

Одабрана проширења Це++ језика за бољи дизајн класа

override, final, explicit, =delete, =default,
делегирање конструктора, кориснички литерали,
мув семантика, „универзалне референце“
(савршено прослеђивање)

- Неке ствари на наредним слајдовима би требало да знамо (бар са Основног Це++ курса), али није лоше да поновимо.

override

- За преклапање виртуалне функције потребно је
 - декларација виртуалне функције
 - да име буде исто
 - да тип функције буде исти

```
struct B {  
    void f1();  
    virtual void f2(char);  
    virtual void f3(char) const;  
    virtual void f4(int);  
};
```

```
struct D : B {  
    void f1();      // не преклапа  
    void f2(int);   // не преклапа  
    void f3(char);  // не преклапа  
    void f4(int);   // преклапа  
};
```

override

```
struct B {  
    virtual void f2(char);  
    virtual void f3(char) const;  
    virtual void f4(int);  
};
```

```
struct D : B {  
    void f2(int); // не преклапа и не пријављује грешку  
    void f3(char); // не преклапа и не пријављује грешку  
    void f4(int); // преклапа  
};
```

override

```
struct B {  
    virtual void f2(char);  
    virtual void f3(char) const;  
    virtual void f4(int);  
};
```

```
struct D : B {  
    void f2(int) override; // пријављује грешку  
    void f3(char) override; // пријављује грешку  
    void f4(int) override; // преклапа и даље  
};
```

final

```
struct B {  
    virtual void f4(int);  
};  
  
struct D : B {  
    void f4(int) override;  
};  
  
struct D1 : D {  
    void f4(int) final;  
}  
  
struct D2 : D1 {  
    void f4(int) override; // грешка  
}
```

final

```
struct B {  
    virtual void f4(int);  
};
```

```
struct D : B {  
    void f4(int) override;  
};
```

```
struct D1 final : D {  
    void f4(int) override;  
}
```

Може и наслеђена класа да буде final.
То значи да нема даљег наслеђивања.

```
struct D2 : D1 { // овде је сада грешка  
    ...  
}
```

Функције чланице (методе) које се аутоматски генеришу

- Конструктор (позива се при стварању променљиве)
- 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)
- Деструктор (када променљива заврши свој животни век)
- Два су разлога зашто се ове функције аутоматски генеришу:
 1. Те операције су толико честе да врло ретко нека од њих не треба.
 2. Да би понашање структура које садрже само податке чланове (атрибуте) остало исто као у Цеу.

Функције чланице (методе) које се аутоматски генеришу

- То значи да су ове две дефиниције 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() {}  
    char kind;  
    double value;  
};
```

Функције чланице (методе) које се аутоматски генеришу

- Посебно су интересантне ове три:
- 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)
- Деструктор (када променљива заврши свој животни век)
- Правило тројке: „Ако вам не одговара подразумевана верзија бар једне од ове три функције, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћеш дефинисати или све три функције, или ниједну“

Функције чланице (методе) које се аутоматски генеришу

- У одређеним случајевима не желимо да имамо неку од ових функција.
- За подразумевани **конструктор** је довољно да се дефинише конструктор који прима неке параметре.

```
struct Token {  
    Token(char ch, double val) : kind(ch), value(val) {}  
    Token(char ch) : kind(ch) {}  
    char kind;  
    double value;  
};
```

// сада ово не може

```
Token x;
```

// већ само ово

```
Token y('8', 9.5); // или ово: Token y('8')
```

=default

- А ако ипак треба и подразумевани празни конструктор, онда може овако:

```
struct Token {  
    Token() = default;  
    Token(char ch, double val) : kind(ch), value(val) {}  
    Token(char ch) : kind(ch) {}  
    char kind;  
    double value;  
};
```

```
// сада може и ово  
Token x;
```

Функције чланице (методе) које се аутоматски генеришу

- У одређеним случајевима не желимо да имамо неку од ових функција.
- **Конструктор копије** и **додела копије** могу да се декларишу као приватни.

```
struct Token {  
    char kind;  
    double value;  
private:  
    Token(const Token& x); // не треба дефиниција  
    Token& operator=(const Token& x); // не треба дефиниција  
};
```

- Али то има неколико мана...

=delete

- А сада може и овако

```
struct Token {  
    Token(const Token& x) = delete;  
    Token& operator=(const Token& x) = delete;  
    char kind;  
    double value;  
};
```

=delete, =default

- Укратко, ове две конструкције нам омогућавају да експлицитно наведемо како желимо спрега наше класе да изгледа.
- На пример:

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

=delete – додатна употреба

- Ова конструкција има још једну употребу.
- Постоје подразумеване конверзије основних типова, нпр.:

```
void foo(long x);
```

```
// може и овако да се зове  
foo(5.0);  
int a;  
foo(a);
```

- Али ако желимо то да забранимо:

```
void foo(long x);  
void foo(int x) = delete;  
void foo(double x) = delete;
```

```
// сада ово не може  
foo(5.0);  
int a;  
foo(a);
```


Делегирање конструктора

- До сада конструктори нису могли да позивају друге конструкторе.
- (Пример из курса Основног Це++-а)

```
struct Rectangle : Shape
{
    Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
        if (h <= 0 || w <= 0) error("Bad arguments");
    }
    Rectangle(Point a, Point b) : x(a), w(b.x-a.x), h(b.y-a.y) {
        if (h <= 0 || w <= 0) error("Bad arguments");
    }
private:
    point x;
    int w;
    int h;
};
```

Делегирање конструктора

- Проблем се делимично могао решавати на следећи начин:

```
struct Rectangle : Shape
{
    Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
        check();
    }
    Rectangle(Point a, Point b) : x(a), w(b.x-a.x), h(b.y-a.y) {
        check();
    }
private:
    void check() {
        if (h <= 0 || w <= 0) error("Bad arguments");
    }
    point x;
    int w;
    int h;
};
```

Делегирање конструктора

- Али, сада конструктори могу да зову друге конструкторе:

```
struct Rectangle : Shape
{
    Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
        if (h <= 0 || w <= 0) error("Bad arguments");
    }
    Rectangle(Point a, Point b) : Rectangle(a, b.x-a.x, b.y-a.y)
    {}
private:
    point x;
    int w;
    int h;
};
```

explicit

- Замислимо да правимо свој тип реалног броја.

```
struct MyReal {  
    MyReal(double x);  
    operator double() const;  
    ...  
};  
  
void foo(MyReal x);  
void bar(double x);  
  
void main() {  
    MyReal a = 5.0;  
    foo(6.0);  
    bar(a);  
    a = a + 7.0;  
    a = a + a; // чак ће и ово да ради  
}
```

explicit

- Али, шта ако је наш тип прецизнији и хоћемо сами аритметику да радимо?

```
struct MyReal {  
    MyReal(double x);  
    operator double() const;  
    friend MyReal operator+(MyReal x, MyReal y);  
    ...  
};  
  
void foo(MyReal x);  
void bar(double x);  
  
void main() {  
    MyReal a = 5.0;  
    foo(6.0);  
    bar(a);  
    a = a + 7.0; // ово се сада неће превести!  
    a = a + a;  // шта ће овде бити?  
}
```

explicit

- Можемо имати директнiju контролу, ако желимо.

```
struct MyReal {  
    explicit MyReal(double x);  
    explicit operator double() const;  
    friend MyReal operator+(MyReal x, MyReal y);  
    ...  
};  
  
void foo(MyReal x);  
void bar(double x);  
  
void main() {  
    MyReal a = 5.0; // не преводи се  
    foo(6.0); // не преводи се  
    bar(a); // не преводи се  
    a = a + 7.0; // не преводи се  
    a = a + a; // али овде је сада ствар врло јасна  
}
```

explicit

- Можемо имати директнiju контролу, ако желимо.

```
struct MyReal {  
    explicit MyReal(double x);  
    explicit operator double() const;  
    friend MyReal operator+(MyReal x, MyReal y);  
    friend MyReal operator+(MyReal x, double y);  
    ...  
};  
  
void foo(MyReal x);  
void bar(double x);  
  
void main() {  
    MyReal a{5.0}; // али овако може  
    foo(MyReal(6.0)); // или foo(static_cast<MyReal>(6.0))  
    bar(double(a)); // или bar(static_cast<double>(a))  
    a = a + 7.0; // сада се преводи  
    a = a + a; // и даље ОК  
}
```

bool као мали изузетак

```
struct MyReal {  
    explicit operator bool() const;  
    ...  
};  
  
void foo(bool x);  
  
void main() {  
    MyReal a;  
    ...  
    foo(a); // ово не може  
    if (a) { // али ово и даље може  
        ...  
    }  
}
```


Литерали

- За уграђене типове постоје литерали (непосредни операнди)
- Код аритметичких литерала, тип је одређен суфиксом и постојањем тачке:

```
2U          -37
3L 31 31    051          // 41
5UL          0x2b        // 43
7LL          0xFFFFFFFFD1 // -47
11ULL
13.0
17.          3.14159      // 3.14159
19.0F        6.02e23      // 6.02 x 10^23
23.F         1.6e-19      // 1.6 x 10^-19
29.0L
31.L
'c' // ово је цео број типа char
```


- А постоје и стринг литерали:

```
"c" // ког је ово типа?
"Djura"
"Pera"s // а ово?
```

Кориснички литерали

- Али, сада можемо и сами правити литерале.

```
struct MyReal {  
    ...  
};  
  
MyReal operator""_mr(long double x);  
  
void main() {  
    std::cout << 9.0_mr;  
}
```



- Број до суфикса ће бити тумачен као ***long double***, и тако ће бити прослеђен функцији ***operator""_mr***.
- Суфикс мора почињати са `_`.
- Ово није добар литерал, у складу са горњим кодом:

9_mr

Кориснички литерали

- На располагању имамо следеће функције:

```
MyReal operator""_mr(long double x);  
    // Хвата ово: 9.0_mr, .5_mr, 1.6e-19_mr
```

```
MyReal operator""_mr(unsigned long long x);  
    // Хвата ово: 9_mr, 0x6_mr, 0b1010_mr, 076_mr
```

```
MyReal operator""_mr(char x);  
    // Хвата ово: 'a'_mr, 'B'_mr, 'v'_mr
```

```
MyReal operator""_mr(const char* x, std::size_t n);  
    // Хвата ово: "a"_mr, "Pera"_mr
```

```
MyReal operator""_mr(const char* x);  
    // Хвата ово: 0xDEADBABA_mr и добија тачно тај стринг  
    // Корисно нпр. када имам бољу прецизност или већи опсег од  
    // long double или long long  
    // 90'223'372'036'854'775'808_mr
```

Кориснички літерали

- Обично нема разлога да не користимо constexpr.

```
struct MyReal {  
    ...  
};  
  
constexpr MyReal operator""_mr(long double x);  
  
void main() {  
    std::cout << 9.0_mr;  
}
```

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 објект 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

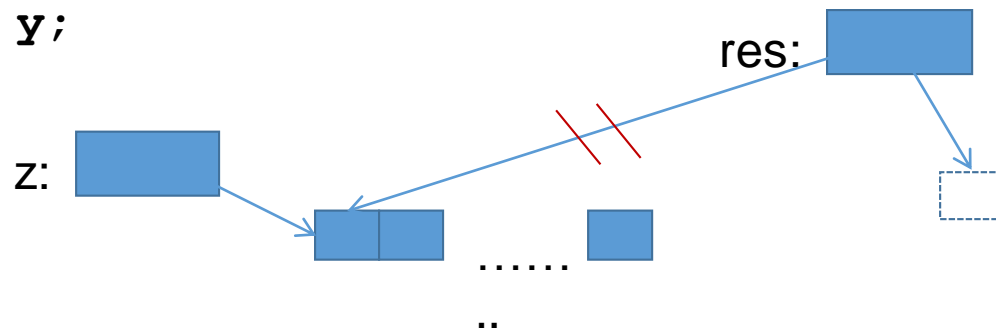
~~• Ко зове delete?~~

Мув семантика

- Нова могућност у Це++11: Мув конструктор

```
class Matrix {  
    // ...  
    Matrix(Matrix&& a)  
    {  
        data = a.data;  
        a.data = nullptr;  
        size = a.size;  
        a.size = 0;  
    }  
};
```


```
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;  
        a.size = 0;  
    }  
};  
  
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 -> Це++>=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) {  
        kind = x.kind; value = x.value;  
    }  
    ~Token() {}  
    char kind;  
    double value;  
};
```

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

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 &).
 - Не морају имати име (тј. нису само променљиве). 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
```

&&

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

```
int x;  
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 C++-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 (али са неким ограничењима)

```
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) { 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

<pre>void foo(T a) { ... T tmp{a}; ... }</pre>	<pre>void foo(const T& a) { ... T tmp{a}; ... }</pre>
<pre>T x; foo(x); // копирање T baz(); foo(baz()); // копирање</pre>	<pre>T x; foo(x); // нема додатног копирања T baz(); foo(baz()); // али ни мув конструктора</pre>

- 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 би сада а трајало 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)
- 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;
};
```