# Přetěžování operátorů, dynamika objektů 2

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze



Programování v C++, A7B36PJC
03/2016, Lekce 6

https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

# Přetěžování operátorů

- Jak jsme si říkali, třídy definují nové datové typy
- Primitivní datové typy mají i operátory
  - Dva `int`y můžeme sčítat, odečítat, dělit ...
- C++ umožňuje přetížit operátory pro třídy, takže se chovají jako primitivní datové typy

- Cílem přetěžování je, aby bylo intuitivní
  - Matice se dají sčítat, odčítat, násobit... ale bitové operace nedávájí smysl
  - Neomezené inty by ale bitové operace podporovat měly
  - Cesty se dají spojovat **/** (separátor cest), ale nemají operátor násobení, sčítání, etc

# Přetížení operátoru +

- Pokud použijeme operaci **c1+c2**, kde **c1** a **c2** jsou objekty našeho nového typu `Complex`, kompilátor nahlásí chybu. Není totiž definováno, jak má operace proběhnout.

```
#include <iostream>

class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    double getReal() const { return real; }
    double getImag() const { return imag; }
};

int main() {
    Complex c1(2.5, -2);
    Complex c2(-0.5, 3);
    Complex c3 = c1 + c2; // Chyba. + pro Complex neexistuje
}
```

# Přetížení operátoru +

Operátor přetížený pomocí metody

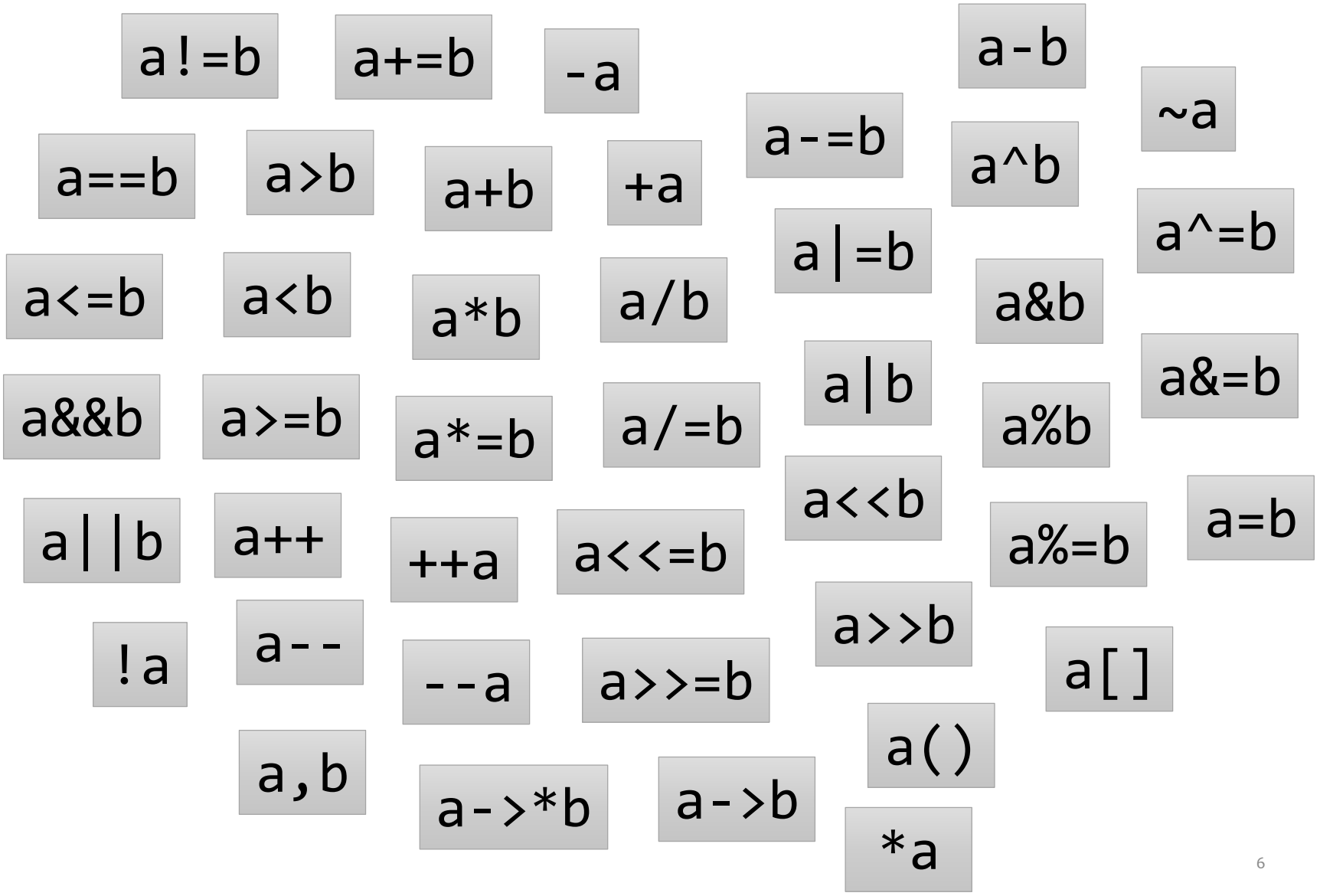
```
#include <iostream>

class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    double getReal() const { return real; }
    double getImag() const { return imag; }
    Complex operator+(const Complex& rhs) const {
        return Complex(real + rhs.real, imag + rhs.imag);
    }
};

int main() {
    Complex c1(2.5, -2);
    Complex c2(-0.5, 3);
    Complex c3 = c1 + c2;
    std::cout << c3.getReal() << " + " << c3.getImag() <<
"i\n"; }
```

# Přetížení operátoru +=

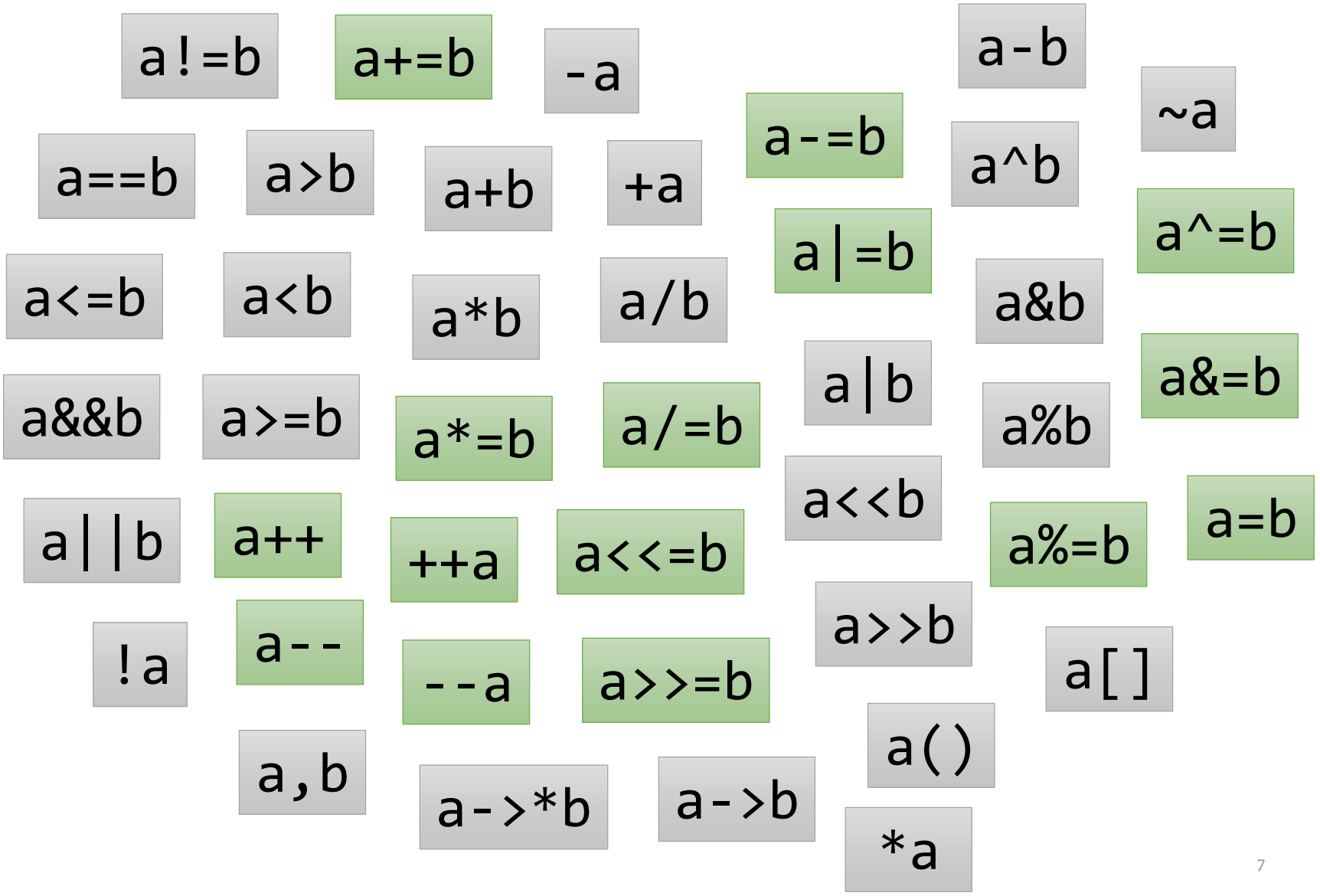
- Podobně jako operátor **+** můžeme přetížit i operátor **+=**.
- Operátor **a+=b** nazýváme modifikující, protože by měl změnit svůj levý operand a; oproti tomu **a+b** je příklad nemodifikujícího operátoru.
- Chování operátorů **+** a **+=** definujeme zvlášť; když zadefinujeme **+**, neznamená to, že bude fungovat **+=**.

```
class Complex {
    double real, imag;
public:
    ...
    Complex& operator+=(const Complex& rhs) {
        real += rhs.real;
        imag += rhs.imag;
        return *this;
    }
};
```

# přetížitelné operátory



modifikující operátory



# Co při přetěžování nedělat

- Systém přetěžování operátorů nám dává jisté svobody, které je lepší nevyužít:
  - Operátor **a+=b** může dělat něco úplně jiného, než **a=a+b**.
  - Nemodifikující operátory mohou modifikovat levý operand, pravý operand, nebo třeba oba.
- Nezapomínejte, že naším cílem je, aby používání našich tříd bylo intuitivní!

```
class Complex {
    ...
    Complex operator+(const Complex& rhs) const {
        std::cout << "Tady je Krakonosovo!!\n";
        std::terminate();
    }
};
```

# Přetížení operátoru pomocí funkce

- Už jsme si ukázali, že přetížení operátoru lze umístit přímo do třídy, jako metodu.
- Přetížení také můžeme definovat mimo třídu, jako funkci.
- Tato funkce ale nemá přístup k soukromým datům třídy, a tak musí používat její veřejné rozhraní.

Operátor přetížený pomocí funkce

```
class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    double getReal() const { return real; }
    double getImag() const { return imag; }
};

Complex operator+(const Complex& lhs, const Complex& rhs) {
    return Complex(lhs.getReal() + rhs.getReal(),
                   lhs.getImag() + rhs.getImag());
}
```

# Přetížení operátoru pomocí funkce

- Funkci umožníme přistupovat k soukromým datům třídy, pokud ji v deklaraci třídy označíme za spřátelenou (`friend`).
- To se při přetěžování operátorů často hodí.

Operátor přetížený pomocí **spřátelené** funkce

```
class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    double getReal() const { return real; }
    double getImag() const { return imag; }

    friend Complex operator+(const Complex& lhs, const Complex& rhs);
};

Complex operator+(const Complex& lhs, const Complex& rhs) {
    return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}
```

# Přetížení operátoru pomocí funkce

- Nabízí se otázka, proč vůbec přetěžovat pomocí funkcí, ne pouze pomocí metod. V některých situacích ale musíme, třeba v případě, že je na levé straně nějaký cizí objekt, který nemůžeme měnit.

Přetížit operátor << lze pouze pomocí funkce

```
class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    double getReal() const { return real; }
    double getImag() const { return imag; }
};

std::ostream& operator<<(std::ostream& out, const Complex& c) {
    return out << c.getReal() << " + " << c.getImag() << "i";
}

int main() {
    Complex c(-0.5, 3);
    std::cout << c << "\n";
}
```

# Přetížení operátoru []

- Toto je třída `zasobnik` z minulé přednášky.

```
class zasobnik {
public:
    zasobnik(int max_prvku);
    zas_typ vezmi();
    void vloz(zas_typ prvek);
    bool je_prazdny() const;
private:
    int max_velikost;
    int aktualni_pozice;
    std::unique_ptr<zas_typ[]> prvky;
};
```

- Chtěli bychom mít operaci [ ], která poskytne prvek zásobníku:

```
zasobnik z(10); z.vloz(11); z.vloz(22); z.vloz(33);
std::cout << z[0] << "\n"; // 11
z[1] = 99;                 // zásobník teď obsahuje 11 99 33
std::cout << z[1] << "\n"; // 99
```

# Přetížení operátoru []

- Chování by se mělo lišit podle toho, zda je náš objekt konstantní:
  - Pokud a není konstantní, a[b] vrací referenci na b-tý prvek.
  - Pokud je a konstantní, a[b] vrací konstantní referenci na b-tý prvek, nebo hodnotu b-tého prvku.

```
class zasobnik {
public:
    ...
    zas_typ& operator[](int i) {              // umožní z[1] = 99,
        return prvky[i];                      // když z je zasobnik
    }

    const zas_typ& operator[](int i) const { // zabrání z[1] = 99,
        return prvky[i];                      // z je const zasobnik
    }
private:
    ...
    std::unique_ptr<zas_typ[]> prvky;
};
```

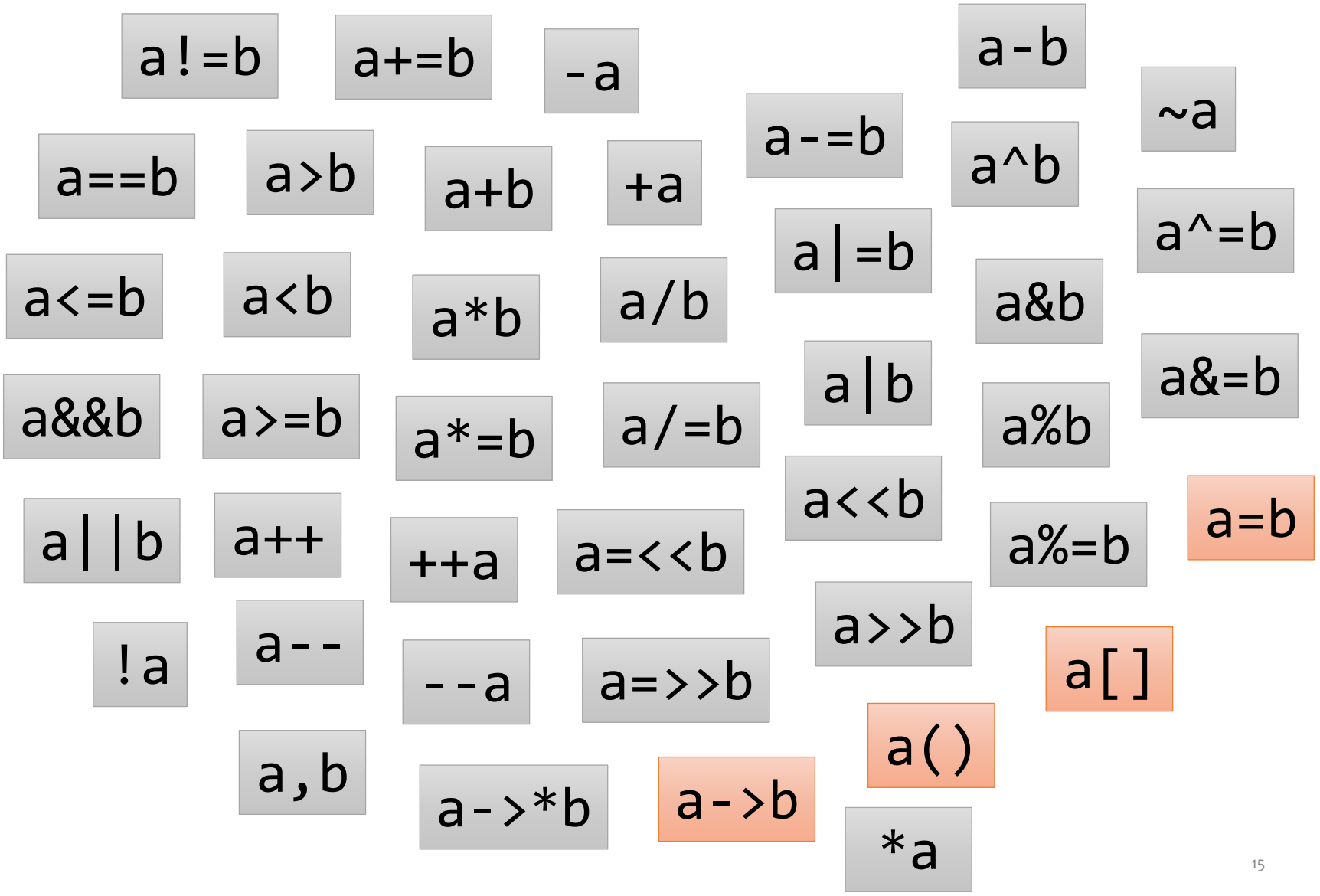
# Přetížení operátoru []

- Takto můžeme vracet hodnotu v případě konstantního objektu:

```
class zasobnik {
public:
    ...
    zas_typ& operator[](int i) {       // umožní z[1] = 99,
        return prvky[i];               // když z je zasobnik
    }
    zas_typ operator[](int i) const { // také zabrání z[1] = 99,
        return prvky[i];               // když z je const zasobnik
    }
private:
    ...
    std::unique_ptr<zas_typ[]> prvky;
};
```

- [ ] je příklad operátoru, který musí být přetížen metodou.

- přetížitelné pouze pomocí metody



# Přetížení operátoru ++

- Mějme třídu Datum.

```
class Datum {
    int den, mesic, rok;
public:
    Datum(int d, int m, int r) : den(d), mesic(m), rok(r) {}
    friend std::ostream& operator<<(std::ostream& out, const Datum& d);
};

std::ostream& operator<<(std::ostream& out, const Datum& d) {
    return out << d.den << ". " << d.mesic << ". " << d.rok;
}
```

- Přidejme této třídě operaci ++, která posune datum o jeden den.

```
int main() {
    Datum d(31, 12, 2015);
    std::cout << d << "\n"; // 31. 12. 2015
    d++;
    std::cout << d << "\n"; // 1. 1. 2016
}
```

# Přetížení operátoru ++ (prefix)

- Operátor ++a má obecně dva úkoly:
  - Změnit objekt a (jedná se o modifikující operátor).
  - Vrátit referenci na a.

```
class Datum {
    int den, mesic, rok;
public:
    ...
    Datum& operator++() {
        ++den;
        if (den > posledniDen(mesic, rok)) {
            den = 1; ++mesic;
            if (mesic > 12) {
                mesic = 1; ++rok;
            }
        }
        return *this; // vrať referenci na tento objekt
    }
};
```

Funkce posledniDen() poskytne pro daný měsíc a rok číslo posledního dne v měsíci.

# Přetížení operátoru ++ (prefix)

Pro úplnost

```
int posledniDen(int mesic, int rok) {
    switch (mesic) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        return 31;
    case 4: case 6: case 9: case 11:
        return 30;
    case 2:
        if (rok % 400 == 0) return 29;
        if (rok % 100 == 0) return 28;
        if (rok %   4 == 0) return 29;
        return 28;
    default:
        throw std::runtime_error("chyba v posledniDen()");
    }
}
```

- Pro objekty typu `Datum` je nyní zavedena operace ++, jenže pouze v prefixové variantě ++a. Operátor a++ musíme přetížit zvlášť.

```
std::cout << d << "\n"; // 31. 12. 2015
++d;
std::cout << d << "\n"; // 1. 1. 2016
```

# Přetížení operátoru ++ (postfix)

- Operátor a++ má rovněž dva úkoly:
  - Změnit objekt a (jedná se také o modifikující operátor).
  - **Vrátit kopii a ve stavu před změnou.**
- a++ se dá vytvořit pomocí ++a. Nejdříve si uložíme kopii našeho objektu, pak provedeme ++a, pak kopii vrátíme.
- Deklarace a++ se od ++a odlišuje nadbytečným parametrem typu `int`.

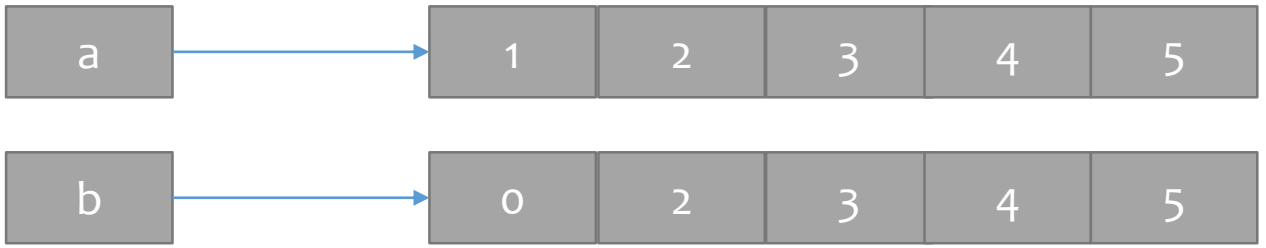
```
class Datum {
    int den, mesic, rok;
public:
    ...
    Datum operator++(int) { // nadbytečný int znamená a++
        Datum kopie(*this); // kopie našeho objektu
        ++(*this);          // proveď ++a
        return kopie;       // vrať kopii
    }
};
```

# Kopie

- Při implementaci postfixové inkrementace jsme potřebovali, aby se třída `Datum` uměla zkopírovat.
- Nemuseli jsme ale říkat jak, protože Datum je jednoduchá třída, pro kterou to za nás zvládne zařídit kompilátor. (Více o tomto později.)
- U složitějších tříd (například `std::vector`) už ale musí programátor specifikovat, jak se taková kopie vytváří.
- V C++ chceme vytvářet tzv. hluboké (nesdílejí data s originálem) kopie.

```
#include <vector>

int main() {
    std::vector<int> a = { 1, 2, 3, 4, 5 };
    std::vector<int> b = a;
    b[0] = 0; //b = {0, 2, 3, 4, 5}, a = {1, 2, 3, 4, 5}
}
```

b je kopie a
a a b jsou nezávislé

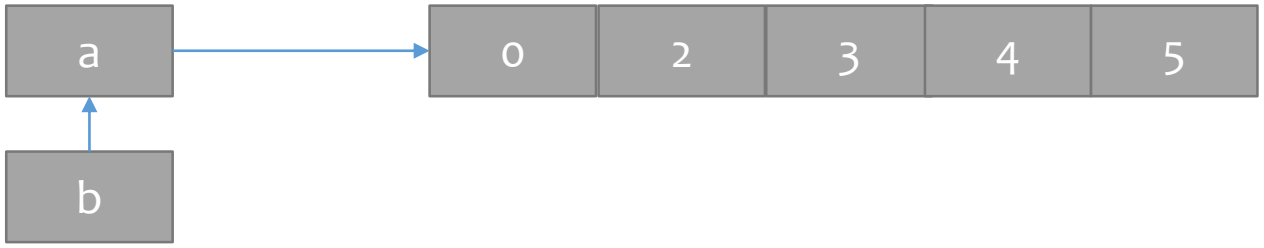


# Kopie (2)

- Někdy je hluboká kopie nežádoucí.
- Často chceme jenom jinak pojmenovat nějaká data, nebo je předat funkci pomocí parametru.
- V tom případě použijeme reference nebo ukazetele.

```
#include <vector>

int main() {
    std::vector<int> a = { 1, 2, 3, 4, 5 };
    std::vector<int>& b = a;
    b[0] = 0; //b = {0, 2, 3, 4, 5}, a = {0, 2, 3, 4, 5}
}
```

b je reference na a
a a b jsou identické



# Kopírující operace

- Existují dvě tzv. kopírující operace: kopírující konstruktor a kopírující přiřazení.
- Konstruktor, jak jméno napovídá, je volán při kopírování do nového objektu.
- Přiřazení je voláno, pokud chceme kopírovat do už existujícího objektu.

```
std::vector<int> a = { 1, 2, 3, 4, 5 };
std::vector<int> b(a); // kopírující konstruktor (b se vytváří)
std::vector<int> c = { 10, 20, 30, 40, 50 };
b = c;                 // kopírující přiřazení (b už existuje)
```

# Kopírující konstruktor

- Kopírující konstruktor je zodpovědný za vytvoření objektu se stavem rovným stavu kopírovaného objektu.
- Má tvar `T(const T& rhs)`
- Pro std::vector musí:
  - Nastavit svou velikost na velikost rhs
  - Alokovat dostatek paměti pro uložení všech prvků z `rhs`
    - Nastavit svou kapacitu dle množství alokované paměti
  - Překopírovat jednotlivé prvky z `rhs`

# Kopírující konstruktor – příklad

```
using zas_typ = int;

class zasobnik {
public:
    zasobnik(const zasobnik& rhs); // Kopírující konstruktor
private:
    int max_velikost;
    int aktualni_pozice;
    std::unique_ptr<zas_typ[]> prvky;
};
```

```
zasobnik::zasobnik(const zasobnik& rhs):max_velikost(rhs.max_velikost),
                        aktualni_pozice(rhs.aktualni_pozice),
                        prvky(std::make_unique<zas_typ[]>(max_velikost) {
    for (int i = 0; i < aktualni_pozice; ++i) {
        prvky[i] = rhs.prvky[i];
    }
}
```

```
int main() {
    zasobnik z(10);
    zasobnik z2(z); // z2 je vytvořeno jako kopie z
}
```

# Kopírující přiřazení

- Kopírující přiřazení musí zohlednit, že cílový objekt již má nějaký stav.
- Nejdříve dojde k úklidu, poté následuje kopírování dat jako v kopírujícím konstruktoru.
- Má tvar `T& operator=(const T& rhs)`
- Pro std::vector musí:
  - Zavolat destruktor všech svých prvků
  - Nastavit svou velikost na `0`
  - Porovnat svou kapacitu s počtem prvků v `rhs`
    - Navýšit svou kapacitu, pokud nedostačuje na uložení všech prvků
  - Nastavit svou velikost na velikost `rhs`
  - Překopírovat jednotlivé prvky z `rhs`

# Kopírující přiřazení – příklad

```
class zasobnik {
public:
    zasobnik& operator=(const zasobnik& rhs); // Kopírující přiřazení
private:
    ...
};
```

```
zasobnik& zasobnik::operator=(const zasobnik & rhs) {
    max_velikost = rhs.max_velikost;
    aktualni_pozice = rhs.aktualni_pozice;
    prvky = std::make_unique<zas_typ[]>(rhs.max_velikost);
    for (int i = 0; i < max_velikost; ++i) {
        prvky[i] = rhs.prvky[i];
    }

    return *this;
}
```

```
int main() {
    zasobnik z(10);
    zasobnik z2(z);  // z2 je vytvořeno jako kopie z
    z = z2;          // z je přiřazené z2
}
```

# Konstruktor vs přiřazení

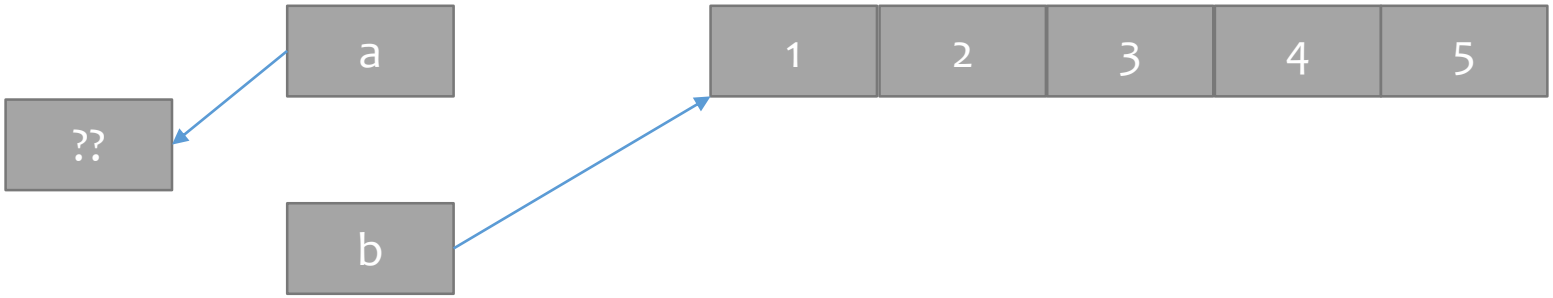
- Jak bylo řečeno, konstruktor se provádí pokud vzniká nový objekt a přiřazení se provádí pokud chceme již existující objekt změnit na kopii jiného.
- Nestačí se ale orientovat pouze dle použití „=“, protože se tak dá volat i konstruktor.
  - Nejjednodušší je se podívat, zda je na levé straně přiřazení proměnná zároveň deklarována.

```
std::vector<int> a = { 1, 2, 3, 4, 5 };
std::vector<int> b(a);  // kopírující konstruktor (b se vytváří)
std::vector<int> c = { 10, 20, 30, 40, 50 };
b = c;                  // kopírující přiřazení (b už existuje)
std::vector<int> d = a; // kopírující !! konstruktor !! (d se vytváří)
```

# Přesun

- Ne vždy je potřeba vytvářet kopii objektu.
  - Můžeme vědět, že originál již nikdy nepoužijeme
  - Nebo originál přestane existovat
- Zároveň může být vytváření kopií drahé a tudíž nežádoucí.
  - Kopírování vektoru s miliony prvků musí okopírovat i všechny prvky.
- Za takových okolností se použijí přesunující operace.
- Pozor, přesun je vždy destruktivní.

```
#include <vector>

int main() {
    std::vector<int> a = { 1, 2, 3, 4, 5 };
    std::vector<int> b(std::move(a));
}
```



# Přesunující konstruktor

- Přesunující konstruktor přebírá stav objektu, ze kterého se konstruuje.
- Při tom má možnost původní objekt znehodnotit.
- Má tvar `T(T&& rhs)`.
- Pro std::vector:
  - Zkopíruje kapacitu, velikost a ukazatel na data z `rhs`.
  - V `rhs` nastaví ukazatel na data na `nullptr`.

# Přesunující konstruktor – příklad

```
using zas_typ = int;

class zasobnik {
public:
    zasobnik(zasobnik&& rhs); // Přesunující konstruktor
private:
    int max_velikost;
    int aktualni_pozice;
    std::unique_ptr<zas_typ[]> prvky;
};
```

```
zasobnik::zasobnik(zasobnik&& rhs):max_velikost{ rhs.max_velikost },
        aktualni_pozice{ rhs.aktualni_pozice },
        prvky{ std::move(rhs.prvky) } {}
```

```
int main() {
    zasobnik z(10);
    zasobnik z2(std::move(z)); // z2 bylo vytvořeno přesunem ze z
                               // z už nesmí být používáno
}
```

# Přesunující přiřazení

- Stejně jako kopírující přiřazení musí přesunující přiřazení zohlednit, že objekt již má nějaký stav.
- Nejdříve dojde k úklidu, poté probíhá přesun jako v přesunujícím konstruktoru.
- Má tvar `T& operator=(T&& rhs)`
- Pro std::vector:
  - Zavolá destruktor všech svých prvků
  - Dealokuje paměť
  - Zkopíruje kapacitu, velikost a ukazatel na data z `rhs`.
  - V `rhs` nastaví ukazatel na data na `nullptr`.

# Přesunující přiřazení – příklad

```
using zas_typ = int;

class zasobnik {
public:
    zasobnik& operator=(zasobnik&& rhs);
private:
    int max_velikost;
    int aktualni_pozice;
    std::unique_ptr<zas_typ[]> prvky;
};
```

```
zasobnik& zasobnik::operator=(zasobnik&& rhs) {
    max_velikost = rhs.max_velikost;
    aktualni_pozice = rhs.aktualni_pozice;
    prvky = std::move(rhs.prvky);
    return *this;
}
```

```
int main() {
    zasobnik z(10);
    zasobnik z2(z);    // z2 je kopie z
    z = std::move(z2); // z byl přiřazen přesun z2
}                      // z2 se už nesmí používat
```

# Kdy tedy dochází k přesunům?

- Při návratu proměnné z funkce

```
std::string foo() {
    std::string temp;
    std::cin >> temp;
    return temp; // temp je ven přesunuto
}
```

- Při předávání dočasných proměnných

```
std::string foo() {return "abcd";} // vytvoří nový řetězec
std::string s = foo(); // dočasný string vrácený z foo je do s
přesunut
```

- Při explicitním požadavku programátora

```
std::string s1;
std::string s2 = std::move(s1); // s1 je do s2 přesunuto, protože
si o to programátor řekl
```

# Speciální funkce

- Existuje 6 speciálních funkcí, které doplní kompilátor, pokud je programátor nezadefinuje: základní konstruktor, kopírující operace, přesunující operace, destruktor.
- Základní konstruktor je doplněn, pokud programátor nezadefinuje žádné konstruktory.
- Kopírující operace jsou doplněny, když chybějí a není definována žádná přesunující operace.
- Přesunující operace jsou doplněny, když chybějí a není definována žádná přesunující operace, kopírující operace, ani destruktor.
- Destruktor je doplněn vždy, když chybí.

# Speciální funkce (2)

- Generování speciálních funkcí je možné explicitně vyžádat, nebo zakázat.
- Například můžeme chtít automaticky generovat přesunující operace, i když jsme zadefinovali kopírující operace.
  - Třeba když chceme v kopírujících operacích sbírat statistiky a přesunující nechat beze změny.
- Nebo můžeme chtít zakázat vytváření kopií, ale nechat si možnost typ přesouvat.
  - `std::unique_ptr` je příklad třídy, která má kopírující operace zakázány, k dispozici jsou pouze přesuny.

# Speciální funkce – příklad

```
class osoba {
public:
    osoba& operator=(const osoba& rhs) {
        jmeno = rhs.jmeno; prijmeni = rhs.prijmeni;
        std::cout << "Prirazuju osobu " << jmeno
                  << ' ' << prijmeni << std::endl;
    }
    osoba(const osoba& rhs):jmeno{ rhs.jmeno },
                            prijmeni{rhs.prijmeni} {
        std::cout << "Kopiruju osobu " << jmeno
                  << ' ' << prijmeni << std::endl;
    }

    // Necháme kompilátor vygenerovat přesunující operace
    osoba(osoba&&) = default;
    osoba& operator=(osoba&) = default;
private:
    std::string jmeno, prijmeni;
};
```

# Speciální funkce – příklad (2)

```
class foo {
public:
    foo(const foo&) = delete;
    foo& operator=(const foo&) = delete;
    foo(foo&&) = default;
    foo& operator=(foo&&) = default;
private:
    std::vector<int> bar;
};
```

# Kompilátorem definované funkce

- Kompilátor generuje funkce dle jednoduchého pravidla: pro každý datový prvek třídy se zavolá její speciální funkce.
- Například:
  - Základní konstruktor je kompilátorem vygenerován jako volání základního konstruktoru pro každý prvek objektu.
  - Kopírující konstruktor je vygenerován jako volání kopírujícího konstruktoru pro každý prvek objektu.
  - atd.

```
class osoba {
public:
    osoba(osoba&&) = default;
    ...
private:
    std::string jmeno, prijmeni;
};
```

```
class osoba {
public:
    osoba(osoba&& rhs):
     jmeno(std::move(rhs.jmeno)),
     prijmeni(std::move(rhs.prijmeni))
     {}
    ...
private:
    std::string jmeno, prijmeni;
};
```

# RAII a speciální funkce

- Jak víme z minulých přednášek, některé objekty při své konstrukci nabývají zodpovědnost za prostředky.
- Takovým objektům definujeme destruktor.
- Kompilátor za nás doplní kopírující a přesunující operace.
- Jenže operace, které kompilátor vygeneroval, neví nic o tom, jak s prostředky nakládat!
- Máme dvě možnosti:
  - Zadefinovat kopírující a přesunující operace tak, aby nakládaly s prostředky korektně.
  - Zakázat kopírující a přesunující operace pomocí `=delete`.

# Pravidlo tří (pěti, nuly)

- Pokud programátor definoval destruktor, kopírující konstruktor, nebo kopírující přiřazení, měl by je definovat všechny tři.
- Pro některé třídy se pravidlo tří vztahuje na destruktor, přesunující konstruktor a přesunující přiřazení.
- Pravidlo pěti říká totéž o destruktoru, kopírujících a přesunujících operacích.
- Občas se vyskytuje také pravidlo nuly: třída by měla být dělaná tak, aby stačilo přenechat generování speciálních funkcí kompilátoru.

# Pravidlo nuly – příklad

```
class osoba {
public:
    osoba() = default;
    osoba(std::string jmeno, std::string prijmeni)
        :jmeno{std::move(jmeno)}, prijmeni{std::move(prijmeni)}
{}
private:
    std::string jmeno, prijmeni;
};
```

```
int main() {
    osoba o{"Jan", "Novak"}, o2{“Petr", “Svoboda"}, o3{“Pavel",
“Novotny"};
    o3 = std::move(o2);
    o2 = o;
    osoba o4(o3);
}
```

# The End