

# Шаблоны (Templates)

## Идея шаблонов

- Некоторые алгоритмы слабо зависят от типа переменных, но зависят от «интерфейса данных»: возможности сравнить, присвоить, сложить. . .
- Достаточно написать «шаблон» такого алгоритма и компилятор будет модифицировать его под нужный тип переменных
- 👉 Шаблоны можно рассматривать как параметрический полиморфизм средствами компилятора

## Пример: алгоритм обмена значениями

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

👉 функции для `double` или `Rational` отличаются только типом переменных

В C++ имеется возможность «параметризовать» тип  
`template<class TYPE>`    или    `template<typename TYPE>`

### Обратите внимание

- В `template<>` 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\")= " << 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;
}
```

```
int x = 10, y = 12;
cout<<"Min(x,y)="<<Min(x,y)<<endl; // Min(x,y)=10
cout<<Min(1,1.5)<<endl; // ERROR no func: Min(int, double)
cout<<Min<double>(1,1.5)<<endl; // 1 explicitly converts int -> double
// reference as a type of template
Min<int&>(x,y) = 1; // присвоить минимальному числу значение
cout << " x = " << x << " y = " << y << endl; // x = 1 y = 12
```

👉 наличие оператора < не гарантирует правильности ответа

```
// pointer as a type of template
cout<<Min("A","B")<<endl; // => Min<const char*>("A","B")
// сравнение указателей может вернуть B
cout<<Min<string>("A","B")<<endl; // A conts char* -> string
```

# Явная специализация шаблона

👉 Позволяет дополнить или изменить шаблон для конкретного типа

specialization Min(T a, T b) for const char\*

```
template <>
const char* Min(const char* a, const char* b) {
    return std::strcmp(a, b) <= 0 ? a : b ;
}
```

```
cout << Min("Bony", "Bob") << endl; // Bob
```

specialization SQR(Atype x) for char

```
template <>
inline char SQR (char c) { // toy example
    if ( std::isupper(c) ) { return std::tolower(c);
    } else {
        return std::toupper(c); }
}
```

```
cout<<SQR('c')<<endl<<SQR('C')<<endl; // SQR('C')=c SQR('c')=C
```

# 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
```



# Шаблоны переменных (C++14)

☞ Можно определять шаблоны отдельных переменных

- обычно применяют для констант таких как  $\pi$

## Пример

```
template<class T> constexpr T pi = T(3.1415926535897932385L);
template<class T> const T N1 = T(0); // universal zero
int main() {
    constexpr double twoPI2 = 2*pi<double>*pi<double>;
    cout << twoPI2 << endl; // 19.7392

    printf("%d, %.1f\n",N1<int>,N1<double>); // 0, 0.0

    N1<int> = 1; // ERROR: read-only
}
```

# Шаблоны с двумя и более типами

## ● Класс

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

Pair<int,double> pid {1,2.};
cout<<pid.first<<"", "<<pid.second<<endl; // 1, 2
```

## ● Функция

```
template <class T, class U>
auto MinTU(T a, U b) -> std::common_type_t<T,U> {
    return (a < b) ? a : b;
}


cout<<Min(1,1.5)<<endl; // ERROR no Min(int, double)
cout << MinTU(1,1.5) << endl; // 1
```

# Объявление возвращаемого типа (C++11)

Trailing return type: `auto fun() -> return_type {...}`

- перед именем функции должно стоять `auto`
- после стрелочки стоит выражение определяющее возвращаемый тип
- если выражение со стрелочкой отсутствует, возвращаемый тип определяется по типу в операторе `return` (return type deduction)

В `return_type` может стоять:

- обычные типы: `int`, `double`, `bool` ...
  - оператор `decltype(exp)`: компилятор определяет **тип** выражения `exp`
-  в нашем примере стоит `std::common_type_t<T,U>`: компилятор возвращает «общий тип» `T` и `U`

# Non-type параметры шаблона

👉 Если параметр шаблона не тип, то должен сам иметь «один из документированных» типов

● **Пример: Size is an integral non-type parameter; = default values**

```
template<class T = double, int Size = 0>
T SumElements(T v[], int n = Size) {
    T sum = 0; // N1<T>
    for(int i = 0; i < n; i++) {sum += v[i];}
    return sum;
}

int a[] {1,2,3,4,5};
auto sum = SumElements(a,std::size(a)); // 15
auto sum3 = SumElements(a,3); // 6
std::array A {1.1,2.2};
auto sumA = SumElements(A.data(),std::size(A)); // 3.3
```

# Шаблоны с пакетом параметров (C++11)

## Variadic template

- Шаблон функции или класса с параметрами заданными в виде так называемого пакета параметров (**parameter pack**)  

```
template<typename ... Args> void fun(Args ... args); // функция  
template<typename ... Args> struct some_type;      // класс
```
- Пакет параметров может содержать ноль или более аргументов шаблона: типов, non-type или других шаблонов
- Используют вариативные шаблоны: отделяют первый аргумент от остальных и затем рекурсивно вызывают шаблон для хвоста  
☞ В C++17 введены «свертки» для упрощения написания

## Пример: функция для суммы произвольного числа аргументов

```
auto SumCpp11() { // must be first  
    return 0;  
}
```

```
template<class Head, class... Tail>    // Variadic template  
auto SumCpp11(Head h, Tail... t) {  
    return h + SumCpp11(t ...);  
}
```

---

\_\_\_\_\_ тест SumCpp11() \_\_\_\_\_

```
auto sum11 = SumCpp11(1,2,3,4,5);  
cout << sum11 << endl; // 15
```

```
auto fsum11 = SumCpp11(1,3.1,'0');  
cout << fsum11 <<endl; // 52.1
```

---

# Выражение для свертки (C++17)

## Fold expressions

инструкции для применения бинарного оператора  $\odot$  к пакету параметров шаблона:

$(E_{\text{pack}} \odot \dots) \equiv (E_1 \odot (E_2 \odot (\dots \odot E_n)))$  – правая унарная свертка

$(\dots \odot E_{\text{pack}}) \equiv (((E_1 \odot E_2) \odot \dots) \odot E_n)$  – левая унарная свертка

$(E_{\text{pack}} \odot \dots \odot \text{Ini}) \equiv (E_1 \odot (E_2 \odot (\dots \odot (E_n \odot \text{Ini}))))$  – правая бинарная свертка

$(\text{Ini} \odot \dots \odot E_{\text{pack}}) \equiv (((((\text{Ini} \odot E_1) \odot E_2) \odot \dots) \odot E_n)$  – левая бинарная свертка

## предыдущая функция с использованием правой унарной свертки

```
template<typename... Args>
auto Sum(Args... args) {
    return (args + ...); // right unary fold
}

_____ тест _____
auto fsum = Sum(1.,3.1,'0'); // fsum=52.1
```

### пример: свертка с оператором запятая

```
template<typename T, typename ... Args>
void FoldPushBack(std::vector<T>& v, Args&& ... args) {
    (v.push_back(std::forward<Args>(args)), ...); // right unary fold
}
```

---

тест

---

```
vector<double> Vec;
FoldPushBack(Vec, 1.1,2.2,3.3,4.4,5.5);
for (const auto& v : Vec ) { cout << v << " "; }
cout << endl; // 1.1 2.2 3.3 4.4 5.5
```

---

### пример: функция печати с левой бинарной сверткой

```
template<typename ... Args>
void FoldPrint(Args&& ... args) {
    (cout << ... << std::forward<Args>(args)) << '\n'; // left binary
}
```

---

тест

---

```
FoldPrint("I=",1," Pi=",3.14," Func=",__func__);
Output: I=1 Pi=3.14 Func=main
```

---



# Универсальная передача параметров в функцию

## Universal (perfect) forwarding (C++11)

- 👉 Если объект надо только передать через параметр одной функции дальше в другую функцию, используйте конструкцию:

```
_____ perfect forwarding arg to fun() _____  
template<typename T>  
void callFun(T&& arg) {                               // void callFun(auto&& arg) {  
    fun(std::forward<T>(arg));  
}
```

- 1 тип параметра `callFun` объявляется как шаблон на R-ссылку, без `const`
- 2 параметры в `fun()` передаются через функцию `std::forward<>()`
  - если `arg` rvalue, то эта функция эквивалентна `std::move()`
  - в противном случае функция «вернет» `arg`

```
struct X {...}; // произвольный класс
X v;           // объект класса - lvalue
const X c;     // - const lvalue
X{}           // в выражениях - rvalue
// три перегруженные функции для объектов класса X
void fun(X& x) {...}; // fun for variable (lvalue)
void fun(const X& x) {...}; // fun for read-only access
void fun(X&& x) {...}; // fun for rvalue (move semantics)
```

### без шаблонов нужны три функ.

```
void callFun(const X& arg) {
    fun(arg); // => fun(const X&)
}
void callFun(X& arg) {
    fun(arg); // => fun(X&)
}
void callFun(X&& arg) {
    fun(std::move(arg)); //=>fun(X&&)
}
```

### с правильным шаблоном

```
callFun(v); // => fun(X&)
callFun(c); // => fun(const X&)
callFun(X{}); // => fun(X&&)

callFun(std::move(v)); //=>fun(X&&)
callFun(std::move(c)); //=>fun(const X&)
```

# Анонимные функции, lambdas (C++11)

## Анонимные == безымянные

- Общая форма: `[capture] (parameters) -> return_type {body}`  
Краткая: `[capture] (parameters) {body}`  
☞ используется trailing return type синтаксис
- Объявляется в месте использования, то есть внутри обычных функций
- Лямбда это функтор: объект с вызовом как у функции, его можно сохранить в `auto` переменную для дальнейшего использования

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

## ● захват переменных (capture)

☞ В квадратных скобках, перед списком аргументов, перечисляются локальные переменные которые можно использовать в `lambdas`:

[ ] внешние переменные не используются, нет захвата

[=] при создании `lambdas` все переменные копируются по значению и используются как константы

[&] все переменные используются как ссылки

[a,b,&c] `a,b` копируются по значению (`const`), `c` по ссылке

[x=sin(a),&ra=a] C++14: позволяет в списке захвата создавать переменные с инициализацией, при этом тип этих переменных определяется как если бы перед ними стояло `auto`  
`x` новая константная переменная, `ra` новая ссылка

## Пример с захватом локальных переменных

```
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
```

## Пример с инициализацией в списке захвата

```
double d = 0.1;
auto f14 = [x=sin(d),&y=d](const string& s) {
    cout << s << " x=" << x << " y=" << y++ << endl;
};
f14("capture initializer:"); // x=0.0998334 y=0.1
cout << "d=" << d << endl;  // d=1.1
```

## ● 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) {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);
```

# Обобщенная анонимная функция (C++14)

## Generic lambdas

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

## C++20: функция с аргументом типа `auto`

```
void f1(auto a); // same as template<class T> void f1(T a)
```

## ● пример: универсальная функция возведения в квадрат

```
auto SQ = [](auto x) -> decltype(x) {  
    return x*x;  
};  
cout << SQ(3) << ", " << SQ(1.1) << endl; // 9, 1.21
```

## ● Using forwarding (universal) references

```
auto pr2 = [](auto&& x, auto&& y) -> void {  
    cout << x << y << endl;  
};  
auto pi2 { SQ(M_PI) }; // variable  
pr2("SQ(pi)=", pi2); // SQ(pi)=9.8696
```

## ● Generic lambdas and fold expression (C++17)

```
auto prtline = [](auto&& ... args) {  
    ((cout << args << " "), ...) << '\n'; // left binary fold  
};  
prtline("pi=", M_PI, "sqrt(pi)=", sqrt(M_PI));  
  
pi= 3.14159 sqrt(pi)= 1.77245
```

# Анонимные функции в константных выражениях

## Constexpr lambdas (C++17)

👉 В C++17 можно использовать lambdas в `constexpr` выражениях

### ① с явной декларацией в спецификации lambdas

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

### 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
```

Дополнительные слайды

# Псевдонимы типов в шаблонах (C++11)

## Using declaration for type aliases

- Использование `using` для псевдонимов типов похоже на `typedef`, но совместимо с шаблонами

```
template <class T>
using Vec = std::vector<T>;
Vec<int> v; // the same as std::vector<int>
```

- кроме того псевдонимы типов с `using` легче читаются

## using for pointer to function

```
using PF = int (*)(int);
template<class T>
using PFT = int (*)(T);
```

## typedef for pointer to function

```
typedef int (*PF)(int); // OK
template<class T>
typedef int (*PFT)(T); // error
```