (operation_code == "COUNT") {
    string word;
    cin >> word;
    cout << GetSynonymCount(synonyms, word) << endl;
} else if (operation_code == "CHECK") {
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Таким образом, весь наш проект представляет собой 7 файлов:

1. **coursera.cpp** – главный файл с `main()`, в котором лежит решение нашей задачи;
2. **synonyms.h** – объявления функций, решающих нашу задачу и **synonyms.cpp** – определения этих самых функций;
3. **test_runner.h** и **test_runner.cpp** – определения и объявления функций и классов, связанных с юнит-тестированием;
4. **tests.h** – объявления тестирующих функций и **test.cpp** – их определение.

Если внести изменения в `test_runner.h`, то у нас пепесоберётся всё: `test_runner.cpp`, `tests.cpp`, `coursera.cpp`. Потому что `coursera.cpp` включает в себя `test.h`, который включает в себя `test_runner.h`. Таким образом:

1. Сборка проектов состоит из трёх стадий: препроцессинг, компиляция и компоновка;
2. При повторной сборке проекта компилируются только изменённые `.cpp`-файлы;

3. Внесение определений в .cpp-файлы позволяет при каждой сборке компилировать только изменённые файлы;
4. Это сильно ускоряет пересборку проекта.

Правило одного определения

Мы ранее говорили, что объявлений может быть сколько угодно, а определение обязательно должно быть ровно одно. И давайте мы ещё раз это продемонстрируем: вот у нас есть функция, например, `GetSynonymCount`, и у неё есть определение в файле `synonyms.cpp`. Если мы просто возьмём и скопируем это определение, а потом запустим компиляцию, то мы получим знакомую ошибку `redefinition`. Однако в больших проектах бывают ситуации, когда в вашем проекте, вроде бы, есть всего одно определение функции, но при этом компилятор сообщает вам, что у вас одна и та же функция определена несколько раз. И давайте посмотрим, как это выглядит и по какой причине случается. Давайте, например, возьмём нашу функцию `GetSynonymCount` и перенесём её определение обратно в заголовочный файл, как было у нас несколькими видео ранее. И скомпилируем наш проект. Давайте мы его соберём. И что-то пошло не так. Нам компилятор написал `first defined here`. А в консоли увидим `"multiple definition of GetSynonymsCount"`. Вроде определение функции одно, но ошибка возникает. Посмотрим на строки запуска компилятора:

```
21:09:19 **** Incremental Build of configuration Debug for proj
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\coursera.o" ".
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\tests.o" "..\
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\synonyms.o" ".
g++ -o coursera.exe "src\\coursera.o" "src\\synonyms.o" "src\\t
src\\synonyms.o: In function `std::_Rb_tree<std::__cxx11::basic_
c:/dev/mingw-w64/mingw64/lib/gcc/x86_64-w64-mingw32/7.1.0/inclu
src\\coursera.o:C:\workspace\coursera\Debug\../src/synonyms.h:18
src\\tests.o: In function `std::_Rb_tree<std::__cxx11::basic_str
C:\workspace\coursera\Debug\../src/synonyms.h:18: multiple defi
src\\coursera.o:C:\workspace\coursera\Debug\../src/synonyms.h:18
collect2.exe: error: ld returned 1 exit status

21:09:25 Build Finished (took 5s.944ms)
```

Рис. 2.8: Настройка компилятора

Каждый .cpp файл успешно скомпилировался. На этапе компоновки возникает ошибка.

Multiple definition of GetSynonymCount

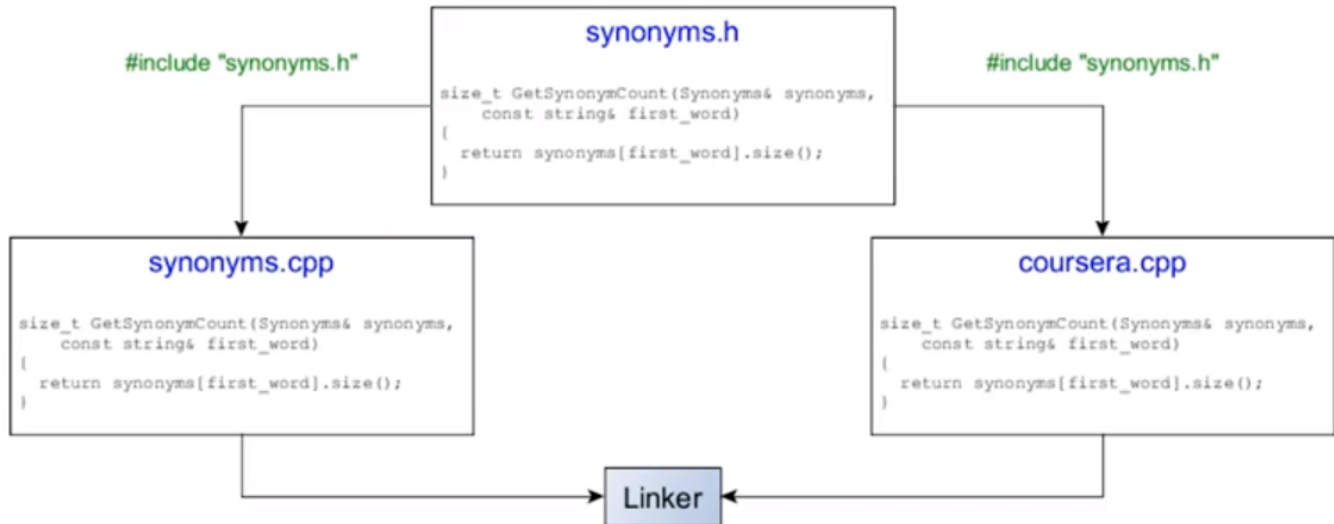


Рис. 2.9: Схема компиляции и сборки проекта

Ошибка происходит, когда компоновщик видит два определения одной и той же `GetSynonymCount` уже на этапе компоновки двух разных `.cpp`-файлов. Он видит, что одна и та же функция определена в двух объектных файлах и сообщает об ошибке. Вспомним, что основной причиной разделения на файлы было ускорение сборки. Теперь у нас есть ещё одна причина помещать определения в `.cpp`-файлы – это позволяет избежать ошибки `Multiple definitions`. Таким образом:

1. В C++ есть One Definition Rule (ODR);
2. Если функция определена в `.h`-файле, который включается в несколько `.cpp`-файлов, то нарушается ODR;
3. Чтобы не нарушать ODR, все определения надо помещать в `.cpp`-файлы.

Итоги

1. Разбиение программы на файлы упрощает её понимание и переиспользование кода, а также ускоряет перекомпиляцию;
2. В C++ есть два типа файлов: заголовочные (чаще .h) и файлы реализации .cpp;
3. Включение одного файла в другой осуществляется с помощью директивы `#include`;
4. Чтобы избежать двойного включения, надо добавлять `#pragma once`;
5. Знаем, что такое объявления и определения. Объявлений может быть сколько угодно, а определение только одно (ODR);
6. В .h-файлы обычно помещают объявления, а в .cpp – определения;
7. Если помещать определения в .h-файлы, то возможно нарушение ODR на этапе компоновки.