Основы разработки на C++: красный пояс

Неделя 1

Макросы и шаблоны классов

Оглавление

Макросы и шаблоны классов

1.1. Введение в макросы

1.1.1. Введение в макросы

Первая тема нашего курса – введение в макросы. В курсе «Жёлтый пояс по C++» мы разработали unit test framework для создания юнит-тестов. Кроме того у нас была задача, которая называлась «Тестирование класса `Rational`», в которой нужно было написать набор юнит-тестов для класса `Rational`. Этот класс представлял собой рациональное число.

```
class Rational {
  public:
    Rational() = default;
    Rational(int nn, int dd);

    int Numerator() const;
    int Denominator() const;

  private:
    int n = 0;
    int d = 1;
};
```

Нам нужно было разработать набор юнит-тестов, которые проверяли, что этот класс реализован корректно. Рассмотрим программу, которая тестирует класс `Rational` с помощью unit test framework'а. В ней есть два теста: `TestDefaultConstructor()`, который проверяет, как работает конструктор по умолчанию в классе `Rational`, и `TestConstruction()`, в котором есть один тест, который проверяет, как ведет себя класс `Rational`, когда ему в конструктор передается числитель и знаменатель.

```
void TestDefaultConstructor() {
  const Rational defaultConstructed;
```

```
  AssertEqual(defaultConstrcuted.Numerator(), 0, "Default constructor
     denominator");
  AssertEqual(defaultConstrcuted.Denominator(), 1, "Default constructor
     denominator");
}
```

```
void TestConstruction() {
  const Rational r(3, 12);
  AssertEqual(r.Numerator(), 1, "3/12 numerator");
  AssertEqual(r.Denominator(), 4, "3/12 denominator");
}
```

В функцию `main` передаётся объект класса `TestRunner`, запускается два теста с помощью метода `RunTest`, куда передаём тест и текстовое сообщение.

```
int main() {
  TestRunner tr;
  tr.RunTest(TestDefaultConstructor, "TestDefaultConstructor");
  tr.RunTest(TestConstruction, "TestConstruction");
  return 0;
}
```

Строковые сообщения мы добавляли в `AssertEqual` для того, чтобы, когда наш `assert` срабатывает, понять, какой именно `assert` сработает. Намеренно допустим в классе `Rational` ошибку (например, вместо знаменателя будем возвращать числитель). С помощью сообщения об ошибке мы можем найти в нашем коде `assert`, который сработал. Нам помогло то, что мы сделали эти сообщения уникальными.

```
int Rational::Denominator() const {
  return n;
}
// TestDefaultConstructor fail: Assertion failed: 0 != 1
// TestConstruction fail: Assertion failed: 1 != 4 hint: 2 unit tests failed. Terminate
```

Посмотрим, как устроен вызов метода `RunTest` в классе `TestRunner`. Он принимает функцию, которая выполняет тестирование, и строчку, которая совпадает с названием функции. Эта строчка нужна, чтобы формировать сообщения "`TestDefaultConstructor fail`, `TestConstruction fail`", то есть чтобы в консоль выводить имя теста, который либо прошёл, либо не прошёл. Это неудобно, потому что это дублирование кода.

Что хотим получить:

- Если срабатывает `AssertEqual(x, y)`, на экран выводится `Assertion failed: 2 != 3 hint: x != y. main.cpp:35`;
- Запускать тесты кодом `tr.RunTest(TestConstruction);`
- Если тест проходит, на экран выводится `TestConstruction OK`.

Для того, чтобы решить эту задачу, вспомним, что сборка проекта на C++ состоит из трёх стадий: препроцессинг, компиляция, компоновка. На стадии препроцессинга выполняются директивы `#include`. Содержимое подключаемых файлов копируется в тело компилируемого файла, после этого наступает стадия компиляции. Также на стадии препроцессинга происходит разворачивание макросов. Они объявляются с помощью ключевого слова `#define`.

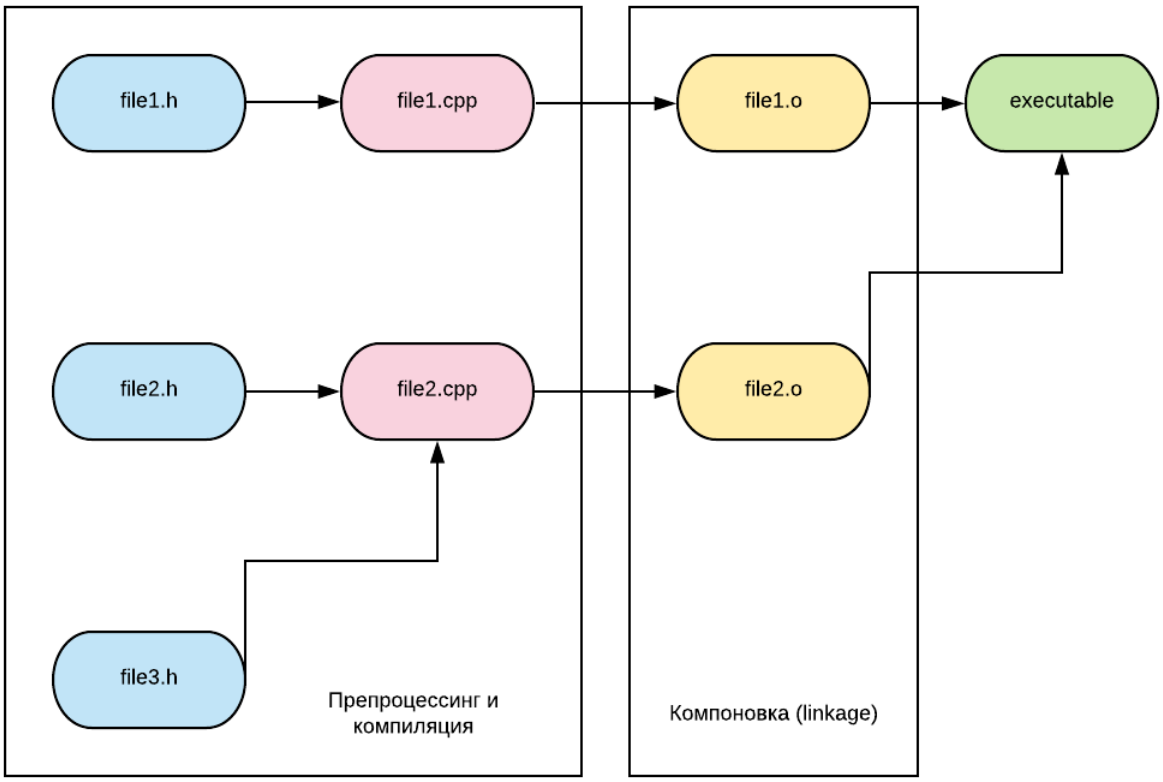

Рис. 1.1: Этапы сборки в C++

Для примера можно объявить макросы:

```
#define MY_MAIN int main()
#define FINISH return 0
```

Исправим функцию

```
int main() {
  return 0;
}
```

на

```
MY_MAIN {
  FINISH;
}
```

Объявленные макросы на этапе препроцессинга будут развёрнуты в свои определения.

Объявим макрос с тремя параметрами `ASSERT_EQUAL`, который разворачивается в вызов шаблона `AssertEqual` с этими тремя параметрами:

```
#define ASSERT_EQUAL(x, y, m) \
  AssertEqual(x, y, m)
```

Можно заменить вызов шаблона на макрос, такая программа скомпилируется.

```
void TestDefaultConstructor() {
  const Rational defaultConstructed;
  ASSERT_EQUAL(defaultConstructed.Numerator(), 0, "Default constructor
     denominator");
  ASSERT_EQUAL(defaultConstrcuted.Denominator(), 1, "Default constructor
     denominator");
}
```

Мы с вами познакомились с макросами, узнали, что они создаются с помощью ключевого слова `#define`, и на этапе препроцессинга происходит текстовая замена имени макроса на его содержимое. При этом макросы могут не содержать параметров или содержать один или несколько параметров.

1.1.2. Оператор

Нам хотелось бы избавиться от дублирования функции и её названия в коде (см. стр. 3).

Напишем макрос `RUN_TEST`, который принимает на вход параметр `tr` – объект класса `TestRunner`, `func` – функция, содержащая тесты. Этот тест будет у объекта `tr` вызывать метод `RunTest` и передавать туда `func` и ещё раз `func`:

```
#define RUN_TEST(tr, func) \
  tr.RunTest(func, func)
```

Применим макрос в функции `main`.

```
int main() {
  TestRunner tr;
  RUN_TEST(tr, TestDefaultConstructor);
  RUN_TEST(tr, TestConstruction);
  return 0;
}
```

Такой код не скомпилируется, потому что наш макрос получает в качестве второго параметра функцию, хотя этот параметр должен быть строкой.

```
#define RUN_TEST(tr, func) \
  tr.RunTest(func, #func)
```

Поставим перед парметром `func` символ решётки. Такая программа работает. Теперь в качестве второго параметра передается имя функции, обёрнутое в кавычки, то есть строковый литерал. Таким образом, с помощью оператора # мы добились того, что когда мы вызываем юнит-тесты, мы можем не дублировать имя функции, а просто передавать эту функцию в макрос и препроцессор за нас в качестве второго параметра передаст имя этой функции.

Другой пример, когда оператор # в макросах бывает полезен. Иногда нам нужно что-то логировать, например, для отладки. Допустим, для отладки мы хотим выводить следующие значения в какой-нибудь поток:

```
int x = 4;
string t = "hello";
bool isTrue = false;
```

Выведем их:

```
cerr << x << " " << t << " " << isTrue << endl;
// 4 hello 0
```

Можно использовать макросы и оператор #, чтобы сделать вывод более понятным:

```
#define AS_KV(x) #x << " = " << x
```

Перепишем код:

```
cerr << boolsalpha;
cerr << AS_KV(x) << endl
  << AS_KV(t) << endl
  << AS_KV(isTrue) << endl;
// x = 4
// t = hello
// isTrue = false
```

С помощью макроса и оператора # мы сделали более читаемый отладочный вывод.

Итоги:

- Оператор # вставляет в код строковое представление параметра макроса;
- Макрос `RUN_TEST` упрощает запуск тестов и избавляет от дублирования.

1.1.3. Макросы __FILE__ и __LINE__

Упростим использование шаблонов `Assert` и `AssertEqual`.

Для того, чтобы понять, какой `assert` сработал, нам достаточно знать название файла и номер строки в этом файле. Эту информацию мы можем получить автоматически. Для этого есть специальные макросы.

Используем макросы `__FILE__` и `__LINE__`.

```
const string file = __FILE__;
const int line = __LINE__;
```

На стадии препроцессинга `__FILE__` раскроется в название файла (в нашем случае это будет macro_intro.cpp). `__LINE__` раскроется в номер строки, в котором был объявлен (в нашем случае 14, поскольку макрос был объявлен на 14 строке).

Воспользуемся этими макросами, чтобы сформировать уникальное сообщение для `assert`'а. Чтобы создавать многострочные макросы, нужно каждую строчку кроме последней (с закрывающейся скобкой) завершать нисходящим слэшом.

```
#define ASSERT_EQUAL(x, y) {            \
  ostringstream os;                     \
  os << __FILE__ << ":" << __LINE__;    \
  AssertEqual(x, y, os.str());          \
}
```

Теперь макрос стал от двух параметров, потому что сообщение генерируется автоматически. Воспользуемся им:

```
void TestDefaultConstructor() {
  const Rational defaultConstructed;
  ASSERT_EQUAL(defaultConstructed.Numerator(), 0);
  ASSERT_EQUAL(defaultConstructed.Denominator(), 1);
}
// TestDefaultConstructor fail: Assertion failed: 0 != 1 hint: ..\src\macro_intro.cpp:19
```

В поле `hint` фреймворк выводит имя файла и строку, в котором вызван `assert`. Усовершенствуем выводимое сообщение:

```
os << #x << " != " << #y << ", "
  << __FILE__ << ":" << __LINE__;
```

В поле `hint` получаем сообщение:

```
// hint: defaultConstructed.Denominator() != 1, ..\src\macro_intro.cpp:20
```

Осталось перенести макрос в файл test_runner.h. Добавим туда также макросы `ASSERT` и `RUN_TEST`.

```
#define ASSERT(x) {                     \
  ostringstream os;                     \
  os << #x << " is false, "             \
    << __FILE__ << ":" << __LINE__;     \
  AssertEqual(x, y, os.str());          \
}
```

1.1.4. Тёмная сторона макросов

Макросы позволили сделать наш код короче и проще в использовании. Но вы могли слышать рекомендации, что макросы в C++ – это зло и что никогда нельзя использовать их в своих программах. Да, действительно, при чрезмерном их использовании могут возникать проблемы. Рассмотрим пример:

```
#define MAX (a, b) a > b ? a : b  // находит максимум из двух своих аргументов

int main() {
  int x = 4;
  int y = 2;
  int z = MAX(x, y) + 5;
  cout << z;
}
// 4
```

Мы ожидаем, что на экран будет выведено `9`, однако получаем `4`. Посмотрим, во что раскрывается наш макрос.

```
int z = x > y ? x : y + 5;
```

Если `x > y`, то в переменную `z` записывается значение `x`, если это не так, то в `z` запишется `y + 5`. Чтобы макрос работал правильно, можно обернуть его в скобки.

```
#define MAX(a, b) (a > b ? a : b)
```

Однако в данном случае гораздо лучше использовать функцию `max` из библиотеки алгоритмов.

Рассмотрим более реальный пример. В стандартной библиотеке нет функции, которая возводит свой аргумент в квадрат. Реализуем макрос, учтём прошлые ошибки и сразу обернём его в скобки.

```
#define SQR(x) (x * x)
```

Реализуем следующий код:

```
int main() {
  int x = 3;
  int z = SQR(x + 1);
  cout << z;
}
```

```
// 7
```

В консоли мы ожидаем увидеть `16`, но видим `7`. Посмотрим вывод препроцессора, чтобы узнать, в какое выражение раскрылся макрос.

```
int z = (x + 1 * x + 1);
```

Ошибку можно быстро исправить, если обернуть `x` в скобки.

```
#define SQR(x) ((x) * (x))
```

Вместо этого макроса лучше написать шаблон.

```
template <typename T>
T Sqr(T x) {
  return x * x;
}
```

Такой шаблон прекрасно справляется с задачей возведения в квадрат, при этом мы можем не бояться забыть обернуть макрос и аргументы в скобки.

Напишем функцию `LogAndReturn` и передадим её в качестве параметра в макрос `SQR`. Мы ожидаем, что в консоли выведется `x = 3`, а потом выведется `9`.

```
int LogAndReturn(int x) {
  cout << "x = " << x << endl;
  return x;
}

int main() {
  int z = SQR(LogAndReturn(3));
  cout << z;
}
// x = 3
// x = 3
// 9
```

Мы не ожидали, что функция `LogAndReturn` выполнится дважды. Посмотрим результаты препроцессирования.

```
int main() {
  int z = ((LogAndReturn(3)) * (LogAndReturn(3)));
```

```
  cout << z;
}
```

Макрос выполнил прямую текстовую замену и вызов функции `LogAndReturn` добавился в код дважды. Это учебный пример. Если бы функция в реальном коде выполняла сложные, долгие вычисления, то мы на ровном месте могли бы получить просадку производительности из-за неудачного использования макроса.

Сохраним результат вызова функции в переменную `x` и передадим её в макрос:

```
int main() {
  x = LogAndReturn(3);
  int z = SQR(x);
  cout << z;
}
// x = 3
// 9
```

Всё будет работать нормально. Допустим, далее нам понадобится переменная `x`, увеличенная на `1`. Ради экономии места увеличим её прямо на месте.

```
int main() {
  x = LogAndReturn(3);
  int z = SQR(x++);
  cout << z;;
}
// x = 3
// 12
```

Вместо ожидаемой `9` получили `12`. В результатах препроцессирования видим:

```
int z = ((x++) * (x++));
```

Когда одну и ту же переменную мы изменяем несколько раз в одном и том же выражении, то результат не определён.

Если вместо макроса можно написать функцию или шаблон, то именно так и нужно сделать. Так вы защититесь от неожиданных ошибок. Следовательно, если макрос не использует `__FILE__`, `__LINE__` или оператор `#`, подумайте, можно ли обойтись без него. Если вы всё же пишете макрос, то старайтесь использовать каждый аргумент только один раз, максимально изолируйте аргументы с помощью скобок.

1.2. Шаблоны классов

В курсе «Жёлтый пояс по C++» мы изучили шаблоны функций. Они позволяют избежать дублирования кода в функциях, который отличаются типами своих аргументов или типом возвращаемого значения. В этом модуле мы изучим шаблоны классов. Они решают ту же самую задачу: позволяют избежать дублирования кода в классах, которые отличаются только типами своих полей, или типами параметров своих методов, или типами возвращаемых значений в методах.

1.2.1. Введение в шаблоны классов

Простейший пример шаблона класса – это пара. Забудем на время, что существует стандартная пара. Напишем соответствующую структуру.

```
struct PairOfStringAndInt {
  string first;
  int second;
};
```

Воспользуемся этой парой.

```
int main() {
  PairOfStringAndInt si;
  si.first = "Hello";
  si.second = 5;
}
```

Потом у нас как-то проект развивается, мы пишем код дальше и понимаем, что нам нужна ещё пара, из логического значения и символа.

```
struct PairOfBoolAndChar {
  bool first;
  char second;
};
```

Добавим в `main`:

```
int main() {
  PairOfStringAndInt si;
```

```
  si.first = "Hello";
  si.second = 5;

  PairOfBoolAndChar bc;
  bc.first = true;
  bc.second = 'z';
}
```

Понятно, что каждый раз, когда у нас возникает необходимость в новых сочетаниях типов, нам приходится объявлять новую структуру. Мы видим, что классы `PairOfStringAndInt` и `PairOfBoolAndChar` структурно одинаковы, но отличаются типами своих полей. Вместо них мы можем написать шаблон класса.

```
template <typename T, typename U>
struct Pair {
  T first;
  U second;
};
```

Мы написали простейший шаблон класса «пара». Воспользуемся им.

```
int main() {
  Pair<string, int> si;
  si.first = "Hello";
  si.second = 5;

  Pair<bool, char> bc;
  bc.first = true;
  bc.second = 'z';
}
```

Важно отметить, что `Pair` – это шаблон класса, а `Pair<bool, char>` – это уже класс, самостоятельный тип. Таким образом, в этой программе мы создаем из шаблона класса два класса. Создание типа из шаблона класса называется **инстанцированием**.

1.2.2. Интеграция пользовательского класса в цикл for

Рассмотрим пример. У нас есть вектор целых чисел, по которому можно итерироваться, используя цикл `range-based for`. Если мы хотим проитерироваться по первым трем элементам вектора, то нам приходится использовать обычный цикл со счетчиком. В этом нет универсальности, к тому же если размер вектора меньше `3`, то цикл может выйти за пределы вектора и программа будет вести себя не так, как мы ожидаем. Давайте напишем функцию `Head`.

```
int main() {
  vector<int> v = {1, 2, 3, 4, 5};
  for (int x : Head(v, 3)) {
    cout << x << " ";
  }
}
```

Эта запись означает, что с помощью удобного цикла `range-based for` мы хотим проитерироваться по первым трем элементам вектора. Кроме того, эта функция должна корректно работать, и когда в векторе меньше чем три элемента. Мы напишем шаблон функции в данном случае.

```
template <typename T>
vector<T> Head(vector<T>& v, size_t top) {
  return {
    v.begin(),
    next(v.begin(), min(top, v.size()))
  };
}
```

Функция принимает на вход вектор `v` по ссылке, параметр `top` задаёт размер префикса, по которому мы хотим проитерироваться. Функция возвращает два итератора: начало вектора и `begin`, который с помощью оператора `next` мы продвинули на минимум из значения параметра `top` и размера вектора.

Функция `Head` создает копию вектора. Это не практично. Более того, в текущей реализации функции мы не можем изменять элементы изначального вектора.

И нам нужен какой-то другой способ, который бы позволил нам из функции `Head` вернуть диапазон внутри исходного вектора, и с помощью этого диапазона обращаться к элементам самого вектора. Например, из функции `Head` вместо копии вектора возвращать лишь пару итераторов на вектор `v` и итерироваться в цикле по ним. Это возможно сделать как раз с помощью шаблонов классов, с которыми мы с вами познакомились в предыдущем уроке.

```
template <typename Iterator>
struct IteratorRange {
  Iterator first, last;
};
```

Применим этот шаблон класса внутри функции `Head`.

```
template <typename T>
IteratorRange<typename vector<T>::iterator> Head(vector<T>& v, size_t top) {
  return {
    v.begin(), next(v.begin(), min(top, v.size()))
  };
}
```

В текущем виде код не скомпилируется, поскольку у структуры `IteratorRange` нет методов `begin` и `end`. Давайте их добавим.

```
Iterator begin() const {
  return first;
}
Iterator end() const {
  return last;
}
```

Чтобы по объекту класса можно было проитерироваться с помощью цикла `for`, он должен иметь методы `begin()` и `end()`. Методы `begin()` и `end()` должны возвращать итераторы.

1.2.3. Разница между шаблоном и классом

Сам по себе `IteratorRange` не является классом, это шаблон класса. В него необходимо подставить конкретный тип, чтобы создать класс. `IteratorRange` у нас параметризован типом итератора, а чтобы создать из него класс, в него нужно подставить какой-то конкретный тип итератора, например, итератор вектора целых чисел.

```
IteratorRange<vector<int>::iterator>
```

Все стандартные контейнеры, которыми мы пользовались, например, `vector`, `map`, `set` являются шаблонами классов.

Допустим, мы хотим посчитать, сколько элементов у нас находится в диапазоне двух итераторов. Мы не можем оформить функцию вот так:

```
size_t RangeSize(IteratorRange r) {
  return r.end() − r.begin();
}
```

Дело в том, что параметр `r` должен иметь тип. `IteratorRange` – не тип, это шаблон типа. Чтобы создать из этого шаблона тип, его нужно инстанцировать.

```
template <typename T>
size_t RangeSize(IteratorRange<T> r) {
  return r.end() − r.begin();
}
```

1.2.4. Вывод типов в шаблонах классов

Допустим, мы хотим обратиться к суффиксу вектора – его второй половине.

```
IteratorRange<vector<int>::iterator> second_half {
  v.begin() + v.size() / 2, v.end()
};
```

Чтобы объявить переменную `second_half`, нам пришлось написать достаточно громоздкую конструкцию. Напишем так называемую **порождающую функцию**.

```
template <typename Iterator>
IteratorRange<Iterator> MakeRange(Iterator begin, Iterator end) {
  return IteratorRange<Iterator>(begin, end);
}
```

Теперь мы можем лаконично объявить переменную `second_half`.

```
auto second_half = MakeRange {
  v.begin() + v.size() / 2, v.end()
};
```

Порождающие функции позволяют возложить на компилятор выведение шаблонных типов при инстанцировании шаблонных классов. Они сокращают код и избавляют от необходимости много

печатать. Однако для каждого шаблона класса порождающую функцию приходится писать самостоятельно. Кроме того, из записи `auto full = MakeRange(t.begin(), t.end())` неочевиден тип переменной `full`. Необходимо отдельно проверить, что возвращает функция `MakeRange`.

Порождающие функции – это не единственный способ возложить на компилятор вывод шаблонных типов при инстанцировании шаблонов класса. Другой способ появился в стандарте C++17. Чтобы им воспользоваться, необходимо настроить вашу среду разработки в соответствии с этим стандартом.

Необходимо убедиться, что у вас стоит компилятор `GCC` версии не младше седьмой. В самой IDE необходимо убедиться, что проект собирается с использованием самого свежего стандарта.

Если в классе есть конструктор, позволяющий определить тип шаблона, компилятор выводит его сам. Рассмотрим пример:

```
template <typename T> struct Widget {
  Widget(T value);
};

Widget w_int(5);
```

У нас есть шаблон класса `Widget`, в котором есть конструктор, принимающий значение `value` типа `T`. Компилятор по этому конструктору может сам вывести тип. Мы можем объявить переменную `w_int`, проинициализировать её значением `5`. При этом в качестве её типа мы просто пишем `Widget`. Компилятор берёт `5` и смотрит, какие конструкторы есть в шаблоне `Widget`. Видит конструктор, принимающий `value` типа `T`. Он понимает, что `5` имеет тип `int`, поэтому мы хотим инстанцировать шаблон типа `Widget` с помощью типа `int`, и он создаёт класс `Widget<int>`.

Рассмотрим другой пример:

```
pair<int, bool> p(t, true);
```

Из документации мы знаем, что у шаблона класса `pair` есть конструктор, который принимает параметры его шаблонных аргументов, поэтому компилятор может воспользоваться конструктором и вывести типа. Код мы можем переписать следующим образом:

```
pair p(5, true);
```

Код скомпилируется, поскольку по `5` и `true` компилятор поймёт, что нам нужно из шаблона `pair` создать класс `pair<int, bool>`.

Переделаем структуру `IteratorRange` в класс, добавим туда конструктор:

```
class IteratorRange {
  private:
    Iterator first, last;

  public:
    IteratorRange(Iterator f, Iterator l)
      : first(f)
      , last(l)
    {
    }
    Iterator begin() const {
      return first;
    }
    Iterator end() const {
      return last;
    }
}
```

Тогда пример с `second_half` мы можем переписать следующим образом:

```
IteratorRange second_half(
  v.begin() + v.size() / 2, v.end()
)
```

Компилятор видит, что мы создаём объект класса с помощью двух аргументов типа `vector<int> iterator`, он понимает, благодаря конструктору, что мы должны инстанцировать шаблон `IteratorRange` с помощью типа `vector<int>::iterator`. Таким образом он создаёт объект класса `IteratorRange` от `vector<int>::iterator`.

Способ выведения типов с помощью конструктора обладает преимуществами:

- не всегда нужно писать дополнительный код;
- `IteratorRange full(t.begin(), t.end())` – проще понять, какой тип у `full`.

Из следующего кода может показаться, что `r_i` и `r_s` имеют один и тот же тип, потому что перед ними стоит `IteratorRange`.

```
vector<int> ints;
vector<string> strs;
IteratorRange r_i(begin(ints), end(ints));
IteratorRange r_s(begin(strs), end(strs));
```

По умолчанию при инстанцировании стоит явно указывать шаблонный тип. Если это не удобно, то используем способ выведения через конструктор, потому что в нём явно указано имя шаблона. Если по какой-то причине мы не можем использовать этот способ, то делаем порождающую функцию.

1.2.5. Автоматический вывод типа, возвращаемого функцией

Вернемся к функции `Head`. Сейчас она работает только для вектора. Перепишем её так, чтобы она позволяла итерироваться по префиксу произвольного контейнера.

```
template <typename Container>
IteratorRange<???> Head(Container v, size_t top) {
  return {
    v.begin(), next(v.begin(), min(top, v.size()))
  };
}
```

Возникает вопрос: что нам написать при инстанцировании шаблона `IteratorRange`? Можем написать `typename Container::iterator`. Воспользуемся функцией `Head` для вывода четырёх минимальных элементов множества:

```
set<int> nums = {5, 7, 12, 8, 10, 5, 6, 1};
for (int x : Head(nums, 4)) {
  cout << x << ' ';
}
// 1 5 6 7
```

Для `deque<int>` код также работает правильно, однако для `const deque<int>` код не скомпилируется. Дело в том, что у `const deque` метод `begin` возвращает `const_iterator`, который не разрешает изменять элементы вектора. Нам нужно уметь выбирать между константным итератором для константных объектов и неконстантным итератором для неконстантных объектов. Напишем `auto` в качестве типа возвращаемого значения функции `Head`. Таким образом мы укажем компилятору взять возвращаемый тип из команды `return`, то есть `IteratorRange`.

```
template <typename Container>
auto Head(Container v, size_t top) {
  return IteratorRange{
    v.begin(), next(v.begin(), min(top, v.size()))
  };
}
```

Такой код работает для константного `deque`. Кроме того, работает пример с модификацией вектора `v` с помощью функции `Head`. Теперь компилятор сам выводит тип итератора, с которым нужно инстанцировать `IteratorRange`.

По умолчанию следует явно указывать тип результата функции. Использовать `auto` в качестве типа результата функции стоит только если:

- тип результата громоздкий;
- тело функции очень короткое.

В противном случае может пострадать понятность кода.