

## **Обектно-ориентирано програмиране (записки)**

- **Марина Господинова**
- **Илиан Запрянов**

# Тема 13. Шаблони

def| Шаблон

- Клас/функция с общо предназначение спрямо типа „ (необходима функция на тип)
- шаблон = параметричен полиморфизъм

Със сегашните знания функцията **max()** между числа и стрингове ще реализираме по следния начин:

```
(Global Scope)

#include <iostream>

int max(int a, int b)
{
    return a > b ? a : b;
}

const std::string& max(const std::string& lhs, const std::string& rhs)
{
    return lhs > rhs ? lhs : rhs;
}

int main()
{
    std::cout << max(3, 9) << std::endl;
    std::cout << max("ABC", "XY") << std::endl;

    return 0;
}
```

Но благодарение на шаблоните, можем да спестим дублирането на код:

```
(Global Scope)

#include <iostream>

//обозначаване на функция, че е темплейтна
template <typename T> //или template <class T>

//навсякъде типа, който не знаем се заменя с T
const T& max(const T& lhs, const T& rhs)
{
    return lhs < rhs ? rhs : lhs;
}

int main()
{
    std::cout << max<int>(3, 9) << std::endl;
    std::cout << max<std::string>("ABC", "XY") << std::endl;

    //<int> и <std::string> генерират нов код, където T е заменено
    //със съответния тип

    return 0;
}
```

```

#include <iostream>

template <typename T>
const T& max(const T& lhs, const T& rhs)
{
    return lhs < rhs ? rhs : lhs;
}

class A {};

int main()
{
    A obj1, obj2;

    //уловка
    max<A>(obj1, obj2);
    // ^
    // |
    // |
    // тъй като синтаксиса е max [<] A > ...
    // се очаква A да има предефиниран оператор <

    return 0;
}

```

[!] Всички колекции трябва да имат дефиниран оператор <

[!] Тъй като шаблонните функции/класове се генерират по време на компилация и типовете за шаблоните трябва да са известни по време на компилация, то можем да кажем, че шаблоните са **параметричен полиморфизъм**

#### Пример **Stack**

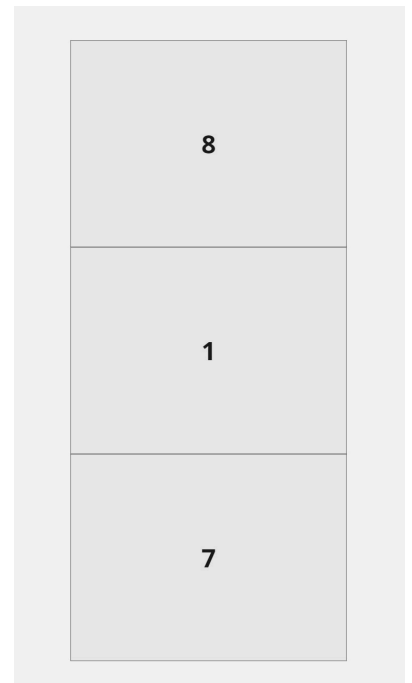
- абстрактна структура от данни
- имаме дефинирано поведение  
(знаем какво трябва да стане, но нямаме дефинирана имплементация)

```

int main()
{
    //стек, работещ с int-ове
    Stack<int> s;
    s.push(7);
    s.push(1);
    s.push(8);

    return 0;
}

```



чрез функцията **push()** добавяме нови елементи в нашия стек

```

s.pop(); //8
s.pop(); //1
s.pop(); //7

return 0;
}

```

чрез функцията **pop()** премахваме най-горния елемент в стека ни (тоест последния добавен)

Искаме максимален капацитет

- капацитетът да е шаблон
- типа на стека също да е шаблон

```

template <typename T, unsigned N>
class Stack
{
    T data[N]; //създаваме масив от тип T с размер N

    //(!) вече сме видели как се създават масиви
    //и можем да направим извода, че задължаваме T да има
    //default-ен конструктор
};

int main()
{
    Stack<int, 33> s1;
    Stack<int, 7> s2;
    Stack<char, 3> s3;

    //създадохме три различни стека
    //с различни типове данни, с които работят,
    //както и различни размери

    //[Note] int заменя T, 33 заменя N
    //
    //или първият аргумент заменя T, вторият заменя N

    return 0;
}

```

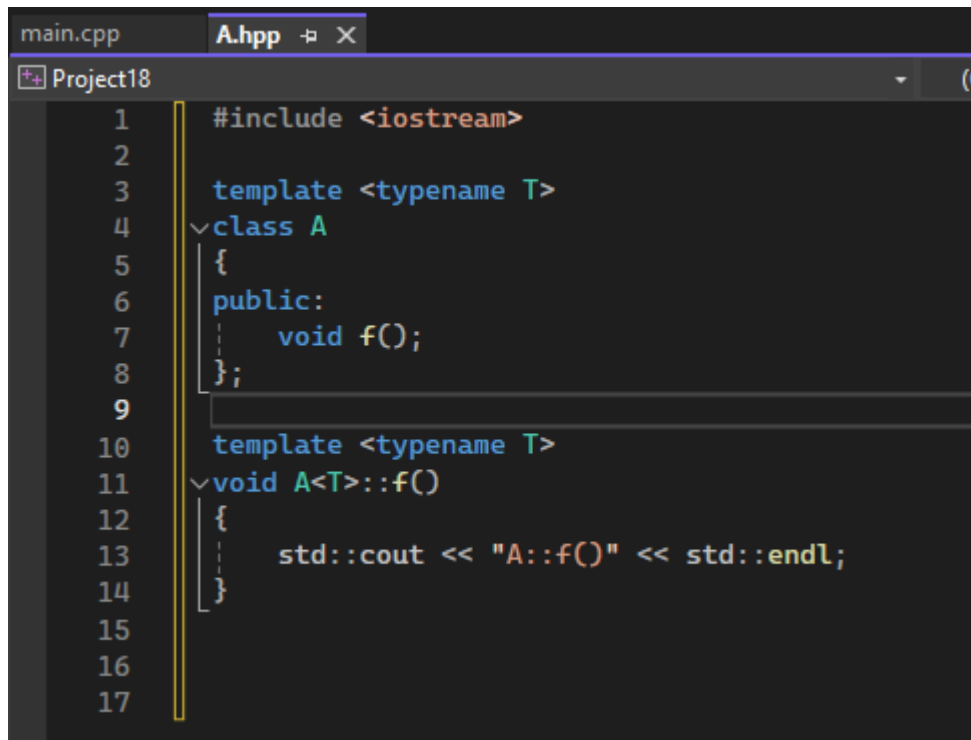
## Проблем при разделна компилация

main вижда .h файлът и ще замени **T** с подходящия тип, но възниква проблем, тъй като .h не вижда .cpp файла и не може **T** да се замени с подходящия тип в .cpp => компилаторът няма да може да генерира необходимия код за шаблона, тъй като не разполага с информацията за неговата имплементация.

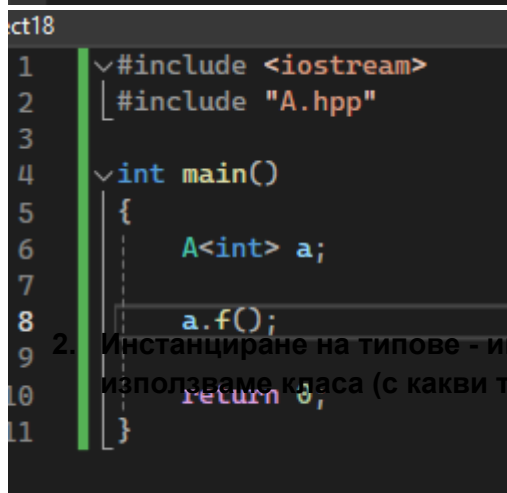
- проблем с разделната компилация - .h не вижда .cpp и не може да бъдат генерирани файлове за конкретния тип, указан от друг файл

## Решения

1. Не използваме разделна компилация и пишем всичко в един .hpp файл



```
main.cpp  A.hpp X
Project18
1  #include <iostream>
2
3  template <typename T>
4  class A
5  {
6  public:
7      void f();
8  };
9
10 template <typename T>
11 void A<T>::f()
12 {
13     std::cout << "A::f()" << std::endl;
14 }
15
16
17
```



```
ct18
1  #include <iostream>
2  #include "A.hpp"
3
4  int main()
5  {
6      A<int> a;
7
8      a.f();
9
10     return 0;
11 }
```

2. Инстанциране на типове - имплицитно ще кажем с какви типове ще използваме класа (с какви типове да генерира кода)

```
A.h  A.cpp  main.cpp
Project18
1  #pragma once
2  #include <iostream>
3
4  template <typename T>
5  class A
6  {
7  public:
8      void f();
9  };
10
11
```

```
A.h  A.cpp  main.cpp
Project18 (Global Scope)
1  #include "A.h"
2
3  template <typename T>
4  void A<T>::f()
5  {
6      std::cout << "A::f()" << std::endl;
7  }
8
9  //експлицитно инстанциране на класа A за конкретни типове
10 template A<int>;
11 template A<char>;
12
```

```
A.h  A.cpp  main.cpp
Project18
1  #include <iostream>
2  #include "A.h"
3
4  int main()
5  {
6      A<int> a;
7
8      a.f();
9
10     return 0;
11 }
```

# Темплейтна специализация

**def|** клас/функция с различно поведение спрямо типа

Темплейтната специализация ни позволява да дефинираме специални версии на шаблоните за конкретни типове. Това е полезно, когато общата реализация на шаблона не е подходяща за даден тип и е необходима специфична реализация.

Напр.:

```
1  #include <iostream>
2
3  template <typename T>
4  class A
5  {
6  public:
7      void f()
8      {
9          std::cout << "A::f()" << std::endl;
10     }
11 };
12
13 //създаваме класът A, специално за типа int
14 template <>
15 class A<int>
16 {
17 public:
18     void f()
19     {
20         std::cout << "int A::f()" << std::endl;
21     }
22 };
23
24 int main()
25 {
26     A<double> a1;
27     a1.f(); //A::f()
28
29     A<int> a2;
30     a2.f(); //int A::f()
31
32     return 0;
33 }
```

```

#include <iostream>

template <typename T>
void print(const T& value)
{
    std::cout << "print " << value << std::endl;
}

// специален print за тип int
template <>
void print<int>(const int& value)
{
    std::cout << "print int " << value << std::endl;
}

int main()
{
    print(3.14); //print
    print(42);   //print int
    print("Hello"); //print

    return 0;
}

```

**def|** Обекти , които се държат като функции

- обекти на клас, който има предефиниран `operator()`, който смята резултат от функцията



## Умни указатели

**def|** Обвиващи класове на указателите, които менажират паметта на обектите, към които сочат; не се интересуваме от триенето, а го прави указателя (delete)

Съществува `auto_ptr`, който е стар и не се използва, затова няма да го разглеждаме

### Unique\_ptr

- имаме точно 1 указател за точно един обект
- трябва да предефинираме операторите \*, ->
- изтрити са копиращия конструктор и оператор=
- разписани са move конструктор и move оператор=

Създаване:

**Unique\_ptr<A> ptr(new A(...))**

//поинтър към динамично заделен обект A

В истинския `unique_ptr` има функция `make_unique`, тя комбинира създаването на обекта и управлението на паметта, т.е. се спестява употребата на `new`, тъй като `make_unique` се грижи за заделянето на памет

Напр.:

**Unique\_ptr<A> ptr = make\_unique<A>(3, 7, 9, 'A');**

//за разлика от първото закачане на поинтъра, тук няма нужда от `new`

```
#include <iostream>

class A{};

int main()
{
    std::unique_ptr<A> ptr(new A()); //уникален поинтър към динамично
    //заделен обект

    std::unique_ptr<A> ptrTwo = std::make_unique<A>(); //спестява new

    return 0;
}
```

## Shared\_ptr

- имаме много указатели за един обект
- имаме copy и move семантики
- при първия указател се заделя обекта
- при изтриването на последния указател към обекта се изтрива и самия обект
- има указател към брояч, който следи броя на shared\_ptr към един обект
- броят на указателите **[!]** НЕ пазим в статична член-данна, а като указател към брояч, за да е един и същи за всички копия на обекта ни

```
int main()
{
    std::shared_ptr<A> sp1(new A(2, 3));

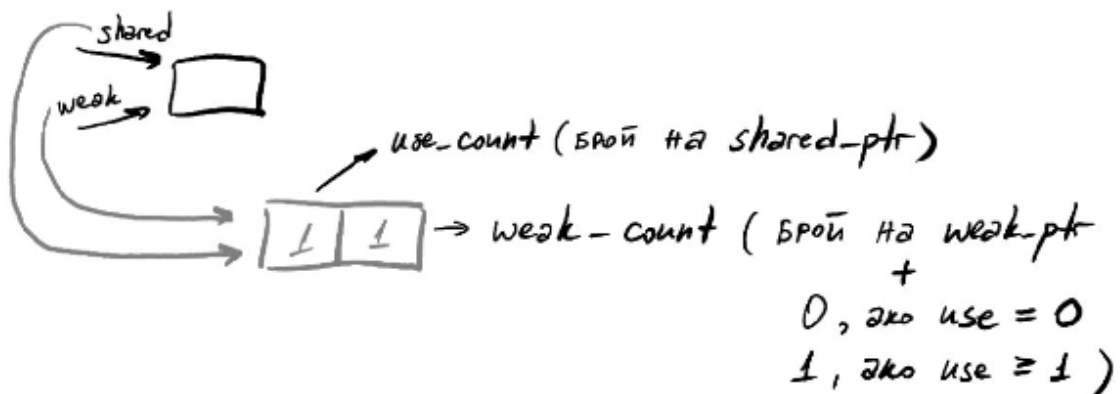
    std::shared_ptr<A> sp1 = std::make_shared<A>(2, 3);

    return 0;
}
```

Синтаксисът за shared\_ptr от stl е същият като този за unique\_ptr

## Weak\_ptr (not-owning ref)

- указател към обект, който е менажират от shared\_ptr
- не влияе на триенето и създаването
- възможно е промотиране от weak в shared
- ако обектът бъде преждевременно изтрит, имаме достъп до брояча на shared\_ptr
- weak\_ptr трябва да има проверка дали обектът още е жив



→ обектът се true, когато use\_count = 0

→ броят се true, когато weak\_count = 0

**[!]** weak\_count = брой на weak\_ptr + 1/0, за да сме сигурни, че имаме достъп до брояча и до проверката дали обектът още е жив