

Одабрана проширења Це++ језика за боље пројектовање класа

override, final, explicit, =delete, =default,
делегирање конструктора, кориснички литерали,
хроно библиотека

- Неке ствари на наредним слајдовима би требало да знамо, 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; // ГРЕШКА
```

```
// 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

- Приметимо како су тела ова два конструктора иста.

```
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 искористе друге конструкторе, тј. да им делегирају део посла:

```
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); //дефинише конверзију double -> MyReal  
    operator double() const; //дефинише конверзију MyReal -> double  
    ...  
};
```

```
void foo(MyReal x);  
void bar(double x);
```

```
void main() {  
    MyReal a = 5.0; //или MyReal a(5.0);, MyReal a{5.0} не може  
    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 директнију 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 директнију 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(Fract(6.0)); // или foo(static_cast<Fract>(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:

```


"с" // 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(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: 0xDEDA'BABA_mr и добија тачно тај стринг  
    // (са све "x" и "'")  
    // Корисно нпр. када имам бољу прецизност или већи опсег од  
    // long double или long long  
    // 90'223'372'036'854'775'808_mr
```

Кориснички литерали

- Обично нема разлога да не користимо constexpr.

```
struct MyReal {  
    ...  
};
```

```
constexpr MyReal operator""_mr(long double x);
```

```
void main() {  
    constexpr auto x = 9.0_mr;  
    std::cout << x;  
}
```

Библиотека за рад са временом

- Ова библиотека се ослања на три концепта:

- Интервал
- Сат (часовник)
- Тренутак

```
#include <chrono>
using namespace std::chrono;

time_point<std::chrono::steady_clock> start, end;

start = steady_clock::now();
foo();
end = steady_clock::now();

duration<double> elapsed_seconds = end - start;

std::cout << "elapsed time: " << elapsed_seconds.count() << "s";
```

Библиотека за рад са временом

- Интервал

```
template<
    class Rep, // тип података у којем ће се чувати број откуцаја
    class Period = std::ratio<1> // број секунди по откуцају
> class duration;

nanoseconds    duration</* бап 64 бита */, std::nano>
microseconds   duration</* бап 55 бита */, std::micro>
milliseconds   duration</* бап 45 бита */, std::milli>
seconds        duration</* бап 35 бита */>
minutes        duration</* бап 29 бита */, std::ratio<60>>
hours          duration</* бап 23 бита */, std::ratio<3600>>
```

Библиотека за рад са временом

- Сат (часовник)

```
std::chrono::system_clock
```

```
std::chrono::steady_clock
```

```
std::chrono::high_resolution_clock // откуцај је најмањег трајања  
                                     // на датом хардверу, обично  
                                     // се своди на један од горња  
                                     // два
```

- Главна метода now

```
static std::chrono::time_point<std::chrono::врста_сата> now()
```

Библиотека за рад са временом

- Тренутак

```
template<
    class Clock,
    class Duration = typename Clock::duration
> class time_point;
```


Библиотека за рад са временом

- Литерали

```
operator""h  
operator""min  
operator""s  
operator""ms  
operator""us  
operator""ns
```

```
???? vreme = 5h + 13min + 5s;
```

Библиотека за рад са временом

- Литерали

```
operator""h  
operator""min  
operator""s  
operator""ms  
operator""us  
operator""ns
```

```
auto vreme = 5h + 13min + 5s;
```

std::array

- Као мешавина између std::vector-а и Цеоовског низа. Има спрегу сличну вектору, али меморија је статички заузета и не може расти (ни смањивати се).

```
template<typename T, std::size_t N> struct array;
```

```
std::array<int, 3> a1;
```

```
std::array<int, 3> a2;
```

```
a2 = a2;
```

```
std::array<int, 3> a3(a2);
```

```
std::sort(a1.begin(), a2.end());
```

Типски алијас - using

- Кључна реч **using** сада имам додатну улогу.
- Сада може да служи да се уведе ново име за тип. Ради исто као typedef али има другачију синтаксу, која се зове алијас декларација (енгл. alias-declaration).
- Разлог за увођење те нове синтаксе није одмах јасан, али добија смисао у контексту генеричког програмирања (што ћемо видети касније).
- За сада ево само једноставне илустрације синтаксе:

```
typedef int MyIntTN;  
using MyIntAD = int;
```

```
typedef int MyArrayTN[10];  
using MyArrayAD = int[10];
```

```
typedef int (*MyFuncTN)(int);  
using MyFuncAD = int(*) (int);
```