Лабораторная работа №8.
ШАБЛОНЫ И ШАБЛОННЫЕ ФУНКЦИИ В C++

Цель работы: Изучение технологии создания шаблонов и шаблонных функций.

Содержание

Шаблонные функции

Рассмотрим простой пример. Допустим, у нас 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 ;
}
```

Как видно из примера, оба алгоритма алгоритм абсолютно одинаковы и отличаются лишь типами параметров и типом 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? Избежать всего этого можно, например, «шаманством» с препроцессором, но это нам ни к чему, нам 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, собственно идет объявление шаблонной сущности (например функция или класс), т. е.

имеет вид:

```
template < template-parameter-list > declaration.
```

Исходя из упомянутой выше структуры объявления н апишем 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 T 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 давайте 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 ;
}
```

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<int> компилятор создаст функцию my_swap в которой T поменяет на int, а при виде my_swap<double> будет создана функция с типом double. Если где-то дальше компилятор опять встретит my_swap<int>, то он ничего генерировать не будет, т.к. код данной функции 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

На самом деле, можно вызвать функцию my_swap не указывая тип в угловых скобках. В ряде случаев компилятор может это сделать самостоятельно.

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 Вы передаете 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. Теперь давайте напишем 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 ;
}

//Функция будет принимать указатель на данные
```

```
//и кол-во элементов массива данных
//Сам алгоритм сортировки можете посмотреть в Интернете.
//Никаких оптимизаций и проверок аргументов применять не будем, нам нужна просто демонстрация.
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
```

Таким образом, 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. Удобно, не правда ли?

Введение в шаблонные классы

Шаблонными могут быть не только функции. Рассмотрим шаблонные классы. Добавим в программный 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. Для этого потребуется создать структуру:

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

А какого же типа будут указатели? Можно сделать их void*, но тогда придется постоянно кастовать их к нужному типу. А что, если 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. Это может быть полезно в других примерах (в данном случае я просто хотел показать, что у шаблона может быть не только один параметр).

```
int main ()
{
    my_pointer_pair<int,double> obj = { new int(10) , new double(67.98) } ;//Создаем объект типа my_pointer_pair<in
    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.

Напишем код шаблонной функции для поиска максимума и минимума:

```
//Шаблон наш будет с одним параметром - тип элементов массива (T)
//Возвращаемое значение - объект типа my_pointer_pair< T , T >
//т.е. first и second в my_pointer_pair будут иметь тип T*.
template <  typename T >
my_pointer_pair< T , T > my_minmax_elements (  T * arr , size_t arrSize )
{
    my_pointer_pair< T , T > result = { 0 , 0 } ;
    if ( arr == 0 || arrSize < 1 )
         return result ;
    result.first = arr ;
    result.second = arr ;
    for ( size_t i = 1 ; i < arrSize ; ++i )
    {
        if ( arr[i] < *result.first )
            result.first = arr+i ;
        if ( arr[i] > *result.second )
           result.second = arr+i ;
    }
    return result ;
}
```

Теперь мы можем вызывать данную функцию:

```
my_pointer_pair< int , int > mm = my_minmax_elements ( arr1 , n ) ;
```

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 ) ;
```

Предлагаю написать 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 ;
}
```

Теперь соберем всё воедино:

```
#include <iostream>


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 ;
}


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


//Шаблон наш будет с одним параметром - тип элементов массива (T)
//Возвращаемое значение - объект типа my_pointer_pair< T , T >
//т.е. first и second в my_pointer_pair будут иметь тип T*.
template <  typename T >
my_pointer_pair< T , T > my_minmax_elements (  T * arr , size_t arrSize )
{
    my_pointer_pair< T , T > result = { 0 , 0 } ;
    if ( arr == 0 || arrSize < 1 )
         return result ;
    result.first = arr ;
```

```
    result.second = arr ;
    for ( size_t i = 1 ; i < arrSize ; ++i )
    {
        if ( arr[i] < *result.first )
            result.first = arr+i ;
        if ( arr[i] > *result.second )
           result.second = arr+i ;
    }
    return result ;
}

template  < typename T >
void out_pair ( const my_pointer_pair< T , T > & 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 ;
}

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 << "Arrays:\n" ;
    out_array ( arr1 , n ) ;//Компилятор сам выведет параметр шаблона исходя из первого аргумента функии
    out_array ( arr2 , n ) ;

    out_pair ( my_minmax_elements ( arr1 , n ) ) ;
    out_pair ( my_minmax_elements ( arr2 , n ) ) ;
}
```

Результат работы программы:

```
Arrays: 10 5 7 3 4 7.62 5.56 38 56 9 min = 3 max = 10 min = 5.56 max = 56
```

Шаблоны и STL

В комплекте с компилятором обычно предоставляется стандартная библиотека шаблонов (Standart 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 которые можно использовать вместо того, чтобы изобретать велосипед. Для небольшого примера возьмем std::vector:

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

int main ()
{
    std::vector <int> arr;
    arr.push_back ( 5 ) ; //Добавляем элемент в конец
    arr.push_back ( 7 ) ;
    arr.push_back ( 3 ) ;
    arr.push_back ( 8 ) ;
    std::cout << "Source vector:\n" ;
    for ( size_t i = 0 , size = arr.size() ; i < size ; ++i )
        std::cout << arr[i] << ' ' ;
    std::cout << std::endl ;

    std::sort ( arr.begin() , arr.end() ) ; //Сортируем вектор

    std::cout << "Sorted vector:\n" ;
    for ( size_t i = 0 , size = arr.size() ; i < size ; ++i )
        std::cout << arr[i] << ' ' ;
   std::cout << std::endl ;
}
```

Заключение

Авторы, которые писали std::vector, понятия не имели, элементы какого типа Вы будете хранить. Изучив технологию создания и применения шаблонов, Вы получите мощный инструмент, позволяющий повысить эффективность разработки и отладки разрабатываемых программ.

Задание

Изучите возможности библиотеки STL и модифицируйте код Вашего приложения из лабораторной работы №1 (8).

[В начало]