

Dynamika objektů

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, 2016

Programování v C++, A7B36PJC

03/2016, Lekce 5

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



Třída vs Instance

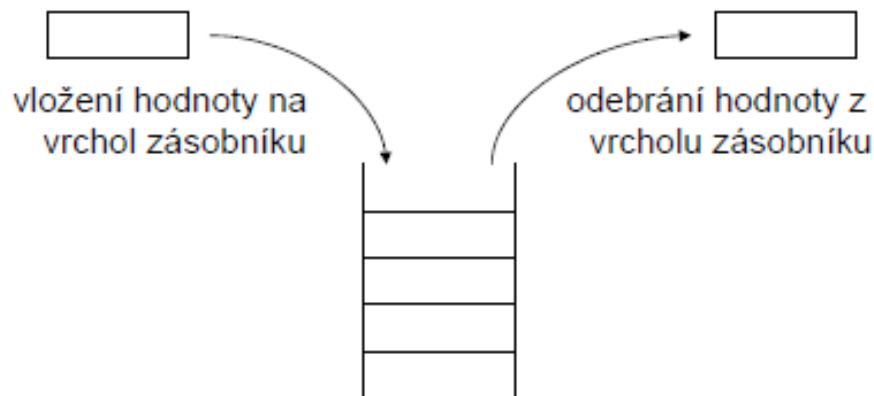
- Třída je abstraktní popis nějaké celku.
 - Data + operace nad daty
 - Například osoba může obsahovat řetězec – jméno
 - Zároveň definuje metody které umožní například získat příjmení
- Instance se tvoří z třídy a obsahuje konkrétní data
 - Instance může obsahovat jméno „Vincent van Gogh“
 - `getLastName()` umožní získat „van Gogh“

Proč objekty?

- **Zapouzdření** (kód je pohromadě se zpracovávajícími daty):
 - Uživatel nespolehá na detaily implementace.
 - Zvýšení bezpečnosti a robustnosti, zamezení nekorektního použití.
 - Usnadnění budoucích modifikací.
- **Polymorfismus** (objekty jsou zaměnitelné)
 - Volání metody na objektu je poslání zprávy, že se má něco vykonat.
 - Objekt sám rozhodne, jak na zprávu zareaguje.
 - Objekty různých typů mohou na stejnou zprávu reagovat různě.
- **Dědičnost:**
 - Tvorba nových tříd na základě stávajících – disciplinovaným způsobem.
 - Mohu přidat data a metody, případně měnit existující metody.

Zásobník

- Zásobník (stack) je abstraktní datový typ, který umožňuje vkládání a odebrání hodnot, přičemž naposledy vložená hodnota se odebere jako první.
- Zásobník je paměť typu LIFO (zkratka z angl. last-in first-out, poslední dovnitř, první ven).



- Základní operace:
 - vložení hodnoty na vrchol zásobníku, v angličtině obvykle push
 - odebrání hodnoty z vrcholu zásobníku, v angličtině obvykle pop
 - test na prázdnotu zásobníku, v angličtině obvykle is_empty

Deklarace třídy a definice metod

```
using zas_typ = int;

class zasobnik {
public:
    zasobnik(int max_prvku);
    zas_typ vezmi();
    void vloz(zas_typ prvek);
    bool je_prazdny();

private:
    int max_velikost;
    int aktualni_pozice;
    zas_typ* prvky;
};
```

```
zasobnik::zasobnik(int max_prvku){
    max_velikost = max_prvku;
    aktualni_pozice = 0;
    prvky = new
    zas_typ[max_prvku];
}

bool zasobnik::je_prazdny() {
    return aktualni_pozice == 0;
}
```

Datové položky, metody, viditelnost

Členská data a metody mají jednu ze 3 možných viditelností

- **Private** (soukromé) – pouze metody daného objektu mají přístup
- **Protected** (chráněné) – metody daného objektu a dědicích objektů mají přístup
- **Public** (veřejné) – volný přístup odkudkoliv

Výchozí viditelnost je private pro třídy (class) a public pro struct.

```
int main() {
    zasobnik z;
    //z.aktualni_pozice == 0; // nelze
    z.je_prazdny();           // OK
    //z.prvky[0] = 1;         // nelze
    z.vloz(1);                // OK
}
```

Členské funkce třídy

- Funkce uvedené v deklaraci třídy se jmenují členské funkce třídy (member functions) nebo metody
- Metody s viditelností **public** tvoří veřejné rozhraní třídy a dají se volat odkudkoliv
- Metody, které nechceme zpřístupňovat, mají viditelnost **private**.
- Metody třídy jsou zodpovědné za udržení invariant třídy.

```
void zasobnik::vloz(zas_typ prvek) {
    if (aktualni_pozice == max_velikost) {
        throw std::runtime_error("Zasobnik je plny");
    }
    prvky[aktualni_pozice++] = prvek;
}

bool zasobnik::je_prazdny() {
    return aktualni_pozice == 0;
}
```

Vložené členské metody (definice v deklaraci)

- Metoda může být v těle třídy nejen deklarována, ale i definována.
- Takové metodě se říká vložená (inline) a je implicitně označena `inline`
- Používá se pro jednoduché funkce.
- Ekvivalentní s použitím `inline` v deklaraci (uvnitř hlavičky).

```
class zasobnik {
public:
    zasobnik(int max_prvku);
    bool je_prazdny() { return aktualni_pozice == 0; }
};

// Stejně, jako:
inline bool zasobnik::je_prazdny() {
    return aktualni_pozice == 0;
}
```


Konstruktor

- Konstruktor je metoda, která vytvoří instanci třídy a nastaví její invarianty.
- Jmenuje se stejně jako třída a nemá návratový typ.
- Může být přetížen jako standardní funkce (metoda).
- Specifické signatury mají specifické jméno:
 - **T()** – tzv. základní (default) konstruktor,
 - **T(const T&)** – tzv. kopírující (copy) konstruktor,
 - **T(T&&)** – tzv. přesunující (move) konstruktor,
 - Více viz později.
- Konstruktor je volán vždy při vytvoření objektu.
- Konstruktor je volán pouze při vytvoření objektu.

Příklad

```
class zasobnik {  
public:  
    zasobnik(int max_prvku);  
    ...  
  
private:  
    int max_velikost;  
    int aktualni_pozice;  
    zas_typ* prvky;  
};
```

```
zasobnik::zasobnik(int max_prvku) {  
    max_velikost = max_prvku;  
    aktualni_pozice = 0;  
    prvky = new zas_typ[max_velikost];  
}
```

```
int main() {  
    zasobnik z1(100); // do zásobníku z1 se vejde 100 prvků  
    zasobnik z2(10); // do zásobníku z2 se vejde 10 prvků  
}
```

Konstruktor – inicializační sekce

- Probíhá před vykonáním těla konstruktoru.
- Přímá inicializace členských proměnných, důležité například pro konstantní proměnné a reference.
- Jednotlivé proměnné se inicializují v pořadí, v jakém jsou zmíněny v definici, ne v pořadí v jakém jsou zmíněny v inicializační sekci.
- Též se může použít pro předání argumentů konstruktoru rodiče.
- Doporučujeme v inicializační sekci inicializovat všechny proměnné.

```
zasobnik::zasobnik(int max_prvku):  
    max_velikost(max_prvku), aktualni_pozice(0),  
    prvky(new zas_typ[max_velikost])  
    {}
```

Základní konstruktor (default)

- Pokud programátor nevytvořil svůj konstruktor, je základní konstruktor definován překladačem.
- Pokud to jde, doporučujeme ho vždy vytvořit. Pokud programátor definoval jiný konstruktor, měl by definovat i základní konstruktor.
- Nebere žádné argumenty, ale může používat implicitní hodnoty argumentů.

```
class zasobnik {  
public:  
    zasobnik(int max_prvku = 20);  
    ...  
};  
  
int main() {  
    zasobnik z1; // do zasobniku se vejde 20 prvku  
}
```

Přetížení a delegace konstruktorů

- Konstruktory lze přetěžovat stejným způsobem jako funkce
- Konstruktory mohou volat ostatní konstruktory. (Delegace)
- Základní konstruktor lze snadno řešit jako delegující konstruktor

```
struct vec3 {  
    int x, y, z;  
    vec3(): vec3(0, 0, 0) {}  
    vec3(int x, int y): vec3(x, y, 0) {}  
    vec3(int x, int y, int z): x(x), y(y), z(z) {}  
};
```

```
int main() {  
    vec3 v1;           // Nulový vektor  
    vec3 v2(1, 2);    // Varianta s dvěma argumenty  
    vec3 v3(1, 2, 3); // Varianta s třemi argumenty  
}
```

Destruktor

- Destruktor je inverzní funkce ke konstrukturu, stará se o zrušení invariant.
- Jeho signatura je vždy $\sim T()$, pro dané T může existovat pouze jeden.
- Pokud není explicitně definován, kompilátor ho doplní.
- Pro každý zkonstruovaný objekt se někdy musí provést destruktork.
 - Pokud k tomu nedojde, je to chyba.

```
class zasobnik {  
    ...  
public:  
    ~zasobnik() {  
        delete[] prvky;  
    }  
};
```

Destruktor

- Destruktor nelze volat explicitně v programu.
- Je volán při ukončení platnosti objektu:
 - Globální nebo statický objekt – na konci programu.
 - Dynamický objekt – řídí uživatel (operátory new a delete).
 - Automatický objekt – ukončen na konci bloku, kde byl definován.

```
class zasobnik {  
    ...  
public:  
    ~zasobnik();  
};  
  
zasobnik::~~zasobnik() {  
    delete[] prvky;  
}
```

Automatické volání destruktorků

- Jak jsme si již řekli, automatické proměnné žijí na zásobníku.
- Programový zásobník funguje podobně jako klasický zásobník.
- Ve chvíli definice proměnné se spustí její konstruktor a po jeho proběhnutí je vložena na zásobník.
- Ve chvíli kdy skončí blok, je každá proměnná v něm definovaná vytažena ze zásobníku a je zavolán její destruktork.
- Toto probíhá zcela automaticky!

Animace

<Anim 1>

Příklad

<příklad 2>

RAII

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

RAII – Příklad

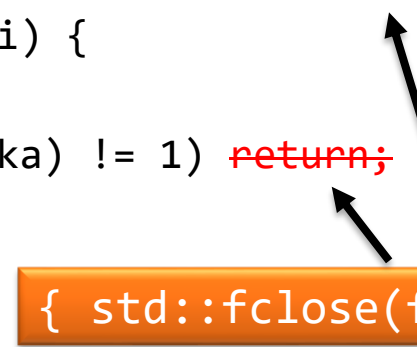
- Některé z příkazů `return` v následujícím kódu způsobí, že se chybně nezavolá `fclose`.
 - Které?

```
void Data::prectiZeSouboru(const char* nazevSouboru) {  
    FILE* fp = std::fopen(nazevSouboru, "r");  
    if (!fp) return;  
  
    int pocetPolozek;  
    if (std::fscanf(fp, "%d", &pocetPolozek) != 1) return;  
    for (int i = 0; i < pocetPolozek; ++i) {  
        int polozka;  
        if (std::fscanf(fp, "%d", &polozka) != 1) return;  
        vec.push_back(polozka);  
    }  
    std::fclose(fp);  
}
```

RAII – Příklad

- Některé z příkazů `return` v následujícím kódu způsobí, že se chybně nezavolá `fclose`.
 - Které?

```
void Data::prectiZeSouboru(const char* nazevSouboru) {  
    FILE* fp = std::fopen(nazevSouboru, "r");  
    if (!fp) return;  
  
    int pocetPolozek;  
    if (std::fscanf(fp, "%d", &pocetPolozek) != 1) return;  
    for (int i = 0; i < pocetPolozek; ++i) {  
        int polozka;  
        if (std::fscanf(fp, "%d", &polozka) != 1) return;  
        vec.push_back(polozka);  
    }  
    std::fclose(fp);  
}
```



The diagram shows an orange box containing the code `{ std::fclose(fp); return; }`. Two black arrows point from this box to the `return;` statements in the code above: one points to the `return;` after the `if (std::fscanf(fp, "%d", &pocetPolozek) != 1)` check, and the other points to the `return;` after the `if (std::fscanf(fp, "%d", &polozka) != 1)` check.

RAII – Příklad

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

void Data::prectiZeSouboru(const char* nazevSouboru) {
    FILE* fp = std::fopen(nazevSouboru, "r");
    if (!fp) return;
    FileSentry sentry(fp);
    int pocetPolozek;
    if (std::fscanf(fp, "%d", &pocetPolozek) != 1) return;
    for (int i = 0; i < pocetPolozek; ++i) {
        int polozka;
        if (std::fscanf(fp, "%d", &polozka) != 1) return;
        vec.push_back(polozka);
    }
    std::fclose(fp);
}
```

Odbočka k new a delete

- Operátor **new** alokuje paměť a spustí konstruktor typu.
 - Uživatel ale dostane pointer na typ a destruktor pointeru je prázdná operace.
- Operátor **delete** paměť dealokuje a spustí destruktor typu.
 - Pokud uživatel ale o pointer přijde, nebo na něj zapomene, destruktor se nikdy nezavolá.
 - Této situaci říkáme **memory leak**.
- Nejde to lépe?
 - 😊

Příklad: memory leak

```
zasobnik* vytvorZRetezce(const char* str) {
    int len = std::strlen(str);
    zasobnik* z = new zasobnik(len);
    for (int i = 0; i < len; i++) {
        z->vloz(str[i]);
    }
    return z;
}
```

```
void vypisObracene(const char* str) {
    zasobnik* zas = vytvorZRetezce(str);

    while (!zas->je_prazdny()) {
        std::cout << (char)zas->vezmi();
    }
    std::cout << '\n';
}
```


Příklad: memory leak

```
zasobnik* vytvorZRetezce(const char* str) {  
    int len = std::strlen(str);  
    zasobnik* z = new zasobnik(len);  
    for (int i = 0; i < len; i++) {  
        z->vloz(str[i]);  
    }  
    return z;  
}
```

```
void vypisObracene(const char* str) {  
    zasobnik* zas = vytvorZRetezce(str);  
  
    while (!zas->je_prazdny()) {  
        std::cout << (char)zas->vezmi();  
    }  
    std::cout << '\n';  
}
```

← delete zas;

`std::make_unique`, `std::unique_ptr`

- Co kdyby operátor `new` nevracel `T*`, ale typ který po sobě uklidí?
- **`std::unique_ptr`** je typ ze standardní knihovny, který se chová jako pointer, ale jeho destruktork volá **`delete`**.
- **`std::make_unique`** vrací **`std::unique_ptr`**.
- Při použití **`unique_ptr`** není možné zapomenout na dealokaci.

Příklad: memory leak

```
zasobnik* vytvorZRetezce(const char* str) {  
    int len = std::strlen(str);  
    zasobnik* z = new zasobnik(len);  
    for (int i = 0; i < len; i++) {  
        z->vloz(str[i]);  
    }  
    return z;  
}
```

```
void vypisObracene(const char* str) {  
    zasobnik* zas = vytvorZRetezce(str);  
  
    while (!zas->je_prazdny()) {  
        std::cout << (char)zas->vezmi();  
    }  
    std::cout << '\n';  
}
```

← delete zas;

Příklad: unique_ptr

```
std::unique_ptr<zasobnik> vytvorZRetezce(const char* str) {  
    int len = std::strlen(str);  
    std::unique_ptr<zasobnik> z(new zasobnik(len));  
    for (int i = 0; i < len; i++) {  
        z->vloz(str[i]);  
    }  
    return z;  
}
```

```
void vypisObracene(const char* str) {  
    std::unique_ptr<zasobnik> zas = vytvorZRetezce(str);  
  
    while (!zas->je_prazdny()) {  
        std::cout << (char)zas->vezmi();  
    }  
    std::cout << '\n';  
}
```

Příklad: unique_ptr

```
std::unique_ptr<zasobnik> vytvorZRetezce(const char* str) {  
    int len = std::strlen(str);  
    auto z = std::make_unique<zasobnik>(len);  
    for (int i = 0; i < len; i++) {  
        z->vloz(str[i]);  
    }  
    return z;  
}
```

```
void vypisObracene(const char* str) {  
    auto zas = vytvorZRetezce(str);  
  
    while (!zas->je_prazdny()) {  
        std::cout << (char)zas->vezmi();  
    }  
    std::cout << '\n';  
}
```

```

struct vec3 {
    int x, y, z;
    vec3() : x(0), y(0), z(0) {}
    vec3(int x, int y) : x(x), y(y), z(0) {}
    vec3(int x, int y, int z) : x(x), y(y), z(z) {}
    void tisk() { std::cout << "(" << x << "," << y << "," << z << ")\n";
}
};

```

```

int main() {
    auto v1 = std::make_unique<vec3>();
    auto v2 = std::make_unique<vec3>(11, 22);
    auto v3 = std::make_unique<vec3>(33, 44, 55);
    auto v4 = std::make_unique<vec3[]>(4);
    v1->tisk();
    v2->tisk();
    v3->tisk();
    for (int i = 0; i < 4; i++) {
        v4[i].tisk();
    }
}

```

(0,0,0)
(11,22,0)
(33,44,55)
(0,0,0)
(0,0,0)
(0,0,0)
(0,0,0)

Příklad: memory leak

```
bool prectiPalindrom(FILE* fp) {
    int length;
    if (std::fscanf(fp, "%d", &length) != 1) return false;

    char* str = new char[length + 1];
    std::fscanf(fp, "%s", str);
    int first = 0;
    int last = length - 1;
    for (; first <= last; first++, last--) {
        if (str[first] != str[last]) {
            return false;
        }
    }


    delete[] str;
    return true;
}
```

Příklad: memory leak

```
bool prectiPalindrom(FILE* fp) {
    int length;
    if (std::fscanf(fp, "%d", &length) != 1) return false;

    char* str = new char[length + 1];
    std::fscanf(fp, "%s", str);
    int first = 0;
    int last = length - 1;
    for (; first <= last; first++, last--) {
        if (str[first] != str[last]) {
            return false;
        }
    }

    delete[] str;
    return true;
}
```



Příklad: unique_ptr

```
bool prectiPalindrom(FILE* fp) {
    int length;
    if (std::fscanf(fp, "%d", &length) != 1) return false;

    auto str = std::make_unique<char[]>(length + 1);
    std::fscanf(fp, "%s", str.get());
    int first = 0;
    int last = length - 1;
    for (; first <= last; first++, last--) {
        if (str[first] != str[last]) {
            return false;
        }
    }

    delete[] str;
    return true;
}
```

Vylepšení zásobníku – `unique_ptr`

```
class zasobnik {  
public:  
    ...  
private:  
    ...  
    std::unique_ptr<zas_typ[]> prvky;  
};  
  
zasobnik::zasobnik(int max_prvku) :  
    max_velikost(max_prvku), aktualni_pozice(0),  
    prvky(std::make_unique<zas_typ[]>(max_velikost))  
    {}  
  
zasobnik::~zasobnik() { ... } // Destruktor už není potřeba
```

Konstantní členská funkce – jen dotaz

- Nemění stav objektu, jedná se jen o dotaz.
- Značí se klíčovým slovem `const` za seznamem argumentů.
- Může se volat i na konstantním objektu.

```
class zasobnik {
public:
    bool je_prazdny() const;
    ...
};
bool zasobnik::je_prazdny() const {
    return aktualni_pozice == 0;
}
void funkce(const zasobnik& z) {
    z.je_prazdny(); // ok, volame const metodu
    //z.vezmi()     // nelze, vezmi() neni const
}
```

Konstanta platná ve třídě

- Existují dva způsoby jak zadefinovat konstantu pro třídu. Starý způsob přes anonymní enumerátor a nový, pomocí static const proměnné.
- Použití enumerátoru umožňuje definování pouze integrálního typu.
- Preferujte static const proměnné.

```
class viewport {
public:
    enum konstanty { max_sirka = 1920, max_vyska = 1080, };
    static const double pomer;
    void zmen_vysku(int nova_vyska) {
        m_vyska = nova_vyska;
        m_sirka = static_cast<int>(std::ceil(m_vyska * pomer));
    }
    int m_vyska, m_sirka;
};
// Mimo třídu kvůli ODR
const double viewport::pomer = static_cast<double>(max_sirka) /
max_vyska;
```

Ukazatel this

- Skrytý argument všech metod na objektu.
- Ukazuje na danou instanci.
- *this lze použít jako referenci na objekt samotný.
- Umožňuje zjistit, zda dvě reference odkazují na stejný objekt.

```
struct dummy {
    dummy() {
        std::cout << "Creating dummy object at: " << std::hex
                   << (uintptr_t(this) & 0xFFFF) << '\n';
    }
    ~dummy() {
        std::cout << "Destroying dummy object at: " << std::hex
                   << (uintptr_t(this) & 0xFFFF) << '\n';
    }
};
```

Pole objektů

- Při vytvoření pole dojde k zavolání základních konstruktorů prvků.
- Při destrukci pole dojde k zavolání destruktorů prvků.
- Konstrukce probíhá zleva doprava, destrukce zprava doleva.

```
int main() {  
    auto foo = std::make_unique<dummy[]>(4);  
}
```

```
>>> Creating dummy object at address: 60e8  
>>> Creating dummy object at address: 60e9  
>>> Creating dummy object at address: 60ea  
>>> Creating dummy object at address: 60eb  
>>> Destroying dummy object at address: 60eb  
>>> Destroying dummy object at address: 60ea  
>>> Destroying dummy object at address: 60e9  
>>> Destroying dummy object at address: 60e8
```

Organizace projektu v C++ --- třídy

- Třídy se obvykle rozdělují do dvou souborů
 - Hlavičkový soubor s deklarací třídy, s příponou .hpp, jméno souboru dle třídy
 - Implementační soubor s příponou .cpp a jménem dle třídy
- Například třída zasobnik by se rozdělila do „zasobnik.hpp“ a „zasobnik.cpp“

```
// zasobnik.h
#ifndef ZASOBNIK_H
#define ZASOBNIK_H
using zas_typ = int;

class zasobnik {
public:
    zasobnik(int max_prvku = 20);
    ...

private:
    int max_velikost;
    int aktualni_pozice;
    std::unique_ptr<zas_typ[]> prvky;
};
#endif
```

```
// zasobnik.cpp
#include "zasobnik.h"

bool zasobnik::je_prazdny() const {
    return aktualni_pozice == 0;
}

zasobnik::zasobnik(int max_prvku):
    max_velikost{ max_prvku },
    aktualni_pozice{ 0 }, prvky{
std::make_unique<zas_typ[]>(max_velikost)}
{}
...
```

The End