Dynamika objektů

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík

Programování v C++, A7B36PJC
09/2015, Lekce 6

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

Dynamika objektů

- Konstruktor, destruktor, životní cyklus proměnné, vlastnictví prostředků, RAII, std::unique_ptr, std::shared_ptr
- Implementace C++ style vektoru, cvičení:

```
(rozhraní Vector na cvičení 6)
class Vector {
public:
   Vector();
   ~Vector();
   void push_back(double);
   double& back();
   void pop_back();
   double& at(int);
   int size();
};
```

Třídy a jejich instance

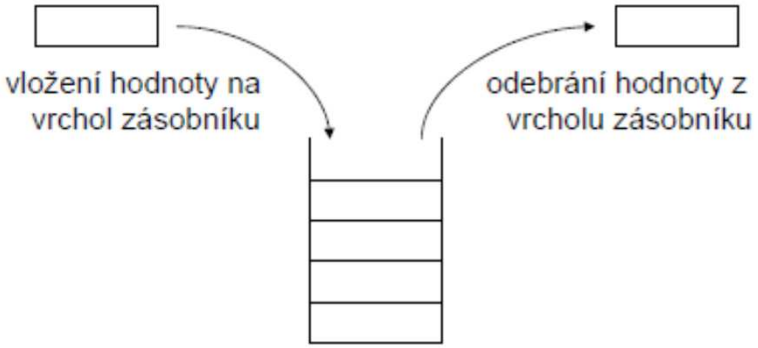
- Objekty s podobnými vlastnostmi sdružujeme do tříd.
- Příslušnost objektu do třídy vyjadřujeme tvrzením, že objekt je instancí dané třídy (termíny objekt a instance třídy jsou synonyma).
- Třída je „forma“ na vytváření svých instancí.
- Třída je zobecněný klasický datový typ:
  - Vedle množiny hodnot definuje i množinu přípustných operací nad těmito hodnotami - třída definuje vlastnosti a schopnosti svých instancí.

Základní pilíře OOP

- **Zapouzdření** (kód je pohromadě se zpracovávanými daty):
  - Uživatel nespoléhá na detaily implementace.
  - Zvýšení bezpečnosti a robustnosti, zamezení nekorektního použití.
  - Usnadnění budoucích modifikací.
  - Objekt může obsahovat jiné objekty.
- **Polymorfismus**
  - Volání metody na objektu je poslání zprávy, že chceme 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 odebírání 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 prázdnost 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

- Datové položky třídy popisují stav objektu.
- Metody určují povolené operace nad objektem.

K určení přístupu ke členským datům a metodám slouží :

- **Private** (soukromé) – pouze členské metody daného objektu mohou přistoupit k položkám.
- **Protected** (chráněné) – jako private, přístup ale mají také členské metody tříd, které z této třídy dědí.
- **Public** (veřejné) – položky jsou přístupné ze všech míst v programu, přes identifikátor objektu a tečkovou notaci.

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
    return 0;
}
```

Členské metody třídy

- V deklaraci třídy se uvádějí deklarace členských metod.
- Členské metody mohou přistupovat k datům i ostatním metodám, bez ohledu na jejich viditelnost.
- Členské metody s viditelností **`public`** tvoří veřejné rozhraní třídy a jsou přístupné ze všech částí programu.
- Metody, které nechceme zpřístupňovat, mají viditelnost **`private`**.
- Členské metody v části **`private`** jsou přístupné pouze z ostatních členských metod (jedná se o pomocné funkce).
- Metody třídy jsou zodpovědné za udržení invariant třídy.

```
void zasobnik::vloz(zas_typ prvek) {
    if (aktualni_pozice == max_velikost) {
        std::cerr << "Vkladam do plneho zasobniku\n";
        exit(1);  // Ošklivé, ale snadné řešení
    }
    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 její obsah je obvykle vložen přímo do místa volání. (Není to ale zaručeno.)
- Používá se pro funkce, kde cena za volání funkce je blízká ceně jejího provedení. (A tudíž jejím vložením do místa volání dojde k signifikatnímu urychlení běhu programu.)
- Tohoto se dá docílit i použitím klíčového slova `inline` v místě definice funkce.

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

// Stejné, 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 přetížena jako standardní funkce (metoda).
- Specifické signatury mají specifické jméno:
  - `T()` – tzv. default (standardní) konstruktor,
  - `T(const T&)` – tzv. copy (kopírující) konstruktor,
  - `T(T&&)` – tzv. move (přesunující) 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.
- Umožňuje přímou inicializaci 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.
- Přímá inicializace členských dat.
- Další využití je předání argumentů konstruktoru rodiče.
- Doporučujeme v inicializační sekci inicializovat všechny proměnné, které se dají.

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

Základní konstruktor (default)

- Základní konstruktor je automaticky vytvořen překladačem, pokud uživatel nevytvořil žádný jiný konstruktor.
- Pokud uživatel vytvořil jiný konstruktor, musí vytvořit i základní konstruktor. Pokud to jde, doporučujeme ho vždy vytvořit.
- 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 konstruktorů

- Konstruktory lze přetěžovat, musí se ale poznat, který se má vyvolat.
- Pokud vytvořím vlastní variantu konstruktoru – překladač nevytvoří základní.
- Pro složitější třídy je radno vždy vytvořit základní konstruktor.
- Základní konstruktor lze řešit jako variantu konstruktoru s implicitními argumenty.

```
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) {}
};
```

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

Destruktor

- Destruktor je inverzní funkce ke konstruktoru, stará se o zrušení invariant.
- Jeho signatura je vždy ~T(), pro dané T může existovat jeden.
- Pokud není explicitně definován, kompilátor ho doplní.
- Pro každé provedení konstruktoru se někdy musí provést destruktor.
  - 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;
}
```

K čemu to všechno je?

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

Příklad

<příklad 1>

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.

Příklad

<příklad 2>

RAII – Příklad

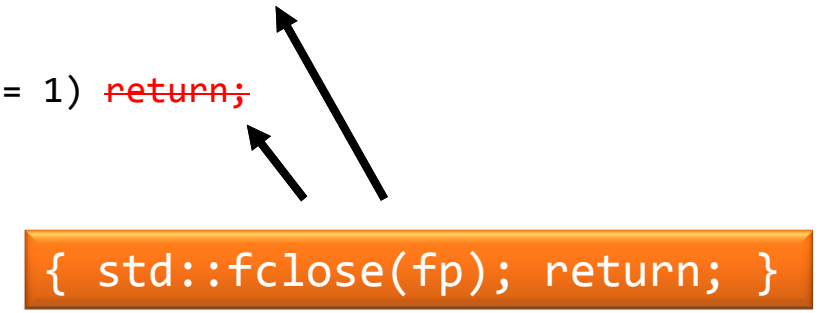
- Některé z příkazů `return` v následujícím kódu způsobí, že se chybně nezavolá `fclose`.
  - Které?

```
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);
}
```



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 destruktor 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++) {
        uptr.get()->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;
}
```

delete[] str;

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 triko {
public:
    enum velikost {
        S,
        M,
        L,
        XL
    };
 static const int i = 1;
 int velikost;
};
```

```
int main() {
    triko t;
    t.velikost = triko::M;
}
```

Ukazatel this

- Každý objekt má ukazatel this – obsahuje adresu tohoto objektu – ukazuje na něj.
- Každá členská funkce jej může použít (skrytý argument všech metod).
- Použit při konfliktu jmen členských dat a argumentu metody.
- *this lze použít jako odkaz na objekt jako celek.
- Umožňuje zjistit, zda dvě reference(pointery) odkazují na stejný objekt.

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

Pole objektů

- Objekt je uživatelský typ - proto lze mít pole objektů.
- Při vytvoření pole dojde k zavolání základních konstruktorů prvků.
- Při zničení pole dojde k zavolání destruktorů prvků.
- Konstrukce probíhá zleva doprava, destrukce zprava doleva.

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

```
>>> 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

- Hlavičkový soubor – deklarace třídy, případně vložené funkce – přípona .h, jméno dle jmena třídy: *trida.h*
- Implementační soubor – se stejným jménem jako hlavička – definice třídy a implementace metod: *trida.cpp*

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

class zasobnik {
public:
    zasobnik(int max_prvku = 20);
    int vezmi();
    void vloz(zas_typ prvek);
    bool je_prazdny() const;

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)(
{}
...
```

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

int main() {
    zasobnik z;
}
```

The End