

Шаблоны (Templates)

Идея шаблонов

- Некоторые алгоритмы слабо зависят от типа переменных, но зависят от «интерфейса данных»: возможности сравнить, присвоить, сложить...
- Достаточно написать «образец» такого алгоритма и компилятор будет модифицировать его под нужный тип переменных

Пример: алгоритм обмена значениями

```
void swap (int& a, int& b) {  
    int tmp = a; a = b; b = tmp;  
}
```

👉 Функция для `double` или `Rational` или двух объектов произвольного класса **будет отличаться только типом переменных**

В C++ имеется возможность «параметризовать» тип с помощью шаблонов:

```
template<class TYPE>  
или    template<typename TYPE>
```

Обратите внимание

👉 В этом контексте использование служебных слов `class` и `typename` практически эквивалентно, а начиная с C++17 полностью эквивалентно

Шаблоны функций

- 👉 Шаблон для функции получается добавлением перед функцией `template<class TYPE>` и `TYPE` используется как «параметр типа»

Пример ① функция Swap()

```
template <class T>
void Swap (T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}

int a = 1, b = 10;
Swap (a, b);                // или Swap<int>(a,b)
Point p1(1,1), p2(-1,2);
Swap (p1, p2);              // или Swap<Point>(p1,p2)
```

- 👉 Здесь `T` – «имя типа» и так же как с обычной переменной `T` можно заменить на любое имя, например на `MyType`

Пример ② возведение в квадрат

```
template <class Atype>
inline Atype SQR (Atype x) {
    return(x*x);
}
```

Проверка

```
cout<<" SQR(5)= "          <<SQR(5)          <<endl; //SQR(5)= 25
cout<<" SQR(sin(0.5))= " <<SQR(sin(0.5))<<endl; //SQR(sin(0.5))=0.229849
cout<<" SQR('c')= "      <<SQR('c')          <<endl; //SQR('c')= I
cout<<" SQR<int>('c')= " <<SQR<int>('c')<<endl; //SQR<int>('c')= 9801
```

👉 Тип 'Atype' должен иметь операцию умножения

```
cout << SQR("c") << endl;
```

Compilation error: invalid operands of types 'const char*' and 'const char*' to binary 'operator*'

Пример 3 минимальное из двух чисел

```
template <class T>
T Min(T a, T b) {
    return (a < b) ? a : b;
}
```

👉 В типе 'T' должен быть определён оператор<()

```
int x = 10, y = 12;
cout << " Min(x,y)=" << Min(x,y) << endl;      // Min(x,y)=10
Min<int&>(x,y) = 1; // присвоить минимальному числу значение
cout << " x = " << x << " y = " << y << endl; // x = 1 y = 12

cout << Min("A","B") << endl;                    // B - сравнили char* указатели!
cout << Min<string>("A","B") << endl;             // A - OK!
```

Шаблоны классов

- 👉 впереди ставится `template<class TYPE>`
- 👉 `TYPE` используется внутри класса как «параметр типа»

Пример: стек для объектов произвольного типа

```
template <class T>
class TStack {
public:
    TStack(int size = 10)    {stackPtr = new T[size];}
    ~TStack()                {delete [] stackPtr;}
    void push(const T& item) {stackPtr[++top] = item;}
    T pop()                  {return stackPtr[top--];}
private:
    int top = 0;
    T* stackPtr = nullptr;
};
```

Использование:

```
typedef TStack<float> FloatStack;
typedef TStack<int>    IntStack;

FloatStack Fs;
Fs.push(2.31);
Fs.push(1.19);
Fs.push(Fs.pop()+Fs.pop());
cout << "FloatStack: " << Fs.pop() << endl; // FloatStack: 3.5

IntStack Is;
Is.push(2);
Is.push(1);
Is.push(Is.pop()+Is.pop());
cout << "IntStack: " << Is.pop() << endl;    // IntStack: 3
```

👉 Начиная с C++14 можно определять шаблоны отдельных переменных

Пример

```
template<class T> T one = T(1);  
...  
one<int> = 2;                // reassign value  
printf("%d\n", one<int>);    // 2  
printf("%.2f\n", one<double>); // 1.00
```

👉 Обычно применяют для констант таких как Pi

```
template<class T> constexpr T pi = T(3.1415926535897932385L);  
...  
constexpr double sqr2pi = sqrt(2*pi<double>); // 2.50663
```


Немного больше о шаблонах

Несколько типов в шаблоне

```
template<class T1, class T2>
struct Pair {
    T1 first;
    T2 second;
};
```

```
Pair<int, double> pr(1, 10.);
```

non-type параметры в шаблоне

```
template<class T, int Size>
T sum_elements(T v[]) {
    T sum; // default ctr!
    for(int i=0; i<Size; i++) sum+=v[i];
    return sum;
}
```

```
int a[] {1,2,3,4,5};
int s5=sum_elements<int, 5>(a); // 15
int s3=sum_elements<int, 3>(a); // 6
```

👉 В случае шаблонов запись `T var;` означает вызов конструктора по умолчанию: «встроенные» типы `int` и `double` инициализуются нулями

Советы и предостережения

- Используйте шаблоны самостоятельно только для **простых задач**
- Не изобретайте велосипед: пользуйтесь библиотеками (**STL, BOOST ...**)
- Не верьте в чудеса: разберитесь, что делают библиотеки, как они это делают, и какова цена их использования

Анонимные == безымянные

- Общая форма: `[capture] (parameters) -> return_type {body}`
Краткая: `[capture] (parameters) {body}`
- Объявляется в месте использования, то есть внутри обычных функций
- По существу является функтором (объект с вызовом как функция) и часто присваивается переменной с типом `auto` для дальнейшего использования

Пример:

```
int main() {  
    auto hw = [](){cout << "hello world\n";};  
    hw(); // hello world  
}
```

● захват (capture) переменных

👉 в `lambdas` можно использовать локальные переменные функции

`[]`: внешние переменные не используются, нет захвата

`[=]`: все переменные по значению, копируются при создании функции и используются как константы

`[&]`: все переменные по ссылке

`[a,b,&c]` перечисление: переменные `a,b` по значению и `c` по ссылке

```
int x = 0, y = 0;
auto f1 = [x,&y](const string& s) {
    cout << s << " x=" << x << " y=" << y << endl;
    // x++; read-only variable 'x'
    y++;
};
x = y = 1;
f1("first:");           // first: x=0 y=1
f1("second:");          // second: x=0 y=2
cout << "x=" << x << " y=" << y << endl; // x=1 y=3
```

● Объявление lambdas со служебным словом mutable

`[capture](parameters) mutable -> return_type {body}`

👉 позволяет модифицировать переменные захваченные по значению

```
x = y = 0;
auto f2 = [x,&y](const string& s) mutable {
    cout << s << " x=" << x << " y=" << y << endl;
    x++; // OK!
    y++;
};
x = y = 10;
f2("first:"); // first: x=0 y=10
f2("second:"); // second: x=1 y=11
cout << "x=" << x << " y=" << y << endl; // x=10 y=12
```

Тип анонимной функции

👉 В ряде случаев требуется указать полную спецификацию анонимной функции: тип аргументов, возвращаемый тип

Например при рекуррентном вызове: лямбда факториал

```
auto f = [&f](int n) -> int{return n < 2 ? 1 : n*f(n-1);};
```

ERROR: use of 'f' before deduction of 'auto'

① STL-шаблон задающий тип функционального объекта

```
std::function<result_type(argument_type_list)>
```

- `argument_type_list` – список типов передаваемых аргументов
- `result_type` – тип возвращаемого значения


```
#include <functional> // header for std::function
function<int(int)> f = [&f](int n)->int{return n < 2 ? 1 : n*f(n-1);};
cout << " 10! = " << f(10) << endl; // 10! = 3628800
```

2 оператор `decltype(expression)`

 компилятор заменяет `decltype(exp)` на **тип** выражения `exp`

Пример: тип в параметре шаблона `std::map`

```
auto LtStr = [](const char* s1, const char* s2) -> bool
    {return strcmp(s1, s2) < 0;};
map<const char*, int, decltype(LtStr)> months(LtStr);
```

 анонимная функция с аргументом типа `auto`, что, по существу, эквивалентно шаблону для типа аргумента

① пример: возведения в квадрат

```
auto SQR = [](auto x) {return x*x;};  
cout << SQR(3) << endl;    // 9  
cout << SQR(1.1) << endl;  // 1.21
```

② пример: универсальный print

```
auto print = [](auto x) {cout << x << endl;};  
print("test"); // test  
print(3.1415); // 3.1415
```


👉 Появилась возможность использовать анонимные функции в `constexpr` выражениях

Пример №1: с явной декларацией в спецификации `lambdas`

```
auto pow3 = [](auto x) constexpr {return x*x*x;};  
constexpr double C = pow3(1.1);  
cout << " C= " << C << endl; // C= 1.331
```

Пример №2: неявно, компилятор поставит спецификацию за вас

```
constexpr auto Len = [](const auto& cont) {return cont.size();};  
constexpr array a {1,2,3};  
static_assert(Len(a)==3,"wrong number of elements"); // compilation  
cout << "Len(a)= " << Len(a) << endl; // Len(a)= 3
```