

Мало другачије решење старог проблема

- Имамо типове разних геометријских облика (и сви они наслеђују Shape)
- Желимо да обављамо већи број разних обрада над структуром таквих елемената – тако да сваки тип елемената има своју верзију те обраде (нпр. draw_shape)
- Два начина која смо видели до сада:
 - Kind поље у класама елемената, па онда једна функција која користи switch/case
 - Виртуелне методе/функције

Проблеми са досадашњим решењима

- Проблем са првим начином:
 - Може бити спорији како расте број класа.
 - Велике switch/case наредбе су незграпне.
- Проблем са другим начином:
 - У свим класама елемената (изведеним из Shape) морамо имплементирати одговарајућу виртуелну функцију која обавља жељену обраду, а то је мукотрпно јер морамо да „скачемо“ по класама. Поготово ако имамо много таквих обрада.
 - Ако имам много обрада, онда претрпавамо типове елемената виртуелним методама.

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct C {  
    void foo(myClass1& x); // 1  
    void foo(myClass2& x); // 2  
    void foo(myClass3& x); // 3  
};
```

```
// ...
```

```
C a;
```

```
// ...
```

```
a.foo(y); // 1, 2 ili 3? Од чега зависи? Када ће се знати?
```

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct C {  
    void foo(myClass1& x); // 1  
    void foo(myClass2& x); // 2  
    void foo(myClass3& x); // 3  
};
```

```
// ...
```

```
C a;
```

```
// ...
```

```
a.foo(y); // 1, 2 ili 3? Од чега зависи? Када ће се знати?
```

Статичко разрешење

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct B {  
    virtual void bar(myClass& x) = 0;  
};
```

```
struct D1 : B {  
    void bar(myClass& x) override { ... }  
};
```

```
struct D2 : B {  
    void bar(myClass& x) override { ... }  
};
```

```
B* p;  
myClass y;  
// ...  
p->bar(y); // B::bar, D1::bar ili D2::bar? Од чега зависи?  
           // Када ће се знати?
```

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct B {  
    virtual void bar(myClass& x) = 0;  
};  
  
struct D1 : B {  
    void bar(myClass& x) override { ... }  
};  
  
struct D2 : B {  
    void bar(myClass& x) override { ... }  
};
```

```
B* p;  
myClass y;  
// ...
```

```
p->bar(y); // B::bar, D1::bar ili D2::bar? Од чега зависи?  
          // Када ће се знати?
```

Динамичко разрешење

Посетилац (engl. Visitor)

- Образац Посетиоца комбинује ова два механизма:

```
struct B {  
    virtual void accept(Visitor& v) = 0;  
};  
struct D1 : B {  
    void accept(Visitor& v) override { v.visit(*this); }  
};  
struct D2 : B {  
    void accept(Visitor& v) override { v.visit(*this); }  
};  
// accept мора постојати за сваку класу у хијерархији  
  
struct Visitor {  
    virtual void visit(D1& x) = 0;  
    virtual void visit(D2& x) = 0; //по једна метода за сваку класу  
};  
  
struct VisForFunc1 : Visitor { // по класа за сваку обраду  
    void visit(D1& x) override { ... }  
    void visit(D2& x) override { ... }  
};
```

```
B* p; VisForFunc1 vis;  
p->accept(vis);
```

7

Полиморфизам

- Ово је релативно честа конструкција:

```
struct BaseClass {
    virtual void specific() =0;

    void function() {
        std::cout << "Some general code";
        specific();
    }
};

struct Derived1 : BaseClass {
    void specific() override { std::cout << "Derived1"; }
};

struct Derived2 : BaseClass {
    void specific() override { std::cout << "Derived2"; }
};

Derived1 x;
Derived2 y;
x.function();
y.function();
```


Полиморфизам

- Али, можемо урадити и овако:

```
template<typename T>
struct BaseClass {
    void function() {
        std::cout << "Some general code";
        static_cast<T*>(this)->specific();
    }
};

struct Derived1 : BaseClass<Derived1> {
    void specific() { std::cout << "Derived1"; }
};

struct Derived2 : BaseClass<Derived2> {
    void specific() { std::cout << "Derived2"; }
};

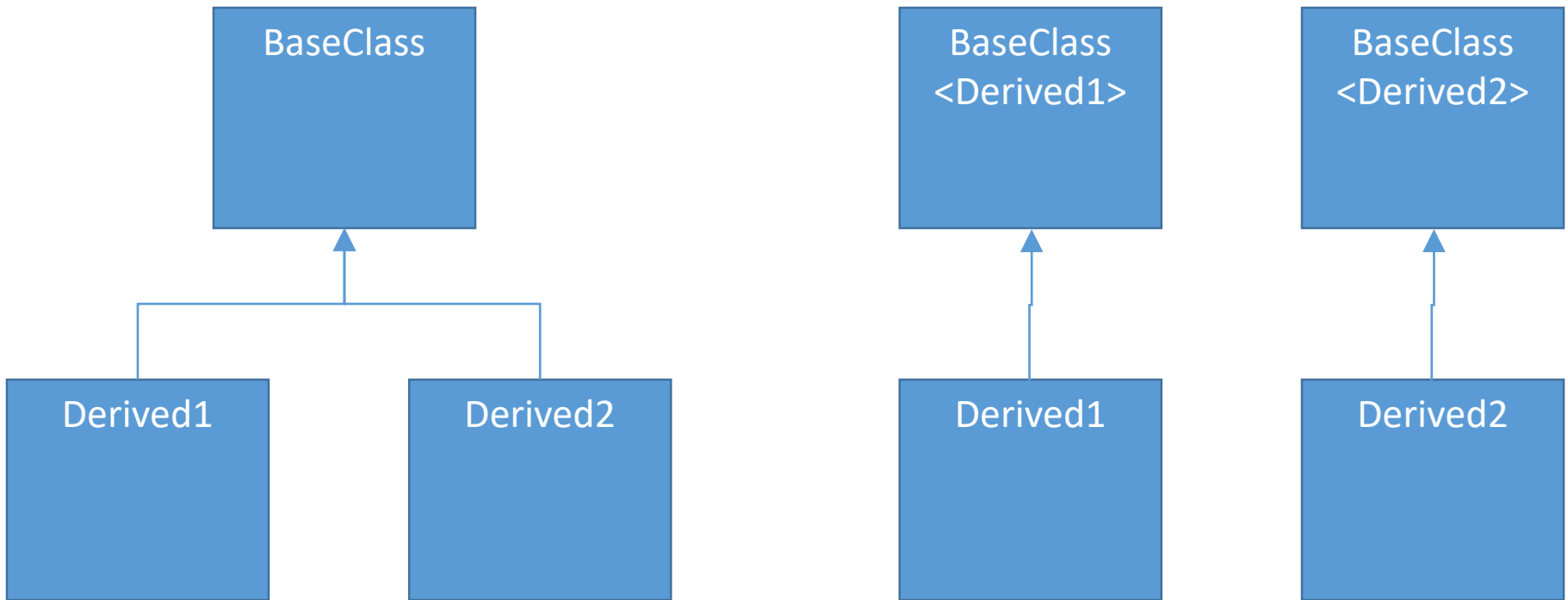
Derived1 x;
Derived2 y;
x.function();
y.function();
```

Статички полиморфизам - CRTP

- То што је показано на претходном примеру представљају статички полиморфизам, а он је постигнут коришћењем нечега што се назива CRTP (Curiously Recurring Template Pattern).
- Статички полиморфизам може бити користан у многим случајевима када немамо показиваче и референце на базну класу, тј. када током превођења имамо информације о типу објекта.
- CRTP може бити коришћен за разне ствари.

Статички полиморфизам - CRTP

- Приметите да је хијерархија класа различита у ова два примера:



C RTP

- C RTP може бити коришћен да дода функционалност типу.
- Ево класе која има функционалност бројања објеката:

```
class MyClass1 {  
    inline static int objCounter = 0;  
  
public:  
    MyClass1() { ++objCounter; }  
    MyClass1(const MyClass1& x) { ++objCounter; } // плус остали код за конструкцију  
    ~MyClass1() { --objCounter; }  
    int getObjectNum() { return objCounter; }  
};
```

- Ствари се компликују када имамо више конструктора и више различитих класа које треба да имају исту ту функционалност.

C RTP

- Са C RTP обрасцем („идиомом“) можемо направити шаблон базне класе која пружа ту могућност свим изведеним класама.

```
template<typename T>
class EnableObjCount {
    inline static int objCounter = 0;

protected:
    EnableObjCount() { ++objCounter; }
    ~EnableObjCount() { --objCounter; }

public:
    int getObjectNum() { return objCounter; }
};

class MyClass1 : public EnableObjCount<MyClass1> {
    //...
};

class MyClass2 : public EnableObjCount<MyClass2> {
    //...
};
```

C RTP

- На сличан начин, C RTP може бити употребљен и у хијерархији класа са динамичким полиморфизмом.

```
struct BaseClass {  
    virtual ~BaseClass(){};  
    virtual BaseClass* clone() const =0;  
};
```

```
struct Derived1 : BaseClass {  
    BaseClass* clone() const override {  
        return new Derived1(*this);  
    }  
};
```

```
struct Derived2 : BaseClass {  
    BaseClass* clone() const override {  
        return new Derived2(*this);  
    }  
};
```

CRTP

- На сличан начин, CRTP може бити употребљен и у хијерархији класа са динамичким полиморфизмом.

```
struct BaseClass {
    virtual ~BaseClass(){};
    virtual BaseClass* clone() const =0;
};

template<typename T>
struct BaseClassCRTP : BaseClass {
    BaseClass* clone() const override {
        return new T(*static_cast<const T*>(this));
    }
};

struct Derived1 : BaseClassCRTP<Derived1> {
};

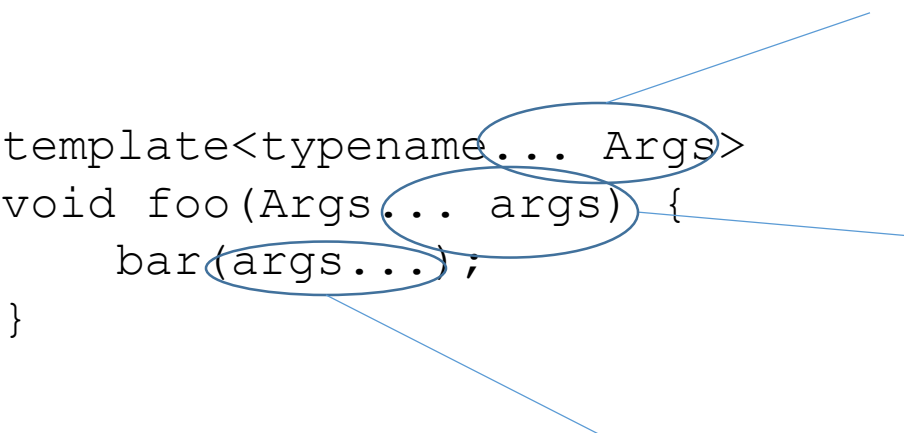
struct Derived2 : BaseClassCRTP<Derived2> {
};
```

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.

- Подсећање, ово је јако користан сајт: cppinsights.io

Шаблони са променљивим бројем параметара

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

A diagram with three ovals: a blue oval around 'Args' in the template parameter list, a purple oval around 'args' in the function parameter list, and a red oval around 'args...' in the function call. A blue line connects 'Args' to 'args'. A purple line connects 'args' to 'args...'. A red line connects 'args...' to the 'args...' in the function call below.

- 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);
}
```

A diagram with three ovals: a blue oval around 'Args' in the first code block, a blue oval around '<int, double, const char*>' in the second code block, and a green oval around 'void foo(int p1, double p2, const char* p3)' in the third code block. A blue line connects 'Args' to the template arguments. A purple line connects 'args' to 'p1', 'p2', and 'p3'. A red oval is around 'bar(p1, p2, p3)' in the third code block. A red line connects 'args...' to 'p1', 'p2', and 'p3'.

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

- Подсећање, ово је јако користан сајт: cprpnsights.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) //...
```

- На овај механизам се ослањају 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);
```


Сетимо се проблема са хетерогеним суповима

Проблем величине

A blue rectangular box with a thin black border.

babe

A red rectangular box with a thin black border.

zabe

A green rectangular box with a thin black border.

krabe

Сетимо се проблема са хетерогеним суповима

Проблем величине

babe

zabe

krabe

krabe be krabe be

std::variant

- std::variant је генерички облик уније.
- Унија омогућава да исто парче меморије посматрамо на различите начине (као да је различитог типа). Свако поље уније представља један начин гледања.
- Варијанта додатно омогућава да се наведе како би требало да се посматра то парче меморије, то јест, ког типа је вредност тренутно у њега постављана. Другим речима, варијанта садржи информацију о типу објекта који се ту тренутно налази.

```
union X {                                std::variant<
    int a1;                               int,
    float a2;                             float> x;
};
```

```
union Y {                                std::variant<
    double a1;                             double,
    float a2;                             float,
    long a3;                               long> y;
};
```

std::variant

- За приступ пољима варијанте, користи се get функција.

```
std::variant<int, float> x;
```

```
std::cout << std::get<1>(x);  
std::get<1>(x) = 5.7f;
```

- Уколико x тренутно садржи алтернативу са индексом 1, биће враћена референца на вредност смештену у x. У супротном, биће бачен овај изузетак: std::bad_variant_access

```
std::variant<int, float> x;
```

```
std::cout << std::get<float>(x); // добавља & или баца изузетак  
std::get<int>(x) = 5.7f; // грешка током превођења
```

std::variant

- Још корисних функција:

```
std::variant<int, float> x;
```

```
x.index(); // индекс тренутне алтернативе
```

```
std::holds_alternative<float>(x);  
    // true уколико је float тренутна алтернатива
```

```
auto p = std::get_if<1>(x);  
if (p)  
    std::cout << *p;
```

```
auto p = std::get_if<float>(x);  
if (p)  
    std::cout << *p;
```

std::variant

- Занимљив пример:

```
struct Shape {  
    void draw() {  
        // постави боју и дебљину  
        drawLines();  
    }  
};
```

```
protected:  
    virtual void drawLines() =0;  
    Color color = Color::RED;  
    int lineThickness = 1;  
};
```

```
struct Rectangle : Shape {  
protected:  
    void drawLines() override { /*...исцртај Rectangle...*/ }  
};
```

```
struct Circle : Shape {  
protected:  
    void drawLines() override { /*...исцртај Circle...*/ }  
};
```

std::variant

- Занимлив пример:

```
struct Canvas {  
    void addShape(Shape& x) { v.push_back(&x); }  
    void drawShapes() {  
        for (auto& it : v)  
            it->draw();  
    }  
private:  
    std::vector<Shape*> v;  
};
```

```
Canvas canv;  
Rectangle r;  
Circle c;
```

```
r.draw();  
c.draw();
```

```
canv.addShape(r); canv.addShape(c);
```

```
canv.drawShapes();
```

std::variant

- Другачији приступ:

```
template<typename T>
struct Shape {
    void draw() {
        // постави боју и дебљину
        static_cast<T*>(this)->drawLines();
    }
protected:
    Color color = Color::RED;
    int lineThickness = 1;
};

struct Rectangle : Shape<Rectangle> {
protected:
    void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape<Circle> {
protected:
    void drawLines() { /*...исцртај Circle...*/ }
};
```


std::variant

- Другачији приступ:

```
template<typename T>
struct Shape {
    void draw() {
        // постави боју и дебљину
        static_cast<T*>(this)->drawLines();
    }
protected:
    Color color = Color::RED;
    int lineThickness = 1;
};

struct Rectangle : Shape<Rectangle> {
    friend Shape<Rectangle>; // постави претка за пријатеља
protected:
    void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape<Circle> { // или уклони protected/private
    void drawLines() { /*...исцртај Circle...*/ }
};
```

std::variant

- Ово ради:

```
struct Canvas {  
    void addShape(Shape& x) { v.push_back(&x); }  
    void drawShapes() {  
        for (auto& it : v)  
            it->draw();  
    }  
private:  
    std::vector<Shape*> v;  
};
```

```
Canvas canv;  
Rectangle r;  
Circle c;
```

```
r.draw();  
c.draw();
```

```
canv.addShape(r); canv.addShape(c);
```

```
canv.drawShapes();
```

std::variant

- A std::variant може помоћи за остало:

```
template<typename... Ts>
struct Canvas {
    using ShapesVar = std::variant<Ts...>;
    void addShape(ShapesVar x) { v.push_back(x); }
    void drawShapes() {
        for (auto& it : v)
            it->draw();
    }
private:
    std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```

std::variant

```
struct V {  
    void operator()(Rectangle x) { x.draw(); }  
    void operator()(Circle x) { x.draw(); }  
};
```

- A std::variant може помоћи за остало:

```
template<typename... Ts>  
struct Canvas {  
    using ShapesVar = std::variant<Ts...>;  
    void addShape(ShapesVar x) { v.push_back(x); }  
    void drawShapes() {  
        for (auto& it : v)  
            std::visit(V{}, it);  
    }  
private:  
    std::vector<ShapesVar> v;  
};
```

```
Canvas<Rectangle, Circle> canv;  
Rectangle r;  
Circle c;
```

```
canv.addShape(r); canv.addShape(c);
```

```
canv.drawShapes();
```

std::variant

```
struct V {  
    template<typename T>  
    void operator()(T x) { x.draw(); }  
};
```

- A std::variant може помоћи за остало:

```
template<typename... Ts>  
struct Canvas {  
    using ShapesVar = std::variant<Ts...>;  
    void addShape(ShapesVar x) { v.push_back(x); }  
    void drawShapes() {  
        for (auto& it : v)  
            std::visit(V{}, it);  
    }  
private:  
    std::vector<ShapesVar> v;  
};
```

```
Canvas<Rectangle, Circle> canv;  
Rectangle r;  
Circle c;
```

```
canv.addShape(r); canv.addShape(c);
```

```
canv.drawShapes();
```

std::variant

- A std::variant може помоћи за остало:

```
template<typename... Ts>
struct Canvas {
    using ShapesVar = std::variant<Ts...>;
    void addShape(ShapesVar x) { v.push_back(x); }
    void drawShapes() {
        for (auto& it : v)
            std::visit([](auto x){ x.draw(); }, it);
    }
private:
    std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```

Торке

- `std::tuple` је, на неки начин, супротно од `std::variant`.
- Оно што је `std::variant` наспрам уније, то је `std::tuple` наспрам структуре.

```
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

- `std::tuple` је, на неки начин, супротно од `std::variant`.
- Оно што је `std::variant` наспрам уније, то је `std::tuple` наспрам структуре.

```
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 начина коришћења торки је за враћање више повратних вредности из функције:

```
std::tuple<int, float, int> foo() {  
    ...  
    return {5, x, y}; // до Це++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.

Торке

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

- 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 { /* ... */ });
```

Торке

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

- 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<sizeof...(Ts)>(x); }

private:
    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 (дакле, 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 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 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<T>::value)
        // if (std::is_integral_v<T>) <- овако може од Це++17
        myAlgSpec(x); // специјална верзија
    else
        // регуларна генеричка верзија
}
```

Својства типова

- Погледајмо још једном овај пример:

```
template<typename T> void myAlg(T x) {  
    if (std::is_integral<T>::value)  
        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  
}
```

Својства типова

- Сличан принцип можемо применити на овај код од раније:

```
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...);
}
```

Својства типова

- 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  
}
```

- Обратите 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<T>::value)  
        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<sizeof...(Ts)>(x); }
private:
    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<sizeof...(Ts)>(x); }
private:
    template<int I>
    int eval(int x) {
        if constexpr (I == 0)
            return 0;
        else {
            auto p = std::get<sizeof...(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>(int x) {
    return 0;
}

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

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

std::tuple

- Стари проблем, решен са std::tuple (овај део је исти као са variant)

```
template<typename T>
struct Shape {
    void draw() {
        // постави боју и дебљину
        static_cast<T*>(this)->drawLines();
    }
protected:
    Color color = Color::RED;
    int lineThickness = 1;
};

struct Rectangle : Shape<Rectangle> {
    friend Shape<Rectangle>; // постави претка за пријатеља
protected:
    void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape<Circle> { // или уклони protected/private
    void drawLines() { /*...исцртај Circle...*/ }
};
```

std::tuple

- Стари проблем, решен са std::tuple:

```
template<typename... Ts>
struct Canvas {
    template<typename T>
    void addShape(T x) { std::get<std::vector<T>>(v).push_back(x); }

    void drawShapes() {

    }
private:
    std::tuple<std::vector<Ts>...> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```

std::tuple

- Стари проблем, решен са std::tuple:

```
template<typename... Ts>
struct Canvas {
    template<typename T>
    void addShape(T x) { std::get<std::vector<T>>(v).push_back(x); }

    void drawShapes() {
        auto f = [](auto x){ for (auto it : x) it.draw(); };
        (f(std::get<std::vector<Ts>>(v)), ...);
    }
private:
    std::tuple<std::vector<Ts>...> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```