C++; delete Java;

Část 1: principy OOP v C++

Kennny

srpen 2017

Kurz - motivace

- základy C++
- jazyk C++ je obrovský, všechny jeho funkce a součásti standardní knihovny nelze obsáhnout v jednom malém kurzu
- přesun C++ -> Java
- zvládnutí KIV/PPR a diplomky bez nějaké větší C++ improvizace

O jazyku

- Bjarne Stroustrup, 1983+
- Původně rozšíření C
- Dialekty
  - C++98 (první oficiální)
  - C++03
  - C++11 (pracovně C++0x, pak C++1x)
  - C++14 (pracovně C++1y)
  - C++1z (připravovaný, předpokládá se C++17)

Navíc proti C

- OOP (třídy, objekty, dědičnost, polymorfismus, ...)
- STL (Standard Template Library)
  - string, vector, list, map, set, ...
  - algorithm, istream/ostream, ...
  - atd.. atd..
- Jiné alokátory (new, delete, delete[])
- Výjimky
- Šablony
- Jmenné prostory
- Odlišná sémantika (scope, namespace, ..)
- atd.. atd..

Jinak proti Javě

- Není čistě objektový - podporuje kód mimo třídy
- Nemá garbagge collector v pravém slova smyslu
- Neběží nad virtuálním strojem
- Vše je naprosto generické (nemá výjimky, jako třeba syntaxe kolem String v Javě)
- Méně bezpečný
- A další...

Hello world

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello World" << endl;
    return 0;
}
```

Třída

- klíčové slovo `class`
- modifikátory viditelnosti atributů:
  - `public` - viditelný všem
  - `protected` - viditelný jen sobě a potomkům
  - `private` - viditelný jen sobě
- sekce s danou viditelností
- konstruktor, destruktor

Schéma třídy

```
class MyClass
{
    public:
        MyClass();  // konstruktor
        ~MyClass(); // destruktor
        void NejakaMetoda(int parametr);

    protected:
        void ProtectedMetoda();
        int mProtParam;

    private:
        void PrivatniMetoda();
        float mPrivParam;
}
```

Konstruktor

- prakticky stejné chování jako v Javě
- může jich existovat více, od C++11 lze i "řetězit" jako v Javě
- *member initializer list* - výčet atributů a jejich inicializačních hodnot v momentě volání konstruktoru

```
MyClass() : mPrivParam(0.0f), mProtParam(10)
{
}
```

- nepovinné, kromě referencí, ty je nutné inicializovat vždy takto (viz dále)
- copy konstruktor - speciální pro kopírování (klonování) objektů

Destruktor

- novinka oproti Javě
- volá se v momentě uvolňování objektu z paměti (resp. těsně před dealokací)
- explicitně nejvýše jeden

```
~MyClass()
{
    delete mSomething;
}
```

Definice a implementace

- lze oddělit definici a implementaci
- v hlavičkovém souboru definice

```
class MyClass
{
    public:
        MyClass();
        void NejakaMetoda();
}
```

- ve zdrojovém souboru implementace

```
MyClass::MyClass()
{
    //
}

void MyClass::NejakaMetoda()
{
    //
}
```

Statická a dynamická alokace

- funguje velmi podobně jako v C
- dynamická alokace: `new` a `delete` (popř. `delete[]`) namísto `malloc` a `free`
  - `new` volá interně konstruktor třídy
  - `delete` destruktor třídy
- statická alokace volá konstruktor a destruktor implicitně
- snaha využít statickou alokaci a obalit tím alokaci dynamickou (viz dále)

Statická a dynamická alokace

- dynamická alokace objektu

```
MyClass* m = new MyClass(1.0, "parametr konstruktoru");
```

- statická alokace objektu

```
MyClass m(1.0, "parametr konstruktoru");
```

- dynamicky alokovaný objekt je nutné ručně uvolnit, statický ne - uvolní se implicitně

Scope

- v kontextu lze přeložit jako "oblast platnosti"
- implicitní (vychází z pravidel syntaxe) a explicitní scope
- implicitní - např. funkční scope nebo scope `if`

```
void funkce()
{ // zacatek implicitni scope
    cout << "Nejaka dulezita prace" << endl;
} // konec implicitni scope
```

- explicitní

```
cout << "Prikaz" << endl;

// zacatek explicitni scope
{
    cout << "Prikaz v explicitni scope" << endl;
} // konec explicitni scope
```

Scope

- i v Javě má funkci oblasti platnosti proměnných - jmen, dosažitelnosti, ..
- navíc ale po jejím opuštění volá destruktory staticky alokovaných objektů

```
void funkce()
{
    cout << "Nejake prikazy, ..." << endl;

    MyClass abc(a); // zde se vola konstruktor

    cout << "Nejake dalsi prikazy, ..." << endl;
} // zde se vola destruktor abc
```

Scope

- často ke scope vážeme tzv. RAII struktury, k těm ale později

```
{
    // ziskani mutexu (konstruktor lock_guard)
    std::lock_guard lck(mMutex);

    ...

} // uvolneni mutexu (destruktor)
```

Příklad

- Prostor pro příklad 01_a_basics
- Note: `#pragma once` je náhražkou bloku:

```
#ifndef MUJ_HEADER_H
#define MUJ_HEADER_H

// vlastni obsah hlavickoveho souboru

#endif
```

Dědičnost

- v hlavičce třídy za dvojtečku se uvádí rodičovská třída a viditelnost
- viditelnost dědičnosti:
  - `public` - všichni ví o vztahu rodič-potomek
  - `protected` - pouze rodič a potomci ví
  - `private` - pouze potomek ví
- v praxi asi nejčastěji `public`
- v konstruktoru potomka by měl být volán konstruktor rodiče

Dědičnost

```
class Rodic
{
    public:
        Rodic(int a)
        {
            ...
        }
};

class Potomek : public Rodic
{
    public:
        Potomek() : Rodic(5) // volani konstruktoru rodice
        {
            ...
        };
};
```

Dědičnost a polymorfismus

- přepisování metod se neděje implicitně
- klíčové slovo `virtual` (rodič)
  - bude uvažovat metodu v tabulce virtuálních metod
  - lze ji tedy přepsat potomkem
- klíčové slovo `override` (potomek)
  - není povinné
  - pouze pro ujištění programátora, že došlo k požadovanému přepisu

Polymorfismus - příklad

```
class Rodic
{
    public:
        virtual void SayHello()
        {
            cout << "Rodic zdravi!" << endl;
        }
};

class Potomek
{
    public:
        virtual void SayHello() override
        {
            cout << "Potomek zdravi!" << endl;
        }
};
```

Dědičnost a polymorfismus

- rozdíl nastává v momentě, kdy máme odlišný typ ukazatele/reference, než je objekt na dané adrese

```
Rodic* r = new Potomek();
r->SayHello();
```

- toto je plně validní, ale jinak se chová s `virtual` u metod a bez nich
  - s `virtual` - "Potomek zdravi!"
  - bez `virtual` - "Rodic zdravi!"
- důvodem je (ne)přítomnost v tabulce virtuálních metod

Dědičnost a polymorfismus

- Speciální případ: destruktor

```
virtual ~Rodic();
```

- rodič by měl mít virtuální destruktor - opět proto, aby se zavolal správný při dealokaci

```
Rodic* r = new Potomek();
r->SayHello();
delete r;
```

- destruktor je jen speciální metoda
  - s `virtual` - zavolá se destruktor potomka
  - bez `virtual` - zavolá se destruktor rodiče

Abstraktní třídy

- abstraktní třída je v C++ vytvořena přítomností tzv. *pure virtual* metody
- speciální signatura, metoda nemá implementaci

```
virtual void DoSomething() = 0;
```

- tato syntaxe ukládá povinnost potomka metodu přepsat
  - resp. pokud chceme vytvořit objekt třídy, musí metoda mít někde v hierarchii implementaci

Metoda s označením const

- metody mohou být označeny klíčovým slovem `const`
- znamená to, že nemění stav objektu

```
float GetX() const
{
    return mPositionX;
}
```

- tyto metody nesmí měnit hodnotu atributů ani ničeho co zaobalují
- mohou volat jen jiné metody označené `const`

Přetypování

- Běžné C-style přetypování

```
Potomek* p = (Potomek*)r;
```

  - "natvrdo" přetypuje
- Statické přetypování

```
int* iptr = static_cast<int*>(iptr2);
```

  - neprovádí kontrolu za běhu, pouze při překladu
- Dynamické přetypování

```
Potomek* p = dynamic_cast<Potomek*>(r);
```

  - provádí kontrolu za běhu
- Reinterpretace

```
IPPacket* pkt = reinterpret_cast<IPPacket*>(ether->payload);
```

  - 1:1 přepis

Příklad

- Prostor pro příklad 01_b_inheritance

Vícenásobná dědičnost

- je možná
- ale opatrně, může se vymstít

```
class DvojPotomek : public Otec, public Matka
```

- dědí metody a atributy obou rodičů
- mnohoznačnost je vyřešena uvozením `Otec::`, popř. `Matka::`, a to jak u atributů, tak metod

Příklad

- Prostor pro příklad 01_c_multiple_inheritance

Operátory

- C++ dovoluje dodefinovat funkce operátorů
- již jsme se setkali s «

```
cout << "Vypis";
```

- začneme něčím jednodušším - typický příklad: 2D vektor
  - z matematiky víme, že vektory lze sčítat, násobit skalárem, násobit vektorem, ... zde se hodí syntax operátorů

Operátory

- sčítání vektorů = sečtení složek vektorů
- uvnitř třídy Vektor:

```
Vektor operator+(const Vektor& second)
{
    return Vektor(x + second.x, y + second.y);
}
```

- zde zachováváme původní vektory a vytváříme novou instanci
- pak lze provést

```
Vektor a(1,2);
Vektor b(2,3);
Vektor c = a + b;
```

Operátory

- Operaci lze dodefinovat i vně třídy Vektor:

```
Vektor operator+(const Vektor& first, const Vektor& second)
{
    return Vektor(first.x + second.x, first.y + second.y);
}
```

- první parametr je levá strana, druhý parametr pravá

Operátory

- takto lze přetížit i operátor bitového posunu
- `std::cout` je globální instance potomka `std::ostream`
- pro výpis vektoru si můžeme přetížit operátor úplně stejně

```
ostream& operator<<(ostream& str, const Vektor& vekt)
{
    return str << "(" << vekt.x << "," << vekt.y << ")";
}
```

- pak můžeme psát

```
Vektor a(1,10);
cout << a << endl;
```

- výsledkem bude výpis: `(1,10)`

Příklad

- Prostor pro příklad 01_d_operators

Další

- reference = v podstatě kompilátorem řízený pointer, nemůže být `null`

```
Vektor& refA = a;
```

- `nullptr` = typově silná náhrada za `NULL`
- `final` = třída, kterou nelze dále dědit

```
class Nededitelna final : public Rodic
{
    ...
```

Namespace

- jmenný prostor = prostor platnosti objektů a proměnných (a jejich jmen, pochopitelně)

```
namespace Prostor
{
    int promenna;

    class Trida
    {
        ...
    }
}
```

- dostupná bez instancování, jen je nutné uvodit názvem namespace

```
Prostor::promenna = 10;
```

- nebo využít `using`, pak není nutné dále uvozovat

```
using namespace Prostor;
promenna = 10;
```

Konec 1. části

```
exit(0);
```