Шаблоны

М. А. Ложников

18 февраля 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Шаблоны функций

Часто возникает ситуация, когда нужно написать набор функций, выполняющих одни и те же операции над объектами разных типов данных. Например, рассмотрим функцию, возводящую число в квадрат

```
1  int Square(int value) {
2    return value * value;
3  }
4
5  double Square(double value) {
6    return value * value;
7  }
```

Отметим, что всевозможных типов данных в C++, представляющих целые или вещественные числа довольно много, поэтому было бы неудобно писать функцию `Square()` для каждого из них. Кроме того, следует учитывать случай новых, неизвестных на данный момент типов данных. Например, если мы напишем класс для работы с большими числами, нам придётся написать для него функцию `Square()`.

Чтобы решить названные проблемы, можно сделать функцию `Square()` шаблонной.

```
1  /*
2    Слово "template" перед функцией говорит о том, что она шаблонная.
3    После него пишутся через запятую шаблонные параметры в угловых скобках. В данном
4    случае параметр один - тип данных T. Ключевое слова "typename" говорит
5    о том, что это некий тип данных. Практически всегда вместо слова "typename"
6    можно писать слово "class".
7  */
8  template<typename T>
9  T Square(T value) {
10   return value * value;
11 }
12
13 int main() {
14   // Ниже в функцию Square() передаётся целое число (типа int). Компилятор сгенерирует код
15   // функции Square() с типом T равным int. В данном случае будет вызвана именно эта
```

```
  // сгенерированная на этапе компиляции функция.
  int x = Square(5);

  // Теперь в функцию подставили вещественное число (типа double). Встретив запись ниже,
  // компилятор сгенерирует код функции Square() с T = double.
  Square(5.0);
  return 0;
}
```

Таким образом, как только компилятор встретит вызов функции `Square()` с новым типом данных, он автоматически сгенерирует функцию `Square()`, которая будет принимать аргумент переданного в неё типа данных. Обратите внимание на важное замечание 3.1.

Иногда вывод шаблонных параметров невозможен:

```
template<typename T>
T Max(T value1, T value2) {
  return (value1 > value2 ? value1 : value2);
}

int main() {
  /* Код ниже вызовет ошибку поскольку в функцию передали два аргумента
     разных типов: int и double. Компилятор не будет знать, как правильно вывести
     шаблонный параметр T. */
    Max(5, 5.9); // Ошибка компиляции.

  // В этом случае можно указать явно нужную специализацию шаблона,
  // передав через запятую все обязательные шаблонные параметры в угловых скобках.
  Max<double>(5, 5.9);
  return 0;
}
```

Замечание 1.1 *В данном случае нельзя сделать шаблон с двумя разными параметрами (тип данных для* `value1` *и тип данных для* `value2`*) поскольку тип данных значения, которое возвращает функция, должен быть определён на этапе компиляции.*

2 Перегрузка шаблонов функций

Мы уже узнали, что компилятор определяет шаблонные параметры автоматически. Теперь поговорим о правилах перегрузки шаблонов. Вкратце эти правила можно сформулировать так: компилятор выбирает наиболее частный случай. Неоднозначности приводят к ошибкам компиляции. Подробное определение можно посмотреть в [1]. Поясним правила перегрузки на примере

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

using namespace std;

// Наиболее общий случай
template<typename T>
```

```
T Square(T value) {
  cout << "General template" << endl;
  return value * value;
}

// Специализация шаблона для типа char.
template<>
char Square(char value) {
  cout << "Specialization for char" << endl;
  return value * value;
}

// А вот это обычная функция, принимающая тип double.
double Square(double value) {
  cout << "Function for double" << endl;
  return value * value;
}

// Случай для произвольного вектора
template<typename T>
vector<T> Square(const vector<T>& vec) {
  vector<T> result;

  cout << "Template with vector" << endl;

  for (size_t i = 0; i < vec.size(); i++)
    result.push_back(Square(i)); // Вызовем Square() для каждого элемента.

  return result;
}

int main() {
  int val = 0;
  /* Этот список инициализации работает начиная с C++11 (brace-enclosed initializer list).
     Позволяет сразу задать элементы вектора, они перечисляются через запятую в фигурных
     скобках. */
  vector<double> vec = {3.14, 2.7};
  double realVal = 0;
  char charVal = '\0';

  Square(realVal); // Вызовется обычная функция с параметром типа double.

  Square(vec); // Вызовется шаблон с вектором как наиболее частный случай.

  Square(charVal); // Вызывается специализация шаблонной функции для типа char.

  /* Вызовется общая шаблонная версия. Функция Square(double) тоже подходит,
     поскольку int неявно преобразуется в double, однако этот вариант будет отброшен.
     Вариант Square(char) тоже будет отброшен. Таким образом, компилятор не делает
     дальнейшее продвижение типов (неявное преобразование) в том случае, если ему
     приходится автоматически выводить шаблонные параметры. */
```

```
59   Square(val);
60
61   return 0;
62 }
```

3 Шаблоны классов

Шаблонными бывают не только функции, но и классы. Рассмотрим несколько примеров.

3.1 Простейший пример. Пара

```
1  // Пара из двух типов данных.
2  template<typename T, typename V>
3  struct Pair {
4    T first;
5    V second;
6
7    Pair()
8    { }
9
10   Pair(const T& f, const V& s) :
11     first(f),
12     second(s)
13   { }
14 };
15
16 // Порождающая функция для нашей пары. Создаёт пару из переданных аргументов.
17 template<typename T, typename V>
18 Pair<T, V> MakePair(const T& first, const V& second) {
19   return Pair<T, V>(first, second);
20 }
21
22 int main() {
23   Pair<int, double> pairIntDouble;
24   pairIntDouble.first = 5; // first имеет тип данных int
25   pairIntDouble.second = 4.5; // second имеет тип данных double
26
27   // До C++17 компилятор не умел автоматически определять шаблонные параметры
28   // из аргументов, переданных в конструктор. Начиная с C++17, если аргументы, переданные
29   // в конструктор позволяют однозначно вывести шаблонные параметры, то эти
30   // параметры указывать не обязательно, компилятор их выведет автоматически.
31   pairIntDouble = Pair<int, double>(10, 12.5);
32
33   // pairIntDouble = Pair(10, 12.5); // Сработает в C++17.
34
35   // Для сокращения записи в старых стандартах C++ можно использовать порождающие функции,
36   // для которых компилятор умеет выводить шаблонные параметры автоматически.
37   pairIntDouble = MakePair(10, 12.5);
```

```
38   return 0;
39 }
```

При работе с шаблонами нужно понимать, что сам шаблон не является классом, классом он становится только после подстановки в него шаблонных параметров. При этом классы, полученные при помощи подстановки в один и тот же шаблон разных шаблонных параметров являются разными классами. Иными словами, `Pair` — это не класс, это всего навсего шаблон класса. А вот `Pair<int, double>` — это уже класс, причём классы `Pair<int, int>` и `Pair<int, double>` являются разными классами, соответственно для них не будет определена операция присваивания, и из одного нельзя будет создать другой при помощи copy-конструктора.

Задача 1 *Напишите шаблонную функцию* `Square()`*, возводящую произвольную пару в квадрат поэлементно.*

3.2 Структура для определения типа данных или его свойств

Иногда бывает нужно узнать какую-либо характеристику шаблонного параметра, например, является ли он числовым типом, является ли он контейнером и т.д. Для этого можно сделать общий шаблон и несколько его специализаций. Правила выбора специализации похожи на правила выбора перегруженной функции: всегда выбирается наиболее частный случай. Неоднозначности приводят к ошибкам компиляции.

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  // Общий шаблон
7  template<typename T>
8  struct IsFloatingPoint {
9    const static bool value = false;
10 };
11
12 // Специализация шаблона для типа float
13 template<>
14 struct IsFloatingPoint<float> {
15   const static bool value = true;
16 };
17
18 // Специализация шаблона для типа double
19 template<>
20 struct IsFloatingPoint<double> {
21   const static bool value = true;
22 };
23
24 // Специализация шаблона для типа long double
25 template<>
26 struct IsFloatingPoint<long double> {
27   const static bool value = true;
28 };
```

```
29
30 template<typename T>
31 bool IsEqual(T value1, T value2) {
32   // Числа с плавающей точкой в большинстве случаев некорректно сравнивать точно.
33   if (IsFloatingPoint<T>::value)
34     return fabs(value1 - value2) < 1e-12;
35
36   return value1 == value2;
37 }
38
39 int main() {
40   // std::boolalpha --- манипулятор для того, чтобы переменные типа bool
41   // выводились как true/false, а не как 1/0.
42   cout << boolalpha;
43   cout << IsFloatingPoint<int>::value << endl; // Выведет false, сработает общий шаблон.
44
45   // Выведет true, сработает специализация шаблона для double.
46   cout << IsFloatingPoint<double>::value << endl;
47   return 0;
48 }
```

Задача 2 *Напишите шаблон* `IsIntegral`*, с единственным булевым полем* `value`*, позволяющим определить, является ли тип целочисленным. Относительно полный список (в различных компиляторах могут быть различные расширения этого списка) можно посмотреть по ссылке* ***https://en.cppreference.com/w/cpp/language/types*** *.*

3.3 Оболочка над статическим массивом

Шаблонными параметрами могут быть не только типы данных, могут быть, например, целые числа.

```
1 template<typename T, int N>
2 class Array {
3  public:
4   // Метод для чтения элемента по индексу
5   const T& operator[](int index) const { return data[index]; }
6   // Метод для записи элемента по индексу
7   T& operator[](int index) { return data[index]; }
8   // Возвращает размер массива
9   int Size() const { return N; }
10 private:
11   T data[N];
12 };
13
14 int main() {
15   Array<int, 10> arrayInt; // Объявили массив из 10 int'ов.
16
17   /* Обратите внимания, что в качестве шаблонного параметра N можно подставлять
18      только значения, известные на момент компиляции то есть константы, поскольку
19      новый класс будет генерироваться из шаблона именно в момент компиляции.
20      Таким образом, классы Array<int, 10> и Array<int, 8> это разные классы
```

```
    и для них будет сгенерирован разный код с разными методами.
  */

  // Можем записать или прочитать элемент массива по индексу.
  arrayInt[3] = 5;
  arrayInt[0] = arrayInt[3];
  return 0;
}
```

Задача 3 *Напишите итератор для шаблона* `Array`*, то есть класс* `Iterator`*, а также методы* `begin()` *и* `end()`*, возвращающие итератор на начало массива и итератор на конец массива соответственно. Для итератора должны быть определены операции сравнения, инкремента и разыменования (унарная операция* `*`*).*

Задача 4 *Напишите шаблон* `Square()`*, который возводит массив* `Array` *в квадрат поэлементно.*

3.4 Оболочка над динамическим массивом

Напишем простую оболочку над динамическим массивом.

```
#include <iostream>

using namespace std;

template<typename T>
class Vector {
 public:
  // Конструктор по умолчанию инициализирует всё нулями.
  Vector() :
    data(nullptr), // В C++11 вместо NULL рекомендуется писать nullptr
    size(0),
    capacity(0)
  { }

  // Создаёт массив размера count, в котором лежит count элементов,
  // заполненных значениями по-умолчанию.
  Vector(size_t count) :
    data(new T[count]),
    size(count),
    capacity(count)
  {
    for (size_t i = 0; i < count; i++)
      data[i] = T();
  }

  // Освобождает память, если она была выделена.
  ~Vector() {
    if (data)
      delete[] data;
  }
```

```
31
32    // Чтение элемента вектора по индексу.
33    const T& operator[](size_t index) const { return data[index]; }
34
35    // Запись элемента по индексу.
36    T& operator[](size_t index) { return data[index]; }
37
38    // Количество элементов, лежащих в векторе.
39    size_t Size() const { return size; }
40
41    // Ёмкость вектора, то есть количество выделенной памяти в элементах.
42    size_t Capacity() const {return capacity; }
43
44    // Сырой доступ на чтение к элементам по указателю.
45    const T* Data() const { return data; }
46
47    void PushBack(const T& elem) {
48      if (!data) { // Если не выделена память
49        data = new T[1];
50        capacity = 1;
51      }
52      if (size >= capacity) {
53        // Если вся память заполнена, выделяем в два раза больший массив.
54        T* tmp = new T[capacity * 2];
55
56        // Копируем в новый массив все элементы
57        for (size_t i = 0; i < size; i++)
58          tmp[i] = data[i];
59
60        // Освобождаем старый массив
61        if (data)
62          delete[] data;
63
64        // Ставим новый массив на место старого
65        data = tmp;
66        capacity *= 2; // Обновляем ёмкость.
67      }
68
69      // Записываем сам элемент.
70      data[size++] = elem;
71    }
72   private:
73    T* data;          // Массив с данными
74    size_t size;      // Количество заполненных элементов
75    size_t capacity; // Размер выделенной памяти
76  };
77
78  int main() {
79    Vector<int> vec;
80
81    for (int i = 0; i < 10; i++)
```

```
82      vec.PushBack(i * i);
83
84    for (size_t i = 0; i < vec.Size(); i++)
85      cout << i << " " << vec[i] << endl;
86
87    return 0;
88  }
```

Задача 5 *Напишите copy-конструктор для шаблона* `Vector`*, оператор присваивания* `operator=()`*, функцию* `PopBack()`*, удаляющую элемент из вектора, а также функцию* `Resize()`*, изменяющую количество элементов в векторе и инициализирующую новые элементы (если есть) значениями по умолчанию* `T()`*. Функция* `PopBack()` *не изменяет ёмкость массива. Функция* `Resize()` *выделяет память только в том случае, если новый размер вектора превышает исходную ёмкость.*

Задача 6 *Напишите итератор для шаблона* `Vector`*, то есть класс* `Iterator`*, а также методы* `begin()` *и* `end()`*, возвращающие итератор на начало массива и итератор на конец массива соответственно. Для итератора должны быть определены операции сравнения, инкремента и разыменования (унарная операция* `*`*).*

Задача 7 *Напишите шаблон* `Square()`*, который возводит вектор в квадрат поэлементно. Шаблон должен работать для вектора из элементов типа* `Array` *или* `Pair` *и наоборот (число вложений может быть произвольным). В тех случаях, когда шаблон вызывает другой шаблон, которой реализован в файле ниже первого шаблона, нужно перед первым шаблоном объявлять прототип второго. Иными словами, сначала нужно объявить прототипы всех шаблонов, а затем написать их реализации (точно также как раньше делали на C). Прототип шаблона состоит из слова* `template`*, всех шаблонных параметров, типа возвращаемого значения, названия шаблона и списка аргументов.*

```
1  // Прототип общего шаблона.
2  template<typename T>
3  T Square(const T& x);
4
5  // Прототип шаблона для массива.
6  template<typename T, int N>
7  Array<T, N> Square(const Array<T, N>& array);
8
9  // Прототип шаблона для вектора.
10 template<typename T>
11 Vector<T> Square(const Vector<T>& vec);
```

3.5 Стек

Методы шаблона не обязательно описывать сразу, можно это сделать после или в отдельном файле. Рассмотрим следующий пример

```
1  template<typename T>
2  class Stack {
3   public:
4    Stack();
5
```

```
6   ~Stack();
7
8   void Push(const T& elem);
9
10  struct StackNode {
11    T element;
12    StackNode* prev; // Предыдущая ячейка стека.
13  };
14 private:
15  StackNode* top; // Верхняя ячейка стека.
16 };
17
18 /* Если реализации методов написаны отдельно, то нужно заново перечислять
19   шаблонные параметры, а также подставить их в шаблон. */
20 template<typename T>
21 Stack<T>::Stack() :
22   top(nullptr)
23 { }
24
25 template<typename T>
26 Stack<T>::~Stack() {
27   while(top) { // Удаляем все ячейки, начиная с верхней.
28     StackNode* tmp = top->prev;
29     delete top;
30     top = tmp;
31   }
32 }
33
34 template<typename T>
35 void Stack<T>::Push(const T& elem) {
36   /* В C++11 можно ещё писать так. Запись StackNode{elem, top}
37     создаст новый элемент StackNode, у которого element = elem,
38     а prev = top. Специальный конструктор для этого писать не нужно. */
39   top = new StackNode{elem, top};
40 }
```

Замечание 3.1 *Ещё одной особенностью шаблонных типов данных и шаблонных функций является следующая: поскольку компилятор генерирует новый тип данных каждый раз, как увидит новую подстановку (инстанцирование) шаблона, то он должен располагать полной информацией для генерации кода. Иными словами, реализацию шаблонных методов, а также шаблонных функций придётся писать в заголовочных* `.hpp` *файлах, а не в файлах исходного кода* `.cpp`*. В противном случае мы сможем использовать шаблон только в том файле, в котором он был реализован.*

Задача 8 *Напишите метод* `Pop()`*, удаляющий верхний элемент из стека и метод* `Top()`*, возвращающий верхний элемент стека (константную и неконстантную версии). В случае, если стек пуст, то метод* `Pop()` *ничего не делает, а метод* `Top()` *генерирует исключение* `std::runtime_error`*.*

3.6 Разные примеры

Вспомним пример использования полиморфизма для класса шахматной фигуры. У нас был абстрактный класс `ChessPart`, описывающий фигуру. На базе него создаются различные фигуры, которые

должны перепреодолеть абстрактный метод `CanMove` класса `ChessPart`, который проверяет, может ли фигура быть переставлена в указанную точку. Сделаем то же, используя шаблоны вместо наследования.

3.6.1 Первый пример

```
#include <iostream>

using namespace std;

/* Этот шаблон описывает абстрактную шахматную фигуру. Он должен быть инстанцирован
  классом, у которого есть статический метод CanMove(int, int, int, int), определяющий,
  может ли фигура пойти из клетки (x, y) в клетку (nx, ny). */
template<typename Policy>
class ChessPart {
 public:
  bool Move(int nx, int ny) {
    if (!Policy::CanMove(x, y, nx,  ny)) {
      cout << "Can't move from (" << x << ", " << y << ") to ("
           << nx << ", " << ny << ")!" << endl;
      return false;
    }

    x = nx;
    y = ny;

    cout << "Moved from (" << x << ", " << y << ") to ("
         << nx << ", " << ny << ")!" << endl;

    return true;
  }
 private:
  // В C++11 можно инициализировать поля классов некоторых типов данных прямо
  // при объявлении. Компилятор сам сгенерирует соответствующий стандартный конструктор.
  int x = 0; // Текущая позиция фигуры.
  int y = 0;
};

class CastlePolicy {
 public:
  static bool CanMove(int x, int y, int nx, int ny) {
    if (nx < 0 || nx >= 8 || ny < 0 || ny >= 8)
      return false;
    if (x != nx && y != ny)
      return false;
    if (x == nx && y == ny)
      return false;

    return true;
  }
```

```
};

int main() {
  ChessPart<CastlePolicy> castle;

  cout << boolalpha << castle.Move(1, 1) << endl
       << castle.Move(0, 5) << endl;

  return 0;
}
```

Преимуществом данного подхода перед подходом, использующим наследование, является отсутствие проверок, выполняемых во время работы программы, в то время, как при использовании наследования и полиморфизма придётся заглянуть в таблицу виртуальных функций для того, чтобы узнать, какой метод `CanMove` нужно вызвать. В случае шаблонов все проверки будут выполнены на этапе компиляции, однако для этого придётся пожертвовать гибкостью кода из-за статической типизации.

3.6.2 Второй пример

Правила, по которым ходит фигура могут быть устроены довольно сложно, в этом случае вместо статического метода можно использовать обычный.

```
#include <iostream>

using namespace std;

/* С помощью ключевого слова "enum class" определим две константы
Team::White и Team::Black, которые не приводяться неявно к типу int. */
enum class Team {
  White,
  Black
};
/* Этот шаблон описывает абстрактную шахматную фигуру. Он должен быть инстанцирован
  классом, у которого есть конструктор, принимающий команду, а также метод
  CanMove(int, int, int, int), определяющий, может ли фигура пойти
  из клетки (x, y) в клетку (nx, ny). */
template<typename Policy>
class ChessPart {
 public:
  ChessPart(Team team) :
    policy(team), // Вызываем конструктор шаблонного параметра Policy
    x(0),
    y(0)
  { }

  bool Move(int nx, int ny) {
    if (!policy.CanMove(x, y, nx,  ny)) {
      cout << "Can't move from (" << x << ", " << y << ") to ("
           << nx << ", " << ny << ")!" << endl;
      return false;
```

```
29      }
30
31      x = nx;
32      y = ny;
33
34      cout << "Moved from (" << x << ", " << y << ") to ("
35           << nx << ", " << ny << ")!" << endl;
36      return true;
37    }
38   private:
39    Policy policy; // Правила, характерные для фигуры.
40    int x, y;      // Текущая позиция фигуры.
41  };
42
43  class PawnPolicy {
44   public:
45    PawnPolicy(Team team) :
46      team(team)
47    { }
48
49    bool CanMove(int x, int y, int nx, int ny) const {
50      if (nx < 0 || nx >= 8 || ny < 0 || ny >= 8)
51        return false;
52
53      if (team == Team::White && y == ny - 1 && x == nx)
54        return true;
55
56      if (team == Team::Black && y == ny + 1 && x == nx)
57        return true;
58
59      // Опускаем случаи, когда пешка кого-то ест.
60      return false;
61    }
62   private:
63    // Команда, которой принадлежит пешка (Team::White или Team::Black).
64    Team team;
65  };
66
67  int main() {
68    ChessPart<PawnPolicy> pawn(Team::White);
69
70    cout << boolalpha << pawn.Move(1, 1) << endl
71         << pawn.Move(0, 1) << endl;
72
73    return 0;
74  }
```

Задача 9 *Напишите класс* `QueenPolicy`*, определяющий правила хода для ферзя. Он должен быть написан аналогично классу* `PawnPolicy`*, то есть его конструктор должен принимать команду, и он должен содержать константный метод* `CanMove()`*, принимающий четыре целых числа и возвращающий* `bool`*. Иными словами, должен компилироваться следующий код:*

```
ChessPart<QueenPolicy> queen(Team::White);
queen.Move(1, 1);
```

4 Заключение

Шаблонные функции и классы представляют уровень абстракции, который разрешается в момент компиляции программы. Таким образом, они позволяют избежать дублирования кода, при этом они не создают условий, которые нужно проверять во время выполнения программы, что положительно сказывается на её скорости работы.

Однако, код, использующий шаблоны, иногда может оказаться сложным для чтения. Кроме того, код, инстанцирующий шаблоны с разными шаблонными параметрами может довольно сильно разбухнуть, что отразится на размере исполняемого файла. Время компиляции программы, использующей шаблоны, может быть довольно большим за счёт того, что компилятор каждый раз генерирует весь код шаблона заново при подстановке в него новых шаблонных параметров.

Шаблоны часто используют в тех случаях, когда хотят избежать дублирования кода, а также структурировать программу, не повлияв негативно на её скорость работы. Взамен приходится расплачиваться читабельностью кода, временем компиляции и размером исполняемого файла.

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`