УДК 004

ШАБЛОННЫЕ ФУНКЦИИ В ЯЗЫКЕ C++

Арташкин Евгений Павлович

студент

Мордовский государственный университет им. Н.П. Огарева, Саранск

author@apriori-journal.ru

Аннотация. В данной статье рассматривается типы шаблонов функций в языке программирования C++, их основное назначение, синтаксис конструкций. Описывается стандартная библиотека шаблонов, методы работы с ней.

Ключевые слова: C++; Visual Studio 2010; шаблон; функция; стандартная библиотека шаблонов.

FUNCTION TEMPLATES IN C++

Artashkin Evgeniy Pavlovich

student

Ogarev Mordovia State University, Saransk

Abstract. This article discusses the types of template functions in C ++ programming language, their primary purpose, syntax structures. Describes the standard template library, methods of working with it.

Key words: C++; Visual Studio 2010; template; function; standard template library.

Язык C предназначался для системного программирования при создании операционных систем, системных утилит и встраиваемого программного обеспечения. Он обладает всеми необходимыми для этого свойствами: программы, написанные на нем, очень эффективны, не требуют специальной среды поддержки времени выполнения [1].

Рассмотрим простой пример. Допустим, есть функция, которая меняет местами значения двух переменных типа int:

```
#include <iostream>
void my_swap ( int & first , int & second )
{
    int temp ( first ) ;
    first = second ;
    second = temp;
}

int main ()
{
    int a = 5 ;
    int b = 10 ;
    std::cout << a << " " << b << std::endl ;
    my_swap ( a , b ) ;
    std::cout << a << " " << b << std::endl ;
}
```

Теперь, допустим, у нас в функции main так же есть две переменные типа double, значения которых тоже нужно обменять. Функция для обмена значений двух переменных типа int нам не подойдет. Напишем функцию для double:

```
void my_swap ( double & first , double & second )
{
    double temp ( first ) ;
    first = second ;
    second = temp ;
}
```

И теперь перепишем main:

```
int main ()
{
    int a = 5 ;
    int b = 10 ;
    std::cout << a << " " << b << std::endl ;
    my_swap ( a , b ) ;
    std::cout << a << " " << b << std::endl ;
    double c = 77.89 ;
    double d = 54.22 ;
    std::cout << c << " " << d << std::endl ;
    my_swap ( c , d ) ;
    std::cout << c << " " << d << std::endl ;
}
```

Алгоритм абсолютно одинаковый, отличаются лишь типы параметров и тип переменной temp. А теперь представьте, что нам еще нужны функции для short, long double, char, string и еще множества других типов. Конечно, можно просто скопировать первую функцию, и исправить типы на нужные, тогда получим новую функцию с необходимыми типами. А если функция будет не такая простая? А вдруг потом еще обнаружится, что в первой функции была ошибка? Помогут шаблоны.

Шаблоны (англ. template) – средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Описание шаблона начинается с ключевого слова template за которым в угловых скобках («<» и «>») следует список параметров шаблона. Далее идет объявление шаблонной сущности (например функция или класс), т. е. имеет вид:template < template-parameter-list > declaration.

Теперь напишем шаблонную функцию my_swap. Исходя из упомянутой выше структуры объявления шаблона следует, что наша функция будет выглядеть так: template < параметры_шаблона > описание_функции.

Напишем функцию:

```
template < typename T >
void my_swap ( T & first , T & second )
{
    T temp(first) ;
    first = second ;
    second = temp ;
}
```

Typename в угловых скобках означает, что параметром шаблона будет тип данных. T – имя параметра шаблона. Вместо typename здесь можно использовать слово class: template < class T >. В данном контексте ключевые слова typename и class эквивалентны. Далее, в тексте шаблона везде, где мы используем тип T, вместо него будет проставляться необходимый тип.

void my_swap (T & first , T & second) //T – тип, указанный в //параметре шаблона

```
{
    T temp(first) ; //временная переменная должна быть того же типа,
//что и параметры
    first = second ;
    second = temp ;
}
```

Теперь напишем функцию main:

```
int main ()
{
    int a = 5 ;
    int b = 10 ;
    std::cout << a << " " << b << std::endl ;
    my_swap<int> ( a , b ) ;
    std::cout << a << " " << b << std::endl ;
    double c = 77.89 ;
    double d = 54.22 ;
    std::cout << c << " " << d << std::endl ;
    my_swap<double> ( c , d ) ;
    std::cout << c << " " << d << std::endl ;
}
```

После имени функции в угловых скобках указываем тип, который нам необходим, он то и будет типом T. Шаблон – это лишь макет, по которому компилятор самостоятельно будет генерировать код. При виде такой конструкции: my_swap<тип> компилятор сам создаст функцию my_swap с необходимым типом. Это называется инстанцирование шаблона. То есть при виде my_swap компилятор создаст функцию my_swap в которой T поменяет на int, а при виде my_swap будет создана функция

с типом double. Если где-то дальше компилятор опять встретит my_swap, то он ничего генерировать не будет, т.к. код данной функции уже есть (шаблон с данным параметром уже инстанцирован). Таким образом, если мы инстанцируем этот шаблон три раза с разными типами, то компилятор создаст три разные функции

Вывод типа шаблона исходя из параметров функции

Можно вызвать функцию my_swap не указывая тип в угловых скобках. В ряде случаев компилятор может это сделать за вас.

Рассмотрим вызов функции без указания типа:

```
int a = 5 ;
int b = 10 ;
my_swap ( a , b ) ;
```

Шаблонная функция принимает параметры типа T&, основываясь на шаблоне, компилятор видит, что ему передается в функцию аргументы типа int, поэтому может самостоятельно определить, что в данном месте имеется ввиду функция my_swap с типом int. Это deducing template arguments.

Рассмотрим пример посложнее. Например, программу сортировки массива (будем использовать сортировку «пузырьком»). Естественно, что алгоритм сортировки один и тот же, а вот типы элементов в массиве будут отличаться. Для обмена значений будем использовать нашу шаблонную функцию my_swap.

```
#include <iostream>
template < typename T >
void my_swap ( T & first , T & second ) //T - тип, указанный в
//параметре шаблона
{
```

```
    T temp(first) ; //временная переменная должна быть того же типа,
//что и параметры
    first = second ;
    second = temp ;
}

template < class ElementType > //Использовал class, но можно и
typename - без разницы
void bubbleSort(ElementType * arr, size_t arrSize)
{
    for(size_t i = 0; i < arrSize - 1; ++i)
        for(size_t j = 0; j < arrSize - 1; ++j)
            if (arr[j + 1] < arr[j])
                my_swap ( arr[j] , arr[j+1] ) ;
}

template < typename ElementType >
void out_array ( const ElementType * arr , size_t arrSize )
{
    for ( size_t i = 0 ; i < arrSize ; ++i )
      std::cout << arr[i] << ' ' ;
    std::cout << std::endl ;
}


int main ()
{
    const size_t n = 5 ;
    int arr1 [ n ] = { 10 , 5 , 7 , 3 , 4 } ;
    double arr2 [ n ] = { 7.62 , 5.56 , 38.0 , 56.0 , 9.0 } ;
```

```
    std::cout << "Source arrays:\n" ;
    out_array ( arr1 , n ) ;//Компилятор сам выведет параметр
//шаблона исходя из первого аргумента функции
    out_array ( arr2 , n ) ;

    bubbleSort ( arr1 , n ) ;
    bubbleSort ( arr2 , n ) ;

    std::cout << "Sorted arrays:\n" ;
    out_array ( arr1 , n ) ;
    out_array ( arr2 , n ) ;
 }
```

Вывод программы:

Source arrays: 10 5 7 3 4 7.62 5.56 38 56 9

Sorted arrays: 3 4 5 7 10 5.56 7.62 9 38 56

Как видно, компилятор сам генерирует out_array для необходимого типа. Так же он сам генерирует функцию bubbleSort. А в bubbleSort у нас применяется шаблонная функция my_swap, компилятор сгенерирует и её код автоматически.

Шаблонными могут быть не только функции. Рассмотрим шаблонные классы. Начнем с простого примера. Мы добавим в наш предыдущий код функцию, которая будет искать максимум и минимум в массиве. При создании функции «упираемся» в проблему — как вернуть два указателя? Можно передать их в функцию в качестве параметров, а можно вернуть объект, который будет содержать в себе два указателя. [2] Первый вариант при большом кол-ве возвращаемых значений приведет к заваливанию функции параметрами, поэтому лучше воспользоваться структурой:

```
struct my_pointer_pair
{
   тип * first ;
   тип * second ;
} ;
```

А какого же типа будут указатели? Можно сделать их void*, но тогда придется постоянно приводить их к нужному типу. Необходимо сделать эту структуру шаблонной:

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

Теперь компилятор при виде кода my_pointer_pair <тип1, тип2> сам сгенерирует нам код структуры с соответствующими типами. В данном примере указатели у нас будут одинакового типа, но структуру мы сделаем такой, чтобы типы указателей могли быть разными.

```
int main ()
{
 my_pointer_pair<int, double> obj = { new int(10) , new double(67.98) };
//Создаем объект типа my_pointer_pair<int,double>
      std::cout << *obj.first << ' ' << *obj.second << std::endl ;
      delete obj.first ;
      delete obj.second ;
}
```

Компилятор не будет автоматически определять типы для шаблона класса, поэтому необходимо их указывать самостоятельно.

Для классов мы должны явно указывать параметры шаблона. В стандарте C++11, устаревшее ключевое слово auto поменяло свое значение и теперь служит для автоматического вывода типа в зависимости от типа инициализатора, поэтому мы можем написать так:

```
auto mm = my_minmax_elements ( arr1 , n ) ;
```

Можно использовать еще одну функцию, которая будет выводить объект my_pointer_pair в стандартный поток вывода:

```
template < typename T1 , typename T2 >
void out_pair ( const my_pointer_pair< T1 , T2 > & mp )
{
    if ( mp.first == 0 || mp.second == 0 )
        std::cout << "not found" << std::endl ;
    else
        std::cout << "min = " << *mp.first << " max = " << *mp.second <<
std::endl ;
}
```

В комплекте с компилятором предоставляется стандартная библиотека шаблонов (Standard Template Library). Она содержит множество шаблонных функций и классов. Например, класс двусвязного списка(list), класс «пара» (pair), функция обмена двух переменных(swap), функции сортировок, динамически расширяемый массив(vector) и т.д. Всё это – шаблоны, их можно использовать.

Список использованных источников

1. Александров Э.Э., Афонин В.В. Программирование на языке C в Microsoft Visual Studio 2010. [Электронный ресурс]. Режим доступа: http://www.intuit.ru/department/pl/prcmsvs2010 (дата обращения: 01.11.2015).
2. Александров Э.Э., Афонин В.В. Введение в программирование в языке C. Саранск: Мордовский гос. университет им. Н.П. Огарева, 2009.