

Лекция 6.

Шаблоны. Специализация шаблонов. Наследование шаблонов

3 семестр

Лектор: ст.пр. Бельченко Ф.М.

Проблемный пример

```
#include <iostream>

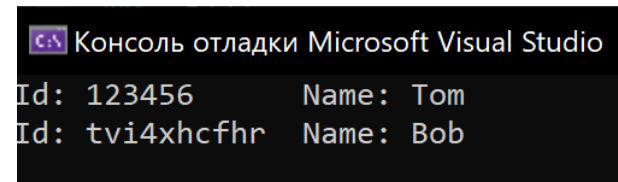
// класс Person, где id - целое число
class UIntPerson {
public:
    UIntPerson(unsigned id, std::string name) : id{id}, name{name}
    { }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    unsigned id;
    std::string name;
};

// класс Person, где id - строка
class StringPerson {
public:
    StringPerson(std::string id, std::string name) : id{id}, name{name}
    { }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    std::string id;
    std::string name;
};

int main()
{
    UIntPerson tom{123456, "Tom"};
    tom.print();    // Id: 123456    Name: Tom
    StringPerson bob{"tvi4xhcfhr", "Bob"};
    bob.print();    // Id: tvi4xhcfhr    Name: Bob
}
```

Здесь класс UIntPerson представляет класс пользователя, где id представляет целое число типа unsigned, а тип StringPerson - класс пользователя, где id - строка. В функции main мы можем создавать объекты этих типов и успешно их использовать. Хотя данный пример работает, но по сути мы получаем два идентичных класса, которые отличаются только типом переменной id.

А что, если для id потребуется использовать какой-то еще тип?



Консоль отладки Microsoft Visual Studio

```
Id: 123456    Name: Tom
Id: tvi4xhcfhr    Name: Bob
```

Шаблоны

Шаблоны — это фрагменты обобщённого кода, в котором некоторые типы или константы вынесены в параметры. Шаблонами могут быть функции, структуры (классы) и даже переменные.

Компилятор превращает использование шаблона в конкретный код, подставляя в него нужные параметры на этапе компиляции.

Для применения шаблонов перед классом указывается ключевое слово **template**, после которого идут угловые скобки. В угловых скобках указываются параметры шаблона. Если несколько параметров шаблона, то они указываются через запятую.

```
template <список_параметров>
class имя класса
{
    // содержимое шаблона класса
};
```

Пример с шаблонами

```
#include <iostream>

template <typename T>
class Person {
public:
    Person(T id, std::string name) : id{id}, name{name}
    { }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    T id;
    std::string name;
};

int main()
{
    Person tom{123456, "Tom"};      // T - число
    tom.print();                   // Id: 123456      Name: Tom
    Person bob{"tvi4xhcfhr", "Bob"}; // T - строка
    bob.print();                   // Id: tvi4xhcfhr  Name: Bob
}
```

Общие сведения

Шаблоны не похожи на обычные классы в том смысле, что компилятор не создает код объекта для шаблона или любого из его членов. Нет ничего, пока шаблон не будет создан с конкретными типами. Когда компилятор обнаруживает создание экземпляра шаблона, такого как `MyClass<int> mc;` и класс с такой сигнатурой еще не существует, он создает такой класс. Он также пытается создать код для любых используемых функций-элементов.

Стандартная библиотека C++ построена на шаблонах. Раньше её даже называли Standard Template Library (STL, стандартная библиотека шаблонов).

Её контейнеры и итераторы являются шаблонными классами, а алгоритмы — шаблонными функциями.

Примеры: контейнер `std::vector` и функция `std::sort`

Несколько параметров в шаблоне

```
#include <iostream>

template <typename T, typename V>
class Transaction
{
public:
    Transaction(T fromAcc, T toAcc, V code, unsigned sum):
        fromAccount{fromAcc}, toAccount{toAcc}, code{code}, sum{sum}
    { }
    void print() const
    {
        std::cout << "From: " << fromAccount << "\tTo: " << toAccount
            << "\tSum: " << sum << "\tCode: " << code << std::endl;
    }
private:
    T fromAccount; // с какого счета
    T toAccount;   // на какой счет
    V code;        // код операции
    unsigned sum;  // сумма перевода
};

int main()
{
    // явная типизация
    Transaction<std::string, int> transaction1{"id1234", "id5678", 2804, 5000};
    transaction1.print(); // From: id1234    To: id5678    Sum: 5000    Code: 2804
    // неявная типизация
    Transaction transaction2{"id6789", "id9018", 3000, 6000};
    transaction2.print(); // From: id6789    To: id9018    Sum: 6000    Code: 3000
}
```

Класс Transaction использует два параметра типа T и V.

Параметр T определяет тип для счетов, которые участвуют в процессе перевода. Здесь в качестве номеров счетов можно использовать и числовые и строковые значения и значения других типов. А параметр V задает тип для кода операции - опять же это может быть любой тип.

Определение функций вне шаблона класса

```
#include <iostream>

template <typename T>
class Person {
public:
    Person(T, std::string);           // обычный конструктор
    Person(const Person&);           // конструктор копирования
    ~Person();                       // деструктор
    Person& operator=(const Person&); // оператор присваивания
    void print() const;             // функция класса

private:
    T id;
    std::string name;
};

// определение конструктора вне шаблона класса
template <typename T>
Person<T>::Person(T id, std::string name) : id{id}, name{name} { }

// определение конструктора копирования вне шаблона класса
template <typename T>
Person<T>::Person(const Person& person) : id{person.id}, name{person.name} { }

// определение деструктора копирования вне шаблона класса
template <typename T>
Person<T>::~~Person(){ std::cout << "Person deleted" << std::endl; }
```

```
// определение оператора присвоения вне шаблона класса
template <typename T>
Person<T>& Person<T>::operator=(const Person& person)
{
    if (&person != this)
    {
        name = person.name;
        id = person.id;
    }
    return *this;
}


// определение функции вне шаблона класса
template <typename T>
void Person<T>::print() const
{
    std::cout << "Id: " << id << "\tName: " << name << std::endl;
}

int main()
{
    Person tom{123456, "Tom"};
    tom.print();

    Person tomas{tom}; // конструктор копирования
    tomas.print();

    Person tommy = tom; // оператор присваивания
    tommy.print();
}
```

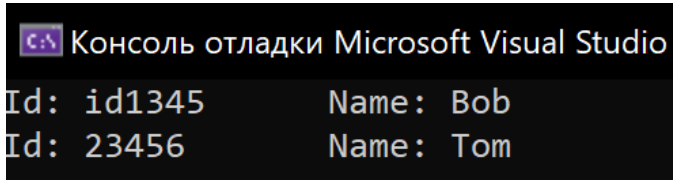
В данном случае все функции, в том числе конструкторы, деструктор, функция оператора присваивания, определяются как функции шаблона класса `Person<T>`. Причем в данном случае конструктор копирования или функция `print` никак не используют параметр `T`, но все равно они определяются как шаблоны. То же самое касается и деструктора.

 Консоль отладки Microsoft Visual Studio

```
Id: 123456      Name: Tom
Id: 123456      Name: Tom
Id: 123456      Name: Tom
Person deleted
Person deleted
Person deleted
```

Параметр шаблонов по умолчанию

Как и параметры функций, параметры шаблонов могут иметь значения по умолчанию - тип по умолчанию, который будет использоваться.



Консоль отладки Microsoft Visual Studio

```
Id: id1345      Name: Bob
Id: 23456      Name: Tom
```

```
#include <iostream>

template <typename T=int>
class Person {
public:
    Person(std::string name) : name{name} { }
    void setId(T value) { id = value;}
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    T id;
    std::string name;
};

int main()
{
    Person<std::string> bob{"Bob"};    // T - std::string
    bob.setId("id1345");
    bob.print();    // Id: id1345  Name: Bob

    Person tom{"Tom"};    // T - int
    tom.setId(23456);
    tom.print();    // Id: 23456   Name: Tom
}
```


Вложенные шаблоны классов

Шаблоны можно определить в классах или шаблонах классов, в этом случае они называются шаблонами элементов. Шаблоны членов, которые являются классами, называются шаблонами вложенных классов. Шаблоны элементов, которые являются функциями, рассматриваются в шаблонах функций-членов.

```
#include <iostream>
using namespace std;

class X
{
    template <class T>
    struct Y
    {
        T m_t;
        Y(T t) : m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()
{
    X x(1, 'a');
    x.print();
}
```

Консоль отладки Microsoft Visual Studio

1 a

C:\Users\phile\source\repos\ClassLec\x64\Debug\ClassLec.exe
(процесс 4120) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:

Специализация шаблонов

В некоторых случаях невозможно или желательно определить точно тот же код для любого типа. Например, может потребоваться определить путь к коду, который необходимо выполнить, только если аргумент типа является указателем или `std::wstring` или типом, производным от определенного базового класса. В таких случаях можно определить специализацию шаблона для конкретного типа. Когда пользователь создает экземпляр шаблона с этим типом, компилятор использует специализацию для создания класса, а для всех других типов компилятор выбирает более общий шаблон.

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

Пример полной специализации шаблонов

В данном случае специализация полная, так как для всех параметров шаблона (по сути для единственного параметра шаблона) указано значение - в данном случае тип unsigned. В этом случае после ключевого слова template идут пустые угловые скобки.

```
#include <iostream>

// шаблон класса
template <typename T>
class Person {
public:
    Person(std::string name) : name{ name }
    { }
    void setId(T value) { id = value; }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    T id;
    std::string name;
};

// полная специализация шаблона для типа unsigned
template <>
class Person<unsigned> {
public:
    Person(std::string name) : name{ name }
    {
        id = ++count;
    }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
private:
    static inline unsigned count{};
    unsigned id;
    std::string name;
};
```

```
int main()
{
    // объекты создаются на основе полной специализации шаблона
    Person<unsigned> tom{ "Tom" };
    tom.print(); // Id: 1    Name: Tom

    Person<unsigned> sam{ "Sam" };
    sam.print(); // Id: 2    Name: Sam

    // объект создается на основе класса, генерируемого компилятором по шаблону
    Person<std::string> bob{ "Bob" }; // T - std::string
    bob.setId("id1345");
    bob.print(); // Id: id1345 Name: Bob
}
```

Консоль отладки Microsoft Visual St...

```
Id: 1    Name: Tom
Id: 2    Name: Sam
Id: id1345    Name: Bob
```

C:\Users\phile\source\repos\Proba\x64\Debug\Proba.exe (процесс 14060) завершил работу с кодом 0.

Нажмите любую клавишу, чтобы закрыть это

Пример частичной специализации шаблонов

```
// шаблон класса
template <typename T, typename K>
class Person {
public:
    Person(std::string name, K phone) : name{ name }, phone{ phone }
    { }
    void setId(T value) { id = value; }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << "\tPhone: " << phone << std::endl;
    }
private:
    T id;
    std::string name;
    K phone;
};

// частичная специализация шаблона для типа unsigned
template <typename K>
class Person<unsigned, K> {
public:
    Person(std::string name, K phone) : name{ name }, phone{ phone }
    {
        id = ++count;
    }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << "\tPhone: " << phone << std::endl;
    }
private:
    static inline unsigned count{};
    unsigned id;
    std::string name;
    K phone;
};
```

Шаблоны классов и наследование

При наследовании класса на основе шаблона нам надо указать значения для параметров шаблона базового класса. И в данном случае мы можем также и производный класс определить как шаблон, и использовать его параметры при установке базового класса:

```
#include <iostream>

template <typename T>
class Person {
public:
    Person(T id, std::string name) : id{ id }, name{ name } { }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
protected:
    T id;
    std::string name;
};

template <typename T>
class Employee : public Person<T> {
public:
    Employee(T id, std::string name, std::string company) : Person<T>{ id, name }, company{ company } { }
    void print() const
    {
        Person<T>::print();
        std::cout << Person<T>::name << " works in " << company << std::endl;
    }
private:
    std::string company;
};

int main()
{
    Employee<unsigned> bob{ 123, "Bob", "Google" };
    bob.print();    // Id: 123 Name: Bob
                  // Bob works in Google
}
```

Консоль отладки Microsoft Visual Studio
Id: 123 Name: Bob
Bob works in Google

C:\Users\phile\source\repos\ClassLec\x64\Debug\ClassLec.exe (процесс 3064) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:

Шаблоны классов и наследование с указанием типа

```
#include <iostream>

template <typename T>
class Person {
public:
    Person(T id, std::string name) : id{id}, name{name} { }
    void print() const
    {
        std::cout << "Id: " << id << "\tName: " << name << std::endl;
    }
protected:
    T id;
    std::string name;
};

class Employee: public Person<unsigned> {
public:
    Employee(unsigned id, std::string name, std::string company) : Person{id, name}, company{company} { }
    void print() const
    {
        Person::print();
        std::cout << name << " works in " << company << std::endl;
    }
private:
    std::string company;
};

int main()
{
    Employee bob{123, "Bob", "Google"};
    bob.print();    // Id: 123 Name: Bob
                  // Bob works in Google
}
```

Консоль отладки Microsoft Visual Studio

Id: 123 Name: Bob
Bob works in Google

C:\Users\phile\source\repos\Proba\x64\Debug\Proba.exe (процесс 17092) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:

Функции и шаблоны

Использование функций даёт несколько преимуществ:

1. Если надо поменять повторяющуюся логику - достаточно сделать это в функции, не надо менять все копии одинакового кода в программе. Если бы в примере выше вариант без функции содержал системную ошибку в тернарных вызовах, с путаницей порядка операндов: `"(a >= b) ? b : a"` и `"(max_ab >= c) ? c : max_ab"` - ошибку пришлось бы искать и править во всех местах использования. Вариант с функцией же требует одной правки - в реализации функции.
2. При грамотном именовании в коде с функциями логика кода становится прозрачнее. В примере без функции внимательного прочтения требует каждая конструкция вида `"(... >= ...) ? ... : ..."`, надо узнавать повторяющуюся логику выбора большего значения из двух каждый раз заново. Функция же во втором варианте *именует* повторяющуюся логику, за счёт чего общий смысл программы понятнее.

Процедурное программирование делает код чище. Однако, что если логику получения максимального элемента надо поддерживать для всех числовых типов: для всех размеров (1, 2, 4, 8 байт), как знаковых, так и беззнаковых (signed / unsigned), для чисел с плавающей точкой ("float", "double")?

Пример функций без шаблонов

Можно воспользоваться перегрузкой функций:

```
#include <iostream>

int add(int, int);
double add(double, double);
std::string add(std::string, std::string);

int main()
{
    std::cout << "int: " << add(4, 5) << std::endl;
    std::cout << "double: " << add(4.4, 5.5) << std::endl;
    std::cout << "string: " << add(std::string("hel"), std::string("lo")) << std::endl;
}

int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

std::string add(std::string str1, std::string str2)
{
    return str1 + str2;
}
```


Шаблоны функций

```
#include <iostream>

template<typename T> T add(T, T); // прототип функции

int main()
{
    std::cout << "int: " << add(4, 5) << std::endl;
    std::cout << "double: " << add(4.4, 5.5) << std::endl;
    std::cout << "string: " << add(std::string("hel"), std::string("lo")) << std::endl;
}

template<typename T> T add(T a, T b)
{
    return a + b;
}
```

Перегрузки функций с повторяющейся логикой можно заменились на одну "функцию" с новой конструкцией - **template<typename Type>**. Слово "функция" взято тут в кавычки намеренно. Это не совсем функция. Данная запись означает для компилятора следующее: "После конструкции **template<typename Type>** описан *шаблон функции*, по которому *подстановкой* типа вместо *шаблонного аргумента Type* порождаются конкретные функции".