

PB161 Programování v jazyce C++

Přednáška 2

Základy objektů

Reference, `const`

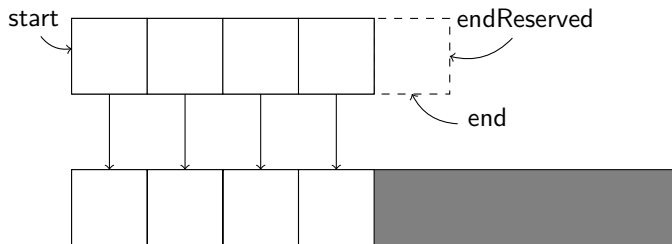
Testování

Nikola Beneš

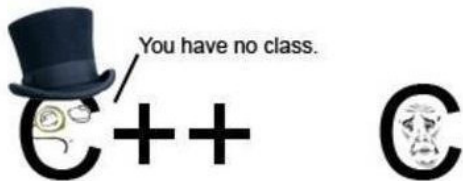
27. září 2016

Typická implementace

- rezervovaná paměť, část z ní je obsazená
- tři ukazatele (nebo jeden ukazatel a dvě čísla)
- začátek rezervované paměti, konec obsazené, konec rezervované
- když je rezervovaná paměť plná, provede se alokace nové a prvky se do ní zkopírují (resp. přesunou)



Třídy, objekty, metody



Znáte z C: `struct`

Třídy v C++: `class`, `struct`

- rozšíření struktur o:
 - přístupová práva
 - metody
 - prvky OOP (dědičnost, pozdní vazba, ...)
- přístupová práva:
 - `public` vidí všichni
 - `private` vidí jen objekty dané třídy
 - `protected` – souvisí s dědičností
 - `class` má implicitně právo `private`,
`struct` má implicitně právo `public`

Poznámka: Zatím používáme třídy jen jako „lepší `struct`“, k OOP se dostaneme později.

Objekty v C++

- proměnné jejichž typem je třída

Atributy (správně „členské proměnné“ – *member variables*)

- jako položky **struct**
- jsou součástí stavu objektu

Metody (správně „členské funkce“ – *member functions*)

- funkce, které nějak pracují s objekty
- smí číst/měnit atributy objektu

[ukázka]

Deklarace a definice metod

- deklarace uvnitř třídy (v hlavičkovém souboru)
- definice:
 - uvnitř třídy (vhodné pro malé metody)
 - odděleně (větší metody)

```
class Person {  
    std::string name;  
public:  
    // deklarace s definicí  
    void rename(std::string newName) {  
        name = newName;  
    }  
  
    // deklarace bez definice  
    int getSalary();  
}
```

Skrytý parametr `this`

- ukazatel na konkrétní objekt

```
void Person::rename(std::string newName);
```

// je jakoby

```
void Person::rename(Person * this, std::string newName);
```

// (to ovšem není validní C++)

```
jimmy.rename("Agent 007");
```

// je jakoby

```
Person::rename(&jimmy, "Agent 007"); // (není C++)
```

Inicializace objektů

Speciální metoda – konstruktor

- jmenuje se stejně jako třída
- nemá návratový typ
- má tzv. **inicializační sekci**

```
class Person {  
    std::string name;  
    int age;  
public:  
    Person(std::string n, int a) : name(n), age(a) {  
        std::cout << "New Person created:" << name << "\n";  
    }  
}
```

[ukázka použití]

Inicializace objektů (pokr.)

Inicializace atributů přímo v deklaraci třídy – od C++11

```
class Person {  
    std::string name;  
    int age = 0;  
public:  
    Person(std::string n) : name(n) {  
        std::cout << "New Person created:" << name << "\n";  
    }  
}
```

- přednost má inicializační sekce konstruktoru
 - to má význam, když definujeme více různých konstruktů
 - o přetěžování metod/funkcí apod. si řekneme více později

Různé způsoby inicializace

- při deklaraci

```
std::string one = "One";  
std::string two{"Two"};  
std::string three("Three");  
std::string empty{};  
std::string alsoEmpty;
```

- dočasný objekt

```
std::cout << std::string();  
std::cout << std::string{};  
std::cout << std::string("Hello");  
std::cout << std::string{"Hello"};
```

- v inicializační sekci (podobně)

Inicializace v C++ (pokr.)

Na co si dát pozor

- u primitivních typů a tříd s implicitním konstruktorem je zde rozdíl:

```
int x;    // neinicializováno
int y{};  // inicializováno na 0
int arrayA[100];    // neinicializované pole
int arrayB[100]{};  // pole nul
```

- kulaté a složené závorky mohou mít u některých tříd různý význam

```
std::string a{65, 'x'}; // "Ax"
std::string b(65, 'x'); // šedesát pět krát x
```

Doporučení

- pokud je to možné, raději proměnné inicializujte
- (na linuxu) používejte *valgrind*

Pro zvědavé:

<http://en.cppreference.com/w/cpp/language/initialization>

Single Responsibility Principle (SRP)

- část tzv. principů SOLID (více později)
- každá třída (modul, funkce, ...) by měla být zodpovědná za jednu konkrétní věc
- Heslo: „Třída by měla mít jen jeden důvod ke změně.“

Princip zapouzdření (*encapsulation*)

- metody a atributy tříd jsou na jednom místě
- přístup k atributům třídy může být omezen právy

Rekapitulace – Třídy a objekty

Třídy v C++ – složené datové typy

Objekty – hodnoty těchto datových typů

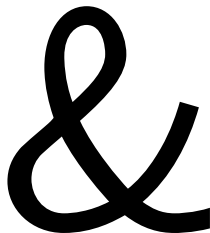
Metody (členské funkce) – funkce deklarované uvnitř tříd, mají skrytý parametr **this** – ukazatel na aktuální objekt

Konstruktor – speciální metoda, která se volá při inicializaci objektu, může mít (a je doporučeno toho využívat co nejvíce) inicializační sekci

Zatím zamlčeno (a řekneme si o tom později):

- destruktory
- přetěžování funkcí, metod
- kopírovací konstruktory
- dědičnost
- virtuální metody, abstraktní třídy

Reference a const



Viděli jsme minule

- parametry se předávají hodnotou (tj. kopie)
- někdy nechceme vytvářet kopie
 - chceme parametr navenek změnit
 - parametr je složitý datový typ a kopírování by bylo zbytečně drahé

Práce s ukazateli (znáte z C)

```
void swap(int *x, int *y) { // toto je C  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

Problémy s ukazateli?

Reference

Reference: slabší, ale bezpečnější forma ukazatele

- nemůže být NULL (od C++11 `nullptr`)
- nemůže se přesměrovat jinam
- musí být vždy inicializovaná
- funguje jako alias

```
int a = 1;
int b = 2;
int& ref = a; // ref je reference na a
// cokoli provedeme s ref, jako bychom provedli s a
ref = b;
ref += 7;
// jaká je teď hodnota a, b?
```

[ukázka: srovnání s C]

Reference vs. ukazatele

Ukazatel může být neinicializovaný

```
int* ptr;  
int& ref; // chyba při kompilaci
```

Ukazatel může být nullptr

```
int* ptr = nullptr;  
int& ref = nullptr; // chyba při kompilaci
```

Ukazatel se může přesměrovat

```
int a = 1;  
int b = 2;  
int* ptr = &a;  
ptr = &b;  
  
int& ref = a;  
ref = b;  
// v čem je rozdíl?
```

Používat reference nebo ukazatele?

- pokud možno dávejte přednost referencím
- ukazatele používejte jen tam, kde dává smysl `nullptr` a přesměrování
- pro dynamickou alokaci jsou vhodnější chytré ukazatele (o těch později)

Na co si dát pozor?

- nedržet si referenci na něco, co už přestalo existovat
- nevracet z funkcí reference na lokální proměnné [ukázka]

Klíčové slovo `const`

Znáte (možná) z C

- deklarace konstant
- použití spolu s ukazateli

Připomenutí: `const` a ukazatele

```
int a = 1;
int b = 2;
int* ptrX = &a;
const int* ptrY = &a;
int* const ptrZ = &a;
ptrX = &b;
*ptrX = 7;
ptrY = &b;
*ptrY = 7;
ptrZ = &b;
*ptrZ = 7;
```

Které řádky obsahují chybu?

Použití s referencemi

```
int a = 1;
const int& cref = a;
std::cout << cref << '\n'; // čtení je OK
cref = b; // zápis/modifikace je chyba při kompilaci
```

Význam konstantních referencí

- deklarujeme záměr neměnit proměnnou (*read-only*)
- zároveň ale nevytváříme kopii
 - vhodné např. pro předávání větších objektů do funkcí
- viděli jsme minule (`string`, `vector`)

Reference a l/p-hodnoty

l-hodnoty a p-hodnoty (*lvalues and rvalues*)

- l-hodnota je to, co může stát na levé straně přiřazení
 - má adresu, má jméno
- p-hodnota je to, co může stát na pravé straně přiřazení
 - číselná hodnota, dočasný objekt
- (v C++11 trochu složitější)

Co „chytají“ reference?

- konstantní reference „chytá“ všechno
- nekonstantní reference „chytá“ jen l-hodnoty

```
void f(const std::string& x) { /* ... */ }
```

```
void g(std::string& x) { /* ... */ }
```

```
int main() {  
    std::string s = "XYZ";  
    f("ABC"); g("DEF"); f(s); g(s);  
}
```

Konstantní metody

```
class Person {  
    int age;  
public:  
    void setAge(int a) { age = a; }  
    int getAge() { return age; }  
};  
void f(const Person& person) {  
    std::cout << person.getAge() << '\n';  
}  
int main() {  
    Person john;  
    john.setAge(20);  
    f(john);  
}
```

Kde je problém?

Konstantní metody (pokr.)

const metody

- pokud metoda nehodlá měnit vnitřní stav objektu, musíme ji takto označit
- klíčové slovo `const` za hlavičkou metody

```
class Person {  
    // ...  
    int getAge() const { return age; }  
};
```

- překladač nás hlídá
 - pokus o změnu stavu objektu v `const` metodě ohlásí jako chybu

Doporučení

- deklarujte jako `const` všechny metody, které nemají měnit stav objektu

Použití `const` u atributů třídy

- atribut se dá nastavit pouze v inicializační sekci konstruktoru
- potom už se nedá vůbec změnit

```
class Person {  
    std::string name;  
    int age;  
    const std::string genome;  
    // ...  
};
```

- využití:
 - neměnné objekty
 - neměnné části objektů (větší bezpečnost)

Kde používat `const`?

- všude, kde dává smysl (tj. skoro všude)
- pište `const` všude a jen tehdy, pokud zjistíte, že danou proměnnou (atribut, referenci) budete chtít měnit, `const` smažte
- pište `const` ke všem metodám a jen tehdy, pokud zjistíte, že chcete, aby daná metoda měnila stav objektu, `const` smažte

Proč?

- překladač vás hlídá → větší bezpečnost
- jasně deklarujete svůj záměr → větší čitelnost kódu
- překladač může využít pro optimalizace

Poznámka: Programátoři v C++ v naprosté většině nepišou `const` k parametrům funkcí předávaným hodnotou. Proč?

Hodnotová sémantika objektů

```
class MyObject {  
    int x, y;  
public:  
    MyObject(int, int);  
    int getX() const;  
    int getY() const;  
    void setX(int);  
    void setY(int);  
};  
  
void f(MyObject a) {  
    MyObject b(10, 12);  
    a.setX(100);  
    a = b;  
    a.setY(50);  
    b.setX(20);  
}  
  
int main() {  
    MyObject o(0, 0);  
    f(o);  
    return o.getX() + o.getY();  
}
```

- co se stane, když změníme hlavičku funkce f, aby brala referenci?
- jaké bude chování podobných programů v Pythonu či Javě?

Testování



Programy obsahují chyby

- je to tak

Testování programů

- důležitá součást vývoje
- různé úrovně testování
- *unit testing* – testování malých jednotek kódu

Nástroje pro unit testing

- velká řada různých frameworků
- my používáme *googletest*

Použití googletest

Psaní testů – makro TEST

```
TEST(test_case_name, test_name) {  
    // some code with asserts  
}
```

Podmínky

```
ASSERT_TRUE(condition); // musí být pravdivá  
ASSERT_FALSE(condition); // nesmí být pravdivá  
ASSERT_EQ(x, y); // musí platit x == y
```

- místo EQ lze použít NE (nerovnájí se), LT, GT, LE, GE
- místo ASSERT lze použít EXPECT
- googletest předpokládá pořadí
(očekávaná hodnota, skutečná hodnota)
 - pište tedy např.

```
ASSERT_EQ("John", Person.getName());
```

Podmínky (pokr.)

- rozdíl mezi ASSERT a EXPECT
 - neplatnost ASSERTu ukončí daný test
 - neplatnost EXPECTu se zaznamená, ale test pokračuje dál
 - primárně preferujte EXPECT, ale použijte ASSERT, pokud by další pokračování nemělo smysl
- k podmínkám je možno přidat zprávu

```
ASSERT_EQ(x, y) << "x is not equal to y";
```

`https://kahoot.it`