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

1.1. Целочисленные типы

1.1.1. Введение в целочисленные типы

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

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

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

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

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

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

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

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

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

Особенности целочисленных типов языка C++:

1. В языке C++ память для целочисленных типов ограничена. Если вам не нужны целые числа размером больше 2 миллиардов, язык C++ для вас выделит вот ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.

2. Некоторые целочисленные типы языка C++ беззнаковые. Используя их, вы сможете хранить больше положительных значений, но не сможете хранить отрицательные.

Виды целочисленных типов:

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

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

Узнаём размеры и ограничения типов:

Как узнать размеры типа? Очень просто:

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

Узнаём ограничения типов:

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

1.1.2. Преобразования целочисленных типов

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

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

Получилось, что `max + 1 = min`, а `min - 1 = max`. Теперь попробуем вычислить среднее арифметическое $1000000000 + 2000000000$.

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

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

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

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

Если значение умещается 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 выведения 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
`int`	/	`size_t`	`size_t`	больший размер
`int32_t`	+	`int8_t`	`int32_t` (`int`)	тоже больший размер
`int8_t`	*	`uint8_t`	`int`	все меньшие приводятся к `int`
`int32_t`	<	`uint32_t`	`uint32_t`	знаковый к беззнаковому

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

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

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

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

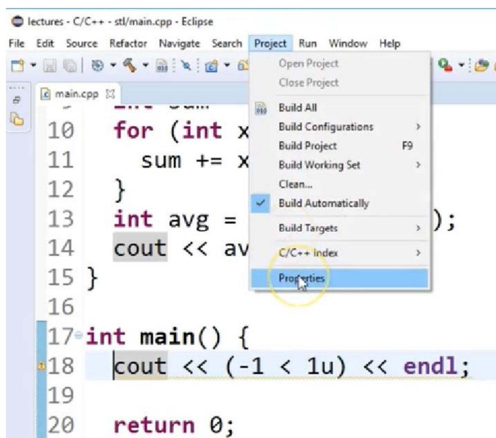
Как было сказано ранее, при операции между знаковым и беззнаковым типом обе переменные приводятся к беззнаковому. `-1`, приведённая к беззнаковому, становится очень большим числом, большим `1`. Суффикс `u` также приводит `1` к `unsigned`, а операция `<` теперь сравнивает `unsigned` `-1` и `1`. Причём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

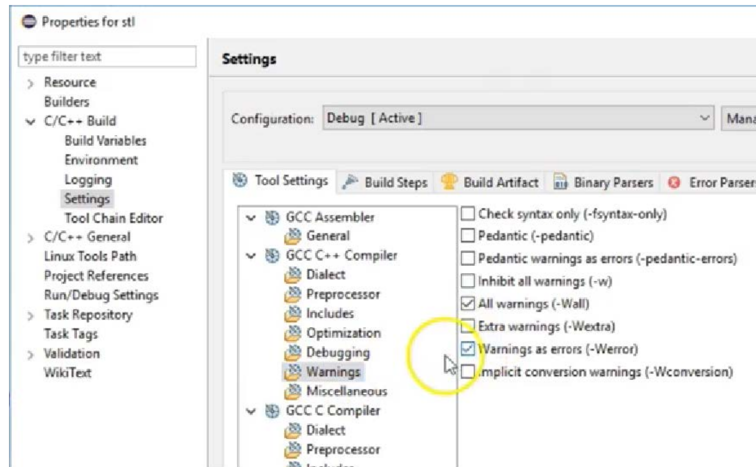
1.1.3. Безопасное использование целочисленных типов

Настроим компилятор:

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

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






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

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

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

Исправляем задачу о температуре:

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

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

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

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

Ещё примеры опасностей с беззнаковыми типами:

Переберём в векторе все элементы кроме последнего:

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

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

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

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

```
int main() {
```

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

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

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

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

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

1.2. Кортежи и пары

1.2.1. Упрощаем оператор сравнения

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

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

```
using namespace std;

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2. Кортежи и пары

Создадим кортеж из трёх элементов:

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

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

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

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

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

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

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

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

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

В результате нам вывело нашу пару. Эту структуру мы уже видели при итерировании по словарю. 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 словарю, не создавая пар.

1.2.3. Возврат нескольких значений из функций

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

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

class Cities {  // класс городов и стран
  public:
    tuple<bool, string> FindCountry(const string& city) const {
      if (city_to_country.count(city) == 1) {
        return {true, city_to_country.at(city)};
        // city_to_country[city] выдало бы ошибку, потому что могло нарушить const словаря
      } else if (ambigious_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> ambigious_cities;
}
int main() {
  Cities cities;
  bool success;
  string message;  // свяжем кортежем ссылок
  tie(success, message) = cities.FindCountry("Volgograd");
  cout success << " " << message << endl;  // вывели результат
  return 0;
 }
//0 Not exists
```

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

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

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

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

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

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

1.3. Шаблоны функций

1.3.1. Введение в шаблоны

Рассмотрим шаблоны функций на примере функции возведения числа в квадрат.

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

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

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

Функция работает. Теперь мы хотим возводить в квадрат дробные числа, например, `2.5`. Получается снова 4. Это неправильно.

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

Это происходит потому, что у нас нет аналогичной функции для работы с дробными числами. Заведём её:

```
int Sqr(int x) {  // функция возведения целого числа в квадрат
  return x * x;
}
double Sqr(double x) {  // функция возведенияя 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.

1.3.2. Универсальные функции вывода контейнеров в поток

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

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
// напишем свой оператор вывода в поток вектора целых типов
ostream& operator<< (ostream& out, const vector<int>& vi) {
  for (const auto& i : vi) {  // 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 поменяем тип вектора с `int` на шаблонный `T`:

```
using namespace std;
template <typename T> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const vector<T>& vi) {  // вектор на шаблон
  for (const auto& i : vi) {
    out << i << ' ';
  }
  return out;
}
int main() {
  vector<double> vi = {1.4, 2,3};  // дробные числа
  cout << vi << endl;
}
// 1.4 2 3
```

Ошибки пропали, вывелся наш вектор из дробных чисел. Мы научились универсальным способом решать задачу для вектора. Таким же универсальным способом научимся решать задачу для других контейнеров.

```
int main() {
  map<int, int> m = {{1, 2}, {3, 4}};
  cout << m << endl;
}
// no match for 'operator <' (operand types are std::ostream and std::map<int, int>)
```

Видим, что оператор вывода для `map` не определён. Определим его так же, как и для вектора:

```
...
template <typename Key, typename Value> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
  for (const auto& i : vi) {
    out << i << ' ';
  }
  return out;
}
...
int main() {
  map<int, int> m = {{1, 2}, {3, 4}};
  cout << m << endl;
}
// no match for 'operator <' (operand types are std::ostream and std::pair<const int, int>)
```

Заметим, что ошибка для map имеет интересный вид: `pair<const int, int>`. Действительно, ведь `map` – это `pair`, в которой `key` нельзя модифицировать, а `value` можно. Получается, нам достаточно определить оператор вывода в поток для пары. Тогда мы сможем вывести и `map`.

```
// весь рабочий код
...
template <typename First, typename Second>  // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
  out << p.first << ", " << p.second;
  return out;
}
template <typename T>  // для vector
ostream& operator<< (ostream& out, const vector<T>& vi) {
  for (const auto& i : vi) {
    out << i << ' ';
  }
  return out;
}
template <typename Key, typename Value>  // для пар
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
  for (const auto& i : vi) {
    out << i << ' ';
  }
  return out;
}
```

```
int main() {
  // vector<double> vi = 1.4, 2,3;
  // cout << vi << endl;
  map<int, int> m = {{1, 2}, {3, 4}};  // целые числа
  map<int, int> m2 = {{1.4, 2.1}, {3.4, 4}};  // дробные числа
  cout << m << '; ' << m2 << endl;
}
//1, 2 3, 4; 1.4, 2.1 3.4, 4
```

Код отработал корректно. Оба `map`'а вывелись, как нам и было нужно. Заметим, что код с вектором, если его добавить, до сих пор будет работать. Но этот код тоже можно доработать. Шаблоны для вектора и для `map`'а выглядят почти одинаково. Кроме того, для читаемости можно сделать улучшение: когда мы выводим `map`, обрамлять его в фигурные скобки, когда вектор – в квадратные, а когда мы выводим пару, обрамлять её круглыми скобками.

1.3.3. Рефакторим код и улучшаем читаемость вывода

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

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
template <typename Collection>  // тип коллекции
string Join(const Collection& c, char d) {  // передаем коллекцию и разделитель
  stringstream ss;  // завели строковый поток
  bool first = true;  // первый ли это элемент?
  for (const auto& i : c) {
    if (!first) {
      ss << d;  // если вывели не первый элемент - кладем поток в разделитель
    }
```

```
    first = false;  // т.к. следующий элемент точно не будет первым
    ss << i;  // кладем следующий элемент в поток
  }
  return ss.str();
}
template <typename First, typename Second>  // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
  return out << '(' <<p.first << ',' << p.second << ')';  // тоже изменили
}
template <typename T> // для vector изменили код и добавили скобочки
ostream& operator<< (ostream& out, const vector<T>& vi) {
  return out << '[' << Join(vi, ',') << ']';
}  // оператор вывода возвращает ссылку на поток
template <typename Key, typename Value>  // для map убрали аналогично vector
ostream& operator<< (ostream& out, const map<Key, Value>& m) {
  return out << '{' << Join(m, ',') << '}';  // и добавили фигурные скобочки
}

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

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

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

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

1.3.4. Указание шаблонного параметра-типа

Рассмотрим случай, когда компилятор не знает, как на основе вызова шаблонной функции вывести тип T на примере задачи о выводе максимального из двух чисел.

```
#include <iostream>
using namespace std;
template <typename T>
T max(T a, T b) {
  if (b < a) {
    return a;
  }
  return b;
}

int main() {
  cout << Max(2, 3) << endl;
  return 0;
}
// 3
```

Если оба числа целые, то всё работает. Но если поменять одно число на вещественное, мы увидим ошибки:

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

Т. е. вывод шаблонного параметра типа `T` не может состояться, потому что компилятор не знает, что поставить: `int` или `double`. В таких ситуациях мы либо приводим переменные к одному типу, либо подсказываем компилятору таким образом:

```
cout << Max<double>(2, 3.5) << endl;  // явно показываем компилятору тип T
// 3.5
```

А если же мы попросим `int`, получим следующее:

```
cout << Max<int>(2, 3.5) << endl;  // 3.5 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 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 ипользуется, 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.