

# Содержание

<b>1</b>	<b>Целочисленные типы, кортежи и шаблоны функции</b>	<b>2</b>
1.1	Целочисленные типы . . . . .	2
1.1.1	Введение в целочисленные типы . . . . .	2
1.1.2	Преобразования целочисленных типов . . . . .	4
1.1.3	Безопасное использование целочисленных типов . . . . .	6
1.2	Кортежи и пары . . . . .	8
1.2.1	Упрощаем оператор сравнения . . . . .	8
1.2.2	Кортежи и пары . . . . .	9
1.2.3	Возврат нескольких значений из функций . . . . .	11
1.3	Шаблоны функций . . . . .	13
1.3.1	Введение в шаблоны . . . . .	13
1.3.2	Универсальные функции вывода контейнеров в поток . . . . .	15
1.3.3	Рефракторим код и улучшаем читаемость вывода . . . . .	17
1.3.4	Указание шаблонного параметра-типа . . . . .	18

# Неделя 1

## Целочисленные типы, кортежи и шаблоны функции

### 1.1 Целочисленные типы

#### 1.1.1 Введение в целочисленные типы

Вы уже знаете один целочисленный тип — это тип `int`. Начнём с проблемы, которая может возникнуть при работе с ним. Для этого вспомним задачу «Средняя температура» из первого курса. Что там нужно было сделать? 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  $(8 + 7 + 3)/3 = 6$ .

Листинг 1.1: Средняя температура

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> t = {8, 7, 3}; //вектор с наблюдениями
    int sum = 0; //завели переменную с суммой
    for (int x : t) { //проитерировались по вектору и нашли суммарную температуру
        sum += x;
    }
    int avg = sum / t.size(); //поделили на размер вектора 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 неотрицательные. 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}; //сумма -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 = 2'000'000000; //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 ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.
2. Возможны проблемы с беззнаковыми типами. Если бы в задаче 1.1 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. Числа по умолчанию имеют тип `int` *auto x = 1;*
  2. Эффективен: операции с ним напрямую транслируются в инструкции процессора.
  3. В зависимости от архитектуры имеет размер 4 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 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 `#include <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?
<b>int</b>	4 (обычно)	$-2^{31}$	$2^{31} - 1$	по умолчанию
<b>unsigned int</b>	4 (обычно)	0	$2^{32} - 1$	только положительные
<b>size_t</b>	4 или 8	0	$2^{32} - 1$ или $2^{64} - 1$	индексы контейнеров
<b>int8_t</b>	1	$-2^7$	$2^7 - 1$	сильно экономить память
<b>int16_t</b>	2	$-2^{15}$	$2^{15} - 1$	экономить память
<b>int32_t</b>	4	$-2^{31}$	$2^{31} - 1$	нужно ровно 32 бита
<b>int64_t</b>	8	$-2^{63}$	$2^{63} - 1$	недостаточно int

### Узнаём размеры и ограничения типов

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

Узнаем ограничения типов:

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе

using namespace std;

int main(){
    cout << sizeof(int16_t) << ", ";
    cout << numeric_limits<int>::min() << ", " << numeric_limits<int>::max() << endl;
    return 0;
}
//4, -2147483648, 2147483647
```

### 1.1.2 Преобразования целочисленных типов

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

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main(){
    cout << numeric_limits<int>::max() + 1 << endl;
    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  $1\,000\,000\,000 + 2\,000\,000\,000$ .

```
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 = -x;
cout << x << ", " << y <<"; " << -x << ", " << z << endl;
//2000000000, 2000000000; -2000000000, 2294967296
```

Если значение помещается 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 1.1, посмотрим, в чем 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;
```

Перед сравнениями и арифметическими операциями числа приводятся к общему типу

### Правила вывода общего типа

1. Все типы размера меньше `int` приводятся к `int`
2. Из двух типов выбирается больший по размеру
3. Если размер одинаковый, выбирается беззнаковый

	Слева	Операция	Справа	Общий тип	Комментарий
	<code>int</code>	<code>/</code>	<code>size_t</code>	<code>size_t</code>	больший размер
Примеры:	<code>int32_t</code>	<code>+</code>	<code>int8</code>	<code>int32_t (int)</code>	тоже больший размер
	<code>int8_t</code>	<code>*</code>	<code>uint8_t</code>	<code>int</code>	все меньшие приводятся к <code>int</code>
	<code>int32_t</code>	<code>&lt;</code>	<code>uint32_t</code>	<code>uint32_t</code>	знаковый к беззнаковому

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

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

И получим ошибку, в логе которой указано, что наше выражение имеет тип `size_t`

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. Второй пример аналогичен первому: в нём тоже сравнивается -1 типа int и 1 типа unsigned.

### 1.1.3 Безопасное использование целочисленных типов

#### Настроим компилятор

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

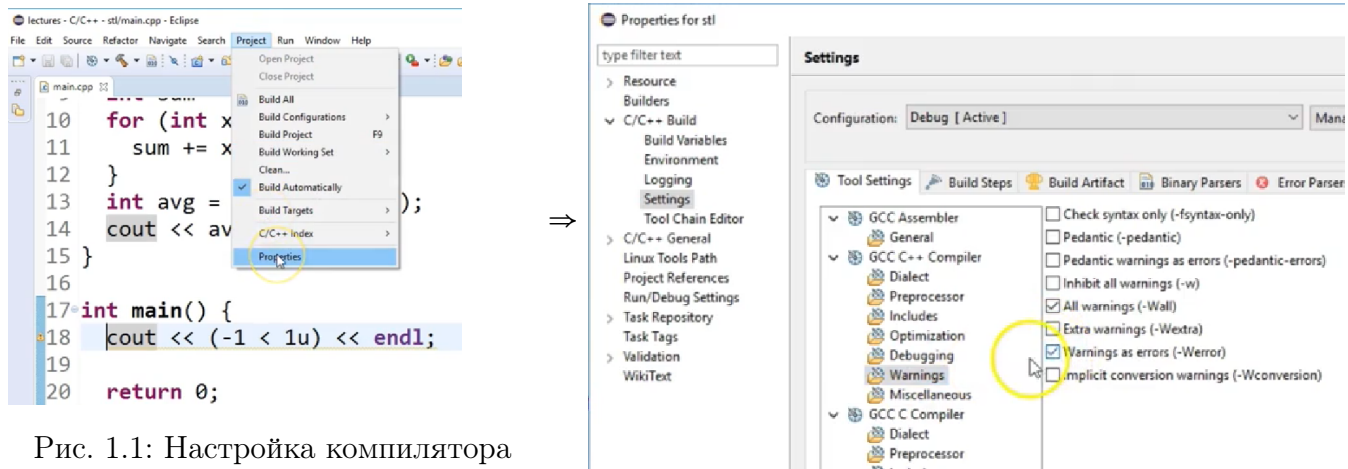


Рис. 1.1: Настройка компилятора

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

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

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

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

## 1.2 Кортежи и пары

### 1.2.1 Упрощаем оператор сравнения

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

Можно сделать это так:

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

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

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

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

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

Выигрыш в том, что для векторов лексикографический оператор сравнения уже определён:



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

Действительно, первая дата больше второй и код выводит 0. Но тип `vector` слишком мощный: он позволяет выполнять `push back`, удалять из середины, и пр. Использование этого типа в данном случае избыточно. Нам нужно всего лишь объединить три значения в одно, для левой даты и для правой даты, объединить их в одну и сравнить. Кроме того, он работает только при однотипных элементах. Если бы у нас месяц был строкой, векторы бы не подошли.

**Как же объединить разные типы в один набора?** 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, какой тип имеют `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 определён для std::tuple<const int&, const std::__cxx11::basic_string ...
```

Т.е. они имеют тип `tuple<const int&, const string&, const int&>`.

`Tuple` — это **кортеж**. Т.е. структура из ссылок на нужные нам данные (возможно, разнотипные).

### 1.2.2 Кортежи и пары

Создадим кортеж из трёх элементов:

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

```
using namespace std;
int main() {
    //tuple<int, string, bool> t(7, "C++" , true); // просто создаём 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); //так тоже можно создать кортеж
auto t = make_tuple(7, "C++", true);
cout << get<1>(t) << endl;
return 0;
}
//C++
```

С помощью функции `get<1>(t)` мы получили 1ый (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 (рис. 1.1) идём в *Project* → *Properties* → *C/C++ Build* → *Dialect* → *Other dialect flags* и пишем `'--std=c++1z'` 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 словарю. 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 набор пар 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

```

Все скомпилировалось, значит мы можем итерироваться по словарю, не создавая пар.

### 1.2.3 Возврат нескольких значений из функций

Возврат нескольких значений из функций — еще 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 из 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"};
        }
        //Выводит значение, нашлась ли единственная страна для города.
    } //И сообщение: либо страна не нашлась, либо их несколько
private:
    map<string, string> city_to_country; //по названию города храним название страны
    set<string> ambiguous_cities; //множество городов, принадлежащих нескольким странам
}

```

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

Мы научились возвращать из метода несколько значений с помощью кортежа. По новому стандарту можно получить кортеж и распаковать его в пару переменных:

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

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, 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 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 операторов сравнения или вернуть несколько значений из функции. Пары — это частный случай кортежа, у которых понятные названия полей и к которым удобно обращаться.

## 1.3 Шаблоны функций

### 1.3.1 Введение в шаблоны

Рассмотрим шаблоны функций на примере функции возведения числа в квадрат.

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

using namespace std;

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

int main() {
    cout << Sqr(2) << endl; //результат функции выведем в поток
    return 0;
}
//4
```

Теперь мы хотим возводить в квадрат дробные числа. Получается неправильный ответ.

```
... cout << Sqr(2.5) << endl; //результат функции выведем в поток
//4
```

Напишем функцию для работы с дробными числами.

```
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
```

Тип `T` компилятор выведет сам, чтобы типы поддерживали умножение. Нужно 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); //создаём пару
    cout << Sqr(p) << endl; //пытаемся возвести её в квадрат
    return 0;
}
//no match for 'operator*' (operand types are std::pair<int,int> and std::pair<int,int>)
```

Видим ошибку так как для оператора умножения не определены аргументы пара и пара. Тогда напомним шаблонный оператор умножения для пар:

```
using namespace std;
template <typename First, typename Second>
//Т.к. умножение 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;
}
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); //создаём 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 дублирования и сильно сокращать код.

### 1.3.2 Универсальные функции вывода контейнеров в поток

В курсах ранее мы часто печатали содержимое наших контейнеров, будь то вектор или `map`, на экран. Для этого мы либо определяли специальную функцию, либо перегружали оператор вывода в поток. Давайте это сделаем и сейчас:

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
//напишем свой оператор вывода в поток вектора целых типов
ostream& operator<< (ostream& out, const vector<int>& vi) {
    for (const auto& i : vi) { //проитерируем по вектору
        out << i << ' '; //выведем все элементы в поток
    }
    return out;
}

int main() {
    vector<int> vi = {1,2,3};
    cout << vi << endl;
}
//1 2 3
```

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 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 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() {
    vector<double> vi = {1.4, 2, 3}; //дробные числа
    cout << vi << 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`.

#### Листинг 1.2: шаблоны для `pair` `vector` 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 map
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
```

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 — круглыми скобками.

### 1.3.3 Рефакторим код и улучшаем читаемость вывода

Исправим предыдущую программу (1.2), и сделаем её вывод более читаемым. Нужно создать универсальную шаблонную функцию, которая на вход будет принимать некую коллекцию, по которой можно итерироваться с помощью цикла range-based for. И на вход этой функции будем передавать некий разделитель, через который надо вывести элементы нашей коллекции. Единственное, что мы требуем от входной коллекции, это условия: по ней можно итерироваться с помощью цикла range-based for, и ее элементы можно выводить в поток.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>

using namespace std;

template <typename Collection> //тип коллекции
string Join(const Collection& c, char d) { //передаём 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 - кладём в поток разделитель
        }
        first = false; //т.к. следующий элемент точно не будет первым
        ss << i; //кладём 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 изменим код и добавим скобочки
ostream& operator<< (ostream& out, const vector<T>& vi) {
    return out << '[' << Join(vi, ',') << ']';
} //оператор вывода возвращает ссылку на поток
template <typename Key, typename Value> //для map убрали аналогично vector
ostream& operator<< (ostream& out, const map<Key, Value>& m) {
    return out << '{' << Join(m, ',') << '}'; //и добавили фигурные скобочки
}

int main() {
    vector<double> vi = {1.4,2,3};
    pair<int, int> m1 = {1,2};
    map<double, double> m2 = {{1.4,2.1} , {3.4,4}};
    cout << vi << ';' << m1 << ';' << m2 << endl;
}
//[1.4,2,3];(1,2);{(1.4,2.1),(3.4,4)}

```

Всё работает. Наша программа вывела сначала вектор, потом пару и затем map. Для более сложных конструкций она тоже будет работать. Например вектор векторов:

```

int main() {
    vector<vector<int>> vi = {{1,2}, {3,4}};
    cout << vi << endl;
}
//[[1,2],[3,4]]

```

В итоге всё работает и на сложных контейнерах. Таким образом мы сильно упростили наш код и избежали ненужного дублирования с помощью шаблонов и универсальных функций.

### 1.3.4 Указание шаблонного параметра-типа

Рассмотрим случай, когда компилятор не знает, как на основе вызова шаблонной функции вывести тип T, на примере задаче 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

```

Но программа работает только для целых чисел.

```

cout << Max(2,3.5) << endl;
//deduce conflicting types for parametr 'T' (int and double)

```

То есть 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 привёлся 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 итог:

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 ипользуется как и любой другой тип
5. Выведение шаблонного типа может происходить либо автоматически, на основе аргументов, либо спомощью явного указания в угловых скобках ( `std::max<double>(2, 3.5)`)
6. Цель шаблонных функций: сделать код короче (избавившись от дублирования) и универсальнее.