

**Подготовительная  
программа по  
программированию на  
C/c++**

Занятие №7

Валентина Глазкова

# Модульное программирование.

## Шаблоны классов и методов

---

- Пространства имён.
- Ортодоксальная каноническая форма класса.
- Разбиение программы на файлы и модули.
- Обобщённое программирование. Шаблоны классов и методов.
- Параметры шаблонов.
- Специализация, конкретизация и перегрузка шаблонов.

# Пространства имён (1/2)

---

- **Пространство имён (namespaces)** - удобный способ автоматического создания **префиксов имён** сразу для многих сущностей (как правило логически взаимосвязанных)
- Эти сущности заключаются внутрь пространства имён (при этом можно размещать *тела* определяемых сущностей внутри пространства имён либо оставлять только заголовки, а тела выносить вовне)
- Пространства имён – это **абстракция** исключительно **периода компиляции** программы. В работающей программе они не существуют, им не соответствуют никакие объекты.

# Пространства имён (2/2)

---

- Пространства имён позволяют **избежать конфликтов имён** (например, при разработке библиотек с одноимёнными сущностями разными разработчиками)
- Пространства имён могут быть вложены друг в друга

```
namespace Model3D
{
    namespace Point{void draw(int x, int y);}
}
namespace GUI
{
    void draw(int x, int y);
}
int main( )
{
    GUI::draw(1,2);
    Model3D::Point::draw(10, 20);
    return 0;
}
```



# Пространства имён: пример

```
namespace N1
{
    class C1
    {
        int v;
    public:
        C1(int _v = 0) { v = _v; }
    };
}
```

```
namespace N2
{
    class C2;
}
class N2::C2
{
    int v;
public:
    C2(int _v = 0);
};
N2::C2::C2(int _v)
{
    v = _v;
}
```

# Пространства имён: отличие от классов

---

- Пространство имён не может иметь сущностей, недоступных извне его (т.е. отсутствуют аналоги квалификаторов доступа `private` и `protected`). **Любой идентификатор, объявленный в пространстве имён, может использоваться вне его** (естественно, если это допускается, например, правами доступа вложенного класса)
- Пространство имён открыто для добавления новых сущностей: в любом месте программы можно добавить новое имя в уже существующее пространство имён

```
namespace A
{
    void func(int);
}
// ...
namespace A
{
    void func(double);
}
```

# Пространства имён: директива `using`

---

- Директива **использования имени из некоторого пространства имён** – позволяет использовать соответствующий идентификатор без указания префикса пространства имён

```
using N1::C1;
```

```
C1 obj; // эквивалентно N1::C1 obj;
```

- Директива **подключения всего пространства имён** (применяется, как правило, если программа использует только одно пространство имён – например, одну библиотеку)

```
using namespace N1;
```

- Директива `using` **не вводит новых имён сущностей**, а позволяют сокращать имя за счёт использования общего префикса

# Анонимные пространства имён

- Безымянное пространство имен позволяет определить уникальность идентификаторов **с областью видимости в пределах единственного файла**
- Неименованные пространства имен — это превосходная замена статическому **объявлению глобальных переменных**
- Всем неименованным пространствам имён присваивается неявный уникальный идентификатор

```
namespace
{ int i; } // unique::i
void f() { i++; } // unique::i++
namespace A
{ namespace
{
    int i; // A::unique::i
    int j; // A::unique::j
}
}
using namespace A;
void h()
{
    i++; // error
    A::i++; // A::unique::i++
    j++; // A::unique::j++
}
```



# Псевдоним пространства имён

---

- Определение псевдонима пространства имен объявляет альтернативное имя (**синоним**) для пространства имен.
- Псевдоним пространства имен (как и само имя пространства) должен отличаться от всех других идентификаторов в данной области видимости

```
namespace a_very_long_namespace_name { ... }  
namespace AVLNN = a_very_long_namespace_name;
```

# Разбиение программы на файлы и модули

---

- **Модуль в C++** — логически обособленная часть программы, которая обычно состоит из двух частей – интерфейса и реализации
- **Реализация** может состоять из одного или более файлов (.c или .cpp); **интерфейс** может быть представлен в виде одного или более заголовочных файлов (.h или .hpp)
- Как правило, файлы, составляющие один логический модуль, собираются в один **объектный модуль** (это указывается при задании правил компиляции в объектные модули)

# Этапы сборки программы

---

- **Препроцессинг (preprocessing)** — директивы препроцессора (например, `#include`) заменяются содержимым указанного в них заголовочного файла, в результате файл с исходным кодом дополняется прототипами указанных там функций и объявлениями глобальных переменных
- **Компиляция (compiling)** — для каждого исходного файла составляются таблицы определенных в нем функций и глобальных переменных; все определенные функции переводятся на машинный язык; результатом работы компилятора являются *объектные файлы* для каждого программного модуля
- **Связывание (linking)** — происходит привязка всех используемых (вызванных) функций и глобальных переменных к той таблице, в которой они определены; если определения не обнаружены ни в одной таблице, происходит ошибка связывания (`unresolved external symbol ...`). Результатом связывания является исполняемый файл

# Разбиение программы на файлы и модули

---

- Эмпирические правила организации исходного кода на языке C++ — результат многолетнего опыта практического программирования специалистов по всему миру:
  - объявления классов, как правило, хранятся в заголовочных файлах с именами *<имя класса>.{h | hpp}*
  - **заголовочный файл**, как правило, включает в себя прототипы функций, определения констант, объявления глобальных переменных - но только для тех элементов модуля, о которых должны знать другие модули
  - код реализации в основном хранится в исходных файлах с именами *<имя класса>.{c | cpp}*
  - файл с исходным кодом (**файл реализации**), как правило, включает в себя определения функций, а также определения глобальных переменных и констант (если они есть); в первой строчке обычно подключается заголовочный файл того же модуля

# Разбиение программы на файлы и модули

---

- **Эмпирические правила организации исходного кода на языке C++ — результат многолетнего опыта практического программирования специалистов по всему миру:**
  - члены класса перечисляются в порядке назначенных им уровней доступа: `public`, `protected`, `private`
  - подставляемые функции обычно выделяются из интерфейса и со спецификатором `inline` размещаются в заголовочном файле после объявления класса
  - полностью заголовочный файл помещается в директивы условной компиляции во избежание повторного включения в сборку при вложенных директивах `#include`

# Ортодоксальная каноническая форма класса (1 / 3)

---

- Ортодоксальная каноническая форма (ОКФ) класса — одна из важнейших идиом C++, согласно которой класс должен содержать:
  - конструктор по умолчанию: `T::T()`
  - конструктор копирования: `T::T(const T&)`
  - операцию-функцию присваивания: `T& T::operator=(const T&)`
  - деструктор: `T::~~T()`
- ОКФ обеспечивает:
  - единый стиль оформления классов;
  - помогает справиться со сложностью классов в процессе развития программы.



# Ортодоксальная каноническая форма класса (2 / 3)

---

```
class String
{
public:
    String();//к о н с т р у к т о р   п о   у м о л ч а н и ю
    String(const String&)//к о н с т р у к т о р   к о п и р о в а н и я
    String& operator=(const String&);//о п е р а т о р   п р и с в а и в а н и я
    ~String(); //д е с т р у к т о р
    //...
private:
    char *rep;
    int len;
};
```

# Ортодоксальная каноническая форма класса (3/3)

---

- ОКФ класса следует использовать, когда:
  - необходимо обеспечить поддержку присваивания для объектов класса или передачу их по значению в параметрах функций;
  - объект содержит указатели на объекты, для которых применяется подсчет ссылок;
  - деструктор класса вызывает `operator delete` для атрибута класса.
- ОКФ класса желательно использовать для всех классов, не ограничивающихся агрегированием данных аналогично структурам C.
- Отклонения от ОКФ позволяют реализовать нестандартные аспекты поведения класса.



# Обобщённое программирование

---

**Шаблон класса (функции)** — элемент языка, позволяющий параметризовать типы и значения, используемые для автоматического создания (**конкретизации**) классов (функций) по обобщенному описанию (**шаблону, алгоритму**).

Использование шаблонов классов — шаг на пути к парадигме **обобщенного программирования**.

Различают **описания и определения шаблонов**. В отличие от «обычного» класса (функции) описания и определения шаблона содержат **списки параметров шаблона**, среди которых выделяются **параметры-типы** и **параметры-константы**.

Параметры-типы шаблона представляют некоторый тип данных, параметры-константы — некоторое константное (**вычисляемое при компиляции**) выражение.

# Шаблонные функции

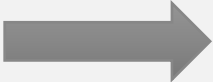
---

- Функции, объявленные с помощью ключевого слова `template`, называются *шаблонными* (template), *обобщенными* (generic) или *родовыми*
- Для каждого конкретного вызова функции компилятор автоматически генерирует функцию с соответствующими значениями параметров; этот процесс называется *конкретизацией* шаблона

# Шаблонные функции

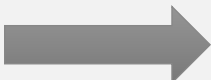
---

Использование шаблонов  
C++: одна шаблонная  
функция для всех типов



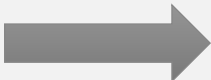
```
template<class T> T abs(T x) {  
    return x >= 0 ? x : -x;  
}
```

Использование перегрузки  
C++: одно и то же имя  
функции для каждого типа,  
дублирование тела каждой  
функции



```
float abs(float x) {  
    return x >= 0 ? x : -x;  
}  
  
int abs(int x) {  
    return x >= 0 ? x : -x;  
}
```

В стиле чистого C: свое имя  
функции для каждого типа,  
дублирование тела каждой  
функции



```
float fabs(float x) {  
    return x >= 0 ? x : -x;  
}  
  
int abs(int x) {  
    return x >= 0 ? x : -x;  
}
```

# Параметрический полиморфизм

---





# Шаблонные функции: пример

Ключевое слово,  
объявляющее шаблон

Ключевые слова `class` и `typename`  
объявляют параметрический тип

```
template<class T> T abs(T x)
{
    return x >= 0 ? x : -x;
}
```

Аргумент параметрического типа

Произвольный идентификатор,  
обозначающий параметрический тип,  
который будет автоматически  
подставляться компилятором на этапе  
конкретизации

```
void main()
{
    int    i = abs( -5);
    double d = abs( 2.0);
    float  f = abs(7.0f);
    long   l = abs(-16L);
}
```

Значение  
параметрического  
типа выводится из  
фактического типа  
аргумента



# Явный выбор конкретизации

```
template<class T> T get_one() {  
    return static_cast<T>(1);  
}  
  
void main()  
{  
    // Error:  
    // could not deduce template argument for 'T'  
    char c = get_one();  
  
    int i = get_one<int>(); // T == int  
    double d = get_one<double>(); // T == double  
    float f = get_one<float>(); // T == float  
    long l = get_one<long>(); // T == long  
}
```

При описании шаблонной функции  
параметрический тип используется  
точно так же, как и обычный,  
заданный в явном виде

Можно в явном виде  
указать явную  
конкретизацию при вызове  
шаблонной функции

# Параметры шаблона

---

- Параметром шаблона может быть:
  - Произвольный тип (в том числе другой шаблон)
  - Значение интегрального или перечислимого типа
  - Указатель или ссылка на объект любого типа
  - Указатель на член класса
- Автоматический выбор значения параметра называется *выводом* аргумента шаблона. При выведении аргументов не допускаются никакие преобразования, кроме точного отождествления.



# Вывод аргумента шаблона

---

```
template<class T> bool less(const T& a, const T& b) {  
    return a < b;  
}  
struct S {};  
  
void main()  
{  
    S s1, s2;  
    less(2, 3);           // T == int  
    less('a', 'b');      // T == char  
    less(1, 1.0);        // Error: T is ambiguous  
    less('1', 1);        // Error: T is ambiguous  
    less<int>('2', 2)     // T == int  
    less(s1, s2);        // Error: "<" not defined  
}
```





# Параметры-константы шаблона

В качестве параметров шаблонной функции допускаются значения интегральных типов и указатели

Это число известно на этапе компиляции

```
template<int N> int fact() {  
    int r = 1;  
    for (int i = 1; i <= N; i++) r *= i;  
    return r;  
}  
  
void main()  
{  
    cout << fact<5>() << endl; // 120  
}
```

Параметры значения всегда должны быть в явном виде указаны при вызове шаблонной функции



# Шаблонные функции: пример 1

Шаблонная функция может иметь произвольное количество параметров

```
template<int N, typename T> T ipow(T x) {  
    T r = 1;  
    for (int i = 1; i <= N; i++) r *= x;  
    return r;  
}
```

Можно опускать указанные в конце списка параметры, значения которых компилятор может определить самостоятельно

```
void main()  
{  
    cout << ipow<5>(2) << endl;           // 32  
    cout << ipow<5, double>(2) << endl; // 32.0  
}
```

Преобразование из int в double



## Шаблонные функции: пример 2

```
void foo(int x, int y = 0) {  
    cout << "1" << endl;  
}  
void foo(double x) {  
    cout << "2" << endl;  
}  
template<class T> void foo(T x) {  
    cout << "3" << endl;  
}  
void main() {  
    foo(1, 2);           // 1  
    foo(3);              // 1  
    foo(5.0);            // 2  
    foo('a');            // 3  
}
```

Перегруженные нешаблонные функции имеют при выборе больший приоритет, чем конкретизация шаблонов



# Перегрузка шаблонных функций

Конструкция, используемая для описания специализации

Описание функции, используемое по умолчанию при автоматической конкретизации

```
template<typename T> T min_val() { return 0; }
```

```
template<int> min_val() { return INT_MIN; }
```

```
template<float> min_val() { return FLT_MIN; }
```

```
void main()  
{
```

```
    cout << min_val<int>();      // -2147483648  
    cout << min_val<float>();    // 1.175494351e-038  
    cout << min_val<unsigned>(); // 0  
    cout << min_val<double>();  // 0
```

```
}
```

Значение параметрического типа определяется из фактически используемых вместо него при описании функции типов



# Перегрузка шаблонных функций

Для шаблонной функции можно объявить прототип точно так же, как и для обычной

Константа описана в файле float.h

```
template<typename T> T max_val();  
template<> unsigned max_val() { return 0xFFFFFFFF; }  
template<> double    max_val() { return FLT_MAX; }  
void main()  
{  
    cout << max_val<unsigned>(); // 4294967295  
    cout << max_val<double>();  // 3.402823466e+038  
    cout << max_val<int>();      // Error  
}
```

Ошибка линковки: компилятор не может автоматически конкретизировать шаблонную функцию без описания



# Перегрузка шаблонных функций

```
template<int N> int fact()
{
    return N * fact<N - 1>();
}

template<> int fact<1>()
{
    return 1;
}

void main()
{
    cout << fact<6>() << endl; // 720
}
```

Рекурсия времени компиляции  
(compile-time recursion)

При специализации можно в явном  
виде указать значения параметров  
шаблона

# Шаблоны и разбиение программы на файлы

---

- При разборе описания шаблонной функции компилятор не генерирует никакого кода
- Реальный код генерируется только в момент конкретизации функции при ее использовании в теле программы
- Полное описание функции должно быть доступно из места конкретизации (если только компилятор не поддерживает *экспорт шаблонов*)

# Шаблоны и разбиение программы на файлы

---

```
//----- abs.h -----//  
template<class T> T abs(T x);
```

Прототип шаблонной функции в .h-файле

```
//----- abs.cpp -----//  
#include "abs.h"  
template<class T> T abs(T x) {  
    return x >= 0 ? x : -x;  
}
```

Описание шаблонной функции само по себе не генерирует объектного кода

```
//----- main.cpp -----//  
#include "abs.h"  
void main() {  
    cout << abs(-2) << endl;  
}
```

Ошибка линковки: при использовании шаблонной функции компилятору доступен только ее прототип



# Шаблоны и разбиение программы на файлы

```
//----- abs.h -----//  
template<class T> T abs(T x) {  
    return x >= 0 ? x : -x;  
}
```

Описание шаблонной  
функции целиком  
вынесено в .h-файл

```
//----- abs.cpp -----//  
// Not present or empty
```

.cpp-файл отсутствует  
или пуст

```
//----- main.cpp -----//  
#include "abs.h"  
void main() {  
    cout << abs(-2) << endl;  
}
```

Нет ошибки: при  
использовании  
шаблонной функции  
компилятору доступно  
ее описание

# Шаблоны классов

---

**Параметры** шаблона класса **не могут быть** **одноименными члену** соответствующего шаблона.

Имя параметра шаблона может присутствовать в списке параметров лишь **один раз**.

Имена параметров в объявлении (**если есть**) и определении шаблона могут различаться.

Параметры могут иметь **значения по умолчанию**.



## Шаблоны классов: пример

---

```
template <class T, class U, int size>
// эквивалентно:
// template <typename T, typename U, int size>
class Test
{
public:
    Test() : _size(size) // ...
private:
    T        _prm_1;
    U*       _prm_2;
    int      _size;
};
```

# Конкретизация шаблонов классов

---

**Конкретизация шаблона** — это автоматическая генерация исходного кода конкретного класса в соответствии с заданным программистом определением шаблона класса.

Конкретизированный шаблон может использоваться везде, где допустимо использование «обычного» класса (**включая описания и определения шаблонов функций**), а экземпляры автоматически сгенерированного класса — везде, где допустимы «обычные» объекты.

Шаблон класса конкретизируется тогда, когда **впервые требуется определение** автоматически генерируемого класса.

```
Test<int, double, 10> t_id10;
```

# Методы шаблонов классов

---

**Методы шаблонов классов** также являются шаблонами и конкретизируются в точке вызова или взятия адреса. При этом конкретизируемый метод относится к тому классу, через объект которого вызывается.

```
template <class T, class U, int size>
Test<T, U, size>::~~Test()
{
    /* ... */
}
```



# Значения параметров шаблона по умолчанию

Параметры классов (как типы, так и значения) могут иметь значения по умолчанию

```
template<class T, int C = 1000> class Stack {  
    T m_Buffer[C];  
    int m_Size;  
public:  
    TStack() : m_Size(0) {}  
};  
  
void main()  
{  
    TStack<int> istack;  
    TStack<double, 10000> dstack;  
}
```

Это значение известно на этапе компиляции, поэтому можно использовать статический массив

При объявлении объектов шаблонного класса нужно всегда в явной форме указывать значения параметров (если нет значения по умолчанию)



## Шаблонные классы: пример (1/2)

```
template<class T, int C = 1000> class Stack {  
    T m_Buffer[C];  
    int m_Size;  
  
public:  
    class StackException {};  
  
    TStack() : m_Size(0) {}  
    void Push(T in_Value);  
    T Pop();  
    bool IsEmpty();  
    void Clear();  
};
```

Внутри описания шаблонного класса допускается произвольное использование параметров шаблона

Каждая конкретизация порождает свой независимый класс, полностью определяемый значениями фактических параметров



## Шаблонные классы: пример (2/2)

```
template<class T, int C> void TStack<T, C>::Push(T in_Value) {  
    if (m_Size == C) throw StackException();  
    m_Buffer[m_Size] = in_Value;  
    m_Size++;  
}
```

```
template<class T, int C> T TStack<T, C>::Pop() {  
    if (m_Size == 0) throw StackException();  
    m_Size--;  
    return m_Buffer[m_Size];  
}
```

```
void main() {  
    TStack<int, 100> istack;  
    try {  
        istack.Pop();  
    }  
    catch (TStack<int, 100>::StackException) {  
        cout << "Stack exception handled" << endl;  
    }  
}
```

При описании методов шаблонного класса необходимо полностью указывать прототип класса

При обращении к конкретизации шаблонного класса необходимо в явном виде указывать значения параметров



# Дружественные объекты в шаблонах классов

---

**Дружественными по отношению к шаблонам классов могут быть:**

- дружественная функция или дружественный класс (**не шаблон**);
- связанный шаблон дружественной функции, взаимно однозначно соответствующий шаблону класса (**с общим для обоих шаблонов списком параметров**);
- связанный шаблон дружественного класса, взаимно однозначно соответствующий шаблону класса (**с общим для обоих шаблонов списком параметров**);
- несвязанный шаблон дружественной функции, соответствующий всем возможным конкретизациям шаблона класса (**с отдельными списками параметров**);
- несвязанный шаблон дружественного класса, соответствующий всем возможным конкретизациям шаблона класса (**с отдельными списками параметров**).

# Статические члены шаблонов классов

## (1/2)

---

Шаблоны классов могут содержать **статические члены данных**, собственный набор которых имеет каждый конкретизированный согласно шаблону класс.

```
template <class T, class U, int size>
class Test
{
    /* ... */
private:
    static Test *_head;
};
```

```
template <class T, class U, int size>
Test<T, U, size> *Test<T, U, size>::_head = NULL;
```



## Статические члены шаблонов классов (2/2)

```
template<class T> class Foo {  
public:  
    static int s_Instances;  
    Foo() { s_Instances++; }  
};
```

Так как для каждой конкретизации компилятор фактически генерирует свой класс, у каждого из них будет свой экземпляр статического поля

```
template<class T> int Foo<T>::s_Instances;
```

```
void main()  
{  
    Foo<int> i1, i2, i3;  
    Foo<double> d1, d2;  
    cout << Foo<int>::s_Instances;    // 3  
    cout << Foo<double>::s_Instances; // 2  
    cout << Foo<char>::s_Instances;   // 0  
}
```

Такая форма записи позволяет выделить память под все сгенерированные конкретизации

# Специализация шаблонов классов.

## Специализация члена класса: пример

---

Шаблоны классов в языке C++ допускают **частичную (полную) специализацию**, при которой отдельные (все) параметры шаблона заменяются конкретными именами типов или значениями константных выражений.

```
// с п е ц и а л и з а ц и я ч л е н а к л а с с а
template<>
void Test<int, double, 10>::foo()
{
    /* ... */
}
```



# Полная и частичная специализация класса: пример

---

```
// полная специализация класса
template<> class Test<int, double, 100>
{
public:
    Test<int, double, 100>();
    ~Test<int, double, 100>();
    void foo(); /* ... */
};

// частичная специализация класса
template <class T, class U> class Test<T, U, 100>
{
public:
    Test();
    ~Test (); /* ... */
};
```

# Характеристические классы

- Шаблонный класс, который
  - не содержит данных
  - не нуждается в инстанцииции
  - имеет множество частичных специализаций

Характеристический класс, описывающий диапазоны встроенных типов данных, реализован в стандартном классе `numeric_limits` в файле `<limits>`

```
template<class T> struct type_traits {  
    static T GetMin();  
    static T GetMax();  
    static bool IsIntegral();  
    static const char* GetName();  
};
```



# Частичная специализация характеристических классов

```
template<> struct type_traits<int>
{
    static int GetMin() { return INT_MIN; }
    static int GetMax() { return INT_MAX; }
    static bool IsIntegral() { return true; }
    static const char* GetName() { return "int"; }
};

template<> struct type_traits<double>
{
    static double GetMin() { return DBL_MIN; }
    static double GetMax() { return DBL_MAX; }
    static bool IsIntegral() { return false; }
    static const char* GetName() { return "double"; }
};
```

Константы из  
файла limits.h

Константы из  
файла float.h



# Характеристические классы

```
template<class T> void print_type_info() {  
    cout << "Info on type "  
        << type_traits<T>::GetName() << ":" << endl;  
    cout << "Size = " << sizeof(T) * 8 << " bits (";  
    if (type_traits<T>::IsIntegral()) {  
        cout << "integral)" << endl;  
        cout << "Min value = ";  
    } else {  
        cout << "floating point)" << endl;  
        cout << "Min abs value = ";  
    }  
    cout << type_traits<T>::GetMin() << endl;  
    cout << "Max value = "  
        << type_traits<T>::GetMax() << endl;  
}
```





# Характеристические классы

```
void main()
{
    print_type_info<int>();    // Output to the left
    print_type_info<double>(); // Output to the right
    // Linker error: Unresolved external symbols
    print_type_info<char>();
}
```

Info on type int:

Size = 32 bits (integral)

Min value = -2147483648

Max value = 2147483647

Info on type double:

Size = 64 bits (floating point)

Min abs value = 2.22507e-308

Max value = 1.79769e+308



# Характеристические классы

---

```
template<class T> T sqr(T x) {
    if (type_traits<T>::IsIntegral())
        if (type_traits<T>::GetMax() / x < x)
            throw "Type overflow!";
    return x * x;
}

void main() {
    try {
        cout << sqr(-15) << endl;    // 225
        cout << sqr(1e15) << endl;    // 1e+030
        cout << sqr(50000) << endl;    // Type overflow!
    }
    catch (const char* e) {
        cout << e << endl;
    }
}
```

# Метапрограммирование

---

- Парадигма широкого использования шаблонных классов в связке с характеристическими называется *метапрограммированием*
- Наиболее распространенные библиотеки, основанные на метапрограммировании:
  - STL (standard template library),
  - boost (Boost C++ Libraries)

**Валентина Глазкова**

**Спасибо за внимание!**