

Одабрана проширења Це++ језика за боље пројектовање класа

мув семантика, rvalue референце,
„универзалне/прослеђивачке референце“
(савршено прослеђивање)

Мув семантика: Проблем који решавамо

- Нека имамо корисничку класу матрице елемената типа `double`

```
struct Matrix {  
    Matrix() {};  
    ...  
    ~Matrix() { delete[] data; }  
    Matrix(const Matrix& x);  
    Matrix& operator=(const Matrix& x);  
private:  
    double* data = nullptr;  
    int size = 0;  
};
```

Преношење матрице

- Посматрајмо ову функцију:

```
Matrix operator+(Matrix a, Matrix b)
{
    Matrix res;
    // Сабери матрице и резултат смести у res
    return res;
}
```

```
Matrix x, y, z;
```

```
z = x + y; // Колико пута се копирају матрице?
```

Преношење матрице

- Улазне матрице се беспотребно копирају
- Компајлер може оптимизовати код тако што ће уклонити беспотребна копирања, али се на то не можемо ослањати у општем случају.
- Решење за улазне параметре:

```
Matrix operator+(const Matrix& a, const Matrix& b)
```

- Шта са повратном вредношћу?

Преношење матрице

- Једна идеја:

- Враћамо показивач на објекат заузет помоћу **new**

```
Matrix* operator+(const Matrix& a, const Matrix& b);  
Matrix& z = *(x + y);
```

- Проблеми:
 - Ружно на месту позива.
 - Ко зове **delete**?

Преношење матрице

- Друга идеја:

- Враћамо референцу на објект заузет помоћу **new**

```
Matrix& operator+(const Matrix& a, const Matrix& b);  
Matrix& z = x + y;
```

- Проблеми:

- ~~Ружно на месту позива.~~

- Ко зове **delete**?

- Који **delete**? Где је овде показивач?

Преношење матрице

- Трећа идеја:

- Прослеђујемо референцу на већ заузет објекат у који треба да се смести резултат

```
void operator+(const Matrix& a, const Matrix& b, Matrix& res);  
Matrix res = x + y;  
void plus(const Matrix& a, const Matrix& b, Matrix& res);  
plus(x, y, res);
```

- Проблеми:

- Ружно на месту позива.

- А и оператор сабирања прима само два параметра

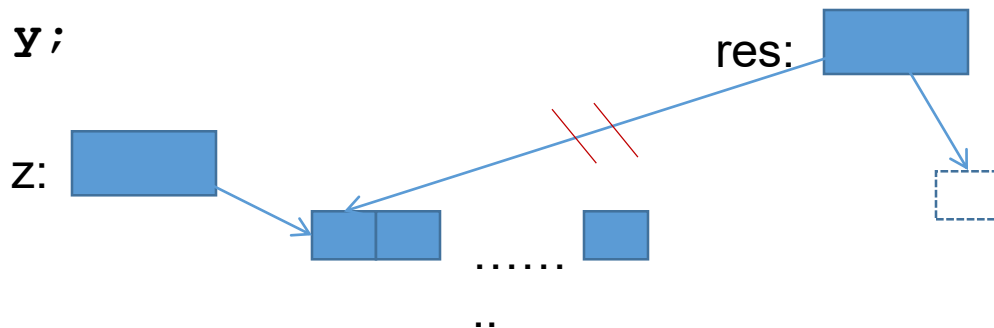
~~• Ко зове delete?~~

Мув семантика

- Нова могућност у Це++11: Мув конструктор

```
class Matrix {  
    // ...  
    Matrix(Matrix&& a)  
    {  
        data = a.data;  
        a.data = nullptr;  
        size = a.size;  
    }  
};
```


`Matrix z = x + y;`



Мув семантика

- А може и мув оператор доделе

```
class Matrix {  
    // ...  
    Matrix& operator=(Matrix&& a)  
    {  
        delete[] data;  
        data = a.data;  
        a.data = nullptr;  
        size = a.size;  
    }  
};  
  
Matrix z;  
z = x + y;
```



Мув семантика

- Сада матрица може овако да изгледа

```
struct Matrix {  
    Matrix() {};  
    ...  
    ~Matrix() { delete[] data; }  
    Matrix(const Matrix& x);  
    Matrix& operator=(const Matrix& x);  
    Matrix(Matrix&& x);  
    Matrix& operator=(Matrix&& x);  
private:  
    double* data = nullptr;  
    int size = 0;  
};
```

Функције чланице (методе) које се аутоматски генеришу -> C++>=11

- Ове две дефиниције су подједнаке:

```
struct Token {
    char kind;
    double value;
};
```

```
struct Token {
    Token() {}
    Token(const Token& x) : kind(x.kind), value(x.value) {}
    Token& operator=(const Token& x) {
        kind = x.kind; value = x.value;
    }
    Token(Token&& x) : kind(x.kind), value(x.value) {}
    Token& operator=(Token&& x) {
    ~Token() {}
    char kind;
    double value;
};
```

Функције чланице (методе) које се аутоматски генеришу -> Це++>=11

- Посебно су интересантне ових пет:
- Конструктор копије (позива се, између осталог, при прослеђивању параметра функцији и враћању повратне вредности)
- Додела копије (представља доделу вредности једног објекта другом објекту истог тог типа)
- Конструктор премештања (мув конструктор)
- Операција премештања (мув оператор доделе)
- Деструктор (када променљива заврши свој животни век)
- Правило петице: „Ако вам не одговара подразумевана верзија бар једне од ових пет функција, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћеш дефинисати или свих пет функција, или ниједну“₁₂

Мув семантика

- Мув конструктор и мув оператор доделе ће бити имплицитно позвани у одређеним случајевима. Суштински, онда када компајлер јасно зна да „десна страна“ у тој наредби завршава сво животни век.
- 1. Повратна вредност.
 - `return` наредба је крај функције и зна се да променљиве аутоматске трајности у локалном досегу престају да живе.

```
Matrix foo() {
    Matrix res;
    ...
    return res; // овде ће бити позван мув конструктор
}
```

- 2. Када је десна страна привремени објекат

```
Matrix a, b, c;
a + b; // резултат функције + је привремени објекат типа Matrix
c = a + b; // биће позван операција премештања да премести
           // привремени објекат у променљиву c
Matrix d = a + b; // биће позван мув конструктор да премести
                 // садржај привременог објекта у нову пром. d
```

&&

- На && у декларацији мув конструктора и мув доделе може да се гледа као само на нешто што прави разлику према обичном конструктору и операцији доделе.
- Али, у питању је, заправо, један шири концепт.
- Да би то разумели, морамо прво разумети ова два појма:
- 1. lvalue (л-вредност – лева вредност)
 - Ствари од којих може да се узме адреса (унарном операцијом &).
 - Обично имају име (променљиве), али не морају. Нпр.:

```
int* p;  
*p; // итекако можемо узети адресу: &(*p), али нема име
```
- 2. rvalue (р-вредност, или д-вредност – десна вредност)
 - Ствари од којих не може да се узме адреса.
 - По правилу немају име.

```
9.0 // литерал је пример д-вредности  
a + b; // резултат функције + је д-вредност
```

&

- Референца може да се односи само на л-вредност.

```
int x;  
int& a = x; // int& a{x}; може
```

```
int& b = 5; // не може
```

```
int foo();  
int& c = foo(); // не може
```

```
void bar(int& a);
```

```
int x;  
bar(x); // може
```

```
bar(5); // не може
```

```
int foo();  
bar(foo()); // не може
```

const &

- Али const референца може да се веже и за д-вредности

```
int x;  
const int& a = x; // int& a{x}; може
```

```
const int& b = 5; // може
```

```
int foo();  
const int& c = foo(); // може
```

```
void bar(const int& a);
```

```
int x;  
bar(x); // може
```

```
bar(5); // може
```

```
int foo();  
bar(foo()); // може
```


&&

- У новом Це++-у је уведена и нова врста референце: rvalue reference

```
int x;
```

```
int&& a = x; // int&& a{x}; не може
```

```
int&& b = 5; // може
```

```
int foo();
```

```
int&& c = foo(); // може
```

```
void bar(int&& a);
```

```
int x;
```

```
bar(x); // не може
```

```
bar(5); // може
```

```
int foo();
```

```
bar(foo()); // може
```

&&

- rvalue reference (или д-референца) има следеће интересантне особине:
- Продужава животни век привремених објеката за које се везује (слично const референци)

```
int foo();  
int&& c = foo();  
std::cout << c;
```

- Има предност при везивању уколико преклапа функцију која прима константну референцу на л-вредност (класична референца, л-референца).

```
void foo(const int& x); // л варијанта  
foo(a); // зове л варијанту  
foo(5); // зове л варијанту  
// али ако имамо ово:  
void foo(const int& x); // л варијанта  
void foo(int&& x); // д варијанта  
foo(a); // зове л варијанту  
foo(5); // зове д варијанту
```

&&

- Међутим, можемо натерати позив функције која прима д-референцу за параметар који је л-вредност.

```
void foo(const int& x); // л варијанта
```

```
void foo(int&& x); // д варијанта
```

```
foo(a); // зове л варијанту
```

```
foo(5); // зове д варијанту
```

```
foo(std::move(a)); // зове д варијанту
```

```
// std::move је суштински static_cast<T&&>(a)
```

- Очекује се да након оваквог позива променљива **a** буде у стању које омогућава њено уништење или доделу нове вредности.
- То са друге стране значи да даље коришћење променљиве **a** треба да обухвати само те операције. У супротном, улазимо у недефинисано стање.
- Један пример употребе std::move је код идиома swap, за неки тип T.

```
void swap(T& a, T& b) {
```

```
    T tmp(a);
```

```
    a = b;
```

```
    b = tmp;
```

```
}
```

&&

- Међутим, можемо натерати позив функције која прима д-референцу за параметар који је л-вредност.

```
void foo(const int& x); // л варијанта
void foo(int&& x); // д варијанта
foo(a); // зове л варијанту
foo(5); // зове д варијанту
foo(std::move(a)); // зове д варијанту
// std::move је суштински static_cast<T&&>(a)
```

- Очекује се да након оваквог позива променљива **a** буде у стању које омогућава њено уништење или доделу нове вредности.
- То са друге стране значи да даље коришћење променљиве **a** треба да обухвати само те операције. У супротном, улазимо у недефинисано стање.
- Један пример употребе std::move је код идиома swap, за неки тип T.

```
void swap(T& a, T& b) { void swap(T& a, T& b) {
    T tmp(a);                T tmp(std::move(a));
    a = b;                    a = std::move(b); // нова вредност у a
    b = tmp;                  b = std::move(tmp); // нова вред. у b
}                             } // уништење tmp
```

Прослеђивање параметара

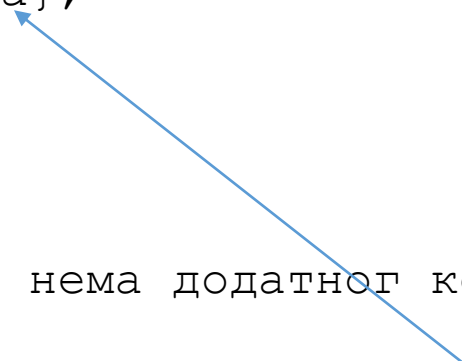
- Постоји још један проблем у чијем решавању учествују д-референце.
- Замислимо тип `T` који има и мув конструктор и мув оператор доделе

```
void foo(T a) {
    ...
    T tmp{a};
    ...
}
```

```
T x;
foo(x); // копирање
T baz();
foo(baz()); // копирање
```

```
void foo(const T& a) {
    ...
    T tmp{a};
    ...
}
```

```
T x;
foo(x); // нема додатног копирања
T baz();
foo(baz()); // али ни мув конструктора
```



- Десна страна нема додатног копирања у првом случају, али се неће позвати мув конструктор у другом случају, иако он постоји за тип `T`. На месту стварања променљиве `tmp` види се само `const T&` и не разликује се први од другог случаја.

Прослеђивање параметара

- Можемо направити две верзије функције, једна која се позива за д-вредност, а друга која се позива за л-вредност

```
void foo(const T& a) { // л верзија
    ...
    T tmp{a};
    ...
}
```

```
void foo(T&& a) { // д верзија
    ...
    T tmp{std::move(a)}; // мора овако, јер сада а траје до
    ...                // краја функције
}
```

```
T x;
foo(x); // нема копирања
T baz();
foo(baz()); // позива се мув конструктор
```

Прослеђивање параметара

- Али шта ако имамо више параметара?

```
void foo(const T& a, const T& b) {  
    ... T tmp1{a}; T tmp2{b}; ...  
}
```

```
void foo(const T& a, T&& b) {  
    ... T tmp1{a}; T tmp2{std::move(b)}; ...  
}
```

```
void foo(T&& a, const T& b) {  
    ... T tmp1{std::move(a)}; T tmp2{b}; ...  
}
```

```
void foo(T&& a, T&& b) {  
    ... T tmp1{std::move(a)}; T tmp2{std::move(b)}; ...  
}
```

Прослеђивање параметара

- У помоћ долазе шаблони и правила за закључивање типова референци.

```
template<typename T>
void foo(T&& a, T&& b) {
    ...
    T tmp1{std::forward<T>(a)};
    T tmp2{std::forward<T>(b)};
    ...
}
```

- Компајлер ће на основу овога генерисати одговарајућу функцију foo, за сваку комбинацију параметара (л-вредност или д-вредност) за коју се јави потреба.
- Ово се зове „савршено прослеђивање параметара“.
- Када && користимо у контексту где се закључују типови (auto или шаблони) онда то називамо „прослеђивачке референце“ (или „универзалне референце“)

Прослеђивање параметара

- Ово је често код конструктора и фабричких функција

```
struct MyType
{
    template<typename T1, typename T2>
    MyType(T1&& a, T2&& b)
        : m_x(std::forward<T1>(a), m_y(std::forward<T2>(b)) {}

private:
    SomeType1 m_x;
    SomeType2 m_y;
};
```

Како то тачно ради?

- Прво је важно да разумемо закључивање параметара функцијског шаблона.

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; //                foo(int), foo(const int&)
5; //                foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T x) {...}
```

```
foo(a);           foo(c);           foo(5);
```

- Шта ће бити T у ова три случаја?

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; // foo(int), foo(const int&)
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T x) {...}
```

```
foo(a);           foo(c);           foo(5);
foo(int x);       foo(int x);       foo(int x);
```

- T је у сва три случаја int.
- Поједностављено: Занемарује се референца и const (и volatile)...

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; //                foo(int), foo(const int&)
5; //                foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T& x) {...}
```

```
foo(a);
```

```
foo(c);
```

```
foo(5);
```

- Шта је T сада?

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; // foo(int), foo(const int&)
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T& x) {...}
```

foo(a);	foo(c);	foo(5);
foo(int&);	foo(const int&);	не преводи се

- Необично је што се други случај преводи, а трећи не проводи.
- Поједностављено: Занемарује се референца и const (и volatile), али се const не занемарује уколико би довео некоректне ситуације.

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)  
const int c = 6; // foo(int), foo(const int&)  
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>  
void foo(const T& x) {...}
```

```
foo(a);           foo(c);           foo(5);
```

- А сада?

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; //                foo(int), foo(const int&)
5; //                foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(const T& x) {...}
```

```
foo(a);           foo(c);           foo(5);
foo(const int&);   foo(const int&);   foo(const int&);
```

- T је опет int у сва три случаја.

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; //                foo(int), foo(const int&)
5; //                foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T&& x) {...}
```

```
foo(a);                foo(c);                foo(5);
```

- Овај случај је посебан.

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; // foo(int), foo(const int&)
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T&& x) {...}
```

foo(a);	foo(c);	foo(5);
?	?	foo(int&&);

- У трећем примеру ствар је јасна и T ће бити int, али шта је са прва два примера?
- Делује да се неће преводити.

Закључивање параметара функцијског шаблона

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; // foo(int), foo(const int&)
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T&& x) {...}
```

foo(a);	foo(c);	foo(5);
?	?	foo(int&&);

- Међутим, одговор је следећи: T ће бити, редом, int&, const int&.
- Али како то формира одговарајући прототип функције foo?
- Сада се у обзир морају узети нова правила за слагање референци у оваквим ситуацијама.
- Тај скуп правила се назива:
 „правила за сажимање референци“
 (енгл. „reference collapsing rules“)

T& &	->	T&
T& &&	->	T&
T&& &	->	T&
T&& &&	->	T&&

Сажимање референци

```
int a; // може бити прихваћено са foo(int), foo(const int&), foo(int&)
const int c = 6; // foo(int), foo(const int&)
5; // foo(int), foo(const int&), foo(int&&)
```

```
template<typename T>
void foo(T&& x) {...}
```

```
foo(a);           foo(c);           foo(5);
foo(int&);       foo(const int&);     foo(int&&);
```

```
foo(int& &&) -> foo(int&)
foo(const int& &&) -> foo(const int&)
```

- Сажимање референци се дешава само у контекстима када се закључују типови. (Другим речима, `foo(int& &&)` не можемо написати директно у коду)

```
T& &    -> T&
T& &&   -> T&
T&& &   -> T&
T&& &&  -> T&&
```

Закључивање типова са auto

- Узгред, све што важи за закључивање типова код шаблона, важи и за закључивање типова при употреби кључне речи auto.

```
int& r = x;  
auto a = r;
```

```
const int& cr = x;  
auto ca = cr;
```

```
int&& rr = foo();  
auto aa = rr;
```

std::forward

- Опет (врло) мало поједностављена представа: `std::forward<T>` се може посматрати као `static_cast<T&&>(x)`

```
template<typename T>
void foo(T&& x) {
    bar(std::forward<T>(x));
}
```

// 1. случај

```
MyType a;
```

```
foo(a);
```

```
// T - int&; a тип x-a је int&
```

```
std::forward<int&>(x);
```

```
// static_cast<int& &&>(x);
```

```
// static_cast<int&>(x);
```

// 2. случај

```
foo(MyType{});
```

```
// T - int; a тип x-a је int&&
```

```
std::forward<int>(x);
```

```
// static_cast<int&&>(x);
```

std::forward

- Али, изгледа као да заправо никаква конверзија није ни потребна, јер је **x** у оба случаја добра врста референце...

```
template<typename T>
void foo(T&& x) {
    bar(std::forward<T>(x));
}
```

```
int&& b = 5;
foo(b);
```

- Међутим, шта овде foo прима? Прима int&, док би за foo(5) примала int&&.
- Другим речима, b јесте rval референца, је се везује само за д-вредности, али она сама бива везана за lval референцу (представља л-вредност).

std::forward

- Али, изгледа као да заправо никаква конверзија није ни потребна, јер је `x` у оба случаја добра врста референце...

```
template<typename T>
void foo(T&& x) {
    bar(std::forward<T>(x));
}
```

```
int&& b = 5;
foo(b);
a = b + 7; // <-!
```

- Међутим, шта овде `foo` прима? Прима `int&`, док би за `foo(5)` примала `int&&`.
- Другим речима, `b` јесте `rval` референца, је се везује само за д-вредности, али она сама бива везана за `lval` референцу (представља л-вредност).
- **Био би проблем да је другачије.**

Свођење rval референце на lval референцу

- rval референца ће се свести на lval уколико има име.
- Уколико нема има, остаће rval.
- Сетимо се примера са једног ранијег слајда:

```
T&& bar();
```

```
void foo(T&& a) { // д верзија
    ...
    T tmp{std::move(a)}; // мора овако, јер сада а траје до
    ...                // краја функције
    T tmp1{bar()};      // али овде је ОК
}
```