

Темплейти (Шаблони).

Необходими функции в темплейтен клас/ функция. Темплейтни специализации. SFINAE. Concepts. Вариадични темплейти и std::tuple (C++ 11)

д-р Тодор Цонков
todort@uni-sofia.bg

Какъв проблем решават темплейтите?

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...
По този начин може да се генерира
5 и повече пъти един и същи код,
който нарушава принципа DRY
(Don't Repeat Yourself).

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 представляват темплейтите?

Template (шаблон) е механизъм в C++, който позволява писане на generic код — код, който работи с различни типове данни, без да се преписва за всеки тип.

Това е compile-time механизъм.

👉 Темплейтът не е функция или клас сам по себе си.

Той е инструкция към компилатора как да генерира код за конкретен тип.

Когато напишем: `max(3, 5);`
Компилаторът създава: `int max(int a, int b);`
Когато напишем: `max(3.2, 5.1);` се създава:
`double max(double a, double b);`
Това става по време на компилация, не по време на изпълнение!

Типът T трябва да поддържа всички използвани операции иначе има компилационна грешка.

Темплейтни класове

```
template<typename T>
class Box {
private:
    T value_;
public:
    Box(T value) : value_(value) {}

    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

Те има различен layout и се генерира различен бинарен код.

Имаме два етапа при ползване -

Етап 1 — Parsing (проверка на синтаксис)

Компиляторът проверява дали template-ът е синтактично валиден.

Етап 2 — Instantiation

Когато използваме конкретен тип, компилаторът генерира конкретната версия

Темплейтни класове продължение

```
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.

```
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
}
```

Темплейтни класове продължение

```
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.

```
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.

Необходими функции при темплейтите

Зависи как ще се използва шаблонът, но често: конструктор, деструктор
Copy / Move, Оператори (=, ==, <)
const коректност.

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>
};

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

error: no match for 'operator>'
```

Специализация при шаблони

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

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

Специализация при шаблони

Частична специализация:

```
template<typename T>
```

```
struct X {};
```

```
template<typename T>
```

```
struct X<T*> {};
```

Компилаторът избира:

1) Non-template функции

2) По-специализирани template-и

3) По-общи template-и

Ако имаме void f(int);

```
template<typename T>
```

```
void f(T);
```

f(5); -> ще извика нетемплейтната

Реално се случва SFINAE

Substitution Failure Is Not An Error.

При substitution:

Ако тип стане невалиден → тази версия се изключва от overload resolution.

SFINAE

SFINAE работи в: Return type,
Template parameter list, Default
template arguments, Requires
expressions (преди concepts — чрез
enable_if)

НЕ работи в: Тялото на функцията

Пример

```
template<typename T>  
  
auto f(T t) -> decltype(t + 1) {  
    return t + 1;  
  
}
```

f(5); // OK

f("hello"); // ?

Стъпка 1: Substitution

За $T = \text{const char}^*$ компилаторът пробва: `decltype(t + 1)`
Това е валидно (pointer arithmetic).
Но ако беше: `decltype(t.non_existing())`

Substitution не успява

Тази версия се изключва

Няма компилационна грешка (ако има друга подходяща версия)

Защо SFINAE е сложен?

Проблеми:

- 1)Грешките са нечетими
- 2)Кодът става метапрограмиране
- 3)Лесно се злоупотребява

Пример за класически “template error wall”:

100 реда грешки от substitution chain.

`std::enable_if_t<std::is_integral_v<T>>`

Синтаксисът се различава
значително от стандартния C++

```
template<typename, typename = void>
struct has_to_string : std::false_type {};  
  
template<typename T>
struct has_to_string<T,
std::void_t<decltype(std::declval<T>().to_string(
))>
> : std::true_type {};
```

Ако `T::to_string()` съществува →
specialization се активира

Ако не → substitution failure

Подобрение - Concepts

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-добър overload 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;
}
```

Concepts - продължение

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;
}
```

Стандартната библиотека (в `<concepts>`) предоставя готови концепти:

- `std::integral`
- `std::floating_point`
- `std::same_as`
- `std::derived_from`
- `std::convertible_to`
- `std::regular`
- `std::invocable`

Concepts - примери

```
template<typename T>
concept Addable = requires(T a, T b)
{ a + b; };

template<typename T>
concept Printable = requires(T a)
{
    std::print("{}", a);
};
```

```
template<typename T>
concept Number = std::integral<T> ||
std::floating_point<T>;
```

Concepts позволяват:

- 1)По-ясни API-та
- 2)Ограничаване на шаблони
- 3)Документиране на намерението
- 4)По-добър generic design
- 5)По-добър type safety

В модерния C++ те са фундаментални за:
Ranges, STL алгоритмите, Модерни
библиотеки

Concepts - примери

Concept е предикат върху тип, който дефинира множество от типове, удовлетворяващи формални синтактични и семантични ограничения.

Функционално, concepts превъръщат implicit template requirements в explicit, проверими constraints.

Concepts са вдъхновени от:
type classes в Haskell и traits в Rust
Но реализацията им е специфична за C++ template системата.

Concepts в C++:

ограничават шаблоните
подобряват compile-time диагностика
правят generic програмирането формално и безопасно
заменят SFINAE в повечето случаи
са ключова част от модерния C++ (след C++20)

Variadic Templates

Вариадичните темплейти са механизъм в C++ (въведен в C++11), който позволява на шаблони (templates) да приемат произволен брой параметри – типове или стойности.

Те решават фундаментален проблем:

Как да дефинираме обобщен код, който работи с неограничен брой аргументи, без да пишем отделни overload-и за 1, 2, 3, ... N параметъра?

Ключовият синтаксис е:

`template<typename... Args>`

Тук:

`typename... Args` → пакет от типове (template parameter pack)

`Args...` → разширяване на пакета (pack expansion)

Variadic Templates

```
template<typename... Args>
auto sum(Args... args)
{
    return (args + ...);
}
auto result = sum(1, 2, 3, 4); // 10
((1 + 2) + 3) + 4
```

```
template<typename... Types>
class Tuple;
std::tuple<int, double, std::string>
```

std::forward в C++

std::forward е utility функция от <utility>, въведена в C++11, чиято цел е да реализира перфектно препращане (perfect forwarding) — т.е. да запази value category (lvalue/rvalue) на аргумент при предаване към друга функция.

std::forward

std::forward в C++

std::forward е utility функция от `<utility>`, въведена в C++11, чиято цел е да реализира перфектно препращане (perfect forwarding) – т. е. да запази value category (lvalue/rvalue) на аргумент при предаване към друга функция.

```
#include <print>
void foo(int& ) { std::print("lvalue\n"); }
void foo(int&&) { std::print("rvalue\n");}
```

```
template<typename T>
void wrapper(T arg)
{
    foo(arg);
}
int x = 5;
wrapper(x); // ?
wrapper(5); // ?
```

И в двета случая ще извика lvalue

Решението

```
template<typename T>
void wrapper(T&& arg)
{
    foo(std::forward<T>(arg));
}
int x = 5;
wrapper(x); // lvalue
wrapper(5); // rvalue
```

Реалната имплементация:

```
template<typename T>
constexpr T&&
forward(std::remove_reference_t<T>& arg)
noexcept
{
    return static_cast<T&&>(arg);
}
```

Ключът е: `static_cast<T&&>(arg)`

std::tuple

std::tuple е хетерогенен 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.

 Размерът и типовете са известни на compile-time

 Няма именовани полета (за разлика от struct)

std::tuple

```
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;
}
auto t = std::make_tuple(1, 2.5, "abc");
Пример за Factory design pattern.
```

templates vs generics в Java/C#

Java / C# – Generics

→ Механизъм за type safety и по-чист API.

→ Разширение върху вече съществуваща runtime система.

Ограничения:

Java → наследяване (extends)

C# → по-гъвкави ограничения (class, struct, new(), интерфейси)

C++ → Concepts (expression-based, структурни ограничения, най-мощи)

Java / C# → няма template specialization

C++ → има full и partial specialization

👉 Само C++ позволява различна имплементация за конкретен тип.

Java → няма compile-time execution

C# → ограничено (reflection, source generators)

C++ → Turing-complete template metaprogramming + constexpr

Въпроси?

Благодаря за вниманието!