Základy programování v C++ 21. cvičení

Zuzana Petříčková

17. prosince 2018

Přehled

Letmý úvod do objektového programování (v C++)

Softwarový objekt

- model nějaké části reálného světa (např. **Pepa**)
- objekty řadíme do **tříd** (např. **člověk, zaměstnanec, zlomek,...**)

Základní myšlenka

- **Datový typ** (jak ho známe, např. **int**)
  - je určen množinou přípustných hodnot a množinou operací, které nad touto množinou lze provádět
- **Objektový datový typ** (**třída**)
  - definuje mnzožinu hodnot
  - navíc definuje i operace nad touto množinou

Letmý úvod do objektového programování (v C++)

Datový typ struktura

- přímo obsahuje (definuje) datové složky
- (bokem) definujeme podprogramy (funkce), které s proměnnými daného typu pracují

Objektový datový typ (třída)

- přímo obsahuje (definuje) datové složky
- přímo obsahuje (definuje) i operace nad objekty dané třídy

Terminologie

- **třída (class)** ... objektový datový typ
- **instance (objekt, object)** ... proměnná objektového typu

Složky třídy

- **atribut** ... datová složka
- **metoda** ... funkce
  - **konstruktor** ... postará se o vytvoření a inicializaci nové instance
  - **destruktor** ... postará se o zrušení instance

1. **instanční** ... metoda pro práci s jednotlivými instancemi
2. **třídní** ... metoda pro práci s třídou jako celek

stručně:

- na rozdíl od struktury definujeme u třídy zároveň **atributy** i **metody**
- **třída** ale přináší další nové možnosti

Příklad ... třída jako rozšíření struktury o metody

```
struct Zamestnanec
{
    unsigned int ID = 0;
    string jmeno = "";
    void vypis()
    {
        cout << ID << " " << jmeno << endl;
    }
};

int main()
{
    Zamestnanec z, *pz;
    z.vypis();
    z.ID = 1;
    z.jmeno = "Josef";
    z.vypis();
    pz = new Zamestnanec;
    pz->ID = 2;
    pz->jmeno = z.jmeno;
    pz->vypis();
    delete pz;
}
```

Vlastnosti objektových typů (nové možnosti)

Zapouzdření (ukrývání implementace, encapsulation)

- datové složky definujeme současně s metodami
- můžeme omezit přístup k určitým datovým složkám
- **pravidlo**: k datovým složkám přistupovat pouze prostřednictvím metod

Dědění (dědičnost, inheritance)

- možnost od jedné třídy (**předek**) odvodit jinou třídu (**potomek**)
- potomek může zastoupit předka
- potomek **zdědí** metody a atributy předka, některé metody může překrýt, další metody a atributy může doplnit

Polymorfismus (mnohotvárnost)

- práce s instancemi neznámého typu (předek? / potomek?)
- určení typu instance za běhu programu

Deklarace objektového typu (prozatím bez dědění)

- klíčová slova **class, struct**

```
class identifikator // struct identifikator
{
    <seznam sekci>
}
```

- tělo třídy je rozděleno do sekcí s různou specifikací přístupu (**public, protected, private**)
- **sekce:**

```
<specifikace pristupu>
<seznam atributu>
<seznam metod>
```

Deklarace objektového typu (prozatím bez dědění)

Přístup ke složkám instancí

- **public** ... veřejné složky (mohou je používat všechny části programu)
- **private** ... soukromé složky (přístupné pouze metodám třídy a tzv. **přátelům**)
- **protected** ... chráněné složky (přístupné pouze metodám třídy a v potomcích)

Rozdíl mezi struct a class

- **struct** ... přístup je implicitně nastaven jako **public**
- **class** ... přístup je implicitně nastaven jako **private**

Příklad

```
class Zamestnanec
{
  public:
    void vypis()
    {
     // cout << ID << " " << jmeno << " " << prijmeni
     //         << " " << plat << endl;
        cout << this->ID << " " << this->jmeno << " "
             << this->prijmeni << " " << this->plat << endl;
    }
   private:
    unsigned int ID = 0;
    string jmeno = "";
    string prijmeni = "";;
    unsigned int plat = 0;
};
```

přístup k datovým i funkčním složkám instance přímo nebo pomocí ukazatele **this**

Deklarace objektového typu

definiční deklarace metod mohou být

1. přímo v těle třídy
2. mimo tělo třídy
   - v těle třídy je jen informativní deklarace
   - definiční deklarace je v tomto případě kvantifikovaná názvem třídy (jinak by překladač nevědel, ke které třídě metoda patří)
   - pro třídy s mnoha složkami je to přehlednější varianta

```
void Zamestnanec::vypis()
{
  cout << ID << " " << jmeno << " " << prijmeni
       << " " << plat << endl;
}
```

Příklad ... pokračování

```
class Zamestnanec
{
  public:
    void vypis();
  private:
    unsigned int ID = 0;
    string jmeno = "";
    string prijmeni = "";;
    unsigned int plat = 0;
};
void Zamestnanec::vypis()
{
  cout << ID << " " << jmeno << " " << prijmeni
          << " " << plat << endl;
}
```

Zapouzdření

Zapouzdření (encapsulation)

- datové složky definujeme současně s metodami
- můžeme omezit přístup k určitým datovým složkám
- **pravidlo**: k datovým složkám přistupovat důsledně prostřednictvím (přístupových) metod
- **proč?**:
  - ukrytí implementace
  - bezpečnost (třída má svoje data pod kontrolou)
  - úspora pozdější práce (snadno mohu datovou reprezentaci časem změnit)

Zapouzdření

Další pojmy

- **Vlastnost** (feature)
  - atribut doplněný o tzv. přístupové metody
- **Přístupová metoda** (accessor)
  - metoda, která nastavuje nebo vrací hodnotu atributu (**setter, getter**)
- **Rozhraní třídy** (interface)
  - seznam **veřejných** metod a **veřejných** datových složek spolu s jejich popisem

Správné zapouzdření $\rightarrow$ pokud změním implementaci třídy a zachovám přitom její rozhraní, nemusím měnit jiné části programu

Zapouzdření ... příklad

```
class Vektor
{
    float x=0;
    float y=0;
public:
    void nastavX(float _x)
    {  // setX
        this->x = _x;
    }
    float vratX() // getX
    {
        return this->x;
    }
    void nastavY(float _y)
    {   // setY
        this->y = _y;
    }
    float vratY() // getY
    {
        return this->y;
    }
};
```

```
int main()
{
    Vektor v;
    v.nastavX(1);
    v.nastavY(2);
    cout << v.vratX() << " "
         << v.vratY() << endl;
    return 0;
}
```

Přístupová práva ... doplnění

Spřátelená funkce

- není to metoda třídy, ale má při přístupu ke složkám stejná práva jako metody:
  - může přistupovat k soukromým i chráněným složkám třídy
- deklarace **friend** kdekoliv v těle třídy (specifikace přístupu zde nehraje roli):

```
friend informativni_deklarace_funkce;
friend definicni_deklarace_funkce;
friend class jmeno_tridy; // vsechny metody tridy jsou spratelene
```

Spřátelená funkce ... příklad

```
class Vektor
{
    float x=0;
    float y=0;
public:
    ...
    friend float vypis(Vektor &v);
    friend float delka(Vektor &v)
    {
        return sqrt(v.x*v.x + v.y*v.y);
    }
};

void vypis(Vektor &v) {
        cout << v.x << " " << v.y << endl;
}
```

```
int main()
{
    Vektor v;
    v.nastavX(1);
    v.nastavY(2);
    cout << v.vratX() << " "
         << v.vratY() << endl;
    vypis(v);
    cout << delka(v) << endl;
    return 0;
}
```

.

Konvence při práci se třídami

- každé třídě odpovídá jeden hlavičkový a jeden zdrojový soubor (oba pojmenované stejně jako třída)
  - hlavičkový soubor ... deklarace třídy (obsahuje pouze informativní deklarace metod)
  - zdrojový soubor ... definiční deklarace metod
- Většina moderních vývojových nástrojů umožňuje vytvářet nové třídy takto rozdělené do souborů ,,v jednom kroku”

<**Ukázka ve Visual Studiu** >

Konstruktor a destruktor

- speciální metody, které slouží (k vytvoření a) inicializaci instance třídy (**konstruktor**) a k zrušení instance třídy (**destruktor**)
- nevoláme je přímo, volají se automaticky při vzniku / zániku instance třídy

Co bývá v konstruktoru:

- inicializace datových složek
- vytvoření (alokace) dynamických datových složek
- příp. otevření souboru ap.

Co bývá v destruktoru:

- zrušení (dealokace) dynamických datových složek
- příp. zavření souboru ap.

Konstruktor

- metoda, která slouží k (vytvoření a) inicializaci instance
- konstruktor nelze volat přímo, je volán automaticky:
  - při vytvoření instance
  - při předávání parametrů objektového typu hodnotou
  - při konverzích
- konstruktor se jmenuje stejně jako identifikátor třídy
- jeho deklarace neobsahuje typ návratové hodnoty
- konstruktor může mít parametry libovolného typu s výjimkou své třídy (ale může mít jako parametr referenci na svou třídu)
- třída může mít několik konstruktorů
- pokud nedeklarujeme žádný konstruktor, překladač vytvoří implicitní konstruktor bez parametrů s prázdným tělem a implicitní tzv. kopírovací konstruktor

Konstruktor ... příklad

```
class Vektor
{
    float x; // konstruktor nelze kombinovat s float x=0;
    float y;
  public:
    ...
    Vektor(float xx, float yy)
    {
        x = xx; // this->x = xx;
        y = yy;
    }
    Vektor()
    {
        x = 0;
        y = 0;
    }
};
```

Příklad na procvičení I : Zlomek

- Přeměňte strukturu Zlomek z cvičení na třídu se soukromými složkami **citatel** a **jmenovatel**.
  - Deklarace třídy bude v hlavičkovém souboru Zlomek.h a definiční deklarace metod budou v souboru Zlomek.cpp.
  - Přidejte Zlomku konstruktor se dvěma parametry (hodnota čitatele a hodnota jmenovatele).
  - Implementujte jednotlivé přístupové metody (gettery a settery) pro Zlomek (konstruktor i settery budou kontrolovat, že jmenovatel je nenulový a převedou zlomek do základního tvaru ap.).
  - Původní funkce nad jedním Zlomkem změňte na metody (které z nich by měly být veřejné a které soukromé?)
  - Funkce nad dvěma Zlomky změňte tak, aby správně a efektivně využívaly přístupové metody.
  - Zkuste implementovat varianty výpočtů nad zlomky ve tvaru:

```
void Zlomek::pricti(const Zlomek &a);// pricte zlomek k akt. insta
void Zlomek::vynasob(const Zlomek &a);// vynasobi aktualni instanc
...                                   // zlomkem
```

Konstantní atributy a metody

Konstantní metoda

- označení metody, která nemění datové složky instance (správně by tak měly být označeny všechny gettery):

```
void Vektor::vratX() const
{
    return x;
}
```

Pravidla

- z konstantní metody lze volat jen konstantní metody
- pro konstantní instance lze volat pouze konstantní metody

Konstruktor a destruktor

- speciální metody, které slouží (k vytvoření a) inicializaci instance třídy (**konstruktor**) a k zrušení instance třídy (**destruktor**)
- nevoláme je přímo, volají se automaticky při vzniku / zániku instance třídy

Co bývá v konstruktoru:

- inicializace datových složek
- vytvoření (alokace) dynamických datových složek
- příp. otevření souboru ap.

Co bývá v destruktoru:

- zrušení (dealokace) dynamických datových složek
- příp. zavření souboru ap.

Konstruktor

- metoda, která slouží k (vytvoření a) inicializaci instance
- konstruktor nelze volat přímo, je volán automaticky:
  - při vytvoření instance
  - při předávání parametrů objektového typu hodnotou
  - při konverzích
- konstruktor se jmenuje stejně jako identifikátor třídy
- jeho deklarace neobsahuje typ návratové hodnoty
- konstruktor může mít parametry libovolného typu s výjimkou své třídy (ale může mít jako parametr referenci na svou třídu)
- třída může mít několik konstruktorů
- pokud nedeklarujeme žádný konstruktor, překladač vytvoří implicitní konstruktor bez parametrů s prázdným tělem a implicitní tzv. kopírovací konstruktor

Konstruktor ... příklad

```
class Vektor
{
    float x; // konstruktor nelze kombinovat s float x=0;
    float y;
  public:
    ...
    Vektor(float xx, float yy)
    {
        x = xx; // this->x = xx;
        y = yy;
    }
    Vektor()
    {
        x = 0;
        y = 0;
    }
};
```

Konstruktor ... příklad (defaultní hodnoty parametrů)

```
class Vektor
{
    float x; // nelze kombinovat s float x=0;
    float y;
  public:
    ...
    Vektor(float xx=0, float yy=0)
    {
        x = xx; // this->x = xx;
        y = yy;
    }
};
int main()
{
    Vektor x, y(3), z(2,1);
    Vektor *px = new Vektor; // new Vektor();  new Vektor(2);
                             // new Vektor(3,4);
    delete px;
}
```

Inicializační část konstruktoru

- alternativní způsob inicializace (nestatických) atributů:

```
Vektor::Vektor() : x(0), y(0)
{
    ...
}

Vektor::Vektor(float xx, float yy) : x(xx), y(yy)
{
    ...
}
```

inicializační část:

- je od hlavičky oddělená dvojtečkou
- proběhne před vstupem do těla konstruktoru
- konstantní datové složky lze nastavit jen v inicializační části

Destruktor

- speciální metoda, která slouží k zrušení instance:
  - uvolnění dynamicky alokované paměti
  - uzavření souborů, s nimiž třída pracuje apod.
  - má i své skryté úkoly
- destruktor je volán automaticky při zániku instance
- jmenuje se stejně jako identifikátor třídy, před identifikátorem je znak
- jeho deklarace neobsahuje typ návratové hodnoty a nesmí mít parametry
- každá třída může mít pouze jeden destruktor

```
Vektor::~Vektor()
{
    ...
}
```

Destruktor

- destruktor je volán automaticky při zániku instance:
  - lokální instance třídy ... destruktor je volán na konci bloku, v němž je instance deklarovaná
  - globální a statické lokální instance třídy ... destruktor je volán po ukončení funkce main()
  - dynamická instance třídy ... operáor delete nejprve zavolá destruktor a pak uvolní paměť, kterou instance zabírala
- pokud nedeklarujeme destruktor, překladač vytvoří implicitní destruktor s prázdným tělem

Příklad: třída s dynamickými datovými složkami

```
class Vektor1
{
  float *a;
  int n;
 public:
  Vektor1(): n(2)
  {
        a = new float[n];
        for (int i = 0; i < n; i++)
              a[i] = i+1;
  }
  ~Vektor1()
  {
        delete[] a;
  }
};
```

Kopírovací konstruktor

(copy-constructor)

- specifický konstruktor, slouží k vyvoření kopie instance
- jeho jedinným parametrem je reference na danou třídu
- použije se:
  - při předávání parametrů objektového typu hodnotou
  - v deklaracích typu:

```
T x;        // zde se vola bezny koonstruktor bez parametru
...
T y = x;    // inicializace jinou instanci tridy
T z(x);     // prime volani kopirovaciho konstruktoru
```

- pokud nedeklarujeme kopírovací destruktor, překladač vytvoří vlastní:
  - neobjektové datové složky přenese
  - objektové složky okopíruje pomocí jejich kopírovacích konstruktorů
  - vytváří tzv. **mělkou** kopii (v případě dynamických složek se překopíruje jen ukazatel)

Kopírovací konstruktor

- pokud instance obsahuje dynamicky alokovanou paměť, nebude implicitní konstruktor pracovat správně:
  - chyba se často projevuje velmi zákeřně a na nečekaných místech

  → **vlastní kopírovací konstruktor**
  - postará se o přidělení odpovídající dynamické paměti a překopírování obsahu
  - kopírováním získáme tzv. **hlubokou** kopii

```
Vektor1::Vektor1(Vektor1 &v) : n(v.n)
{
    a = new float[n];
    for (int i = 0; i < n; i++)
        a[i] = v.a[i];
}
```

Operátor přiřazení

- podobně jako kopírovací konstruktor vytváří tzv. **mělkou** kopii (v případě dynamických složek se překopíruje jen ukazatel) → pokud má třída dynamické složky, je na místě **přetížit operátor přiřazení**
  - tak, abychom kopírováním získali **hlubokou** kopii

```
Vektor1& operator=(const Vektor1& v)
{
    for (int i = 0; ((i < n) && (i < v.n)); i++)
        a[i] = v.a[i];
    return *this; // return v;
}
```

Příklad na procvičení II : Chytré pole

- Přeměňte strukturu **ChytrePole** z cvičení na třídu se soukromými datovými složkami.
  - Deklarace třídy bude v hlavičkovém souboru **ChytrePole.h** a definiční deklarace metod budou v souboru **ChytrePole.cpp**.
  - Přidejte třídě **ChytrePole** konstruktor a destruktor (na základě původních funkcí **vytvor()** a **zrus()**).
  - Implementujte metodu **vypis()** (na základě původní funkce)
  - Implementuje metody **pridejNaKonec()** a **smazPrvek()** (na základě původních funkcí **pridej()** a **smaz()**)
  - Implementujte kopírovací konstruktor, aby vytvářel **hlubokou** kopii chytrého pole. Vyzkoušejte, že funguje.
  - Přetěžte pro chytré pole operátor přiřazení, aby vytvářel **hlubokou** kopii. Vyzkoušejte, že funguje.

Příklad na procvičení III : Spojový seznam

Na rozmyšlenou

- Přeměňte struktury **Prvek** a **Seznam** ze cvičení na třídy se soukromými datovými složkami.
  - Přidejte třídě **Seznam** konstruktor a destruktor (na základě původních funkcí **vytvor()** a **zrus()**).
  - Implementujte kopírovací konstruktor, aby vytvářel **hlubokou** kopii seznamu.
  - Přetěžte pro seznam operátor přiřazení, aby také vytvářel **hlubokou** kopii.