



# Шаблоны. Стандартная библиотека шаблонов

## Лекция 4

# Понятие шаблона

*Шаблон (template)* – средство программирования, которое позволяет создавать функции и классы, в которых тип задается в качестве параметра.

В *шаблоне функции* определяется алгоритм, который может применяться к данным разных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу.

*Шаблоны классов* обычно используются, когда класс предназначен для хранения специальным образом организованных данных и работы с ними.

# Шаблоны функций

В простейшем случае функция-шаблон определяется так:

```
template <class Type> тип_ф имя_ф(параметры) {  
    /* тело функции */ }
```

Type – это имя фиктивного типа. Компилятор автоматически заменит его именем реального типа данных при создании конкретной версии функции. Вместо слова **class** может использоваться **typename**.

В угловых скобках указываются параметры шаблона через запятую. Их может быть несколько. Параметр может быть не только типом, но и просто переменной:  

```
template <class T1, class T2, int i> void F() {...}
```

# Пример функции-шаблона

```
#include <iostream>
using namespace std;
template <class T> void Myswap(T &x, T &y) {
    T a=x; x=y; y=a; }
int main() {
    int i=10, j=20;
    char s1='a', s2='b';
    Myswap(i, j);
    cout<<"i="<<i<<" j="<<j<<endl;
    Myswap(s1,s2);
    cout<<"s1="<<s1<<" s2="<<s2<<endl;
    return 0;
}
```

# Шаблоны и перегруженные функции

Функции-шаблоны похожи на перегружаемые функции, но являются более ограниченными, т.к. всегда выполняют один и тот же алгоритм для различных типов данных.

При необходимости функцию-шаблон можно перегрузить явным образом. При явной перегрузке компилятор не создает автоматическую версию функции для параметров указанных типов.

Вывод: если алгоритм обработки для разных типов одинаков, используют функции-шаблоны. Если нет — перегруженные функции.

# Пример использования шаблона и перегруженной функции

```
#include <iostream>
using namespace std;
#include <string.h>
template <class T> T Summa(T x, T y) {
    T S; S=x+y; return S; }
char * Summa(char *s1, char *s2) {
    strcat(s1, s2); return s1;}
int main() {
    float i=3.2, j=2.7;
    char s1[80]="Hello "; char s2[]="World!";
    cout<<Summa(i, j)<<endl;
    cout<<Summa(s1, s2)<<endl;
    return 0; }
```

# Шаблоны классов

Синтаксис шаблона класса:

```
template <описание_параметров_шаблона> class имя_класса  
{ /* описание класса */};
```

Параметры перечисляются через запятую. Параметрами могут быть типы (class Ttype), шаблоны и переменные.

Если метод описывается вне шаблона, его заголовок должен иметь вид:

```
template <описание_параметров_шаблона>  
возвр_тип имя_класса <параметры_шаблона>::  
    имя_функции (список_параметров_функции)
```

Создание экземпляра класса:

```
имя_класса <аргументы> объект[(параметры_конструктора)];
```

Аргументы – значения с которым будет работать класс.

# Пример класса-шаблона

```
template <class T1, class T2> class MyClass {
    T1 i; T2 j;
public: MyClass(T1 a, T2 b) {i=a; j=b;}
    void Show();
};

template <class T1, class T2>
    void MyClass <T1, T2>::Show()
        {cout<<i<<" "<<j<<endl;}

int main() {
    MyClass <int, double> ob1(22, 3.5);
    MyClass <char*, char*> ob2("text1", "text2");
    ob1.Show();    ob2.Show();
    return 0;
}
```



# Библиотека стандартных шаблонов

Стандартная библиотека C++ (Standard Template Library, STL) содержит шаблоны для хранения и обработки данных.

Основные элементы библиотеки:

- контейнеры;
- итераторы;
- алгоритмы;
- функциональные объекты.

# Контейнеры

Контейнеры – это объекты, предназначенные для хранения других однотипных объектов.

Могут содержать простые объекты (целые, вещественные, символьные и т.д.), структурные данные (массивы, строки, структуры), объекты классов. Эти объекты должны допускать копирование и присваивание.

В контейнерах можно хранить сами объекты или указатели на них.

В каждом классе-контейнере определен набор функций для работы с этим контейнером.

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл.

# Классификация контейнеров

*Последовательные контейнеры* обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности.

К базовым последовательным контейнерам относятся *векторы* (vector), *списки* (list) и *двусторонние очереди* (deque).

Специализированные контейнеры (или *адаптеры* контейнеров) реализованы на основе базовых. Это *стеки* (stack), *очереди* (queue) и *очереди с приоритетами* (priority\_queue).

*Ассоциативные контейнеры* обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Есть пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

# Контейнеры, определенные в STL

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двунаправленный список	<deque>
list	Линейный список	<list>
map	Список для хранения пар ключ/значение , где с ключом связано только одно значение	<map>
multimap	Список для хранения пар ключ/значение , где с ключом связано 2 или более значений	<map>
multiset	Множество не уникальных элементов	<set>
priority_queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>
set	Множество уникальных элементов	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

# Итераторы

Итераторы – это объекты, которые по отношению к контейнерам играют роль указателей.

Для всех контейнерных классов STL определен тип `iterator`, но его реализация в разных классах разная. Поэтому при объявлении объектов типа `iterator` всегда указывается область видимости:

```
vector <int>:: iterator i;  
list <double>:: iterator j;
```

# Основные операции с итераторами

- Разыменование итератора: если  $p$  — итератор, то  $*p$  — значение объекта, на который он ссылается.
- Присваивание одного итератора другому.
- Сравнение итераторов на равенство и неравенство ( $==$  и  $!=$ ).
- Перемещение его по всем элементам контейнера с помощью префиксного ( $++p$ ) или постфиксного ( $p++$ ) инкремента.

# Просмотр элементов контейнера

Если  $i$  — некоторый итератор, то используется следующая форма цикла:

```
for (i = first ; i != last; ++i)
```

$first$  — значение итератора, указывающее на первый элемент в контейнере.

$last$  — значение итератора, указывающее на воображаемый элемент, который следует *за* последним элементом контейнера.

Операция сравнения  $<$  здесь заменена на операцию  $!=$ , поскольку операции  $<$  и  $>$  для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы `begin()` и `end()`, возвращающие адреса  $first$  и  $last$  соответственно.

# Способы создания объекта-последовательного контейнера

1. Создать пустой контейнер:

```
vector<int> vec1;
```

```
list<string> list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1(100);
```

```
list<double> list1(20);
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1(100, "Hello!");
```

```
deque<int> decl(300, -1);
```



# Способы создания объекта-последовательного контейнера

4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last) элементов другого контейнера:

```
int arr[7] = {15, 2, 19, -3, 28, 6, 8};  
vector<int> v1(arr, arr + 7);  
list<int> lst(v1.begin() + 2, v1.end());
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;  
// добавить в v1 элементы  
vector<int> v2(v1);
```

# Алгоритмы

Алгоритм — это функция, которая выполняет некоторые действия над содержимым контейнера.

Чтобы использовать обобщенные алгоритмы нужно к программе подключить заголовочный файл `<algorithm>`.

В списках параметров всех алгоритмов первые два параметра задают *диапазон обрабатываемых элементов* в виде полуинтервала [ first , last), где

first — итератор, указывающий на начало диапазона,

last — итератор, указывающий на выход за границы диапазона.

# Пример использования векторов

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    double arr[] = {5.1, 2.2, 1.3, 4.4 };
    int n = sizeof(arr)/sizeof(double);
    vector<double> v1(arr, arr + n); // Инициализация вектора массивом
    vector<double> v2; // пустой вектор
    vector<double>::iterator i;
    for (i = v1.begin(); i != v1.end(); ++i)
        cout << *i <<" ";
    sort(v1.begin(), v1.end());
    for (i = v1.begin(); i != v1.end(); ++i)
        cout << *i <<" ";
    return 0; }
```