

Přetěžování metod a operátorů

Karel Richta a kol.

Katedra technických studií
Vysoká škola polytechnická Jihlava

© Karel Richta, 2020

Objektově-orientované programování, OOP 02/2020, Lekce 6

<https://moodle.vspj.cz/course/view.php?id=200875>
<https://moodle.vspj.cz/course/view.php?id=200875>



Vysoká škola
polytechnická
Jihlava

Polymorfismus

- **Přetěžování funkcí** – stejný identifikátor, ale různé chování, kompilátor se rozhoduje dle datových typů.
- **Přetěžování operátorů** – další postup: operátory mohou mít více významů, respektive mohou provádět různé činnosti.
- Rozlišení činnosti dle datových typů jako operandů: celočíselné dělení, float dělení, &bitový operátor, operátor získání adresy, atd.
- Standard definuje chování operátorů pro standardní datové typy.
- Třída je uživatelem definovaný typ => uživatel musí definovat, jak má operátor pracovat.
- Cíle přetěžování operátorů:
 - Intuitivní používání objektů třídy.
 - Snižování chyb při práci s objekty – uživatel definuje, jak bude operace probíhat.
 - Možnost použití operátoru << pro výstupní operace.
- Operátor pro danou třídu může být definován jako metoda třídy, nebo jako samostatná funkce.

Motivační příklad

Komplexní číslo (Re, Im):

- Chceme zvýšit hodnotu obou složek o 1 -> chceme operátor ++.
- Operátor ++ neví jak má pracovat s komplexním číslem.
- Řešení:
 - Funkce, která provede navýšení.
 - Přetížený operátor ++.

```
class Komplex {  
private:  
    int m_real; int m_imag;  
public:  
    Komplex (int re, int im);  
    void operator++();  
    void tisk();  
};
```

```
Komplex::Komplex (int re, int im) {  
    m_real= re; m_imag = im;  
};  
void Komplex::tisk() { cout<<"\nrealna:  
    "<<m_real<<"imaginarni: "<<m_imag;  
};  
void Komplex::operator++() {  
    m_imag++; m_real++;  
};
```

```
int main() {  
    Komplex k1(10,5);  
    k1.tisk();  
    ++k1; // k1.operator  
    ++(); // členská metoda  
    k1.tisk();  
    return 0;  
}
```

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
}
```

Syntaxe přetíženého operátoru

- Funkce/metoda, která má na místo jména klíčové slovo **operator** - po něm následuje označení operátoru (např. **operator+**).
- Celková syntaxe závisí na konkrétním operátoru:
 - zda je unární, binární,
 - na datových typech jeho argumentů a
 - jeho návratové hodnotě.

void operator++(); //1 operand k dispozici

Komplex operator+(Komplex&) ; //2 operandy k dispozici

- Přetížený operátor = Členská metoda – má svoji deklaraci a definici:

```
void operator++() { m_imag++; m_real++; }
```

Přetížení operátoru +

```
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& p) const {  
        return Complex(real + p.real, imag + p.imag);  
    }  
};  
  
int main() {  
    Complex c1(2.5, -2);  
    Complex c2(-0.5, 3);  
    Complex c3 = c1 + c2;  
    cout << c3.getReal() << " + " << c3.getImag() << "i\n";  
}
```

Operátor přetížený
pomocí metody

Rozšíření třídy Trojúhelník

- Proveďte přetížení operátoru + aby fungovala operace `Trojuhelnik(1,10,20) + 5` a vznikl trojúhelník se stranami 6,16,26

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.

```
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; }  
};
```

Operátor přetížený
pomocí funkce

```
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.

```
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);  
}
```

Operátor přetížený pomocí
spřátelené funkce

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.

```
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; }  
};
```

Přetížit operátor << lze
pouze pomocí funkce

```
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";  
}
```

Metoda vs. funkce

Mezi přetěžováním operátoru metodou a funkcí je několik odlišností.

- Metody mají přístup k privátním položkám; funkce nemají.
- To lze vyřešit pomocí "**friend**" funkcí:

```
class Complex {  
    double real, imag;  
public:  
    Complex(double re, double im) ...  
    friend Complex operator+(const Complex& l, const Complex& p);  
};  
Complex operator +(const Complex& l, const Complex& p) {  
    return Complex(l.real+p.real, l.imag+p.imag);  
}
```

privátní datové položky

friend funkce má přístup k
privátním datovým položkám

Spřátelené funkce

- Členská data jsou přístupná pouze přes veřejné metody dané třídy.
- Existuje mechanismus spřátelených funkcí a tříd.
- Umožňuje přistoupit k privátním položkám třídy bez použití veřejných metod.
- Spřátelené funkce nemají stejná práva jako členské metody.
- Prototyp spřátelené třídy ve třídě, ke které dostane práva (později).
- Deklarace spřátelené funkce:

friend navr_typ nazev(arg);

```
//pridano do komplex.h
friend void tisk_friend(Komplex);

//pridano do komplex.cpp
void tisk_friend(Komplex k) {
    cout << k.m_real << "re " << k.m_imag << "im ";
};
void Komplex::tisk() {
    cout << m_real << "re " << m_imag << "im ";
};
```

Použití

```
//complex.h
class Komplex {
private:
    int m_real;
    int m_imag;
public:
    Komplex (int re, int im);
    void inc();
    void operator++();
    friend void tisk_friend(Komplex);
};

//program komplex.cpp
int main() {
    Komplex k1(10,5);
    k1.tisk();
    tisk_friend(k1);
    return 0;
};
```

Operátor jako spřátelená funkce

- Tzv. nečlenský operátor.
- Operátor je implementován jako samostatná funkce:
 - $A + B$, `operator+(A, B)`, tj.:
- všechny operandy musí být uvedeny v hlavičce funkce.
- Použijte vždy u I/O operátorů, jinak nastane "obrácená" syntaxe např. `operator<<` (dej na „výstup“, např. `cout << a;`)
- Motivace : $B = A * k$ – lze přetížit pomocí členské funkce / $B = k * A$ – nelze (`k.operator*(A)`).

```
//B = A*k
//pridano do komplex.h
Komplex operator*(int);
//pridano do komplex.cpp
Komplex Komplex::operator*(int n) {
    Komplex pom; pom.m_real = n*m_real; pom.m_imag = n*m_imag;
    return pom;
};
```

Použití

- Logicky:

Komplex operator*(int, Komplex&)

- členská funkce má k dispozici implicitně členská data;
- * binární opeátor – (předány 2 argumenty + 1 implicitně).
- Jediné řešení je přetížený operátor pomocí spřátelené funkce:

```
//complex.h
class Komplex {
private:
    int m_real;
    int m_imag;
public:
    Komplex (int re, int im);
    void inc();
    void operator++();
    friend void tisk_friend(Komplex);
};
```

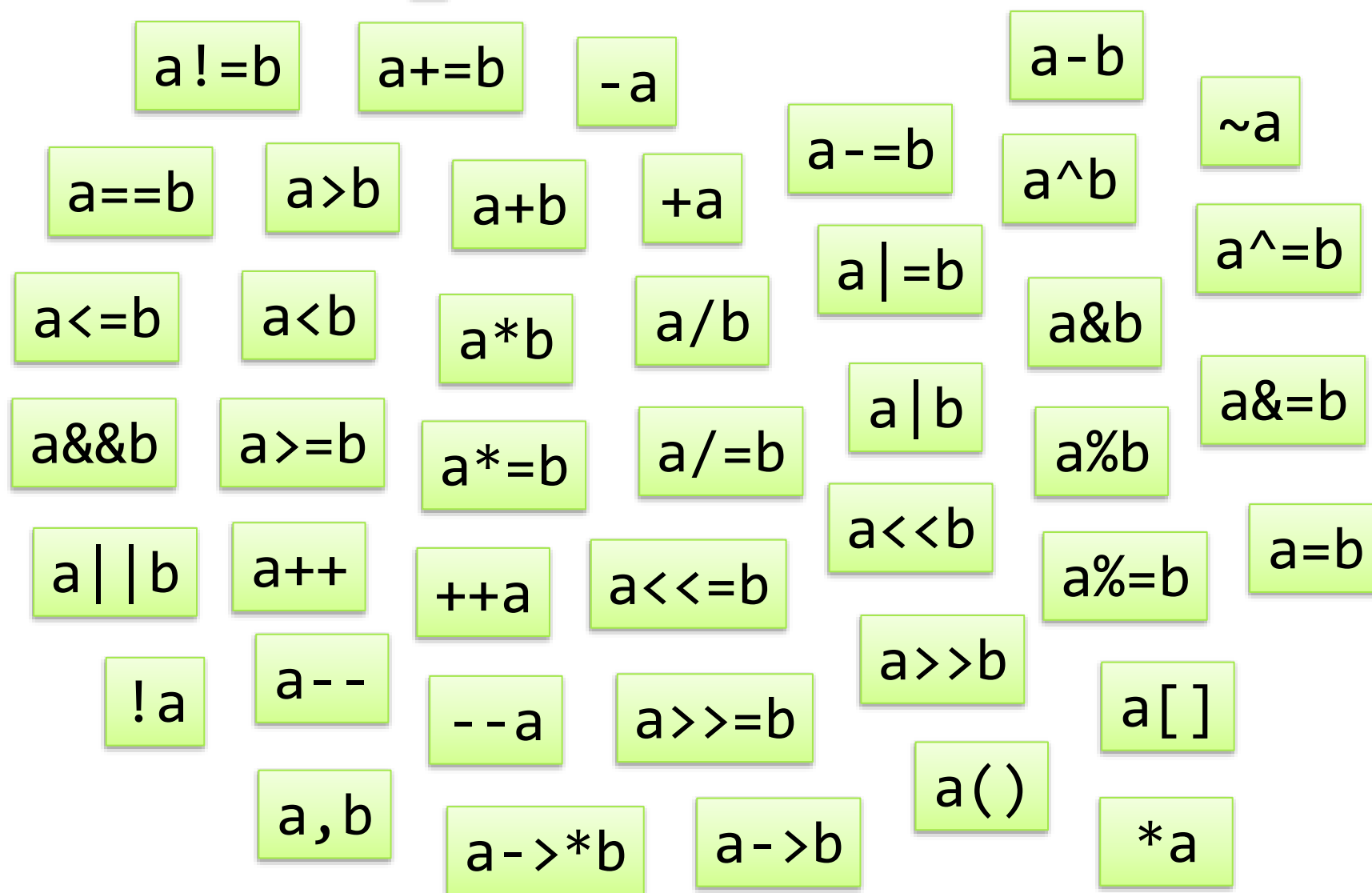
```
//program komplex.cpp
int main() {
    Komplex k1(10,5);
    k1.tisk();
    tisk_friend(k1);
    return 0;
}
```

Přetížení operátoru +=

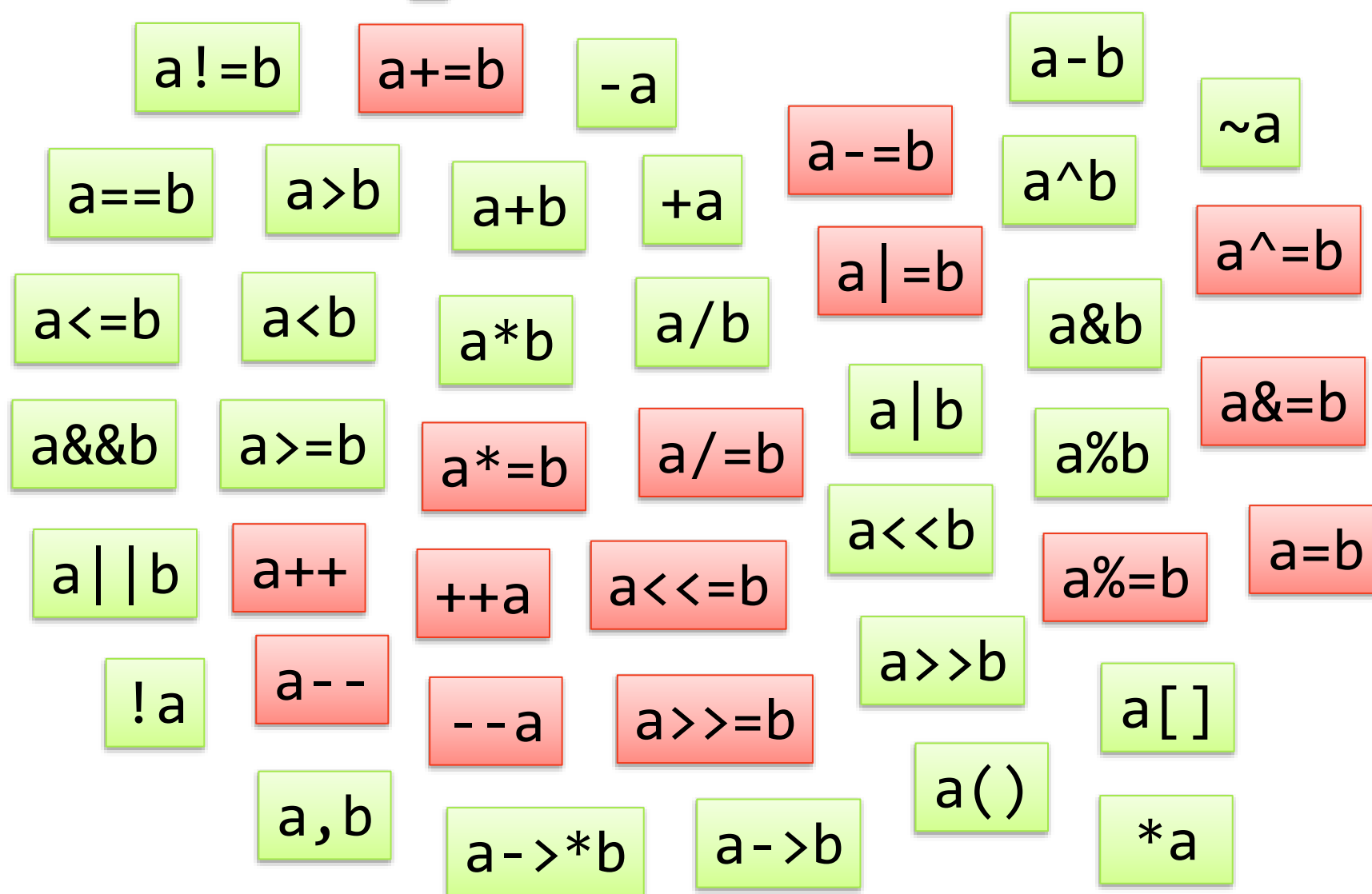
- Podobně jako operátor + můžeme přetížit i operátor +=.
- Operátor **a+=b** nazýváme modifikující, protože by měl změnit svůj levý operand a; oproti tomu **a+b** je příklad nemodifikujícího operátoru.
- Chování operátorů + a += definujeme zvlášť; když zdefinujeme +, neznamená to, že bude fungovat +=.

```
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 []

- Seznamte se s třídou zasobnik.

```
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;
};
```

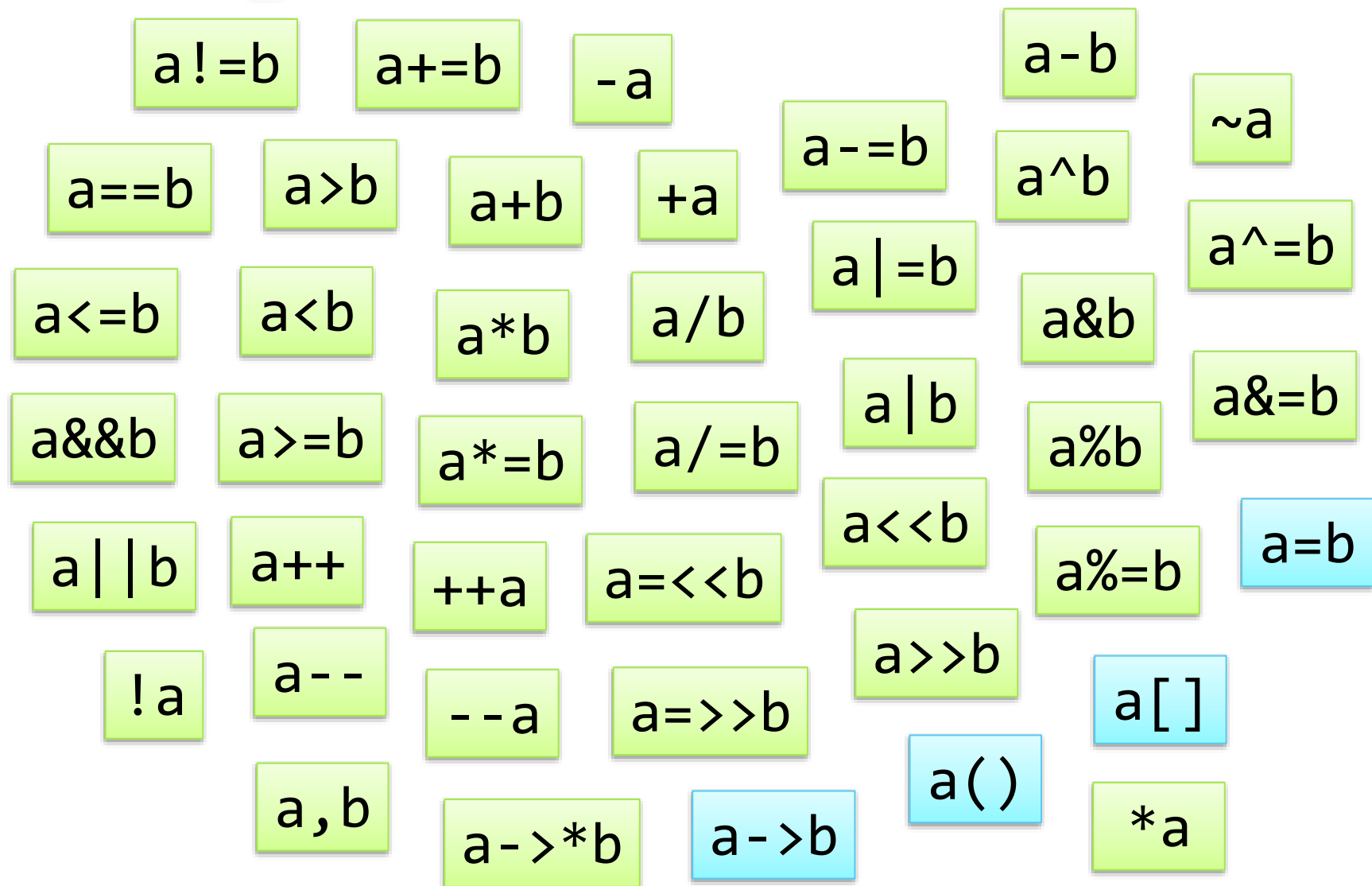
Přetížení operátoru []

- 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)

```
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 úplnost

- 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);          // proved' ++a  
        return kopie;       // vrať kopii  
    }  
};
```

Třída String s operátory

```
class String {
    int pocet;        // aktuální délka řetězce
    char *znaky;      // ukazatel na dyn. alokované pole
public:
    String();          // vytvoří prázdný řetězec
    String(const String&); // vytvoří řetězec jako kopii jiného
    String(const char *); // konverze z C-stringu
    ~String();         // destruktork
    int delka() const; // vrátí délku řetězce
    String& operator=(const String&); // přiřazení řetězců
    friend ostream& operator<<(ostream&, const String&); // výstup řetězce do
                                                    // streamu

    bool operator>=(const String&) const; // porovnání řetězců
    String operator+(const String&) const; // sřetězení
    operator const char *() const; // konverze na C-string
private:
    String(const String&, const String&); // podpora sřetězení
};
```

Třída String s operátory

- přiřazení:

```
String& String::operator=(const String& p) {  
    if (this != &p) {  
        delete [] znaky;  
        pocet = p.pocet;  
        znaky = new char[pocet+1];  
        strcpy(znaky, p.znaky);  
    }  
    return *this;  
}
```

- výstup do datového proudu:

```
ostream& operator<<(ostream& str, const String& p) {  
    return str << p.znaky;  
}
```

Třída String s operátory

- konverze na C-string:

```
String::operator const char *() const {  
    return znaky;  
}
```

- zpřístupnění i-tého znaku (pro čtení):

```
char String::operator[](int i) const {  
    if (i < 0 || i >= pocet) throw ChybnyIndex();  
    return znaky[i];  
}
```

- zpřístupnění i-tého znaku (pro změnu):

```
char& String::operator[](int i) {  
    if (i < 0 || i >= pocet) throw ChybnyIndex();  
    return znaky[i];  
}
```

Třída String s operátory

- použití:

```
int main() {  
    String A, B("abc"), C(B);  
    A = B; // přiřazení  
    C = A + "de"; // sřetězení, konverze  
    if (C >= B) // porovnání  
        cout << C << " je větší nebo rovno " << B;  
    else  
        cout << C << " je menší než " << B;  
    cout << endl << "Délka " << C << " je: " << C.delka() << endl;  
    return 0;  
}
```

Výstup:

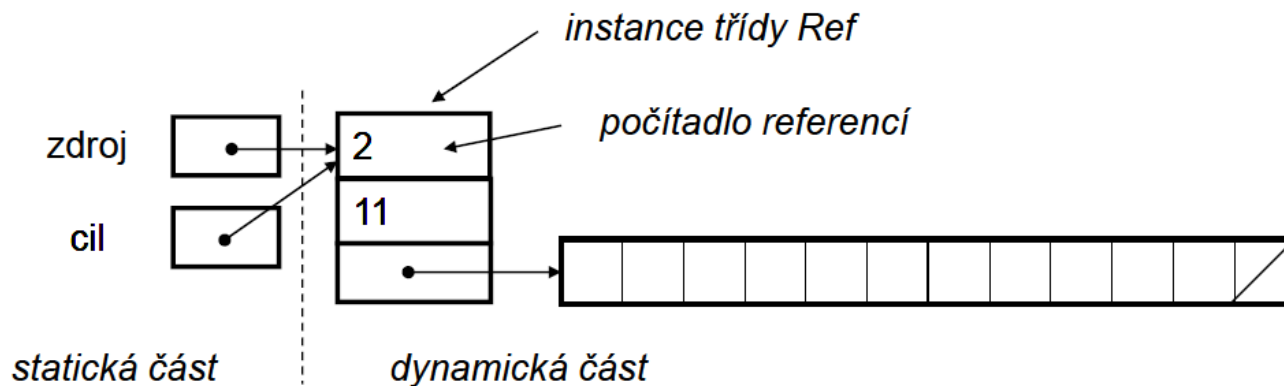
abcde je větší nebo rovno abc
Délka abcde je: 5

Počítané reference

- Sdílení dynamické části:

je třeba vyřešit dva problémy: modifikaci a destrukci

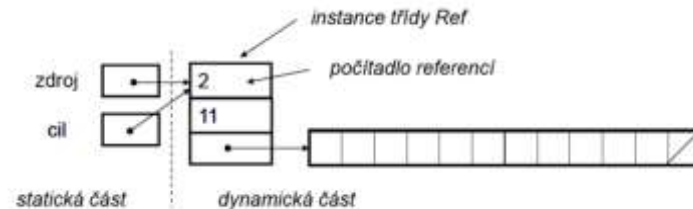
cil = zdroj;



Počítané reference

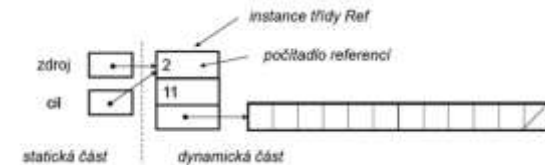
```
class Ref {
public:
    int reference;
    int pocet;
    char *znaky;
public:
    Ref(const char *str) {
        reference = 1; pocet = strlen(str);
        znaky = new char[pocet+1];
        strcpy(znaky, str);
    }
    Ref(int delka) {
        reference = 1; pocet = delka;
        znaky = new char[pocet+1];
    }
    ~Ref() {
        delete [] znaky; pocet = 0;
        reference = 0; znaky = 0;
    }
};

Ref prazdny("");
```



Počítané reference

```
class String {  
    Ref *ref;  
    String(const String& l, const String& p); // konstruktor pro sretezeni  
public:  
    String() { // prazdny retezec  
        ref = &prazdny; prazdny.reference++;  
    }  
    String(const char *str) { // C-string  
        ref = new Ref(str);  
    }  
    String(const String& str) { // kopie jineho retezce  
        ref = str.ref; ref->reference++;  
    }  
    ~String() { // destruktork  
        ref->reference--;  
        if (ref->reference == 0) { delete ref; ref = 0; }  
    }  
    int delka() const { // délka řetězce  
        return ref->pocet;  
    }  
    operator const char *() const { // konverze na C-string  
        return ref->znaky;  
    }  
}
```



Počítané reference

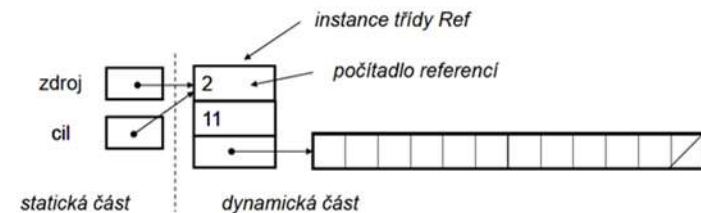
```
String& operator=(const String& p) { // přiřazení
    if (ref != p.ref) {
        this->~String(); // explicitní volání destruktoru!
        ref = p.ref;
        ref->reference++;
    }
    return *this;
}

friend String operator+(const String& l, const String& p) { // sřetězení
    return String(l, p);
}

char operator[](int i) const; // vrať i-tý znak - později
void zmen(int i, char novy); // změň i-tý znak - později
bool operator>=(const String& p) const { // větší nebo rovno
    return strcmp(ref->znaky, p.ref->znaky) >= 0;
}

bool operator==(const String& p) const { // rovnost
    return strcmp(ref->znaky, p.ref->znaky) == 0;
}
};

class ChybyIndex {}; // třída pro výjimku
```



Počítané reference

```
String::String(const String& l, const String& p) {  
    ref = new Ref(l.ref->pocet + p.ref->pocet);  
    strcpy(ref->znaky, l.ref->znaky);  
    strcat(ref->znaky, p.ref->znaky);  
}  
  
char String::operator[](int i) const {  
    if (i < 0 || i >= ref->pocet) throw ChybnyIndex();  
    return ref->znaky[i];  
}  
  
void String::zmen(int i, char novy) {  
    if (i < 0 || i >= ref->pocet) throw ChybnyIndex();  
    if (ref->reference > 1) {  
        ref->reference--;  
        ref = new Ref(ref->znaky); // vytvoříme dynamickou kopii  
    }  
    ref->znaky[i] = novy;  
}
```

The End