Dynamika objektů

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

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

Programování v C++, B6B36PCC
10/2024, Lekce 5

https://cw.fel.cvut.cz/wiki/courses/b6b36pcc/start

Třídy vs. instance

- **Třída** (*class*) je abstraktní popis nějakého celku.
  - Třída zahrnuje
    1. popis dat – typy a názvy datových položek
    2. popis operací s daty
- **Instance (objekt)** třídy T obsahuje konkrétní data a provádí s nimi operace tak, jak předepisuje třída T.
- Příklad: třída `Person` předepisuje, že každá její instance obsahuje řetězec – jméno osoby. Zároveň popisuje např. operaci `getLastName()`, která s řetězcem pracuje. Instance `painter` třídy `Person` obsahuje jméno „Vincent van Gogh“ a volání `painter.getLastName()` umožní získat „van Gogh“.

Píšeme třídy (1/4)

```
struct Person {
    std::string name;
};

int main() {
    Person painter;
    painter.name = "Vincent van Gogh";
}
```

- Jak napsat třídu?
- Ve cvičení jsme si už napsali strukturu.
- **Struktura je třída.**

Píšeme třídy (2/4)

```
struct Person {
    std::string name;
};

std::string getLastName(const Person& person) {
    ... person.name ...
}

int main() {
    Person painter;
    painter.name = "Vincent van Gogh";
    std::cout << getLastName(painter) << '\n'; // van Gogh
}
```

- Také už umíme `getLastName()` implementovat pomocí funkce.

Píšeme třídy (3/4)

funkce

```
struct Person {
    std::string name;
};


std::string getLastName(const
            Person& person) {
    ... person.name ...
}

int main() {
    ... getLastName(painter) ...
}
```

metoda

```
struct Person {
    std::string name;
    std::string getLastName() const;
};

std::string Person::getLastName()
            const {
    ... name nebo this->name ...
}

int main() {
   ... painter.getLastName() ...
}
```

- Druhý způsob, jak implementovat operaci, je pomocí **metody** – podobně jako v jazyce Java.
  - Metoda třídy T musí být deklarována ve třídě T a její název u definice musí být uvozen `T::`
  - K datům instance lze přistupovat přímo nebo přes ukazatel `this`

Píšeme třídy (4/4)

funkce

```
struct Person {
    std::string name;
};


std::string getLastName(const
            Person& person) {
    ... person.name ...
}

int main() {
    ... getLastName(painter) ...
}
```

metoda

```
struct Person {
    std::string name;
    std::string getLastName() const;
};

std::string Person::getLastName()
            const {
    ... name nebo this->name ...
}

int main() {
   ... painter.getLastName() ...
}
```

- Metody lze definovat i v těle třídy.
  - Ještě podobnější Javě
  - Nepíšeme `Person::`

metoda varianta 2

```
struct Person {
    std::string name;

    std::string getLastName() const {
        ... name nebo this->name ...
    }
};
```

Konstantní metody (1/5)

funkce

```
struct Person {
    std::string name;
};


std::string getLastName(const
            Person& person) {
    ... person.name ...
}

int main() {
    ... getLastName(painter) ...
}
```

metoda

```
struct Person {
    std::string name;
    std::string getLastName() const;
};

std::string Person::getLastName()
            const {
    ... name nebo this->name ...
}

int main() {
   ... painter.getLastName() ...
}
```

- Všimněte si `const` na konci, za seznamem parametrů.
- Co by to mohlo znamenat?

metoda varianta 2

```
struct Person {
    std::string name;

    std::string getLastName() const {
        ... name nebo this->name ...
    }
};
```

Konstantní metody (2/5)

```
struct Person {
    std::string name;
    void setName(const std::string& aName);
    std::string getLastName() const;
};
void Person::setName(const std::string& aName) {
    name = aName;
}
std::string Person::getLastName() const {
    ... name ...
}
```

- Abychom pochopili význam `const` za seznamem parametrů, rozšiřme příklad o metodu `setName()`.
- Všimněte si, že `setName()` není označena `const`.
- V čem se liší `setName()` a `getLastName()`?

Konstantní metody (3/5)

```
struct Person {
    std::string name;
    void setName(const std::string& aName);
    std::string getLastName() const;
};
void Person::setName(const std::string& aName) {
    name = aName;
}
std::string Person::getLastName() const {
    ... name ...
}
```

- Vidíme, že `setName()` mění data instance – pozmění hodnotu `name`, zatímco `getLastName()` ne.
- **Jako konstantní tedy označujeme ty metody, které nemění data své instance.**

Konstantní metody (4/5)

```
struct Person {
    std::string name;
    void setName(const std::string& aName);
    std::string getLastName() const;
};
void Person::setName(const std::string& aName) {
    name = aName; // OK
}
std::string Person::getLastName() const {
    name = aName; // chyba
}
```

- V metodě označené jako konstantní je zakázáno měnit data své instance.
- Pokud bychom např. v `getLastName()` přiřadili do `name`, nastane kompilační chyba.

Konstantní metody (5/5)

```
void renaissanceOverflow1(Person& person) {
    std::cout<<"A "<<person.getLastName()<<" no longer!"; // OK
    person.setName("Leonardo da Vinci");                  // OK
}
void renaissanceOverflow2(const Person& person) {
    std::cout<<"A "<<person.getLastName()<<" no longer!"; // OK
    person.setName("Leonardo da Vinci");                  // chyba
}
```

- Konstantní metody pomáhají odhalit chyby.
  - Pokud jsme získali např. konstantní referenci na nějaký objekt, tak volající spoléhá na to, že jej nezměníme.
  - Díky konstantním metodám víme přesně, které metody objekt změní a které ne.
  - Kompilátor za nás automaticky kontroluje, že pro konstantní objekty – typu `const T`, `const T*`, `const T&` – používáme pouze konstantní metody. V opačném případě dojde k chybě.

Viditelnost

```
struct Person {
private:
    std::string name;                              // private
public:
    void setName(const std::string& aName); // public
    std::string getLastName() const;          // public
};
```

- Možné viditelnosti jako v Javě: **public**, **protected**, **private**
- Viditelnost určujeme pomocí klíčových slov následovaných dvojtečkou
- Viditelnost je platná, dokud není stanoveno jinak

Klíčové slovo `class`

```
struct S {
    void a() const; // public
private:
    int b;          // private
};
```

```
class C {
    void c() const; // private
public:
    int d;          // public
};
```

- Klíčové slovo `class` také značí třídu.
- Rozdíl je ve výchozí viditelnosti: `public` pro `struct`, `private` pro `class`.

Konstruktor *(constructor)* (1/5)

- Vždy, když vznikne instance nějaké třídy, je skrytě (implicitně) zavolána metoda zvaná **konstruktor**.
- Konstruktory mají zvláštní syntax:
  - Jejich název je shodný s názvem třídy.
  - Za seznamem parametrů se může nacházet tzv. inicializační seznam (*initializer list*, uvozený dvojtečkou), který používáme k inicializaci datových položek třídy.

```
struct Person {
    Person() : name("?") {}
    Person(const std::string& aName) : name(aName) {}
private:
    std::string name;
};
```

Konstruktor *(constructor)* (2/5)

```
struct Person {
    Person();
    Person(const std::string& nm, int ag);
private:
    std::string name;
    int age;
};
Person::Person() : name("?"), age(0) {
    std::cout << "vychozi\n";
}
Person::Person(const std::string& nm, int ag)
    : name(nm), age(ag) {
    std::cout << name << ", " << age << '\n';
}
int main() {
    Person default;                          // vychozi
    Person painter("Vin. van Gogh", 164); // Vin. van Gogh, 164
}
```

Třída může mít libovolný počet konstruktorů.

Konstruktor bez parametrů se nazývá výchozí.

Tělo konstruktoru se provede po vyhodnocení inicializačního seznamu.

Položky v inicializačním seznamu oddělujeme čárkou.

Konstruktor *(constructor)* (3/5)

```
struct Person {
    Person();
    Person(const std::string& nm, int ag);
private:
    std::string name;
    int age;
};
Person::Person() : name("?"), age(0) {
    std::cout << "vychozi\n";
}
Person::Person(const std::string& nm, int ag)
    : age(ag), name(nm) {
    std::cout << name << ", " << age << '\n';
}
int main() {
    Person default;                          // vychozi
    Person painter("Vin. van Gogh", 164); // Vin. van Gogh, 164
}
```

Co se stane, když prohodíme pořadí v inic. seznamu?

Nic. Položky inicializačního seznamu jsou provedeny v tom pořadí, v jakém jsou deklarována data ve třídě.

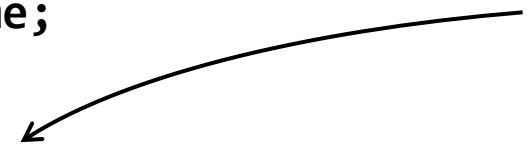
Na pořadí položek v inicializačním seznamu se nehledí. Zde tedy bude pořadí: name, potom age.

```
struct Person {
    Person::Person(const std::string& nm);
private:
    std::string greet;
    std::string name;
};

Person::Person(const std::string& nm)
    : name(nm), greet("Dear " + name) {}
```

Tento program je špatně. Proč?

Konstruktor *(constructor)* (4/5)

```
struct Person {
    Person();
    Person(const std::string& nm, int ag);
private:
    std::string name;
    int age;
};
Person::Person() {
    std::cout << "vychozi\n";
}
Person::Person(const std::string& nm, int
    : name(nm), age(ag) {
    std::cout << name << ", " << age << '\n';
}
int main() {
    Person default;                          // vychozi
    Person painter("Vin. van Gogh", 164); // Vin. van Gogh, 164
}
```



Co se stane, když vynecháme inic. seznam?

Pokud v inicializačním seznamu vynecháme datovou položku, tak se inicializuje výchozím způsobem.

Zde to znamená, že name bude obsahovat prázdný řetězec, a age bude mít **nedefinovanou hodnotu**.

Konstruktor *(constructor)* (5/5)

```
struct Person {
    Person();
    Person(const std::string& nm, int ag);
private:
    std::string name{"?"};
    int age = 0;
};
Person::Person() : name("?"), age(0) {
    std::cout << "vychozi\n";
}
Person::Person(const std::string& nm, int ag)
    : age(ag), name(nm) {
    std::cout << name << ", " << age << '\n';
}
int main() {
    Person default;                          // vychozi
    Person painter("Vin. van Gogh", 164); // Vin. van Gogh, 164
}
```

Můžu ve třídě stanovit výchozí hodnotu položky?

Ano, počínaje standardem C++11 lze výchozí způsob inicializace určit při deklaraci dat ve třídě. Ten pak platí pro všechny konstruktory, které vynechají danou datovou položku v inicializačním seznamu.

Příklad: problémy s inic. seznamem (1/2)

```
class Foo {
    int a;
    int b;
    int c;
    int d = 1;
    int e;
    const int f;
public:
    Foo();
};

Foo::Foo() : b(2),
             a(b + 1),
             d(c + 3) {
    e = 4;
    f = 5;
}
```

Vidí někdo nějaký problém?

Příklad: problémy s inic. seznamem (2/2)

```
class Foo {
    int a; // (*) nejdříve se inicializuje a...
    int b; //      ...až poté b
    int c;
    int d = 1;
    int e;
    const int f;
public:
    Foo();
};

Foo::Foo() : b(2),       // OK, b je 2
             a(b + 1),   // chyba, b má nedefinovanou hodn. (*)
             d(c + 3) {  // chyba, c má nedefinovanou hodn.
    e = 4;               // OK, e je 4
    f = 5;               // chyba, f je konstantní, nejde měnit
}                        // oprava: přidat do inic. seznamu f(5)
```

Význam inicializačního seznamu

- **Všechny datové položky třídy jsou inicializovány před vstupem do těla konstruktoru.**
- Inicializační seznam předepisuje, **jakým způsobem** jsou inicializovány.
- Proč je lepší používat inicializační seznam, než přiřazení (=) v těle konstruktoru?
  - Inicializační seznam může být efektivnější, protože samotná inicializace je potenciálně méně práce, než inicializace a následné přiřazení.
- Jaké datové položky musí být v inicializačním seznamu?
  - Ty, které nelze po inicializaci měnit. Konkrétně: konstantní proměnné a reference.

Možné způsoby vzniku objektu (1/2)

`Obj o;`
`Obj o{};`
`Obj* o = new Obj;`
`Obj* o = new Obj();`
`Obj* o = new Obj{};`
	V těchto případech se zavolá výchozí konstruktor.

`Obj o = 42;`
`Obj o(42);`
`Obj o{42};`
`Obj* o = new Obj(42);`
`Obj* o = new Obj{42};`
	V těchto případech se zavolá konstruktor, který přijímá parametr typu `int`.

`Obj o[42];`
`Obj o[42]{};`
`Obj* o = new Obj[42];`
`Obj* o = new Obj[42]();`
`Obj* o = new Obj[42]{};`
	V těchto případech se 42-krát zavolá výchozí konstruktor.

`Obj o();`	Zrada (známá jako *most vexing parse*): toto není vznik objektu, ale deklarace funkce.

Možné způsoby vzniku objektu (2/2)

`void receivesObj(Obj o); ...`
`receivesObj({});  // výchozí`
`receivesObj(42);  // přijímající int`
	Vznik objektu při předávání hodnoty do funkce

`Obj returnsObj() { ...`
`  return {};  // výchozí`
`  return 42;  // přijímající int`
	Vznik objektu při vracení hodnoty z funkce

`... Obj() ...;  // výchozí`
`... Obj{} ...;  // výchozí`
`... Obj(42) ...;  // přijímající int`
`... Obj{42} ...;  // přijímající int`
	Vznik dočasného *(temporary)* objektu – objekt nemá název

`struct Foo {`
`  Obj o;`
`  Foo() : o(),  // výchozí`
`  o{},  // výchozí`
`  o(42),  // přijímající int`
`  o{42},  // přijímající int`
`  chybí  // podle deklarace`
	Vznik objektu v inicializačním seznamu konstruktoru

Destruktor *(destructor)*

- Vždy, když zanikne instance nějaké třídy, je skrytě (implicitně) zavolána metoda zvaná **destruktor**.
- Destruktor třídy T je její metoda pojmenovaná ~T.
- Rozdíly oproti konstruktoru:
  - Nemá parametry.
  - Nemá inicializační seznam.
  - V dané třídě může být pouze jeden.

```
struct Person {
    ~Person() {
        std::cout << "destruktor\n";
    }
};
```

Význam destruktoru

- Provádí úklid; uvolňuje **prostředky**, za které má daná instance **odpovědnost**.
  - Prostředky mohou být: paměť, soubory, připojení k databázím, a další.

```
struct User {
    User() : data(new UserData) { // získej paměť
        systemLogIn(*this);       // přihlas se
    }
    ~User() {
        systemLogOut(*this);      // odhlas se
        delete data;              // uvolni paměť
    }
private:
    UserData* data;
};
```

Invariant (1/2)

- Skutečnosti, které platí po celou dobu existence objektu, nazýváme **invarianty**.
- V tomto příkladu: paměť je alokovaná, uživatel je přihlášen.

```
struct User {
    User() : data(new UserData) { // získej paměť
        systemLogIn(*this);       // přihlas se
    }
    ~User() {
        systemLogOut(*this);      // odhlas se
        delete data;              // uvolni paměť
    }
private:
    UserData* data;
};
```

Invariant (2/2)

- Abychom zavedli invarianty, používáme:
  - konstruktory – nastolíme invarianty
  - zapouzdření pomocí viditelnosti – povolíme jen ty operace, které invarianty neporuší
  - destruktor – uklidíme, pokud je to potřeba

```
struct User {
    User() : data(new UserData) { // získej paměť
        systemLogIn(*this);       // přihlas se
    }
    ~User() {
        systemLogOut(*this);      // odhlas se
        delete data;              // uvolni paměť
    }
private:
    UserData* data;
};
```

Zánik objektu

- Kdy objekt zaniká?
  - Lokální proměnná: když program opustí blok (`{}`), ve kterém objekt vznikl.
  - Datová položka třídy T: když zaniká instance třídy T, hned po provedení destruktoru ~T.
  - Objekt, který vznikl příkazem `new`: když zavoláme příkaz `delete` na ukazatel, který na objekt ukazuje.
  - Globální proměnná: když program opustí funkci `main`.
  - Dočasný objekt: po provedení řádku.
- Když zaniká více objektů najednou, tak zanikají přesně **v opačném pořadí**, než v jakém vznikly.

Příklady: pořadí vzniku a zániku (1/3)

```
struct Obj {
    Obj(int num) : mNum(num) { std::cout << mNum << ' '; }
    ~Obj() { std::cout << '~' << mNum << ' '; }
private:
    int mNum;
};

int main() {
    Obj o1 = 1;
    Obj o2(2);
    Obj o3{3};
}
```

```
1 2 3 ~3 ~2 ~1
```

Příklady: pořadí vzniku a zániku (2/3)

```
struct Obj {
    Obj(int num) : mNum(num) { std::cout << mNum << ' '; }
    ~Obj() { std::cout << '~' << mNum << ' '; }
private:
    int mNum;
};

int main() {
    Obj o0 = 0;

    for (int i = 1; i <= 3; i++) {
        Obj o = i;
    }
}
```

```
0 1 ~1 2 ~2 3 ~3 ~0
```

Příklady: pořadí vzniku a zániku (3/3)

```
struct Obj {
    Obj(int num) : mNum(num) { std::cout << mNum << ' '; }
    ~Obj() { std::cout << '~' << mNum << ' '; }
private:
    int mNum;
};

void foo(Obj o) {
    std::cout << "foo ";
    Obj o2 = 2;
}

int main() {
    foo(1);
}
```

```
1 foo 2 ~2 ~1
```

Kopírující konstruktor *(copy constructor)* (1/2)

- Konstruktor třídy T s parametrem typu `const T&` se nazývá **kopírující konstruktor** třídy T.
- Jeho úkolem je zkopírovat data do nového objektu.

```
struct Person {
    Person();
    Person(const std::string& name, int age);
    Person(const Person& rhs);
private:
    std::string name;
    int age;
};

Person::Person(const Person& rhs) : name(rhs.name),
                                    age(rhs.age)
{}
```

Kopírující konstruktor *(copy constructor)* (2/2)

- Kopírující konstruktor se zavolá v následujících situacích (o je typu `Obj`):

```
Obj o2 = o;
Obj o2(o);
Obj o2{o};
... new Obj(o);
... new Obj{o};
```

```
receivesObj(o);
... Obj(o) ...;
... Obj{o} ...;
... : o2(o) ...
```

Příklady: kopírující konstruktor (1/2)

```
struct Obj {
    Obj(char data) : mData(data), mNum(1) {
        std::cout << "char " << mData << '-' << mNum << '\n';
    }
    Obj(const Obj& rhs) : mData(rhs.mData), mNum(rhs.mNum + 1) {
        std::cout << "copy " << mData << '-' << mNum << '\n';
    }
    ~Obj() {
        std::cout << "destr " << mData << '-' << mNum << '\n';
    }
private:
    char mData;
    int mNum;
};
```

```
int main() {
    Obj o1{'A'};
    Obj o2 = o1;
}
```

```
char A-1
copy A-2
destr A-2
destr A-1
```

Příklady: kopírující konstruktor (2/2)

```
struct Obj {
    Obj(char data) : mData(data), mNum(1) {
        std::cout << "char " << mData << '-' << mNum << '\n';
    }
    Obj(const Obj& rhs) : mData(rhs.mData), mNum(rhs.mNum + 1) {
        std::cout << "copy " << mData << '-' << mNum << '\n';
    }
    ...
};
```

```
void receivesObj(Obj o) {
    Obj o2 = 'B';
}
int main() {
    Obj o1{'A'};
    receivesObj(o1);
    Obj o3(o1);
}
```

```
char A-1
copy A-2
char B-1
destr B-1
destr A-2
copy A-2
destr A-2
destr A-1
```

Kopírující přiřazení *(copy assignment)* (1/2)

- Metoda třídy T nazvaná `operator=` s parametrem typu `const T&` se nazývá **kopírující přiřazení.**
- Jejím úkolem je zkopírovat data do **již existujícího** objektu a vrátit referenci na `*this`.

```
struct Person {
    Person& operator=(const Person& rhs);
    ...
};

Person& Person::operator=(const Person& rhs) {
    name = rhs.name;
    age = rhs.age;
    return *this;
}
```

Kopírující přiřazení *(copy assignment)* (2/2)

- Kopírující přiřazení se zavolá **pouze** v případě, že objekt na levé straně rovnítka už existuje.

```
// (o, o2 jsou typu Obj)
o2 = o;
```

- Přítomnost rovnítka na řádku tedy nutně neznamená, že se jedná o kopírující přiřazení. Porovnejte:

```
Obj o2 = o; // kopírující konstruktor
o2 = o;     // kopírující přiřazení
```

Příklady: kopírující přiřazení (1/2)

```
struct Obj { ... // rozšíření Obj z příkladů na kop. konstr.
    Obj& operator=(const Obj& rhs) {
        std::cout << "forget " << mData << '-' << mNum << ", ";
        mData = rhs.mData;
        mNum = rhs.mNum * 10;
        std::cout << "assign " << mData << '-' << mNum << '\n';
        return *this;
    }
};
```

```
int main() {
    Obj o1 = 'A';
    Obj o2 = o1;
    o1 = o2;
}
```

```
char constr A-1
copy constr A-2
forget A-1, assign A-20
destr A-2
destr A-20
```

Příklady: kopírující přiřazení (2/2)

```
struct Obj { ... // rozšíření Obj z příkladů na kop. konstr.
    Obj& operator=(const Obj& rhs) {
        std::cout << "forget " << mData << '-' << mNum << ", ";
        mData = rhs.mData;
        mNum = rhs.mNum * 10;
        std::cout << "assign " << mData << '-' << mNum << '\n';
        return *this;
    }
};
```

```
int main() {
    Obj o1 = 'A';
    Obj o2('B');
    Obj o3{'C'};
    o1 = o2 = o3;
}
```

```
char constr A-1
char constr B-1
char constr C-1
forget B-1, assign C-10
forget A-1, assign C-100
destr C-1
destr C-10
destr C-100
```

Hluboká a mělká kopie (1/2)

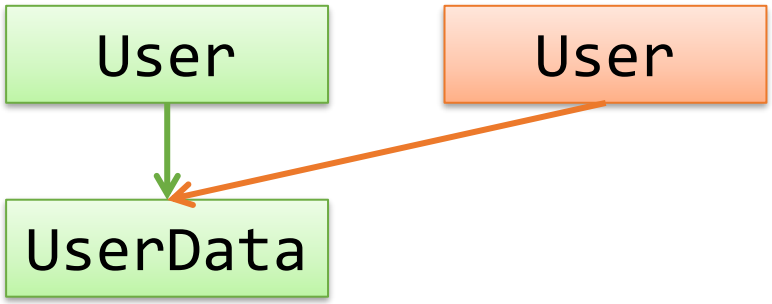
- Kopírování v C++ je zpravidla **hluboké**. Výsledkem kopírování je tzv. **hluboká kopie** originálu.
- To znamená, že vzniklá kopie je na originálu nezávislá.
- Operace provedené na kopii orginál nijak neovlivní.

Hluboká a mělká kopie (2/2)

- Mělké kopírování v C++ neprovádíme.
- Problém mělkých kopií je, že dva a více destruktorů se snaží uvolnit ty samé prostředky.
- U některých prostředků je to velká chyba, která způsobí, že program spadne.

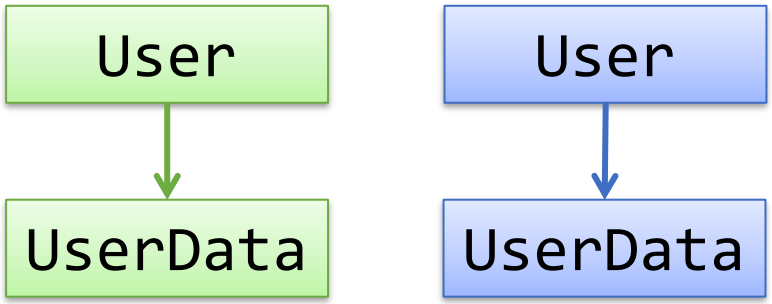
Příklad: mělká kopie

```
struct User {
    User() : data(new UserData) {}
    ~User() { delete data; } // destruktor uvolňuje prostředky
    User(const User& rhs) : data(rhs.data) {} // mělká kopie
private:
    UserData* data;
};

int main() {
    User u1;
    User u2{u1}; // kopírující konstruktor
} // chyba: dvojí smazání objektu "data", program spadne
```



Příklad: hluboká kopie

```
struct User {
    User() : data(new UserData) {}
    ~User() { delete data; } // destruktor uvolňuje prostředky
    User(const User& rhs) : data(new UserData{rhs.data}) {}
private:                          // ^^ hluboká kopie ^^
    UserData* data;
};

int main() {
    User u1;
    User u2{u1}; // kopírující konstruktor
} // v pořádku, každý destruktor smaže vlastní data
```



Přesuny (1/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?
- Jak zrychlit program při přesouvání?

Přesuny (2/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?

```
Obj o = returnsObj();
```

```
o = returnsObj();
```

Kopírujeme-li něco vráceného z funkce, originál bude bezprostředně smazán, a tak je přípustné ho "poškodit" za účelem zrychlení programu.

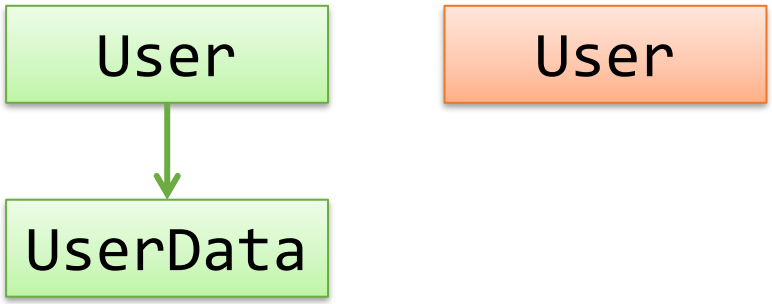
Přesuny (3/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?

```
Obj o = returnsObj();
```

```
o = returnsObj();
```

- Jak můžeme zrychlit program při přesouvání?
  1. namísto hluboké provedeme **mělkou** kopii



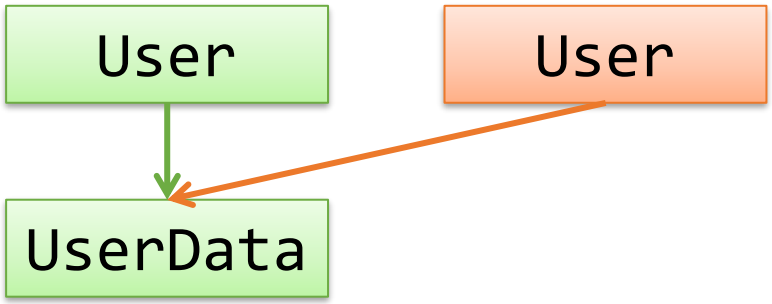
Přesuny (4/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?

```
Obj o = returnsObj();
```

```
o = returnsObj();
```

- Jak můžeme zrychlit program při přesouvání?
  1. namísto hluboké provedeme **mělkou** kopii



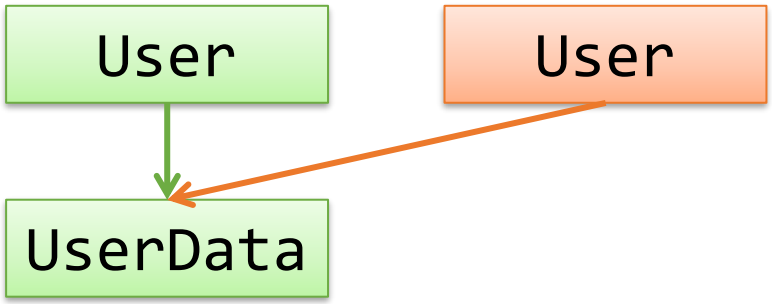
Přesuny (5/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?

```
Obj o = returnsObj();
```

```
o = returnsObj();
```

- Jak můžeme zrychlit program při přesouvání?
  1. namísto hluboké provedeme **mělkou** kopii
  2. vyprázdníme originál tak, aby nemohlo dojít k chybě



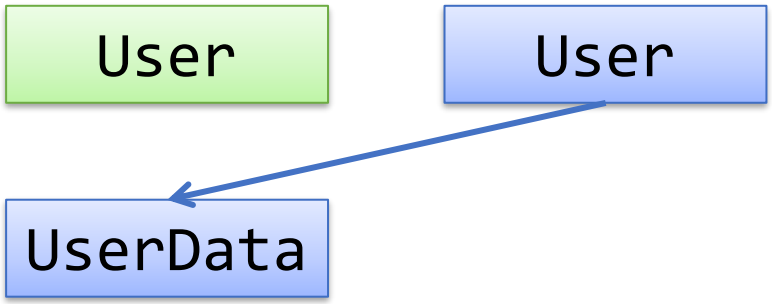
Přesuny (6/6)

- Přesouvání může zrychlit program. Je podobné kopírování, ale originál se při něm může změnit.
- V jaké situaci se něco takového může hodit?

```
Obj o = returnsObj();
```

```
o = returnsObj();
```

- Jak můžeme zrychlit program při přesouvání?
  1. namísto hluboké provedeme **mělkou** kopii
  2. vyprázdníme originál tak, aby nemohlo dojít k chybě



Přesunující konstruktor *(move constructor)*

- Konstruktor třídy T s parametrem typu T&& se nazývá **přesunující konstruktor** třídy T.
- Jeho úkolem je zkopírovat data do nového objektu, přičemž původní objekt může být změněn.

```
struct User {
    ...
    User(User&& rhs) : data(rhs.data) { // mělká kopie
        rhs.data = nullptr;             // vyprázdnění originálu
    }
    ...
};
```

Přesunující přiřazení *(move assignment)*

- Metoda třídy T nazvaná `operator=` s parametrem typu T&& se nazývá **přesunující přiřazení** třídy T.
- Jejím úkolem je zkopírovat data do již existujícího objektu, přičemž původní objekt může být změněn.

```
struct User {
    ...
    User& operator=(User&& rhs) {
        data = rhs.data;     // mělká kopie
        rhs.data = nullptr; // vyprázdnění originálu
        return *this;
    }
    ...
};
```

Kdy dojde k přesunu?

- K přesunu automaticky dochází v případě, když se pokusíme zkopírovat:
  - výsledek vrácený funkcí

```
Obj o = returnsObj();
```

```
new Obj{returnsObj()}
```

```
o = returnsObj();
```

```
return returnsObj();
```

  - objekt, na který bezprostředně aplikujeme std::move()

```
Obj o = std::move(o);
```

```
new Obj{std::move(o)}
```

```
o = std::move(o);
```

```
return std::move(o);
```

  - dočasný objekt
- Pokud jsme neposkytli přesunující operace, zavolá se v těchto případech operace kopírující.

Příklady: přesuny (1/2)

```
struct A {
    ~A() { std::cout << "destr\n"; }
    A() { std::cout << "default\n"; }
    A(int) { std::cout << "int\n"; }
    A(const A&) { std::cout << "copy\n"; }
    A(A&&) { std::cout << "move\n"; }
    A& operator=(const A&) { std::cout<<"copy=\n"; return*this;}
    A& operator=(A&&) { std::cout << "move=\n"; return *this; }
};
```

```
int main() {
    A a1{1};
    A a2 = std::move(a1);
    a1 = a2;
}
```

```
int
move
copy=
destr
destr
```

Příklady: přesuny (2/2)

```
struct A {
    ~A() { std::cout << "destr\n"; }
    A() { std::cout << "default\n"; }
    A(int) { std::cout << "int\n"; }
    A(const A&) { std::cout << "copy\n"; }
    A(A&&) { std::cout << "move\n"; }
    A& operator=(const A&) { std::cout<<"copy=\n"; return*this;}
    A& operator=(A&&) { std::cout << "move=\n"; return *this; }
};
```

```
A returnsA() {
    A a2(3);
    return a2;
}

int main() {
    A a1;
    a1 = returnsA();
}
```

```
default
int
move
destr
move=
destr
destr
```

Bez (N)RVO

```
default
int
move=
destr
destr
```

S (N)RVO

Konec