

Объектно-ориентированное программирование. Лекция 8. Шаблоны

Макаров П. А.

15 декабря 2023 г.

Содержание

1	Необходимость шаблонов	1
2	Шаблоны функций	2
3	Шаблоны классов	4
4	Специализация шаблонов	8
5	Контрольные вопросы и задания	11

1 Необходимость шаблонов

Часто возникают ситуации когда для решения типовых, но несколько отличающихся друг от друга задач требуется писать почти одинаковые фрагменты программного кода. Очевидно, что дублировать код в таких ситуациях — это самый худший вариант разработки, приводящий к большому количеству проблем. Не останавливаясь на них подробно, отметим только одну, наиболее трудоёмкую проблему, — обеспечение корректной синхронизации такого рода разрозненных фрагментов кода.

В том случае, когда фрагменты кода различаются только некоторыми *значениями*, то решение очевидно: достаточно вынести этот фрагмент кода в отдельную функцию и вызывать её с разными значениями параметров. Аналогично можно поступить, когда фрагменты различаются *именами переменных*, над которыми выполняются одинаковые действия.

В таком случае можно написать функцию, фактическим параметром которой будет указатель на требуемую переменную.

Более сложной является ситуация, когда различия фрагментов кода не сводятся к *значениям выражений*. Самый простой и часто встречающийся пример — различие *типов* переменных и выражений. При решении подобных проблем в программах на языке С прибегают к макросам. В языке С++ для этого используют шаблоны.

- Шаблоны функций;
- Шаблоны классов.

И в том, и в другом случае пишется не сам код, а только его “заготовка”, которая при конкретизации неких параметров “превращается” в функцию или класс. Параметрами шаблона чаще всего являются имена типов выражений, но могут быть и значения целого типа, либо адресные выражения. Один шаблон может использоваться многократно для создания разных функций или классов.

Замечание 1. Допустимо считать, что шаблоны представляют собой ещё один вид полиморфизма — **параметрический полиморфизм**.

Так как этот вид полиморфизма реализуется на стадии компиляции, то он представляет собой частный случай статического полиморфизма и, строго говоря, к ООП никакого отношения не имеет¹.

2 Шаблоны функций

Пример 1. Функция, сортирующая «пузырьком» массив целых чисел

```
1 void sort_int(int *array, int len) {
2     for(int start = 0; ; start++) {
3         bool done = true;
4         for(int i = len-2; i >= start; i--)
5             if(array[i+1] < array[i]) {
6                 int tmp = array[i];
7                 array[i] = array[i+1];
8                 array[i+1] = tmp;
9                 done = false;
10            }
11        if(done)
```

¹Безусловно, это не делает его менее важным!

```
12         break;
13     }
14 }
```

Листинг 1: Сортировка «пузырьком» массива чисел типа `int`

Пусть теперь потребовалась аналогичная функция для сортировки массива чисел типа `double`. Приведённую в Листинге 1 функцию по очевидным соображениям использовать нельзя. Решение “в лоб” с описанием функции `sort_double` также некорректно, поэтому выход из этой ситуации выглядит иначе. В языке C для этого с помощью директивы `#define` приходится писать многострочные макросы с одним параметром, задающим тип элементов сортируемого массива и, соответственно, тип переменной `tmp`.

В C++ подобные задачи решаются с помощью шаблонов. Шаблон сам по себе не является функцией, а в некотором роде является её заготовкой. Конечную функцию по заданному шаблону генерирует компилятор при указании всех необходимых параметров. В рассматриваемом нами примере параметром шаблона является тип элементов массива.

```
1 template <class T>
2 void sort(T *array, int len) {
3     for(int start = 0; ; start++) {
4         bool done = true;
5         for(int i = len-2; i >= start; i--)
6             if(array[i+1] < array[i]) {
7                 T tmp = array[i];
8                 array[i] = array[i+1];
9                 array[i+1] = tmp;
10                done = false;
11            }
12        if(done)
13            break;
14    }
15 }
```

Листинг 2: Шаблон функции для сортировки «пузырьком» произвольного массива

Определение 1. *Ключевое слово `template` указывает компилятору, что далее следует шаблон. В угловых скобках перечисляются параметры шаблона, причём слово `class` здесь указывает на произвольный (обобщённый) тип, а не только на класс.*

Для построения функции на основе шаблона следует указать его имя, после чего в угловых скобках привести конкретные значения его параметров.

```
1 int a[30];
2 // ...
3 sort<int>(a, 30);
```

Приведённый пример шаблона можно использовать для сортировки не только целых, но и вещественных чисел. Для этого надо предложить компилятору построить функцию `sort<double>`. Более того, такой шаблон годится для сортировки массивов из элементов произвольного типа. Для этого необходимо только выполнение следующих условий:

1. для этого типа существует конструктор по умолчанию (это используется нами в 7-й строке Листинга 2);
2. определены операция присваивания (также 7-я строка);
3. и операция сравнения `<` (используемая в 6-й строке того же Листинга).

Определение 2. *Получение функции из шаблона называется **инстанциацией** шаблона.*

Компилятор инстанцирует каждую функцию только один раз.

В некоторых случаях инстанцированную функцию можно вызвать, не указывая параметры шаблона. Например, вызов

```
1 sort(a, 30);
```

будет вполне корректным, т. к. по типу параметра `a` компилятор определит, что имеется в виду именно `sort<int>`. Эта возможность называется **автоматическим выводом аргументов шаблона**.

3 Шаблоны классов

С помощью шаблонов классов можно описывать классы, использующие обобщённые типы данных. Синтаксис описания шаблона класса следующий:

```
1 template <list_of_generalized_types >
2 class Template_Class_Name {
3 // ...
4 };
```

Здесь `list_of_generalized_types` — это список вида `class T1 [, class T2, ..., class TN]`, где `Ti` обозначает произвольный идентификатор обобщённого типа.

Все методы шаблона класса являются шаблонами функций. Поэтому при определении каждого метода следует указать на то, что он является шаблоном:

```
1 template <list_of_generalized_types >
2 type Template_Class_Name<list_of_generalized_types >::
3     Method_Name(list_of_formal_parameters) {
4     // ...
5 }
```

Список формальных параметров метода может содержать параметры как конкретных типов данных (`int`, `double`, `Student` и т. д.), так и обобщенных. То же относится и к типу возвращаемого значения метода.

Пример 2. *Шаблон класса «Квадратная матрица», в котором обобщённым типом данных является тип элемента матрицы.*

```
1 template <class T> class QMatrix {
2     T** element;    // elements array
3     int N;        // matrix size
4 public:
5     QMatrix(int n = 0);
6     QMatrix(const QMatrix<T>&);
7     ~QMatrix();
8     int GetSize() const;
9     QMatrix<T> operator+(const QMatrix<T>&) const;
10    QMatrix<T> operator*(const QMatrix<T>&) const;
11    QMatrix<T> operator=(const QMatrix<T>&) const;
12    QMatrix<T> SubMatrix(int, int) const;    // minor
13    QMatrix<T> Inverse() const;            // inverse matrix
14    T Determinant() const;
15 };
```

Листинг 3: Шаблон класса QMatrix

Следует обратить внимание на то, что в Листинге 3 везде (за исключением имени класса, его конструкторов и деструктора) к слову `QMatrix` добавляется параметр `<T>`. Причина этого в том, что относительно самого класса компилятор и так знает, что описывается шаблон (то же, в равной мере, относится и к конструкторам и деструктору), однако когда речь идёт о типах параметров принимаемых и возвращаемых функциями; типах указателей и т. п., то теоретически в качестве них могут выступать любые классы, в том числе созданные из данного шаблона, но с другим параметром `T`. Поэтому тип в большинстве случаев необходимо указывать полностью.

Функции-методы шаблона класса также необходимо описывать как шаблоны. Приведём в качестве примеров определения двух методов шаблона класса `QMatrix`.

```

1 template <class T>
2 QMatrix<T>::QMatrix(int n) {
3     N = n;
4     element = new T* [N];
5     for(int i = 0; i < N; i++)
6         element[i] = new T [N];
7     for(int i = 0; i < N; i++)
8         for(int j = 0; j < N; j++)
9             element[i][j] = 0;
10 }

```

Листинг 4: Шаблон конструктора для шаблона класса QMatrix

```

1 template <class T>
2 QMatrix<T> QMatrix<T>::operator+(const QMatrix<T> & second)
3     const {
4     if (second.N == N) {
5         QMatrix<T> tmp(N);
6         for(int i = 0; i < N; i++)
7             for(int j = 0; j < N; j++)
8                 tmp.element[i][j] = element[i][j] + second.
9                 element[i][j];
10        return tmp;
11    }
12    else
13        throw 1;

```

Листинг 5: Шаблон операции сложения для шаблона класса QMatrix

При анализе Листингов 4 и 5 обратите внимание на то, что при раскрытии области видимости в явном виде указывается какой именно класс (а не шаблон класса!) имеется в виду, поэтому приходится писать `QMatrix<T>`, а не просто `QMatrix`.

Правило 1. *Везде, где по смыслу предполагается имя конкретного типа, при использовании шаблона необходимо указывать значения для параметров шаблона.*

На основе шаблона класса, фрагменты кода которого приведены в Листингах 3–5, можно создавать объекты сгенерированных классов `QMatrix` с указанием конкретных типов данных для элементов матрицы (`int`, `double`, `Complex` — комплексное число, `Fraction` — дробь и т. д.).

Замечание 2. *При генерации класса с указанием пользовательских типов данных нужно убедиться, что данный пользовательский тип содержит переопределение основных функций и операций, которые используются в методах шаблона.*

Например, пусть разработан класс «Рациональная дробь» `Fraction`, и создается матрица `A` размерности 4×4 , элементами которой являются рациональные дроби:

```
1 QMatrix<Fraction> A(4);
```

При таком создании объекта `A` происходит генерация класса `QMatrix` с элементами типа `Fraction`. Также будут сгенерированы и все методы данного класса. Затем будет вызван сгенерированный конструктор с одним параметром. При выполнении данного конструктора (его определение было приведено в Листинге 4) происходит инициализация элементов матрицы нулями (как следует из строки 9 соответствующего Листинга). Для инициализации дроби необходимо, чтобы в классе `Fraction` был определен конструктор с одним параметром целого типа, который преобразует целое число (в данном случае 0) в дробь. Также для корректной работы метода сложения двух матриц придётся определить в классе `Fraction` операторы сложения, умножения и присваивания для дробей.

Особое внимание следует обратить на использование дружественных функций в шаблонах классов. Часто с помощью дружественных функций переопределяются операции ввода/вывода для конкретного класса. При определении таких функций для шаблона класса заранее неясен тип объекта, который следует ввести или вывести, т.к. он зависит от типа данных, для которого формируется объект на основе этого шаблона. Из этого вытекает следующее важное правило.

Правило 2. *Дружественные функции шаблона класса также должны быть шаблонными, и это должно быть явно указано при их объявлении.*

Для иллюстрации этого правила укажем, что к шаблону класса `QMatrix`, приведённому в Листинге 3 можно добавить содержимое Листинга 6.

```
1 template <class T>
2 friend istream& operator >> (istream&, QMatrix<T>&);
3
4 template <class T>
5 friend ostream& operator << (ostream&, const QMatrix<T>&);
```

Листинг 6: Заголовки шаблонов дружественных функций для шаблона класса `QMatrix`

Реализация данных дружественных функций может быть, например, такой как в Листингах 7 и 8

```
1 template <class T>
2 istream& operator >> (istream& in, QMatrix<T>& ob) {
```

```

3     if(ob.N != 0) {
4         for(int i = 0; i < ob.N; i++)
5             for(int j = 0; j < ob.N; j++)
6                 in >> ob.element[i][j];
7     }
8     return in;
9 }

```

Листинг 7: Шаблон операции ввода квадратной матрицы

```

1 template <class T>
2 ostream& operator << (ostream& out, QMatrix<T>& ob) {
3     if(ob.N != 0) {
4         out << "Matrix:" << endl;
5         for(int i = 0; i < ob.N; i++) {
6             for(int j = 0; j < ob.N; j++)
7                 out << ob.element[i][j] << "\t";
8             out << endl;
9         }
10    }
11    else
12        out << "Matrix is empty" << endl;
13    return out;
14 }

```

Листинг 8: Шаблон операции вывода квадратной матрицы

4 Специализация шаблонов

Язык C++ позволяет задать специализированный (отличный от общего) вид шаблона для конкретных частных случаев его параметров.

В качестве наглядного примера, рассмотрим следующую ситуацию. Предположим, нам потребовалось использовать шаблон функции сортировки, приведённый в Листинге 2, для сортировки массива указателей на строки (типа `char*`), чтобы упорядочить адресуемые ими строки в «алфавитном порядке». Проблема в том, что шаблон в той его версии, которая была приведена в Листинге 2, для сравнения элементов использует операцию «<», а это в данной ситуации, очевидно, приведёт к некорректному результату.

Во-первых, заменим символ «меньше» на вызов функции `sort_less`.

```

1 template <class T>
2 void sort(T *array, int len) {
3     for(int start = 0; ; start++) {
4         bool done = true;
5         for(int i = len-2; i >= start; i--)
6             if(sort_less(array[i+1], array[i])) {

```



```

7         T tmp = array[i];
8         array[i] = array[i+1];
9         array[i+1] = tmp;
10        done = false;
11    }
12    if(done)
13        break;
14 }
15 }

```

Листинг 9: Вторая версия шаблона функции для сортировки «пузырьком» произвольного массива

Во-вторых, саму функцию `sort_less` также опишем с помощью шаблона.

```

1 template <class T>
2 bool sort_less(T a, T b) {
3     return a < b;
4 }

```

Листинг 10: Шаблон функции `sort_less`

В результате для всех типов, для которых операция «меньше» имеет нужный нам смысл (обычное сравнение значений), шаблон `sort` продолжит работать по-прежнему; функцию `sort_less` компилятор всякий раз будет генерировать автоматически как обычное сравнение. Осталось предусмотреть особый случай для сравнения элементов типа `char*`, т.е. прибегнуть к **явной специализации**.

Описание случая явной специализации также начинается со слова `template`, однако угловые скобки после него оставляют пустыми, чтобы показать, что конкретно этот шаблон пишется для частного случая, не зависящего от дополнительных параметров. Далее записывается заголовок соответствующей шаблонной функции с указанием всех параметров² шаблона.

```

1 template<>
2 bool sort_less<const char*>(const char *a, const char *b) {
3     return strcmp(a, b) < 0;
4 }

```

Листинг 11: Специализация шаблона функции `sort_less`

Шаблоны классов также подвержены специализации. Не будем подробно обсуждать этот механизм для простейших ситуаций, заметим только, что он в целом аналогичен специализации шаблонов функций.

²В примере, приведённом в Листинге 11, параметр всего один.

Чуть более подробно остановимся на том факте, что язык C++ допускает для шаблонов классов **частичную специализацию**, при которой специализирующий вариант шаблона сам по себе тоже зависит от параметров. Такой специализатор начинается со слова `template`, после которого следует **непустой список параметров шаблона**. При этом параметры в заголовке шаблона в общем случае могут не совпадать (или совпадать не полностью) с параметрами описываемого шаблона. Фактические параметры шаблона указываются в угловых скобках после его имени.

Простейший пример ситуации с частичной специализацией шаблона класса — это случай, когда задаётся конкретное значение **нескольких, но не всех** параметров шаблона.

Пример 3. Частичная специализация шаблонов класса.

Пусть описан шаблон `Cls`, зависящий от двух параметров:

```
1 template <class A, class B>
2 class Cls {
3     // ...
4 };
```

Теперь можно описать специализированный вариант, например, для случая, когда параметр B есть тип `int`:

```
1 template <class X>
2 class Cls<X, int> {
3     // ...
4 };
```

Более сложный случай частичной специализации возникает, когда в заголовке шаблона указывается некий тип (`class T`), а в описываемом типе используется указатель или ссылка на `T`, например:

```
1 template <class Z>
2 class Foo {
3     // general realization of template Foo
4 };
5
6 template <class T>
7 class Foo<T*> {
8     // special realization of Foo for pointers
9 };
```

В последнем фрагменте кода мы сообщаем компилятору, что шаблон `Foo` следует инстанциировать специальным способом, если в качестве его параметра задан тип-указатель, и обычным способом — если заданный параметр не является указателем.

Замечание 3. Для шаблонов функций *частичная специализация запрещена*. Это является решением, не допускающим проблем, вызванных сочетанием частичной специализации с перегрузкой.

Замечание 4. Специализированный вариант шаблона в тексте программы *всегда должен располагаться после общего*.

5 Контрольные вопросы и задания

1. Почему функцию, приведённую в Листинге 1 нельзя использовать для сортировки массива элементов типа `double`? Можно ли это обойти, и как это сделать? К каким последствиям такое решение может привести?
2. По каким причинам решение “в лоб” с описанием функции `sort_double`, аналогичной функции `sort_int` Листинга 1, некорректно?
3. Прочитайте самостоятельно о многострочных макросах и макросах с параметрами препроцессора языка C. В чём их преимущества и недостатки? Можно ли использовать их в программах на языке C++? Если да, то рекомендуется ли это делать на практике и почему?
4. Каково предназначение шаблонов функций и механизма перегрузки функций? Действительно ли это одно и то же? Перечислите сходства и различия этих механизмов.
5. Обратите внимание на заголовок метода `Determinant`, приведённый в строке 14 Листинга 3, описывающего заголовок шаблона класса `QMatrix`. Приведите примеры ситуаций, когда использование такого шаблона метода некорректно. Продумайте возможные выходы из данных ситуаций.