

# Напредно генеричко програмирање 1

шаблони са променљивим бројем параметара,  
торке, `if constexpr`

# Шаблони функције (функцијски шаблони, темплејти функције)

- Да се подсетимо...

- Дефиниција шаблона функције:

```
template<class T>
T sumPow23(T x, T y) {
    return x * x + y * y * y;
}
```

- Инстанцирање шаблона функција (компајлер прави функцију за одговарајући тип на основу задатог шаблона)

```
int r1 = sumPow23<int>(3, 4);
int r2 = sumPow23(3, 4); // исто као и горњи код
```

- Параметри шаблона функције се закључују из типова стварних параметара.

# Шаблони функције (функцијски шаблони, темплејти функције)

```
template<typename T>
T sumPow23(T x, T y) {
    return x * x + y * y * y;
}
```

- На основу горњег шаблона, овај код ће довести до тога да компајлер направи функцију као на десној страни

```
int r1 = sumPow23(3, 4);      int sumPow23(int x, int y) {
                               return x * x + y * y * y;
                               }
```

```
int r1 = sumPow23(3, 4);
```

- Са другим стварним параметрима направиће се друга функција:

```
int r2 = sumPow23(3.0, .5); double sumPow23(double x, double y) {
                               return x * x + y * y * y;
                               }
```

```
int r2 = sumPow23(3.0, .5);
```

# Појмовник

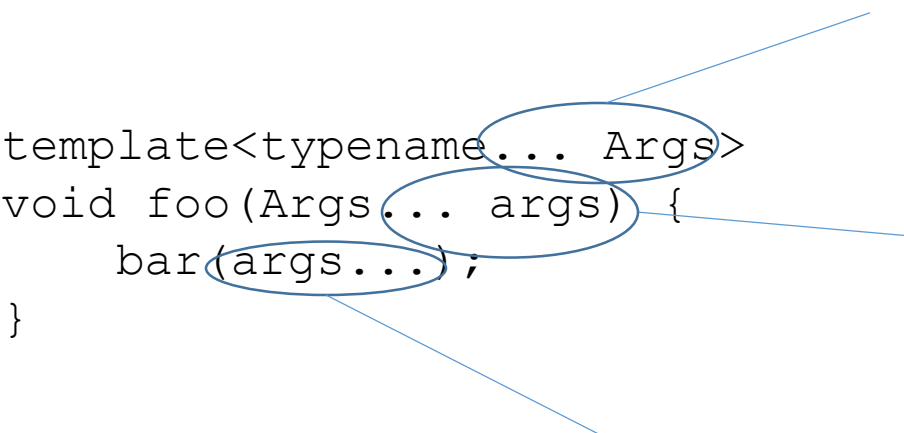
- Најбоље је прихватити овакву употребу појмова:
  - Исправно: шаблон класе, class template, темплејт класе, класни шаблон
  - **Неисправно: шаблонска класа, template class, тамплејт класа**
  - Исто је и за шаблон функције.
  - „vector је шаблон класе.“ – исправно.
  - **„vector је шаблонска класа.“ – неисправно.**
- Врло често ћете наилазити на неисправну употребу појмова. Није само по себи страшно, али може подсвесно створити погрешну слику о томе шта се заправо дешава.
- Сам шаблон није класа, или функција, већ се по том шаблону могу правити класе, или функције (множина!)
- У том смислу, појмови као што су „шаблонска класа“ могу се употребити да означе појединачну инстанцу шаблона („класа која је настала од шаблона“), нпр. „vector<int> је шаблонска класа“.

# Шаблони са променљивим бројем параметара

- Шаблони са променљивим бројем параметара омогућавају да се направи један шаблон који покрива случајеве са произвољно много параметара различитог типа.

... означава да је у питању произвољан број параметара и то се назива „пакет параметара шаблона“. Args је само назив за касније идентификовање (и може бити било шта).

```
template<typename... Args>  
void foo(Args... args) {  
    bar(args...);  
}
```



Ова употреба је врло специфична и означава „распакивање“ пакета параметара шаблона у листу параметара функције, која се зове „пакет параметара функције“. args је такође произвољан назив.

Ово је тзв. „распакивање“ параметара функције. Означава да на том месту треба да се нађе листа свих функцијских параметара одвојених зарезима.

# Шаблони са променљивим бројем параметара

```
template<typename... Args>
void foo(Args... args) {
    bar(args...);
}
```

- Илустрација употребе оваквог шаблона

```
foo(5, 0.5, "hik");
// је исто што и ово
foo<int, double, const char*>(5, 0.5, "hik");
// што чини да компајлер генерише овакву функцију:
void foo(int p1, double p2, const char* p3) {
    bar(p1, p2, p3);
}
```

- Наравно, p1, p2 и p3 су измишљена имена.

- Узгред, ово је јако користан сајт: [c++.io](http://c++.io)

# Шаблони са променљивим бројем параметара

```
template<typename... Args>
void foo(Args... args) {
    bar(args...);
}
```



- Илустрација употребе оваквог шаблона

```
foo(5, 0.5, "hik");
// је исто што и ово
foo<int, double, const char*>(5, 0.5, "hik");
// што чини да компајлер генерише овакву функцију:
void foo(int p1, double p2, const char* p3) {
    bar(p1, p2, p3);
}
```

- Наравно, p1, p2 и p3 су измишљена имена.

- Узгред, ово је јако користан сајт: [cppinsights.io](http://cppinsights.io)

# Шаблони са променљивим бројем параметара

- Лако се може одабрати како се преносе параметри.

```
template<typename... Args>
void foo(Args&... args) {
    bar(args...);
}
```

```
foo(5, 0.5, "hik");
```

```
void foo(int & p1, double & p2, const char* & p3) {
    bar(p1, p2, p3);
}
```

- Наравно, може и овако:

```
template<typename... Args> void foo(const Args&... args) //...
template<typename... Args> void foo(Args&&... args) //...
```

- На овај механизам се ослањају конструктори `std::thread` и `std::scoped_lock`, као и функције `std::make_shared` и `std::make_unique`.



# Шаблони са променљивим бројем параметара

- Важно је имати у виду да се параметри распакују по обрасцу који је задат.
- Овај механизам се назива „fold expression“.

```
template<typename... Args>
void foo(Args... args) {
    bar(2*args...);
}
```

```
foo(5, 0.5, 4);
```

```
void foo(int p1, double p2, int p3) {
    bar(2*p1, 2*p2, 2*p3);
}
```

```
template<typename... Args>
void foo(Args... args) {
    bar(baz(args)...);
}
```

```
foo(5, 4);
```

```
void foo(int p1, int p2) {
    bar(baz(p1), baz(p2));
}
```

# Шаблони са променљивим бројем параметара

- Ово је облик шаблона где је једини параметар заправо „пакет“ параметара.

```
template<typename... Args>  
void foo(Args... args) //...
```

- Мада врло често имамо и овако нешто:

```
template<typename T, typename... Args>  
void foo(T x, Args... args) {  
    // сада имамо x као први параметар и args као остали параметри  
    // сада можемо рекурзивно ићи кроз листу параметара  
    // тј. “љуштимо” листу  
}
```

# Шаблони са променљивим бројем параметара

- Једноставни пример суме:
- Желимо да нам функција враћа суму свих параметара, нпр.:

```
int a = sum(1, 2, 3); // a треба да буде 6
int b = sum(6, 7, 5, 2); // b треба да буде 20
```

- Имплементација би могла овако да изгледа:

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
    return x + sum(args...);
}
```

```
sum(3, 4, 5);
```

```
int sum(int x, int p1, int p2) { return x + sum(p1, p2); }
int sum(int x, int p1) { return x + sum(p1); }
int sum(int x) { return x + sum(); }
int sum() ??? // по шаблону мора бити бар један параметар
```

- ...али проблем је што sum() функција није дефинисана.

# Шаблони са променљивим бројем параметара

- Једно решење може бити да дефинишемо ту недостајућу функцију `sum()`, али ту би наишли на неколико (мањих) проблема.
- Боље решење је да урадимо специјализацију шаблона за један параметар:

```
template<typename T>
```

```
T sum(T x) {  
    return x;  
}
```

```
template<typename T, typename... Args>
```

```
T sum(T x, Args... args) {  
    return x + sum(args...);  
}
```

```
sum(3, 4, 5);
```

```
int sum(int x, int p1, int p2) { return x + sum(p1, p2); }
```

```
int sum(int x, int p1) { return x + sum(p1); }
```

```
int sum(int x) { return x; }
```

- Сад је све ОК.

# Шаблони са променљивим бројем параметара

- Корисно је знати и за **sizeof...** операцију.
- Та операција се примењује на пакет параметара и враћа број параметара.
- Илустровано на функцији која враћа средњу вредност својих параметара.

```
template<typename... Args>
auto average(Args... args) {
    return sum(args...) / sizeof...(args);
}
```

- Подсећање: шта ће бити тип повратне вредности функције **average**?
- Заправо, и тип повратне вредности **sum** би требало тако да дефинишемо, ако желимо да ради исправно и са разноврсним типовима.

```
template<typename T> T sum(T x) { return x; }
```

```
template<typename T, typename... Args>
auto sum(T x, Args... args) { return x + sum(args...); }
```

```
// сад ће радити исправно и за овакав позив:
double x = sum(5, 0.5, 7);
```

# Шаблони са променљивим бројем параметара

- Функције настале од шаблона могу бити срачунљиве током превођења:

```
template<typename T>
constexpr T sum(T x) {
    return x;
}
```

```
template<typename T, typename... Args>
constexpr auto sum(T x, Args... args) {
    return x + sum(args...);
}
```

```
template<typename... Args>
constexpr auto average(Args... args) {
    return sum(args...) / sizeof...(args);
}
```

```
constexpr int x = average(5, 6, 7);
```

# Торке

- Торке би биле нека врста шаблона класе са променљивим бројем атрибута.
- Оне нам омогућавају да генерички јединствено изразимо све ове класе:

```
struct X {  
    int a1;  
    float a2;  
};
```

```
struct Y {  
    double a1;  
    std::string a2;  
    long a3;  
};
```

```
struct Z {  
    long long a1;  
    const double* a2;  
    char* a3;  
    std::vector<int> a4;  
};
```

# Торке

- Торке би биле нека врста шаблона класе са променљивим бројем атрибута.
- Оне нам омогућавају да генерички јединствено изразимо све ове класе:

```
struct X {  
    int a1;  
    float a2;  
};  
  
std::tuple<int, float> x;  
// или  
auto x = std::make_tuple(5, 0.5);
```

```
struct Y {  
    double a1;  
    std::string a2;  
    long a3;  
};  
  
std::tuple<  
    double,  
    std::string,  
    long> y;
```

```
struct Z {  
    long long a1;  
    const double* a2;  
    char* a3;  
    std::vector<int> a4;  
};  
  
std::tuple<  
    long long,  
    const double*,  
    char*,  
    std::vector<int>> z;
```



# Торке

- Постоји могућност и да се елементи торке „распакују“ у појединачне променљиве:

```
std::tuple<int, float, int> x;  
int a1, a3;  
float a2;  
std::tie(a1, a2, a3) = x;  
// а може и овако:  
std::tie(std::ignore, a2, std::ignore) = x;
```

- Један од најчешћих начина коришћења торки је за враћање више повратних вредности из функције:

```
std::tuple<int, float, int> foo() {  
    ...  
    return {5, x, y}; // до C++17 је морало овако да се пише:  
    // return std::tuple<int, float, int>{5, x, y};  
}  
int a1, a3;  
float a2;  
std::tie(a1, a2, a3) = foo();
```

# Торке

- Приступ елементима торке:

```
std::tuple<int, float, int> x;  
std::get<1>(x)  
std::get<0>(x)  
std::get<float>(x)  
std::get<int>(x) // грешка!!! који елемент типа int? има их два
```

- Аутоматско распакивање торки

```
std::tuple<int, float, int> foo() {  
    ...  
    return {5, x, y};  
}  
  
auto [a1, a2, a3] = foo();  
// auto [a1, a2, a3] {foo()};  
// auto [a1, a2, a3] (foo());
```

# Торке

- Распакивање ради и за редовне структуре, али и низове:

```
structure S {  
    int x;  
    float y;  
    int z;  
};
```

```
S x;  
auto [a1, a2, a3] = x;
```

```
int ary[10];  
auto [a1, a2, a3] = ary;
```

# Торке

- Али, постоје и друге користи од торки.
- У суштини, то је алтернативни начин прављења хетерогених скупова.
- На пример...
- Желимо да имамо објекат који представља сложену функцију која прима цео број и враћа цео број, и која је следећег облика:

$$F(x) = \sum_{i=1}^n f_i(x)$$

- где су  $f_i$  било какви функцијски објекти који имају дефинисан оператор  $()$  који прима цео број и враћа цео број.
- Сваки позив оператора  $()$  тог сложеног објекта над целобројним параметром  $x$ , треба да проследи то  $x$  свим функцијама  $f_i$  и да сумира резултате.

$$F(x) = \sum_{i=1}^n f_i(x)$$

## Торке

- Један начин је да успоставимо хијерархију типова, где базна класа има виртуелну методу (и `operator()` може бити виртуелан) за срачунавање функције  $int \rightarrow int$ .
- Затим, да направимо да наша класа сложене функције има вектор (или неки други контејнер) чији елементи су показивачи на базну класу, и у који тако можемо трпати показиваче на објекте било које класе изведене из те базне.
- Срачунавање функције у сложеној класи би се онда свело на пролазак кроз контејнер и сумирање резултата које даје позив виртуелне метода за параметар  $x$ .

```
struct SumFunc : BaseFunc {  
    int operator()(int x) override {  
        int sum{0};  
        for (auto it : m_fs)  
            sum += (*it)(x); // виртуелна метода  
        return sum;  
    }  
private:  
    std::vector<BaseFunc*> m_fs;  
};
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

## Торке

- Код тог приступа морамо мислити о томе ко ствара/уништава функције у вектору (тј. ко је њихов власник), имамо ограничење да се рачунају само функцијски објекти класе која наслеђује нашу базну, позив је спорији због индирекције и виртуелног позива, а и смештање низа функција у меморију није компактно, па је итерирање спорије...
- Део проблема можемо решити употребом `std::function`.
- Али то се испод хаубе своди на скоро исти механизам.

```
int foo(int x) { /* ... */ }
```

```
std::function<int(int)> x(foo);
```

```
std::function<int(int)> y([](int x) -> int { /* ... */ });
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

# Торке

- Коришћењем торки можемо понудити другачије решење које има неколико предности:

```
template<typename... Ts>
struct SumFunc {
    SumFunc(std::tuple<Ts...> x) : m_tuple(x) {}
    int operator()(int x) { return eval<m_ts>(x); }

private:
    constexpr static std::size_t m_ts = sizeof...(Ts);

    template<int I>
    int eval(int x) {
        ...
    } // пролази кроз елементе торке, зове operator() и збраја

    std::tuple<Ts...> m_tuple;
};
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

# Торке

- Сада стварање нове сложене функције може да изгледа овако:

```
template<typename... Ts>
SumFunc<Ts...> makeSumFunc(Ts... ts) {
    return SumFunc<Ts...>(std::make_tuple(ts...));
}

...

auto f = makeSumFunc(
    [](int x){return 5 * x;}, [](int x){return 6 + x;});
std::cout << f(4); // исписаће 5*4 + (6+4) = 30

auto g = makeSumFunc(
    [](int x){return x / 2;}, f);
std::cout << g(4); // исписаће 4/2 + f(x) = 32
```

- Обратити пажњу да променљиве f и g нису истог типа, те – на пример – не може да се уради ово:

```
f = g
```



# Разрешење преклопљених функција

- Замислимо да имамо неки генерички алгоритам који ради нешто, и усредсредимо се на један његов параметар (остале параметре ћемо занемарити у коду):

```
template<typename T> void myAlg(T x);
```

- Узмимо да уколико је тај параметар цео број онда имам врло специјалан алгоритам, који је различит од генеричког и који сјајно ради за тај тип.
- Желимо да имамо униформан начин позивања функције која обавља алгоритам, али да се у случају целобројног параметра у ствари позове она специјална функција.
- Могли би додати и обичну функцију само за long long:

```
void myAlg(long long x); // специјална верзија
```

```
int aInt;
```

```
long long aLongLong;
```

```
myAlg(aLongLong); // Биће позвана специјална верзија
```

```
myAlg(aInt); // Неће бити позвана специјална верзија :
```

# Разрешење преклопљених функција

- Зашто специјална верзија није позвана и у случају *int* параметра?
- Три корака у разрешавању преклопљених функција:
  - 1. Прво се шаблони функција игноришу. Уколико постоји редовна функција која одговара по имену и тачно по типу параметара (по потпису), онда ће се та функција позвати (објашњава зашто се за **long long** зове специјална верзија)
  - 2. Сада се гледају шаблони функција. Уколико се неки шаблон функције може инстанцирати да тачно договара типовима параметара (даје тачан потпис), онда ће се та функција позвати (и биће направљена на основу тог шаблона – ако већ није)
  - 3. Тек у трећем кораку се разматрају редовне (нешаблонске) функције које би се могле позвати уз примену неких имплицитних конверзија (када би се са *int* параметром могла позвати функција која прима **long long**, али у претходном кораку се проналази одговарајућа инстанца (специјализација) шаблона и до трећег корака неће ни доћи)

# Својства типова

- Дакле, треба нам да некако изразимо „за све целе бројеве“, односно да наша функција ради на специјалан начин ако је тип параметра цео број, а на генерички начин за све остале случајеве.
- Прво да видимо како да утврдимо да ли је нешто цео број.
- Стандардна библиотека нуди помоћ:

```
#include <type_traits>
```

```
template<typename T> void myAlg(T x) {  
    if (std::is_integral_v<T>)  
        // if (std::is_integral<T>::value) <- овако мора до Це++17  
        // и ослања се на шаблоне променљивих  
        myAlgSpec(x); // специјална верзија  
    else  
        // регуларна генеричка верзија  
}
```

## if constexpr

- Погледајмо још једном овај пример:

```
template<typename T> void myAlg(T x) {
    if (std::is_integral_v<T>)
        myAlgSpec(x); // специјална верзија
    else
        // регуларна генеричка верзија
}
```

- Ово су два изгледа кода, у зависности да ли јесте или није цео број:

```
void myAlg(int x) {
    if (true)
        myAlgSpec(x); // специјална верзија
    else
        // регуларна генеричка верзија
}

void myAlg(float x) {
    if (false)
        myAlgSpec(x); // специјална верзија
    else
        // регуларна генеричка верзија
}
```

## if constexpr

- Сличан принцип можемо применити на овај код са почетка предавања:

```
template<typename T>
T sum(T x) {
    return x;
}
template<typename T, typename... Args>
T sum(T x, Args... args) {
    return x + sum(args...);
}
```

- и написати га на овај начин:

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
    if (sizeof...(args) == 0)
        return x;
    else
        return x + sum(args...);
}
```

## if constexpr

- Али биће грешка у превођењу јер од шаблона настају следеће функције:

```
sum(3, 4);
```

```
int sum(int x, int p1) {  
    if (1 == 0)  
        return x;  
    else  
        return x + sum(p1);  
}
```

```
int sum(int x) {  
    if (0 == 0)  
        return x;  
    else  
        return x + sum(); // sum() није дефинисано  
}
```

- Обратити пажњу да се `sum()` никада не би ни позвало, али током превођења мора бити дефинисано (или бар декларисано)

## if constexpr

- За овакве случајеве, када имамо услов који је срачунљив током превођења, имамо на располагању наредбу `if constexpr`

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
    if constexpr(sizeof...(args) == 0)
        return x;
    else
        return x + sum(args...);
}
```

- Сада ће функције настале од шаблона `sum` изгледати овако:

```
sum(3, 4);
```

```
int sum(int x, int p1) {
    return x + sum(p1);
}
```

```
int sum(int x) {
    return x;
}
```

## Дигресија: sum на најлакши начин 😊

- Коришћењем „fold expression“ механизма:

```
template<typename... Args>  
auto sum(Args... args) {  
    return (args + ...);  
}
```

- Важне су заграде око **args + ...**



# if constexpr

- И ово је сада боље да пишемо овако:

```
template<typename T> void myAlg(T x) {  
    if constexpr(std::is_integral_v<T>)  
        myAlgSpec(x); // специјална верзија  
    else  
        // регуларна генеричка верзија  
}
```

- Обратите пажњу да ваљаност одбачене гране није битна само у случају да услов зависи од шаблонских параметара.
- На пример, ово је и даље грешка у превођењу:

```
if constexpr(false) {  
    int x = "43";  
}
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

## if constexpr

- Сада можемо елегантно имплементирати и eval методу:

```
template<typename... Ts>
struct SumFunc {
    SumFunc(std::tuple<Ts...> x) : m_tuple(x) {}
    int operator()(int x) { return eval<m_ts>(x); }
private:
    constexpr static std::size_t m_ts = sizeof...(Ts);
    template<int I>
    int eval(int x) {

    }
    std::tuple<Ts...> m_tuple;
};
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

## if constexpr

- Сада можемо елегантно имплементирати и eval методу:

```
template<typename... Ts>
struct SumFunc {
    SumFunc(std::tuple<Ts...> x) : m_tuple(x) {}
    int operator()(int x) { return eval<m_ts>(x); }
private:
    constexpr static std::size_t m_ts = sizeof...(Ts);
    template<int I>
    int eval([[maybe_unused]] int x) {
        if constexpr (I == 0)
            return 0;
        else {
            auto p = std::get<m_ts - I>(m_tuple);
            return p(x) + eval<I - 1>(x);
        }
    }
    std::tuple<Ts...> m_tuple;
};
```

$$F(x) = \sum_{i=1}^n f_i(x)$$

## if constexpr

- Верзија класе за две функције ће имати следеће eval методе:

```
template<>
int SumFunc::eval<0>([[maybe_unused]] int x) {
    return 0;
}
```

```
template<>
int SumFunc::eval<1>([[maybe_unused]] int x) {
    auto p = std::get<1>(m_tuple);
    return p(x) + eval<0>(x);
}
```

```
template<>
int SumFunc::eval<2>([[maybe_unused]] int x) {
    auto p = std::get<0>(m_tuple);
    return p(x) + eval<1>(x);
}
```