



# Тема 15

## Шаблоны

# Зачем нужны шаблоны?

Сильно типизированный язык , такой, как C++, создает препятствия для реализации совсем простых функций. Реализация таких функций должна быть многократно повторена для различных типов, хотя алгоритм остаётся одним и тем же.

```
int Min(int a, int b ) {  
    return a < b ? a : b;  
}
```

Эта функция неприменима к типу **double**:

```
cout << Min(4.5, 6);
```

результат:

4

# Решение проблемы: перегруженные функции

```
int Min(int a, int b ) {  
    return a < b ? a : b;  
}
```

```
double Min(double a, double b ) {  
    return a < b ? a : b;  
}
```

```
long long Min(long long a, long long b ) {  
    return a < b ? a : b;  
}
```

**И Т.Д.**

# Преимущества и недостатки этого подхода

## Преимущество:

- возможность реализовать особенности работы с определёнными типами

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

## Недостатки:

- дублирование кода;
- невозможно реализовать функцию (особенно стандартную) для абсолютно всех типов данных, которые потребуются в дальнейшем.

# Решение проблемы: ИСПОЛЬЗОВАНИЕ МАКРОСОВ

```
#define Min(a, b) ((a) < (b) ? (a) : (b))
```

## Недостатки:

- одинаковая реализация для всех типов;
- это – не вызов функции, а текстовая подстановка, так что выражения будут рассчитываться дважды.

```
int p=6;  
std::cout << Min(p++, 8);
```

Результат вывода – 7, поскольку вторая строка преобразуется в строку

```
std::cout << ((p++) < (8) ? (p++) : (8));
```

# Определение шаблона

**Шаблон** – это реализация функции или класса для типов, которые передаются в шаблон в качестве параметров

Синтаксис задания шаблона функции:

```
template <параметры_шаблона>  
реализация_функции ; // объявление или определение
```

Категории параметров шаблона:

- параметры-типы

```
class имя_параметра  
typename имя_параметра
```

- параметры-переменные

```
тип имя_параметра
```

# Задание шаблона для функции min()

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

**ИЛИ**

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

**Использование шаблона:**

```
cout << Min(5, 8); // результат - 5
cout << Min(5.4, 3.8); // результат - 3.8
cout << Min("Serge", "Kashkevich");
        // неверный результат - Serge
```

# Реализация шаблонов

1. При компиляции текста шаблона проводится лишь синтаксический анализ заголовка, программный код не генерируется;
2. генерация программного кода выполняется при его вызове. При этом определяются значения параметров-типов и для каждого нового параметра-типа строится свой экземпляр шаблона. Такой механизм называется **инстанциацией** (**конкретизацией**) шаблона.



# Ошибки компиляции при использовании шаблонов

- I. Ошибки компиляции возможны при использовании недопустимых синтаксических конструкций в заголовке шаблона или функции:

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

Эта ошибка обнаружится всегда, даже если нет ни одной конкретизации шаблона.

# Ошибки компиляции при использовании шаблонов

2. Ошибки компиляции возможны при использовании недопустимых синтаксических конструкций в тексте функции:

```
template <typename Type>
    Type Min( Type a, Type b ) {
        return a < b ? a? b;
    }
```

Эта ошибка обнаружится при первой же конкретизации шаблона.

# Ошибки компиляции при использовании шаблонов

3. Ошибки компиляции возможны при использовании недопустимых синтаксических конструкций в процессе конкретизации шаблона:

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

Эта ошибка обнаружится при конкретизации шаблона для типа, у которого не определена операция «меньше».

```
Person p1("John"), p2("Richard");
...
Person p3(Min(p1, p2)); // Ошибка!
```

# Явное и неявное указание параметров при конкретизации шаблонов

1. `cout << Min(2, 4) << endl;`

**Создаётся неявная конкретизация для типа `int`**

2. `cout << Min(2.8, 4.7) << endl;`

**Создаётся неявная конкретизация для типа `double`**

3. `cout << Min(2LL, 4LL) << endl;`

**Создаётся неявная конкретизация для типа `long long`**

# Явное и неявное указание параметров при конкретизации шаблонов

4. `cout << Min(2, 4.7) << endl;`

## Ошибка компиляции:

```
error C2782: 'Type Min(Type,Type)' : template parameter 'Type' is ambiguous
```

## Исправление ошибки:

### *Первый способ:*

```
cout << Min((double)2, 4.7) << endl;
```

### *Второй способ:*

```
cout << Min <double>(2, 4.7) << endl;
```

# Специализация шаблонов (определение)

В некоторых случаях имеется специальная информация о типе, позволяющая написать более эффективную функцию, чем конкретизированная по шаблону. А иногда общее определение, предоставляемое шаблоном, для некоторого типа просто не работает.

***Явное определение специализации*** – это такое определение, в котором за ключевым словом `template` следует пара угловых скобок `<>`, а за ними – определение специализированного шаблона:

```
template <> тип_возврата  
функция <параметры шаблона>  
(параметры_функции) {реализация}
```

# Специализация шаблонов (часть I)

```
cout << Min("Serge", "Kashkevich");  
// неверный результат - Serge  
// Причина: выполняется сравнение указателей,  
// а не сравнение строк
```

*Первый вариант решения: перегрузка функции Min*

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

## Специализация шаблонов (часть 2)

```
cout << Min("Serge", "Kashkevich");  
// неверный результат - Serge  
// Причина: выполняется сравнение указателей,  
// а не сравнение строк
```

**Второй вариант решения: специализация Min для типа `const char *`**

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



# Специализация шаблонов (ограничения)

Специализация шаблона должна быть определена до конкретизации!

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

```
void my_func() {
...
cout << Min("Serge", "Kashkevich");
...
}
```

```
// компилятор ещё не видит, что была выполнена
// специализация!
```

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

# Пример шаблона с параметрами-типами и параметрами-переменными

Функция находит минимальный элемент в массиве, размер которого задаётся параметром шаблона (**так делать категорически не рекомендуется!**):

```
template <class Type, int size>
    Type Min( Type *arr) {
        Type m = arr[0];
        for (unsigned i=1; i<size; i++)
            if (m>=arr[i])
                m = arr[i];
        return m;
    }
```

**Конкретизация:**

```
int M[50];
//заполнение массива M
cout << Min <int, 10> (M);
```

# Для чего нужны шаблоны классов?

Наиболее очевидное использование - адаптивные объекты памяти или контейнеры.

Вернёмся к примеру из предыдущих лекций:

```
typedef int InfoType;  
  
class LQueue {  
...  
}
```

## Преимущества:

- легко переходить к другому типу хранимых данных.

## Недостатки:

- невозможно в одном приложении организовать очереди для различных типов;
- не для всех типов данных такая реализация работает

# Описание шаблона класса

```
template <параметры_шаблона>  
class имя_класса { ... };
```

...

```
template <параметры_шаблона>  
реализация_методов;
```

## Реализация метода:

```
template <параметры_шаблона>  
тип_возврата имя_класса<значения_параметров_шаблона>  
::имя_метода { ... }
```

В параметрах шаблона, в отличие от шаблонов функций, могут быть заданы значения по умолчанию!

# Описание очереди через шаблон

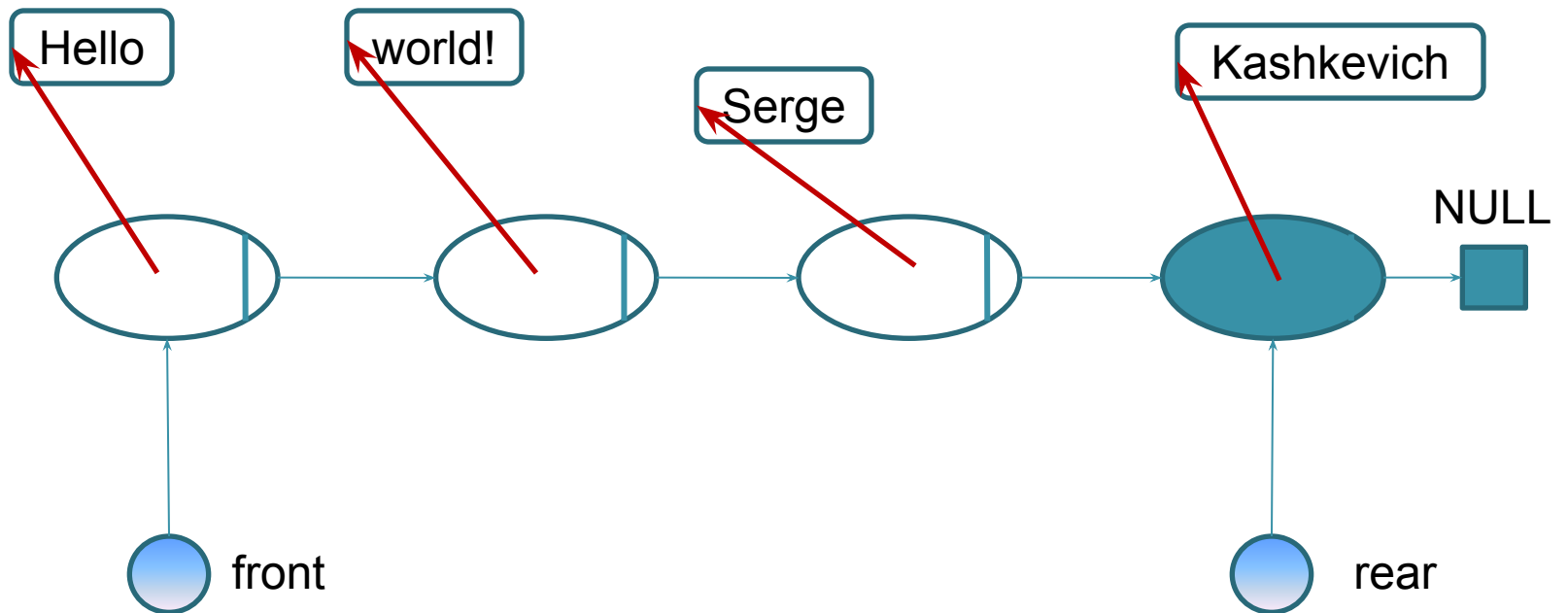
```
#ifndef __LQueue_defined__  
#define __LQueue_defined__  
  
#include <iostream>  
using namespace std;  
  
template <typename InfoType = int>  
class LQueue {  
    ...  
};  
  
#endif
```

# Реализация отдельных методов очереди

```
template <typename InfoType>
void LQueue <InfoType> ::Erase() {
    while (Pop());
    size = 0;
}
```

```
template <typename InfoType>
void LQueue <InfoType> ::Clone(const LQueue& Q) {
    QItem *tmp = Q.front;
    for (unsigned i=0; i<Q.size; i++) {
        Push(tmp->info);
        tmp = tmp->next;
    }
}
```

# Особенности работы с типом `const char *`



# Общий шаблон для метода Pop()

```
template <typename InfoType>
bool LQueue <InfoType> ::Pop() {
    if (size==0)
        return false;
    QItem *tmp = front;
    front = front->next;
    delete tmp;
    size--;
    if (size==0)
        rear = NULL;
    return true;
}
```



# Специализация метода Pop() для класса const char \*

```
template <>
bool LQueue <const char *> ::Pop() {
    if (size==0)
        return false;
    QItem *tmp = front;
    front = front->next;
    delete [] tmp->info;
    delete tmp;
    size--;
    if (size==0)
        rear = NULL;
    return true;
}
```

# Конструктор QItem и его специализация для класса const char \*

```
template <typename InfoType>
class LQueue {
...
    struct QItem {
...
        QItem(InfoType Ainfo): info(Ainfo), next(NULL) {}
    };
...
};

template <>
LQueue <const char *> ::QItem::QItem(
    const char * AInfo): next(NULL) {
    info = new char[strlen(AInfo)+1];
    strcpy((char *)info, AInfo);
}
```

# Особенности компоновки проекта при создании шаблона классов

Шаблоны компилируются только в момент их конкретизации, и разнесение реализации класса и его использования в различные файлы без связи друг с другом становится невозможным.

Возможные способы решения этой проблемы:

- дать пользователю класса описание шаблонов в исходных текстах;
- реализовать собственные классы на основе шаблонов для наиболее употребимых типов.

# Особенности доступа к элементам очереди

Оператор `[]` возвращает ссылку на хранящийся в очереди элемент, позволяя его изменять. Для типа `char*` это может привести к изменению указателя.

Следствия:

- утечка памяти;
- доступ к памяти, выделенной вне класса.

Следовательно, этот оператор нельзя применять для указанного типа.

# Решение возникшей проблемы

- запрет выполнения оператора [] для класса `const char *`;
- Предоставление вместо этого метода `SetByIndex` для выполнения замены строк

Кроме того, имеет смысл написать защищённый метод, возвращающий указатель на элемент очереди с указанным индексом.

# Метод PtrByIndex()

```
template <typename InfoType>
void* LQueue <InfoType> ::PtrByIndex (unsigned k)
    const {
    if ((k<0) || (k>=size))
        throw exception("Impossible to obtain
                           queue item: invalid index");
    QItem *tmp = front;
    for (unsigned i=0; i<k; i++)
        tmp = tmp->next;
    return tmp;
}
```

# Вызовы метода PtrByIndex() из ОТКРЫТЫХ МЕТОДОВ

```
template <typename InfoType>
const InfoType& LQueue <InfoType> ::GetByIndex
(unsigned k) const {
    return ((QItem*)PtrByIndex(k))->info;
}

template <typename InfoType>
void LQueue <InfoType> ::SetByIndex(InfoType AInfo,
    unsigned k) {
    ((QItem*)PtrByIndex(k))->info = AInfo;
}
```

# Специализация метода SetByIndex()

```
template <>
void LQueue <const char *> ::
SetByIndex(const char * AInfo, unsigned k) {
    char * curr =
        (char *) ((QItem*) PtrByIndex(k)) ->info;
    delete [] curr;
    curr = new char[strlen(AInfo)+1];
    strcpy(curr, AInfo);
    ((QItem*) PtrByIndex(k)) ->info = curr;
}
```



# Оператор typeid()

Оператор typeid имеет две формы:

- **typeid(выражение)**
- **typeid(тип)**

Он создаёт объект класса `type_info` с информацией о типе операнда.

Пример использования typeid:

```
cout << typeid("Hello, world!").name() << endl;  
// Результат - "char const [14]"
```

# Использование typeid() вместо специализации

```
template <typename InfoType>
InfoType& LQueue <InfoType> ::operator [] (unsigned k)
{
    if (typeid(InfoType) == typeid(const char *))
        throw exception("Using of operator[] is
prohibited; use SetByIndex or GetByIndex instead");
    return (InfoType&) ((QItem*)PtrByIndex(k)) ->info;
}
```

# Использование typeid() вместо специализации (продолжение)

```
template <typename InfoType>
void LQueue <InfoType> ::Browse(void
    ItemWork(InfoType&)) {
    if (typeid(InfoType)==typeid(const char *))
        throw exception("Using of
            Browse(void(InfoType&)) is prohibited for
            null-terminated strings");
    QItem *tmp = front;
    for (unsigned i=0; i<size; i++) {
        ItemWork(tmp->info);
        tmp = tmp->next;
    }
}
```