

Темплейти (Шаблони).

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)



практически
примери



теория

От какво ще се състои настоящата лекция?

Необходими функции в темплейтен клас/ функция. Темплейтни специализации. Concepts.
Примери за темплейтни класове/функции от стандартната библиотека. Вариадични
темплейти и std::tuple (C++ 11)

Проблем без темплейти?

Нека имаме функция, която прави едно и също, но за различни типове:

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

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

същото за `float`, `long`, `string`...

Подпомага DRY принципа - Don't Repeat Yourself

Решение

Въвеждаме темплейтна функция, която работи с всички типове данни

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

`<typename T>` и `<class T>` са еквивалентни!

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

Какво са темплейтите?

Темплейтът е механизъм за generic програмиране

Позволява писане на код, независим от типа.

Генерира се код по време на компилация.

```
template<typename T>
```

- T add(T a, T b) {
- return a + b;
- }

```
add(2, 3);           // T = int
```

```
add(2.5, 3.1);     // T = double
```

► Типът се извежда по време на компилация

Ограничения

Типът T трябва да поддържа всички използвани операции:

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

✓ T трябва да има:

```
operator>  
копиране / връщане по стойност  
иначе грешка при инстанциране
```

Какво е темплейтен клас

```
template<typename T>

class Box {
    T value;

public:
    Box(T v) : value(v) {}

    T get() const { return value; }

};

Box<int> b1(5);
Box<std::string> b2("Hello");
```

📌 Box<int> и Box<std::string> нямат нищо общо като типове

Пример

```
template<typename T>

class Repository {
    std::vector<T> items;

public:
    void add(const T& t) {
        items.push_back(t);
    }
};

Repository<Student> students;
Repository<int> ids;
```

Темплейтен клас с множествен брой параметри

```
template<typename T, typename U>

class Pair {

    T first; U second;

public:

    Pair(T a, U b) : first(a), second(b) {}

};
```

- 1) Т и U са параметри за типове.
- 2) При създаване на обект задаваме конкретните типове.
- 3) Може да се комбинират различни типове за максимална гъвкавост.
- 4) Темплейт може да има повече от един параметър, тип или константа.

Пример за използването им

```
int main() {
    Pair<int, double> p1(42, 3.14);
    Pair<std::string, int> p2("Age", 25);

    p1.print(); // 42, 3.14
    p2.print(); // Age, 25
}
```

Темплейтен клас с дефолтен тип

```
template <typename T = int>  
  
class DefaultBox {  
  
    T value;  
  
public:  
  
    DefaultBox(T v) : value(v) {}  
  
};  
  
DefaultBox<> b(42); // int по подразбиране
```

Може да зададем тип по подразбиране, ако потребителят не го уточни.

Улеснява използването на шаблони без дълги декларации.

Пример за използването им

```
template <int size>  
class Array {  
    int data[size];  
};  
Array<10> arr;
```

Шаблоните могат да имат параметри, които не са типове, напр. числа, указатели, bool. Полезно за статични масиви и compile-time оптимизации.

Какви функции са необходими при шаблонен клас?

Проблем

Зависи как ще се използва шаблонът, но често:

Конструктор

Деструктор

Copy / Move

Оператори (=, ==, <)
const коректност

Грешки при шаблони

Грешките са:
дълги, трудни за четене,
голям и неясен кол стек.

📌 Причина: компилаторът генерира код, който не може да се компилира.

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

```
struct Student {
    int grade;
    // ЛИПСВА operator>
};
```

Грешка

```
int main() {
    Student s1{5};
    Student s2{6};
    auto s = maxValue(s1, s2);
}

error: no match for 'operator>'  
(operand types are 'Student' and  
'Student')
```

Темплейтна специализация

Темплейтната специализация позволява различна имплементация на шаблон за конкретен тип или форма на тип.

„Общ шаблон → специален случай“. Видовете специализация са: Първичен шаблон (Primary template), пълна специализация и частична специализация (само за класове).

```
template<typename T>
struct TypeTraits {
    static void info() {
        std::cout << "Generic type\n";
    }
}; //първичен темплейт
```

Пълна специализация – конкретен тип

```
template<>
struct TypeTraits<int> {
    static void info() {
        std::cout << "Integer type\n";
    }
}; //пълна специализация
```

📌 Само за int и напълно заменя първичния template

- 1) Търси пълна специализация
- 2) Търси частична специализация (най-специфичната)
- 3) Използва първичния template

Variadic Templates

```
template <typename... Args>

void printAll(Args... args) {
    (std::cout << ... << args) << "\n";
}

printAll(1, 2.5, "Hello");
```

1) Поддържат неограничен брой параметри.

2) Много полезно за универсални функции и логване.

`typename... Args` – означава „неограничен брой типове“.

`Args... args` – параметрите съответстват на всеки тип.

`(std::cout << ... << args)` – fold expression, слепва всички параметри с оператор `<<`.

Пример за рекурсия

```
void print() {} // базов случай
```

```
template <typename T, typename... Args>
void print(T first, Args... rest) {
    std::cout << first << " ";
    print(rest...); // рекурсивно извикване
}
```

```
int main() {
    print(1, 2.5, "Hello", 'A');
}
```

Какво са concepts, въведени в C++ 20?

Concepts (въведени в C++20) са механизъм за описание на ограничения върху шаблонни параметри. Те позволяват да кажем какви свойства трябва да има даден тип, за да бъде използван в шаблон, и така:

- 1) дават по-ясни съобщения за грешка
- 2) позволяват по-добро претоварване и избиране на функции
- 3) правят кода по-четим и по-семантичен
- 4) проверяват изискванията по време на компилация, а не чрез сложни SFINAE конструкции

```
template<std::integral T>
T gcd(T a, T b) {
    while (b != 0) std::swap(a, b = a % b);
    return a;
}
```

Concept може да описва: операции. типове членове връщани типове. свойства (наследяване, copyable и др.)

Пример:

```
template<typename T>
requires Incrementable<T>
void foo(T x) {
    ++x;
}
```

Полиморфизъм чрез концепции

Полиморфизъмът чрез концепции в C++ позволява compile-time полиморфизъм, като се дефинира интерфейс чрез `concept`, който проверява наличието на определени методи или свойства на типа.

Функции или класове, използващи този `concept`, могат да работят с всякакви типове, които отговарят на условията, без да се използват виртуални функции. Това осигурява `type-safe` и високопроизводителен код, тъй като компилаторът избира правилните методи още при компилация. По този начин можем да постигнем полиморфно поведение за различни типове, без `runtime overhead`.



Пример за полиморфизъм

```
#include <concepts>
#include <iostream>

template <typename T>
concept Speakable = requires(T t) {
    { t.speak() } -> std::same_as<void>;
};

template <Speakable T>
void makeSpeak(T& obj) {
    obj.speak();
}

struct Dog { void speak() { std::cout << "Woof\n"; } };
struct Cat { void speak() { std::cout << "Meow\n"; } };

int main() {
    Dog d;
    Cat c;
    makeSpeak(d);
    makeSpeak(c);
}
```

Какво е STL - продължение от предни лекции

STL (Standard Template Library) е част от стандартната библиотека на C++ и представлява колекция от готови, генерични (темплейтни) компоненти за работа с данни и алгоритми.

Основната идея на STL е да раздели данните от алгоритмите и да ги свърже чрез итератори. STL се състои от 4 основни неща:

- 1) Контейнери – съхраняват данни
- 2) Алгоритми – обработват данните
- 3) Итератори – свързват контейнерите с алгоритмите
- 4) Функционални обекти (**functors, lambdas**) – поведение/логика

Контейнерите са темплейтни класове, които съхраняват обекти. Биват Последователни (Sequence containers)

```
std::vector<int> v = {1, 2, 3};  
std::list<std::string> names;
```

STL продължение

Асоциативни (Associative containers)

```
std::map<int, std::string> idToName;  
std::set<int> uniqueValues;
```

Подредени по ключ и Реализирани чрез дървета ($\log N$)

```
std::unordered_map<int, std::string> um; std::unordered_set<int> us;
```

Хеш-базирани, средно $O(1)$ достъп

Алгоритмите са темплейтни функции, които работят с итератори.

```
std::sort(v.begin(), v.end());  
std::find(v.begin(), v.end(), 3);
```

Често използвани:

```
std::sort, std::find, std::count, std::copy
```

Итераторите са обобщени указатели.

```
for (auto it = v.begin(); it != v.end(); ++it) {  
    std::cout << *it << " ";  
}
```

Видове:

Input, Output, Forward

Какво е std::tuple?

std::tuple е контейнер за съхранение на фиксиран брой елементи, всеки от които може да е различен тип.

Може да мислим за него като разширение на std::pair, но с неограничен брой елементи.

Фиксиран размер – броят на елементите се определя при създаване и не може да се променя по време на изпълнение.

Различни типове – всеки елемент може да има различен тип.

Типово-безопасен достъп – използваме std::get<index>(tuple) за достъп до елементи.

Поддържа structured bindings (C++17) – може да „разгънем“ tuple директно в променливи.

```
int main() {  
    // Създаваме tuple с различни типове  
  
    std::tuple<int, double, std::string> person(25, 72.5,  
                                                "Todor");  
  
    std::cout << "Age: " << std::get<0>(person) << "\n";  
  
    std::cout << "Name: " << std::get<2>(person) << "\n";  
  
    // Пример за промяна на стойностите  
  
    std::get<1>(person) = 75.0;  
  
    // Деструктуриране с C++17 (structured binding)  
  
    auto [age, weight, name] = person;  
  
    std::cout << "Destructured: " << age << ", " << weight <<  
           ", " << name << "\n";  
  
    return 0;  
}
```

Въпроси?

Благодаря за вниманието!