RAII, speciální metody, přetěžování operátorů

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad

© Karel Richta, Martin Hořeňovský, Aleš Hrabalík, 2017

Programování v C++, A7B36PJC
06/2017, Lekce 6

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

RAII

- Jedná se o základní koncept programování v C++.
- *Resource Allocation Is Initialization*
  - Nebo také *Responsibility Acquisition Is Initialization*
- Myšlenka: Při svém vzniku získá objekt nějakou zodpovědnost. Při svém zániku tuto zodpovědnost musí naplnit.
  - Např. **`std::vector`** má zodpovědnost za alokovanou paměť a naplní ji dealokací své paměti.
- Zodpovědnost může mít mnoho podob.
  - Např. **`std::ifstream`** otevírá a zavírá soubor.

Příklad: RAII (1/5)

```
void Data::readFromFile(const char* filename) {
    FILE* fp = std::fopen(filename, "r");
    if (!fp) return;

    int numItems;
    if (std::fscanf(fp, "%d", &numItems) != 1) return;
    for (int i = 0; i < numItems; ++i) {
        int item;
        if (std::fscanf(fp, "%d", &item) != 1) return;
        mItems.push_back(item);
    }
    std::fclose(fp);
}
```

- Pokud se nám povede otevřít soubor (`fopen`), nesmíme ho zapomenout zavřít (`fclose`).
- Jenže to není vždy tak jednoduché...

Příklad: RAII (2/5)

```
void Data::readFromFile(const char* filename) {
    FILE* fp = std::fopen(filename, "r");
    if (!fp) return;

    int numItems;
    if (std::fscanf(fp, "%d", &numItems) != 1) return;
    for (int i = 0; i < numItems; ++i) {
        int item;
        if (std::fscanf(fp, "%d", &item) != 1) return;
        mItems.push_back(item);
    }
    std::fclose(fp);
}
```

- Když je funkce složitější, je snadné na něco zapomenout. Zde při provedení vnořeného příkazu `return` soubor nezavřeme.

Příklad: RAII (3/5)

```
void Data::readFromFile(const char* filename) {
    FILE* fp = std::fopen(filename, "r");
    if (!fp) return;

    int numItems;
    if (std::fscanf(fp, "%d", &numItems) != 1) return;
    for (int i = 0; i < numItems; ++i) {
        int item;
        if (std::fscanf(fp, "%d", &item) != 1) return;
        mItems.push_back(item);
    }
    std::fclose(fp);
}
```

- Při čtení kódu jsou nenaplněné odpovědnosti téměř neviditelné.
- Jak mít jistotu, že jsme na něco nezapomněli?

Příklad: RAII (4/5)

```
void Data::readFromFile(const char* filename) {
    FILE* fp = std::fopen(filename, "r");
    if (!fp) return;
    FileSentry sentry(fp);
    int numItems;
    if (std::fscanf(fp, "%d", &numItems) != 1) return;
    for (int i = 0; i < numItems; ++i) {
        int item;
        if (std::fscanf(fp, "%d", &item) != 1) return;
        mItems.push_back(item);
    }
    std::fclose(fp);
}
```

soubor uzavřen automaticky

- Uvažme objekt třídy `FileSentry`, který:
  - má zodpovědnost za zavření souboru, který poskytneme
  - zodpovědnost naplní při svém zániku, tedy v destruktoru

Příklad: RAII (5/5)

- Jakmile se provede konstruktor objektu `sentry`, destruktor – a tudíž uzavření souboru – se provede automaticky při opuštění funkce `readFromFile()`. Problém vyřešen!
- Třídu `FileSentry` si můžeme napsat třeba takto:

```
class FileSentry {
public:
    FileSentry(FILE* in) : fp(in) {}
    ~FileSentry() { fclose(fp); }
private:
    FILE* fp;
};
```

std::ifstream

- Třída `std::ifstream` (input file stream):
  - umožňuje používat operaci >> stejně jako `std::cin`
  - její konstruktor otevře soubor pro čtení
  - její destruktor soubor zavře

  } RAII

```
void Data::readFromFile(const char* filename) {
    std::ifstream in{filename};
    if (!in) return;
    int numItems;
    if (!(in >> numItems)) return;
    for (int i = 0; i < numItems; ++i) {
        int item;
        if (!(in >> item)) return;
        mItems.push_back(item);
    }
}
```

```
#include <fstream>
```

RAII vs. memory leak (1/3)

- Vždy, když použijeme operátor new, můžeme zapomenout použít operátor `delete`.
  - Když delete zapomeneme, nastává tzv. **memory leak**.
- Jak tomu zabránit pomocí RAII?

```
int main() {
    Person* p = new Person{"Vincent van Gogh", 164};

    if (...) {
        reportName(p->getLastName());
        return; // oops, zapomněli jsme na delete
    }
    delete p;
}
```

RAII vs. memory leak (2/3)

- Můžeme si napsat třídu `PersonDeleter`, která
  - má zodpovědnost za smazání ukazatele na `Person`, tedy
  - v konstruktoru přebírá ukazatel na `Person` a
  - v destruktoru jej smaže.

```
int main() {
    Person* p = new Person{"Vincent van Gogh", 164};
    PersonDeleter del{p};
    if (...) {
        reportName(p->getLastName());
        return; // delete automaticky
    }
    // delete automaticky
}
```

RAII vs. memory leak (3/3)

- Ještě lepší je použít třídu `unique_ptr<Person>` ze standardní knihovny, protože
  - je odpovědná za smazání, stejně jako `PersonDeleter`, a
  - má další přednosti, viz dále.

```
int main() {
    Person* p = new Person{"Vincent van Gogh", 164};
    std::unique_ptr<Person> up{p};
    if (...) {
        reportName(p->getLastName());
        return; // delete automaticky
    }
    // delete automaticky
}
```

```
#include <memory>
```

std::unique_ptr<T> (1/6)

- Je možné používat šipku -> (a hvězdičku *) pro přístup k datovým položkám a metodám objektu, stejně jako kdybychom pracovali s ukazatelem.

```
int main() {
    Person* p = new Person{"Vincent van Gogh", 164};
    std::unique_ptr<Person> up{p};

    if (...) {
        reportName(up->getLastName()); //up-> je jako p->
        return;
    }
}
```

```
#include <memory>
```

`std::unique_ptr<T>` (2/6)

- Protože se `unique_ptr` chová jako ukazatel, nepotřebujeme původní proměnnou.
- Říkáme, že `unique_ptr` je **chytrý ukazatel** (*smart pointer*).

```
int main() {
    std::unique_ptr<Person> p{new Person{
                                 "Vincent van Gogh", 164}};

    if (...) {
        reportName(p->getLastName());
        return;
    }
}
```

```
#include <memory>
```

std::unique_ptr<T> (3/6)

- auto a make_unique umožňují zjednodušení syntaxe pro vytvoření unique_ptr.
  - make_unique provede operaci new a vrátí unique_ptr.
  - Zvláště vhodné pro třídy s dlouhým názvem.

```
int main() {
    auto p = std::make_unique<Person>("Vincent van Gogh",
                                       164);

    if (...) {
        reportName(p->getLastName());
        return;
    }
}
```

```
#include <memory>
```

std::unique_ptr<T> (4/6)

- Objekty typu `unique_ptr` **lze přesouvat.**
  - Po dokončení přesunu bude původní `unique_ptr` prázdný – bude ukazovat na `nullptr`.
- Objekty typu `unique_ptr` **nelze kopírovat.**
  - Proč `unique_ptr` neprovádí mělkou kopii?
  - Proč `unique_ptr` neprovádí hlubokou kopii?

```
int main() {
    auto p = std::make_unique<Person>("Vincent van Gogh",
                                      164);

    auto q2 = std::move(p); // lze
    auto q1 = p; // nelze
}
```

```
#include <memory>
```

`std::unique_ptr<T>` (5/6)

- Proč `unique_ptr` neprovádí mělkou kopii?
  - Kdyby ji provedl, tak dva objekty budou mít odpovědnost za stejná data. Nastane dvojí smazání objektu typu `Person`.
- Proč unique_ptr neprovádí hlubokou kopii?
  - Pokud zkopírujeme obyčejný ukazatel, provede se mělká kopie. Bylo by překvapivé, kdyby se chytrý ukazatel choval jinak.
  - Nebylo by možné mít `unique_ptr` na takové typy, které samy o sobě nelze kopírovat, jako např. `std::mutex`.

`std::unique_ptr<T>` (6/6)

- K čemu slouží `unique_ptr`?
  - Umožňuje **exkluzivní vlastnictví** objektu na haldě, tj. odpovědnost za smazání má právě jeden objekt.
  - Přitom zabraňuje chybám práce s pamětí, jako je ztráta paměti (memory leak) nebo dvojí smazání (double delete).
- Co když více součástí programu sdílí stejné prostředky?
  - Je třeba použít `std::shared_ptr`, který umožňuje **sdílené vlastnictví**, tj. odpovědnost za smazání je sdílena větším počtem objektů.

Příklad: RAII podruhé (1/3)

```
Person* makePerson(Profession prof) {
    switch (prof) {
    case Profession::painter:
        return new Person{"Vincent van Gogh", 164};
    case Profession::composer:
        return new Person{"Johann Sebastian Bach", 332};
    }
}
int main() {
    Person* person = makePerson(Profession::painter);
    std::cout << person->getLastName() << '\n';
    // tady něco chybí
}
```

- Funkce `makePerson()` vrací nové objekty.
  - Jedná se o tzv. tovární funkci, továrnu.
- Jenže ve funkci `main()` jsme na něco zapomněli…

Příklad: RAII podruhé (2/3)

```
Person* makePerson(Profession prof) {
    switch (prof) {
    case Profession::painter:
        return new Person{"Vincent van Gogh", 164};
    case Profession::composer:
        return new Person{"Johann Sebastian Bach", 332};
    }
}
int main() {
    Person* person = makePerson(Profession::painter);
    std::cout << person->getLastName() << '\n';
    delete person;
}
```

- Po každém volání `new` musí následovat `delete`!
- Chyba není zřejmá, zvláště kdybychom před sebou neviděli implementaci `makePerson()`...

Příklad: RAII podruhé (3/3)

```
std::unique_ptr<Person> makePerson(Profession prof) {
    switch (prof) {
    case Profession::painter:
        return std::make_unique<Person>("Vinc. van Gogh", 164);
    case Profession::composer:
        return std::make_unique<Person>("Johann S. Bach", 332);
    }
}

int main() {
    auto person = makePerson(Profession::painter);
    std::cout << person->getLastName() << '\n';
    // OK, delete zavoláno automaticky
}
```

- Řešením je používat `unique_ptr` jako návratový typ továrny.

Speciální metody (1/4)

- Speciální metody (*special member functions*), též kompilátorem generované metody, jsou:
  - **Konstruktory: výchozí, kopírující, přesunující**
  - **Přiřazení: kopírující, přesunující**
  - **Destruktor**
- Jsou charakterizovány tím, že je někdy nemusíme psát – za jistých okolností je kompilátor automaticky vytvoří.

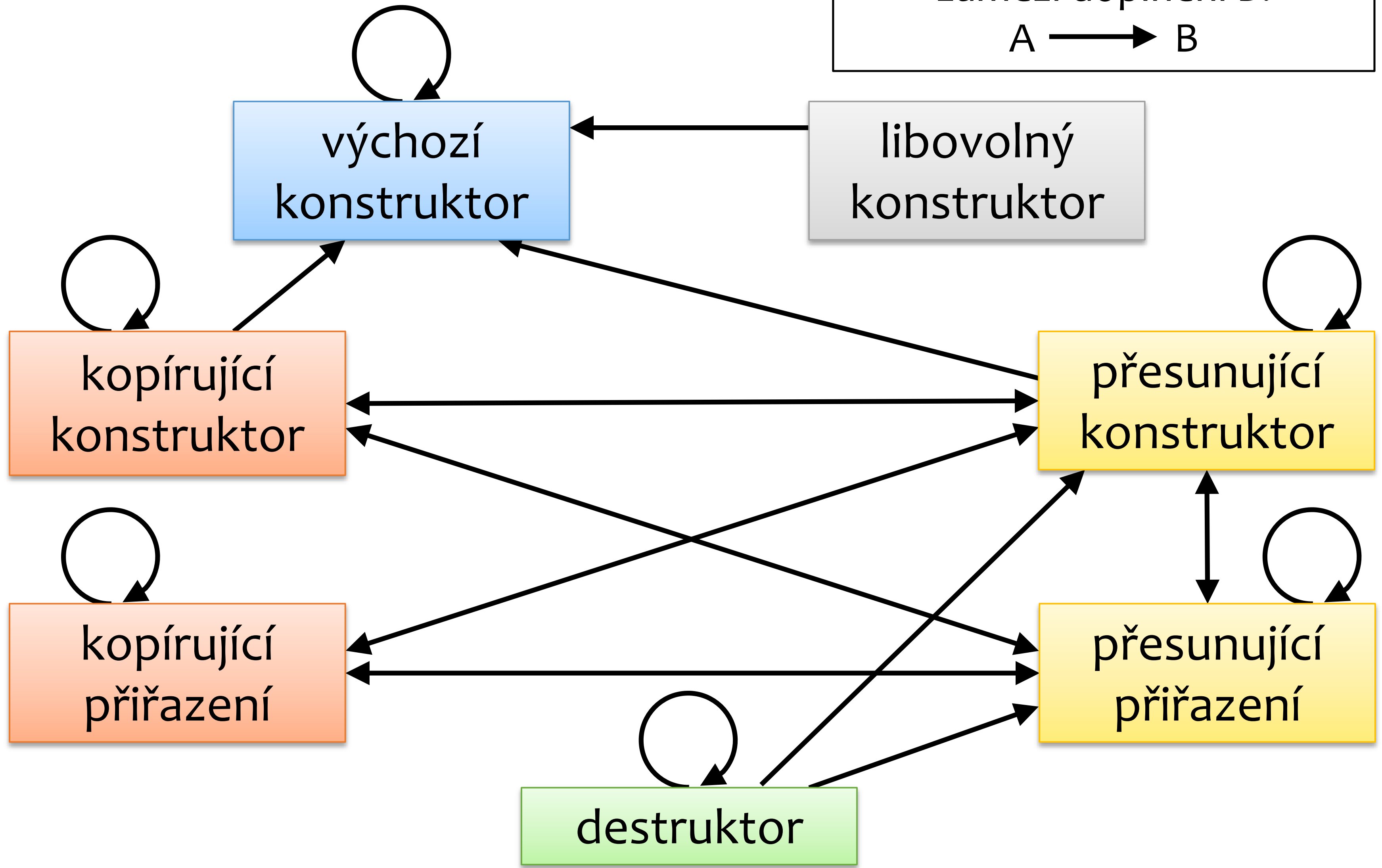
Speciální metody (2/4)

- Chování kompilátorem vygenerovaných speciálních metod je přímočaré.
  - Výchozí konstruktor zavolá výchozí konstruktory všech datových položek.
  - Kopírující konstruktor zavolá kopírující konstruktory všech datových položek.
  - Kopírující přiřazení zavolá kopírující přiřazení všech datových položek.
  - atd.

Speciální metody (3/4)

- Pravidla určující, za jakých okolností jsou speciální metody vygenerovány, přímočará *nejsou*.
  - Výchozí konstruktor je doplněn, když nejsou deklarovány žádné konstruktory.
  - Kopírující operace jsou doplněny, když ony samy nejsou deklarovány a zároveň není deklarována žádná přesunující operace.
  - Přesunující operace jsou doplněny, když nejsou deklarovány a zároveň není deklarována žádná přesunující operace, kopírující operace, ani destruktor.
  - Destruktor je doplněn vždy, když není deklarován.

Speciální metody (4/4)



`default` a `delete`

- Pro danou třídu lze generování konkrétní speciální metody...
  - ...vynutit pomocí klíčového slova `default`:

```
struct Person {
    Person(const std::string& name, int age);
    Person() = default;
    ...
}
```

  - ...zakázat pomocí klíčového slova `delete`:

```
struct SmartPtr {
    SmartPtr(const SmartPtr& rhs) = delete;
    ...
};
```

Pravidlo tří

- **Pokud je pro danou třídu potřeba napsat kopírující konstruktor, kopírující přiřazení, nebo destruktor, pravděpodobně je potřeba napsat všechny tři.**
- Proč?
  - Pokud musíte některou z těchto speciálních metod napsat, téměř jistě nakládáte zvláštním způsobem s daty nebo invarianty.
  - Pokud práce s daty nebo invarianty vyžaduje zvláštní péči, automaticky vygenerované speciální metody jsou pravděpodobně špatně.
  - Proto je potřeba ostatní z těchto tří metod napsat nebo zakázat.

Příklad: pravidlo tří

- Vezměme `FileSentry` tak, jak jsme ho zadefinovali dříve:

```
class FileSentry {
public:
    FileSentry(FILE* in) : fp(in) {}
    ~FileSentry() { fclose(fp); }
private:
    FILE* fp;
};
```

- Pravidlo tří je porušeno. Automaticky vygenerované kopírující operace jsou špatně.
- Proč jsou špatně? Jak to spravit?

Pravidlo nuly

- **Často není třeba psát žádnou ze speciálních metod.**
- Proč?
  – Každá datová položka, která řádně používá RAII, se postará sama o sebe. Postačí automaticky vygenerované speciální metody.

```
struct File {
    std::string path;
    std::unique_ptr<std::ifstream> inputStream;
    // není nutné definovat speciální metody
};
```

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

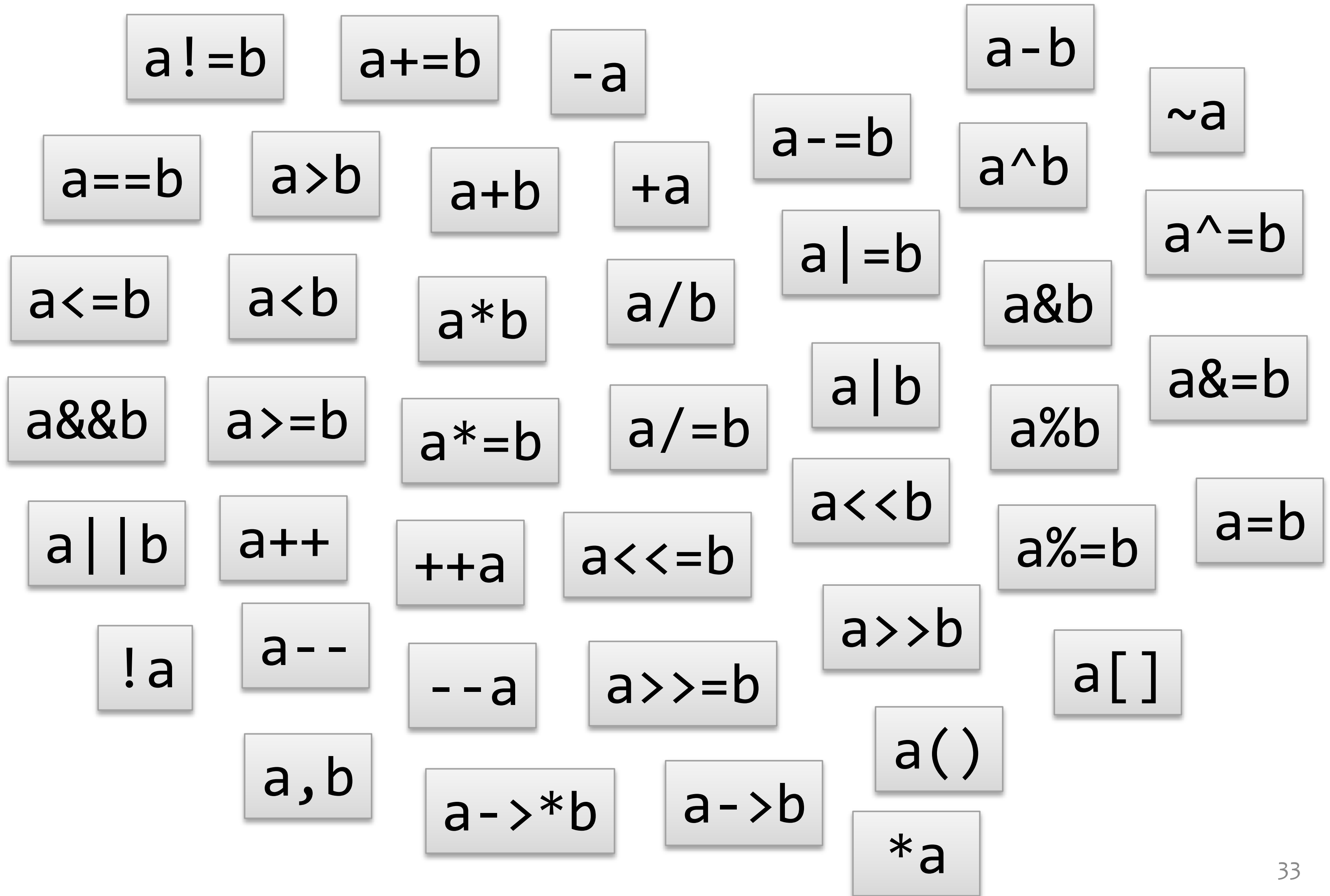
```
#include <iostream>

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 +=

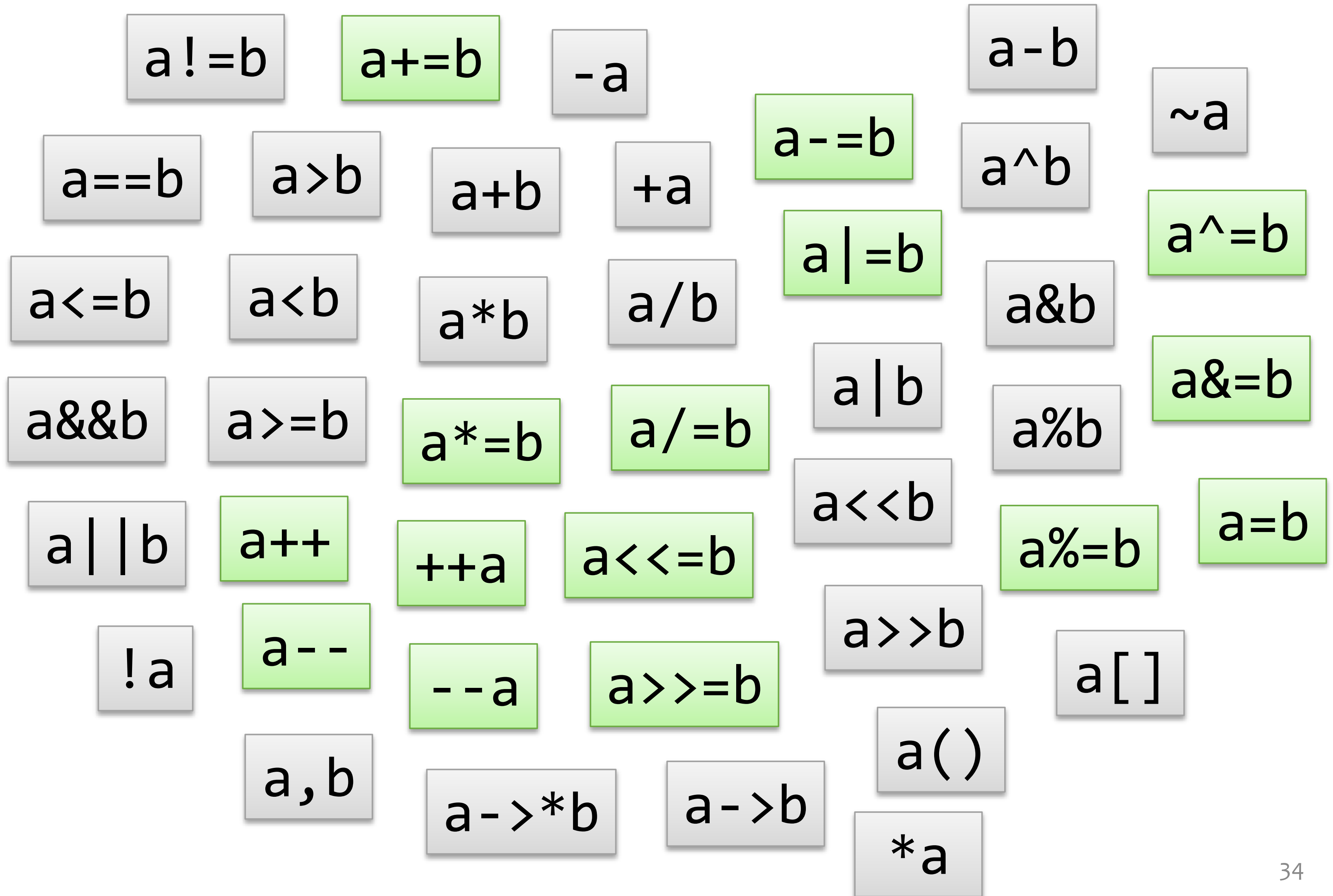
- Podobně jako operátor **+** můžeme přetížit i operátor **+=**.
- Operátor **a+=b** nazýváme modifikující, protože by měl změnit svůj levý operand a; oproti tomu **a+b** je příklad nemodifikujícího operátoru.
- Chování operátorů **+** a **+=** definujeme zvlášť; když zadefinujeme **+**, neznamená to,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

- 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

a!=b a+=b -a a-b ~a

a==b a>b a+b +a a-=b a^b a^=b

a<=b a<b a*b a/b a|=b a&b

a&&b a>=b a*=b a/=b a|b a%b a&=b

a||b a++ ++a a=<<b a<<b a%=b a=b

!a a-- --a a=>>b a>>b a[]

a,b a->*b a->b a() *a

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.

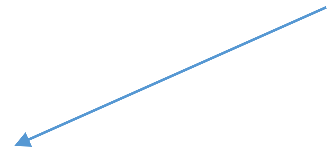
```
int main() {
    Datum d(31, 12, 2015);
    std::cout << d << "\n"; // 31. 12. 2015
    d++;
    std::cout << d << "\n"; // 1. 1. 2016
}
```

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
    }
};
```

Konec