

Изузеци и паметни показивачи

noexcept, new (address), unique_ptr, shared_ptr,
weak_ptr

Exception safety

```
void doSomething(Class1& x, Class2& y) {  
    ...  
    throw SomeException();  
    ...  
}
```

```
try {  
    doSomething(o1, o2);  
}  
catch (SomeException x) {  
    ...
```

← У каквом стању су **o1** и **o2** у овој тачки? У ком стању је динамички заузета меморија у функцији?

Exception safety

```
void doSomething(Class1& x, Class2& y) {
    ...
    throw SomeException();
    ...
}
```

```
try {
    doSomething(o1, o2);
}
```

```
catch (SomeException x) {
    ...
```

← У каквом стању су **o1** и **o2** у овој тачки? У ком стању је динамички заузета меморија у функцији?

- Три одговора:
 - Могу бити у било ком стању. То значи да у случају изузетка, функција не гарантује ништа по питању објеката са којима ради, нити заузете меморије.
 - Објекти су у употребљивом стању и нема цурења меморије. Али садржај објеката није загарантован.
 - У стању које је претходило позиву функције. Дакле, функција гарантује да ће у случају неуспеха (изузетка) оставити објекте у првобитном стању (или га у њега вратити).

Exception safety

- Та три одговора се пресликавају у три нивоа гаранција у случају изузетака, које нека функција може да нуди:
 - **Без гаранција (енгл. no exception safety (guarantee))**
 - по питању стања објеката над којима ради и динамички заузете меморије, у случају баченог изузетка.
 - **Основна гаранција**, или гаранција да неће бити цурења (енгл. **basic exception safety (guarantee)**, no-leak guarantee)
 - Гарантује да ће у случају баченог изузетка објекти бити у употребљивом стању и да неће бити цурења меморије.
 - **Јака гаранција (енгл. strong exception safety (guarantee)**, commit-or-rollback semantics)
 - Гарантује да ће у случају баченог изузетка објекти бити у стању у којем су били пре позива функције. Такође неће бити цурења меморије.

Exception safety

- Та три одговора се пресликавају у три нивоа гаранције у случају изузетака, које нека функција може да нуди:
 - **Без гаранција (енгл. no exception safety (guarantee))**
 - по питању стања објеката над којима ради и динамички заузете меморије, у случају баченог изузетка.
 - **Основна гаранција**, или гаранција да неће бити цурења (енгл. **basic exception safety (guarantee)**, no-leak guarantee)
 - Гарантује да ће у случају баченог изузетка објекти бити у употребљивом стању и да неће бити цурења меморије.
 - **Јака гаранција (енгл. strong exception safety (guarantee)**, commit-or-rollback semantics)
 - Гарантује да ће у случају баченог изузетка објекти бити у стању у којем су били пре позива функције. Такође неће бити цурења меморије.
- А намеће се и четврти ниво гаранције:
 - **Гаранција да неће бити изузетака (енгл. no-throw guarantee)**
 - Само име јој каже: изузетака неће бити. (То може да значи да ће функција увек успети да уради то што треба, али такође може да значи да сваки неуспех представља фаталну грешку, на коју се не реагује; слично асерту)

Exception safety

- Ови нивои нису део Це++ стандарда, али јесу концептуални оквир на који се стандард ослања.
- Неки елементи стандардне библиотеке су специфицирани тако да дају неки од ових нивоа гаранције (иако се сами називи тих нивоа, са претходног слајда, не спомињу)
- Најистакнутији пример су неке од операција над контејнерима. Рецимо, `push_back` или `insert`.
- Који ниво гаранција вектор може пружити за ове операције?
- Како то зависи од типа елемената вектора?

Exception safety

- У основи тих операција можемо замислити овакав шаблон

```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = new T[old_capacity * 2];

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new_ptr[i] = old_ptr[i];
    } catch (...) {
        delete[] new_ptr;
        throw;
    }

    delete[] old_ptr;

    return new_ptr;
}
```

- Који ниво гаранција ово нуди?

Exception safety

- Желимо да искористимо мув операцију код типова који је имају

```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = new T[old_capacity * 2];

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new_ptr[i] = std::move(old_ptr[i]);
    } catch (...) {
        delete[] new_ptr;
        throw;
    }

    delete[] old_ptr;

    return new_ptr;
}
```

- Који сада ниво гаранција ово нуди?

Exception safety

- Најпре зависи од тога да ли T уопште има мув операцију
- Ако нема – онда је исто као и са обичним копирањем/доделом (уколико и копирање постоји): јака гаранција, по цени копирања
- Ако има – онда се проблем дели на додатна два случаја
 - Ако мув не баца изузетак, онда нема проблема: јака гаранција, по (јефтинијој) цени мува
 - А ако баца изузетак... онда нема ни јаке ни основне гаранције.
 - Али, ако би тада ипак желели јаку гаранцију, могли бисмо да употребимо копирање/доделу (уколико постоји), и платимо ту гаранцију мањом ефикасношћу.
- Међутим, како да знамо да ли мув операција баца или не баца изузетак и да се прилагодимо томе?

noexcept

- Ова кључна реч има две употребе:

- Да саопшти да нека функција не баца изузетак

```
void foo(Class1 o1) noexcept;  
void foo(Class1 o1) noexcept(true);
```

```
void bar(Class1 o1);  
void bar(Class1 o1) noexcept(false);
```

- Да провери да ли нека функција баца изузетак

```
noexcept(foo); // једнако true  
noexcept(bar); // једнако false
```

```
template<typename T>  
void foo(T func) noexcept( noexcept(T()) ) {  
    ...  
    func();  
    ...  
}
```

noexcept

- Ово је јако ретко важно ван контекста генеричког програмирања.
- Рецимо, у овом случају `noexcept` ће учинити да компајлер уклони сав код за хватање изузетка

```
void foo() noexcept;
```

```
void bar() {  
    try {  
        foo();  
    } catch (...) {  
        //...  
    }  
}
```

- Али овакав код ретко ко пише, јер зашто би писали **try catch** блок ако знамо да **foo()** не баца изузетак.
- Међутим, јасно је да у генеричком коду на овакве ствари можемо наилазити.

Exception safety

- Онај код би сада могао овако да изгледа:

```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = new T[old_capacity * 2];

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new_ptr[i] = std::move_if_noexcept(old_ptr[i]);
    } catch (...) {
        delete[] new_ptr;
        throw;
    }

    delete[] old_ptr;

    return new_ptr;
}
```

- За сада није битно како је **std::move_if_noexcept** имплементирано.

Exception safety

- За сада није битно како је `std::move_if_noexcept` имплементирано, али суштински обезбеђује овакву гаранцију:
 - Функције направљене по том шаблону дају јаку гаранцију уколико је шаблонски параметар `T`, тип који има оператор/конструктор копије или има мув оператор/конструктор.
 - Тј. само у случају да `T` нема копирање, већ само мув који баца изузетак, онда функција настала од тог шаблона неће давати јаку гаранцију.
 - Такође, уколико постоји мув који не баца изузетак, функција настала од шаблона за такво `T` ће употребити мув за пребацивање елемената.

	Has move		Does not have move
	<code>noexcept(true)</code>	<code>noexcept(false)</code>	
Has copy	Strong guarantee move used	Strong guarantee copy used	Strong guarantee copy used
Does not have copy	Strong guarantee move used	No guarantee move used	Compile error

Exception safety

- Узгред, онај код би могао да се још мало побољша, тако што не би радили прво празну конструкцију, па затим копирање.

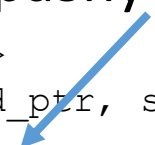
```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = new T[old_capacity * 2];

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new_ptr[i] = std::move_if_noexcept(old_ptr[i]);
    } catch (...) {
        delete[] new_ptr;
        throw;
    }

    delete[] old_ptr;

    return new_ptr;
}
```



Exception safety

- Узгред, онај код би могао да се још мало побољша, тако што не би радили прво празну конструкцију, па затим копирање.

```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = reinterpret_cast<T*>(new std::byte[sizeof(T) * old_capacity * 2]);

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new (new_ptr + i) T(std::move_if_noexcept(old_ptr[i]));
    } catch (...) {
        for (size_t j = 0; j < i; ++j)
            new_ptr[j].~T();
        delete[] reinterpret_cast<std::byte*>(new_ptr);
        throw;
    }

    for (i = 0; i < old_capacity; ++i)
        old_ptr[i].~T();
    delete[] reinterpret_cast<std::byte*>(old_ptr);

    return new_ptr;
}
```

Exception safety

- Узгред, онај код би могао да се још мало побољша, тако што не би радили прво празну конструкцију, па затим копирање.

```
template<typename T>
T* reallocate(T* old_ptr, size_t old_capacity) {

    T* new_ptr = reinterpret_cast<T*>(new std::byte[sizeof(T) * old_capacity * 2]);

    size_t i = 0;
    try {
        for (; i < old_capacity; ++i)
            new (new_ptr + i) T(std::move_if_noexcept(old_ptr[i]));
    } catch (...) {
        for (size_t j = 0; j < i; ++j)
            new_ptr[j].~T();
        delete[] reinterpret_cast<std::byte*>(new_ptr);
        throw;
    }

    for (i = 0; i < old_capacity; ++i)
        old_ptr[i].~T();
    delete[] reinterpret_cast<std::byte*>(old_ptr);

    return new_ptr;
}
```


Који алат употребити?

- Нож
- Јер може све!



Који алат употребити?

- А шта ћемо са овим?
- Сличан је однос показивача и референци.
- Као и још неких елемената језика...



Показивачи

- (Чисте) показиваче можемо користити за све!
- Али треба да их користимо јако ретко...
- Само у јако ограниченим блоковима кода, или специјалним функцијама, где је ефикасност критична, и где је врло мала могућност да настану негативне последице (јер је блок мали, па се целокупна употреба показивача може лако сагледати).
- У скоро свим другим употребама, предност треба давати језичким конструкцијама или елементима стандардне библиотеке који те случајеве боље покривају.

Показивачи

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара

```
int foo(const int* x) {  
    return *x + 1;  
}
```

```
int foo(const int& x) {  
    return x + 1;  
}
```

Показивачи

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе

```
struct vector {  
    int* at(int x) {  
        return &elem[x];  
    }  
}
```

```
*v.at(i) = 5;  
cout << *v.at(i);
```

```
struct vector {  
    int& at(int x) {  
        return elem[x];  
    }  
}
```

```
v.at(i) = 5;  
cout << v.at(i);
```

Показивачи

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе
 - За краћи назив променљиве

```
if (x->clan1->clan2->clan3.get() == 5)
{
    x->clan1->clan2->clan3.set(8);
}
```

```
clan3Type* y = &(x->clan1->clan2->clan3);
if (y->get() == 5)
{
    y->set(8);
}
```

```
clan3Type& y = x->clan1->clan2->clan3;
if (y.get() == 5)
{
    y.set(8);
}
```

Показивачи

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе
 - За краћи назив променљиве
 - За везе између објеката које увек морају постојати

```
struct zavisnaKlasa{  
    refKlasa* ptr; // мора увек показивати  
                  // не неку инстанцу refKlasa типа  
    zavisnaKlasa() : ptr(&globRefKlasa) {}  
    zavisnaKlasa(int count) {} // грешка неће бити откривена  
};
```

```
struct zavisnaKlasa{  
    refKlasa& ptr; // мора увек показивати  
                  // не неку инстанцу refKlasa типа  
    zavisnaKlasa() : ptr(globRefKlasa) {}  
    zavisnaKlasa(int count) {} // грешка у превођењу  
};
```

Показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животни век траје дужи од досега у којем је направљен.

```
мојТип* makeМојТип() {  
    return new мојТип(1, 2, 3);  
}
```

```
void foo() {  
    //...  
    мојТип* p = makeМојТип();  
    // p је сада "власник" објекта  
    //...  
    // је ли било delete? је ли био изузетак пре delete?  
}
```


Показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животни век траје дужи од досега у којем је направљен.

```
std::unique_ptr<mojTip> makeMojTip() {  
    return std::unique_ptr<mojTip>(new mojTip(1, 2, 3));  
}
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = makeMojTip();  
    // p је сада "власник" објекта  
    //...  
    // не треба delete... јер ће бити позвано у деструктору p-а  
}
```

Показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животни век траје дуже од досега у којем је направљен.

```
std::unique_ptr<mojTip> makeMojTip() {  
    return std::make_unique<mojTip>(1, 2, 3);  
}
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = makeMojTip();  
    // p је сада "власник" објекта  
    //...  
    // не треба delete... јер ће бити позвано у деструктору p-а  
}
```

Паментни показивачи

- `Y <memory>`
 - `unique_ptr`

```
struct mojTip {  
    mojTip(int a, int b, int c) : x(a), y(b), z(c) {}  
    int x, y, z;  
};
```

```
void bar(mojTip& x);
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);  
    // auto p = std::make_unique<mojTip>(1, 2, 3);  
    // p је сада "власник" објекта  
    //...  
    cout << p->x;  
    bar(*p);  
    p++; // грешка!  
    p[5]; // грешка!  
}
```

Паметни показивачи

- `Y <memory>`
 - `unique_ptr`

```
struct mojTip {  
    mojTip(int a, int b, int c) : x(a), y(b), z(c) {}  
    int x, y, z;  
};
```

```
void zol(mojTip* x);
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);  
    // p је сада "власник" објекта  
    //...  
    zol(p.get()); // али zol не сме звати delete!  
    std::unique_ptr<mojTip> q{p}; // грешка! може бити само један  
    q = p; // грешка!  
    q = std::move(p);  
    mojTip* r = q.release(); // p више није власник објекта  
}
```

Паметни показивачи

- `У <memory>`
 - `unique_ptr`

```
void bar(int x);
```

```
void foo() {  
    //...  
    std::unique_ptr<int[]> p(new int[7]);  
    // auto p = std::make_unique<int[]>(7);  
  
    //...  
    cout << p->x; // грешка!  
    bar(*p); // грешка!  
    p++; // грешка!  
    p[5];  
}
```

Паметни показивачи

- Основни принцип RAll: „власништво“ над ресурсом (објектом који нема досег) доделити некој променљивој која има досег.

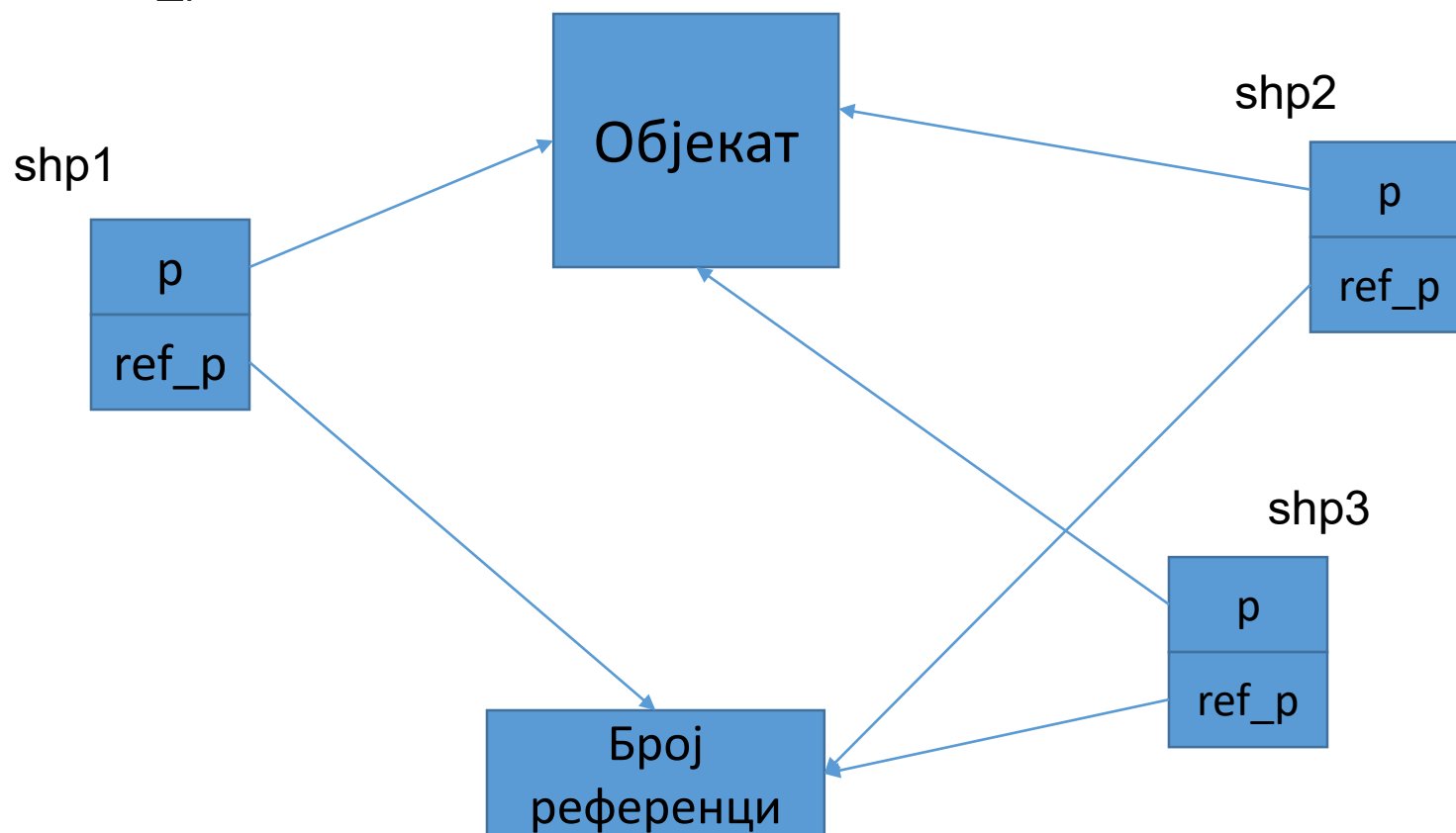
Паметни показивачи

- У <memory>
 - unique_ptr
 - shared_ptr
- Шта ако имамо више показивача који показују на исти објекат?
- Ко је задужен за његово брисање? (Ко је власник?)
- Не постоји једноставан одговор на питање: „Ко показује на овај објекат?“
- У деструктору „дељеног показивача“ проверава се колико још других „дељених показивача“ показује на тај објекат. **delete** се позива тек ако је тај показивач последњи који показује на објекат.
- Та техника се зове бројање референци.
- Колико референци – толико „власника“.

Паметни показивачи

- У <memory>

- unique_ptr
- shared_ptr



Паметни показивачи

- `У <memory>`

- `unique_ptr`
- `shared_ptr`

```
void foo() {  
    //...  
    std::shared_ptr<мојТip> p = std::shared_pointer<мојТip>(  
        new мојТip(1, 2, 3)); // 2 new, за објекат и за референце  
    // број референци: 1 - p.use_count()  
    std::shared_ptr<мојТip> q{p}; // број реф.: 2  
    {  
        std::shared_ptr<мојТip> r = {p}; // број реф.: 3  
    } // број реф.: 2  
    p = nullptr; // број реф.: 1  
} // број реф.: 0 -> зове се delete
```

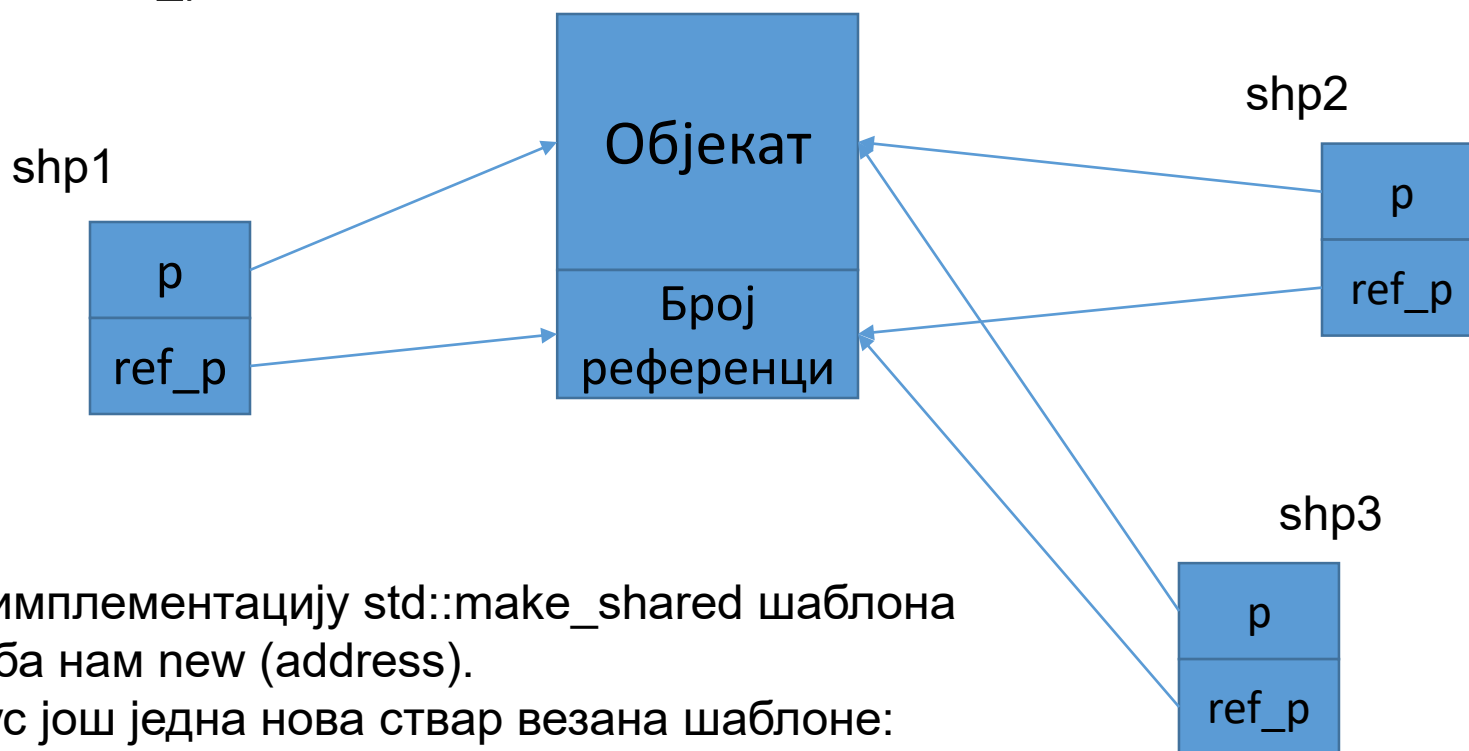
Паметни показивачи

- `У <memory>`
 - `unique_ptr`
 - `shared_ptr`

```
void foo() {  
    //...  
    std::shared_ptr<mojTip> p = std::make_shared<mojTip>(1, 2, 3);  
    // број референци: 1 - p.use_count()  
    std::shared_ptr<mojTip> q{p}; // број реф.: 2  
    {  
        std::shared_ptr<mojTip> r = {p}; // број реф.: 3  
    } // број реф.: 2  
    p = nullptr; // број реф.: 1  
} // број реф.: 0 -> зове се delete
```

Паметни показивачи

- `У <memory>`
 - `unique_ptr`
 - `shared_ptr`



За имплементацију `std::make_shared` шаблона треба нам `new (address)`.

Плус још једна нова ствар везана шаблоне: шаблони са променљивим бројем параметара.

Памятни показивачи

- `memory`

- `unique_ptr`
- `shared_ptr`

```
struct Node {  
    string name;  
    shared_ptr<Node> left;  
    shared_ptr<Node> right;  
    Node(string x) : name(x) { cout<<"Konstruktor " << name << endl; }  
    ~Node() { cout<<"Destruktor " << name << endl; }  
};
```

```
shared_ptr<Node> foo() {  
    shared_ptr<Node> root = make_shared<Node>("koren");  
    root->left = make_shared<Node>("levi");  
    shared_ptr<Node> r = make_shared<Node>("desni");  
    root->right = r;  
    return root;  
}
```

Паметни показивачи

- У <memory>
 - unique_ptr
 - shared_ptr
 - weak_ptr
- Концепт бројања референци има проблема кад постоји кружно референцирање.
- Са „слабим показивачем“ чинимо да један показивач у том кругу буде лабаво повезан са објектом, тј. да има лабаво власништво.
- То значи да слаби показивач није власник, али може привремено да постане (само у тренуцима када се нешто конкретно са објектом кроз тај показивач ради).

Паметни показивачи

- `У <memory>`

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

```
struct mojTip {  
    void do_something();  
};
```

```
shared_ptr<mojTip> makeMojTip();
```

```
void foo() {  
    //...  
    weak_ptr<mojTip> p = makeMojTip();  
    // p није власник објекта, али показује на њега  
    //...  
    p.lock()->do_something();  
    // током извршавања ове линије, p ће бити власник  
}
```

Паметни показивачи

- `U <memory>`

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

```
weak_ptr<mojTip> p = nullptr; // ово не може
```

```
// не може ни ово:
```

```
if (p == nullptr) ...
```

```
// већ мора `вако:
```

```
weak_ptr<mojTip> p; // p празно
```

```
if (p.expired()) ...
```

Compiler explorer godbolt.org

- Коришћењем алата **Compiler explorer** проверите у разним околностима колико је скупа (у смислу додатних инструкција и меморије) употреба **unique_ptr**.

