Шаблоны

1. Механизм шаблонов реализует в C++ **параметрический полиморфизм.**

2. Шаблон представляет собой предварительное описание **функции** или **класса**, конкретное представление которых зависит от параметров шаблона.

3. Для описания шаблонов используется ключевое слово **template**, вслед за которым указываются аргументы (параметры шаблона), заключенные в угловые скобки.

4. Параметры шаблона перечисляются через запятую, и могут быть:
   а) именами типов (перед именем типа надо указывать ключевое слово ***class*** или ***typename***).
   б) объектами следующих типов:
      - **целочисленного,**
      - **перечислимого,**
      - **указательного (в том числе указатели на члены класса),**
      - **ссылочного;**

5. Параметры-объекты являются **константами**, их нельзя изменять внутри шаблона.

Шаблоны функций.

template **<** *список_параметров_шаблона* **>**
тип_рез-та имя_функции (список_аргументов_функции) { /...*/ }*

Обращение к функции-шаблону: *имя_функции* **<** *список_фактич._пар._шаблона* **>**
(список_фактич_аргументов_функции);

Пример: функция суммирования элементов типа *T* массива размера *size*

```
template < class T >
T  sum ( T  array[ ],  int size ) {
        T  res = 0;
        for ( int i = 0; i < size; i++ )    res += array[ i ];
        return res;
 }
```

Использование шаблона для массива из 10 целочисленных элементов :

```
int  iarray [10];
int  i_sum;
//...
i_sum = sum < int > ( iarray, 10 );
```

Можно задать аргумент ***size*** в виде параметра шаблона:

```
template < class  T, int  size >
T  sum ( T  array [ ] ) { /* ... */ }
```

Тогда вызов *sum* будет таким: `i_sum = sum < int, 10 > ( iarray );`

Неявное определение параметра-типа шаблона
Пример 1.

```
class complex {...
public:
        complex ( double r = 0, double i = 0 );
        operator double ();       .....
};
template < class T >
T  f ( T& x, T& y )  {
        return x > y ? x : y;
}
double f ( double x, double y ) {
        return x > y ? -x : -y;
}
int main ( ) {
        complex a ( 2 , 5 ), b ( 2 , 7 ), c;
        double x = 3.5, y = 1.1;
        int  i,  j = 8,  k = 10;
        c = f ( a , b );        // f < complex > (a , b)
        x = f ( a , y );        // f (a , y)
        i = f ( j , k );        // f  < int > (j , k)
        return 0;
}
```

Пример 2.

```
template < class T >
T max (T & x, T & y) {
    return x > y ? x : y;
}

int main ( ) {
    double x = 1.5,  y = 2.8,  z;
    int  i = 5, j = 12,  k;
    char * s1 = "abft";
    char * s2 = "abxde", * s3;

    z = max ( x, y );                  // max <double>
    k = max < int > (i, j);            // max <int>
    //z = max (x, i);                  // Err! -  неоднозначный выбор параметров
    z = max < double > ( y, j );
    s3 = max  (s2, s1);                // max < char * >,
                                       // но происходит сравнение адресов
    return 0;
}
```

Пример 3.

```
template <class T>  T  m1 (T a, T b) {
  cout << "m1_1\n";
  return a < b ? b : a;
}

template <class T, class B> T m1 (T a, T b, B c) {
 cout << "m1_2\n";
 c = 0;  return a < b ? b : a;
}
template <class T, class Z> T m1 (T a, Z b) {
 cout << "m1_3\n";
 return a < b ? b : a;
}
```

```
int m1 (int a, int b) {
        cout << "m1_4\n";
        return a < b ? b : a;
}

int m1 (int a, double b) {
        cout << "m1_5\n";
        return a;
}
```

```
int main () {
    int i;
    m1 <int> (2, 3);              // m1_1
    m1 <int, int> (2, 3);         // m1_3
    m1 <int> (2, 3, i);           // m1_2
    m1 (1, 1);                    // m1_4
    m1 (1.3, 1);                  // m1_3
    m1 (1.3, 1.3);                // m1_1
    return 0;
}
```

// Если убрать первый шаблон:

```
// m1_3
// m1_3
// m1_2
// m1_4
// m1_3
// m1_3
```

Алгоритм выбора оптимально отождествляемой функции с учетом шаблонов

- Для каждого шаблона, **подходящего** по набору формальных параметров функции, осуществляется формирование перегруженной функции, соответствующей списку фактических параметров шаблона.

- Если есть два шаблона функции и один из них более специализирован (т.е. каждый его допустимый набор фактических параметров также соответствует и второй специализации), то далее рассматривается только он.

- Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству и типу параметров. При этом если параметры некоторого шаблона функции были определены путем **выведения** по типам фактических параметров вызова функции, то при дальнейшем поиске оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **Точного** отождествления.

- Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.

- Если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Шаблоны классов.

Шаблоны создаются для классов, имеющих общую логику работы.

Для определения шаблона класса перед ключевым словом **class** помещается **template**-квалификатор.

template **<***список_параметров_шаблона***> class** *имя_класса { /*...*/ };*

Конкретный экземпляр шаблона класса (объект класса) можно создать так:

имя_класса **<** *список фактич_парам* **>** *объект;*

Для шаблонов класса никакие фактические параметры по умолчанию не выводятся.

Функции-члены класса-шаблона **автоматически** становятся функциями-шаблонами.

Шаблоны методов.

Можно описывать шаблонные методы в классах, не являющихся шаблонами.

Запрещено определять шаблоны для виртуальных методов, из-за возникающих больших накладных расходов на возможную перестройку таблиц виртуальных методов при компиляции.

Шаблонный класс *stack*.

```
template <class T, int max_size >
class stack {
        T s [max_size];
        int top;
public:
        stack ( ) { top = 0;}
        void reset ( ) { top = 0;}
        void push (T i);
        T pop () ;
        bool is_empty ( ) { return top == 0;}
        bool is_full  () { return top == max_size;}
};
```

```
template <class T, int max_size >
void stack <T, max_size > :: push (T i) {
        if ( ! is_full () ) {
                s [top] = i;
                top ++;
        }
        else
                throw "stack_is_full";

}
```

```
template <class T, int max_size >
T stack <T, max_size > :: pop ( ) {
        if ( ! is_empty () ) {
                top --;
                return s [top];
        }
        else
                throw
                "stack_is_empty";
}
```