

# Шаблоны функций и шаблоны классов

Тема 10

# Шаблоны функций

```
template<class T>
void printArray(T *array, const int count){
    for (int i = 0; i< count; i++)
        std::cout << array[i] << '\t';
    std::cout << std::endl;
}
```

```
int main(){
    const int aCount = 5, bCount = 7, cCount = 6;
    int a[aCount] = {1,2,3,4,5};
    float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
    char c[cCount] = "HELLO";
    std::cout << "Array of integer: \n";
    printArray(a, aCount); //printArray<int>(a, aCount);
    std::cout << "Array of float: \n";
    printArray<float>(b, bCount); явный аргумент шаблона
    std::cout << "Array of char: \n";
    printArray(c, cCount); //printArray<char>(c, cCount);
    return 0;
}
```

Выведение  
(deduction)

Выведение  
(deduction)

# Шаблоны функций

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

```
int main(){  
    int a = 5, b = 7;  
    double c = 3.3, d = 5.6;  
    swap(a, b);  
    swap(c, d);  
    swap(a, d); //ошибка  
}
```

# Шаблоны функций

```
template <class X, class Y>
void foo(X x, Y y) { }
```


```
int main()
{
    foo(4.5, 5);           //выведение
    foo(4, 5);             //выведение
    foo<int, double>(5, 6); //приведение типов
    foo<int, double>(4.5, 5); //приведение типов
}
```

# Явный аргумент шаблона

```
template <class T>
T* foo() {
    return new T();
}
```

```
int main() {
    std::cout << *foo<int>() << std::endl;
    std::cout << *foo<double>() << std::endl;
    int * p = foo(); //ошибка
    int * p = foo<int>(); //хорошо
}
```

явный аргумент шаблона



# Перегрузка шаблонов функций

```
template<class T> T foo(T);  
template<class T> vector<T> foo(vector<T>);  
int foo(int);
```

```
void f()  
{  
    vector<double> z;  
    foo(2);    //foo(int);  
    foo(2.0);  //foo<double>(double);  
    foo(z);    //foo<double>(vector<double>);  
}
```

# Специализация шаблона

```
template<class T>
void printArray(T *array, const int count){
    for (int i = 0; i < count; i++)
        std::cout << array[i] << '\t';
    std::cout << std::endl;
}
```

```
//вариант 1
template<>
void printArray<char*>(char **array, const int count){
    for (int i = 0; i < count; i++)
        std::cout << array[i] << std::endl;
}
```

```
//вариант 2
template<>
void printArray(char **array, const int count){
    for (int i = 0; i < count; i++)
        std::cout << array[i] << std::endl;
}
```

# Использование шаблона функции

```
int main()
{
    const int aCount = 5, bCount = 7, cCount = 2;
    int a[aCount] = {1,2,3,4,5};
    float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
    char* c[cCount] = {"HELLO, WORLD!", "WORLD WIDE WEB"};
    std::cout << "Array of integer: \n";
    printArray(a, aCount);
    std::cout << "Array of float: \n";
    printArray(b, bCount);
    std::cout << "Array of char*: \n";
    printArray(c, cCount); //явная специализация
    return 0;
}
```



# Шаблоны классов

это параметризованные типы. Они задают семейства классов

## Объявление шаблонов

```
template < class T >  
class Vector {  
...  
};
```

# Описание шаблона вектора

```
template < class T >
class Vector {
    T * v;
    int sz;
public:
    Vector ( int s ) :sz(s)
    {
        v = new T [sz];
    }
    T & operator [] ( int i );
    int size ()const { return sz; }
    // ...
};
```

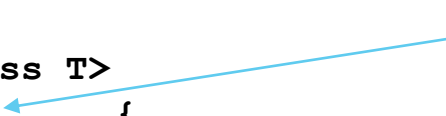
# Использование шаблона вектора

```
main ()
{
    // вектор из 100 целых
    Vector <int> v1 (100);
    //вектор из 200 чисел двойной точности
    Vector <double> v2 (200);
    // ...
    v2 = v1; //ошибка
}
```

# Шаблоны методов класса

```
class Example{  
public:  
    template<class T>  
    void f(T a) {  
        std::cout << a << std::endl;  
    }  
};
```

Не могут быть виртуальными



```
int main(){  
    Example ex;  
    ex.f(5);  
    ex.f(6.7);  
    ex.f(std::complex<double>(5,5));  
}
```

# Шаблоны методов класса

```
template<class X>
class Example{
public:
    template<class T>
    void f(T a)      {
        std::cout << a << std::endl;
    }
};
```

```
int main(){
    Example <int> ex;
    ex.f(5);
    ex.f(6.7);
    ex.f(std::complex<double>(5,5));
}
```

# Шаблоны методов класса

```
template<class X>
class Example
{
    X x;
public:
    X getValue()const { return x; }
    void f(const Example<X>& ex) { x = ex.x; }
};
```

```
int main()
{
    Example<int> ex, ex2;
    Example<double> ex1;
    ex.f(ex2); //хорошо
    ex.f(ex1); //ошибка
}
```

# Шаблоны методов класса

```
template<class X>
class Example{
    X x;
public:
    X getValue()const { return x; }
    void f(const Example<X>& ex) { x = ex.x; } //1
    template<class T>
    void f(const Example<T>& ex) { //2
        //x = ex.x;
        x = ex.getValue();
    }
};
```

```
int main(){
    Example<int> ex, ex2;
    Example<double> ex1;
    ex.f(ex2); //1
    ex.f(ex1); //2
}
```

# Шаблоны вложенных классов

```
class A {};
```

```
class X{  
    template <class T>  
    class Nested{  
        T a;  
    };  
    Nested<int> i;  
    Nested <char> c;  
    Nested <A> a;  
};
```

```
int main(){  
    X x;  
}
```



# Шаблоны вложенных классов

```
template <class Y>
class X{
    template <class T>
    class Nested{
        T a;
    public:
        T& get();
    };
public:
    Nested<int> i;
    X(Y y){
        i.get() = y;
    }
};
```

```
template <class Y>
template <class T>
T& X<Y>::Nested<T>::get()
{
    return a;
}
```

```
int main()
{
    X<char> x('a');
    std::cout << x.i.get() << std::endl; //97
}
```

# Инстанцирование

- ▶ Процесс генерации объявления класса по шаблону класса и аргументу шаблона
- ▶ Версия шаблона для конкретного аргумента шаблона называется специализацией
- ▶ Генерация версий шаблона - задача компилятора
- ▶ Инстанцирование может быть явное
- ▶ Явное инстанцирование используется, если:
  - ▶ инстанцирование шаблонов отнимает слишком много времени
  - ▶ порядок компиляции должен быть абсолютно предсказуем

# Параметр не-типа

```
template<typename T, int size>
class Vector
{
public:
    Vector();
    ~Vector();
    T& operator[] (int);
private:
    T* arr;
};
```

Параметром не-типа может быть:

- целое число
- указатель
- ссылка

## Текст шаблона целиком определяется в заголовочном файле

```
template<typename T, int size>
Vector<T, size>::Vector()
{
    arr = new T[size];
    for(int i = 0; i < size; i++)
        arr[i] = 0;
}
```

```
template<typename T, int size>
Vector<T, size>::~~Vector()
{
    delete[] arr;
}
```

```
template<typename T, int size>
T& Vector<T, size>::operator [](int index)
{
    if(index < 0 || index >= size)
        throw int(index);
    return arr[index];
}
```

# Использование параметра не-типа

```
int main()
{
    const int size1 = 10, size2 = 15;
    Vector<int, size1> obint1;
    Vector<int, size2> obint2;
    ...
}
```

Может быть инициализирован  
только константой

# Аргументы по умолчанию

```
template<typename T=int, int size=10>  
class Vector  
{  
    ...  
};
```

Теперь объект типа Vector можно создать тремя способами:

1. Вообще без задания типа и размера массива, тогда тип элементов будет равен int, а размер массива будет 10 элементов.
2. Указав только тип элементов, размер будет равен 10.
3. Указав и тип элементов, и размер массива.

# Ключевое слово `typename`

```
template <class X>
class Vector{
public:
    class Iterator{
    public:
        Iterator& operator++();
    };
};
```

```
template<class X>
typename Vector<X>::Iterator& Vector<X>::Iterator::operator++() {}
```

```
int main()
{
    Vector<int> v;
    Vector<int>::Iterator it;
}
```

# Ключевое слово `typename`

```
template <typename X>
class Vector{
public:
    class Iterator{
    public:
        Iterator& operator++();
    };
};
```

```
template <typename X>
typename Vector<X>::Iterator& Vector<X>::Iterator::operator++() {}
```

```
int main()
{
    Vector<int> v;
    Vector<int>::Iterator it;
}
```



# Шаблоны и наследование

- ▶ Шаблон класса может производным от шаблона класса
- ▶ Шаблон класса может являться производным от обычного класса
- ▶ Шаблон класса может быть производным от специализации шаблона класса
- ▶ Обычный класс может быть производным от специализации шаблона класса

# Шаблоны и наследование

```
template<class T>  
class A  
{};
```

```
template<class T>class B :public A<T>  
{};
```

```
int main()  
{  
    B<int> b;  
}
```

# Шаблоны и наследование

```
template<class T>  
class A  
{};
```

```
template<class T>class B :public A  
{};
```

```
int main()  
{  
    B<int> b;  
}
```

# Шаблоны и наследование

```
template<class T>  
class A  
{};
```

```
template<class T>class B :public A<int>  
{};
```

```
int main()  
{  
    B<double> b;  
}
```

# Шаблоны и наследование

```
template<class T>  
class A  
{};
```

```
template class B : public A<int>  
{};
```

```
int main()  
{  
    B b;  
}
```

# Шаблоны и дружелюбность

```
template<class T>class A;
```

```
template<class T>
class B {
    friend class A<T>;
    T t;
public:
    B(T a) :t(a) {}
};
```

```
template<class T>
class A {
public:
    void test(B<T>& b) {
        std::cout << b.t << std::endl;
    }
};
```

```
int main() {
    B<int> b(7);
    A<int> a;
    a.test(b);
    A<double> a1;
    a1.test(b); //ошибка
}
```

# Шаблоны и дружелюбность

```
template<class T>class A;
```

```
template<class T>
class B {
    template <class X> friend class A;
    T t;
public:
    B(T a) :t(a) {}
};
```

```
template<class T>
class A {
public:
    template <class X> void test(B<X>& b) {
        std::cout << b.t << std::endl;
    }
};
```

```
int main() {
    B<int> b(7);
    A<double> a;
    a.test(b);
}
```

Конец