

PB161 Programování v jazyce C++

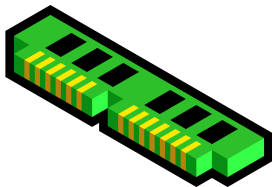
Přednáška 5

Práce s pamětí
Princip RAI

Nikola Beneš, Vladimír Štill

18. října 2016

Práce s pamětí



Ukazatel

- proměnná, která drží adresu v paměti
- aritmetika ukazatelů (přičtení čísla k ukazateli, rozdíl ukazatelů)
- rozdíl mezi `const int*` a `int* const`
 - a případně `const int* const`
- speciální ukazatel `nullptr` (od C++11)
 - dříve `NULL` – nepoužívat v novém kódu!

Alokace paměti – znáte z C

Alokace na zásobníku (stack)

- lokální proměnné
- automatické uvolnění paměti na konci bloku
(v C++ navíc automatické volání destruktorků)

Statická alokace

- globální proměnné, statické proměnné ve funkcích
(v C++ navíc statické atributy – uvidíme později)
- alokace před spuštěním programu
- inicializace před začátkem funkce `main` (složitější)
- dealokace po skončení programu
(v C++ navíc volání destruktorků)

Dynamická alokace na haldě (heap)

- explicitní žádost o paměť
- explicitní uvolnění paměti
- znáte z C: `malloc/free` (v C++ nepoužíváme)
 - nebezpečné: žádná typová kontrola, bez inicializace

Nízkoúrovňová (ruční)

- operátory **new** a **delete**
 - **new** alokuje paměť na haldě a zavolá konstruktor
 - **delete** zavolá destruktory a dealokuje paměť
- moderní C++: příliš nepoužívat, jen v nutných případech
- nicméně je dobré vědět, jak funguje

Vysokoúrovňová (automatická)

- využití standardní knihovny, příp. jiných knihoven (*boost* apod.)
- už jsme vlastně viděli
 - `vector`, `string` a jiné kontejnery vnitřně alokují a dealokují paměť na haldě
- jiné možnosti: chytré ukazatele
 - automatické uvolnění paměti
 - `unique_ptr` (od C++11 ve standardní knihovně) a jiné

Dva způsoby alokace


- jeden objekt
- pole objektů

```
int* ptr    = new int;  
int* array = new int[20];
```

- když se alokace nezdaří, vyhodí výjimku (o těch bude řeč později)
 - není proto třeba kontrolovat na `nullptr`

Odpovídající způsoby dealokace

```
delete ptr;  
delete [] array;
```

- musí odpovídat použitému `new`
- nehlídá kompilátor (někdy ovšem umí dát warning)! Proč? 

Nízkoúrovňová správa paměti (pokr.)

Inicializace alokovaných objektů

- funguje podobně jako inicializace lokálních objektů

```
int* ptr1 = new int;           // neinicializovaná paměť
int* ptr2 = new int(17);       // paměť inicializovaná na 17
int* ptr3 = new int();         // paměť inicializovaná na 0

Obj* ptr4 = new Obj;           // zavolá se konstruktor Obj::Obj()
Obj* ptr5 = new Obj(3, 5);     // konstruktor Obj::Obj(int, int)
```

alokace polí

```
int* arr1 = new int[200];      // neinicializovaná paměť
int* arr2 = new int[200]();     // paměť inicializovaná na 0

Obj* arr3 = new Obj[200];      // zavolá se konstruktor Obj::Obj()
```

- není jak alokovat pole a volat jiný než bezparametrický konstruktor



Problémy správy paměti (některé znáte z C)

- memory leak (alokovaná paměť, na kterou již nemáme ukazatel)
- zápis do/čtení z nealokované paměti
- čtení z neinicializované paměti
- volání `delete` na špatný ukazatel
- volání `delete` místo `delete []` a naopak
- volání `free` na paměť alokovanou `new`, nebo `delete/delete[]` na paměť alokovanou `malloc`
 - v C++ nikdy nepoužíváme `free/malloc` (leďa by to vyžadovala použitá C knihovna)!

Motivace pro chytré ukazatele

- chceme ukazatel, který sám provede `delete`, když je potřeba

Chytré ukazatele

- několik druhů
- ve standardní knihovně (od C++11), příp. v jiných knihovnách (*boost*)
- v tomto předmětu si ukážeme pouze jeden

`std::unique_ptr`

- nejjednodušší, ale pokrývá většinu potřeb
- má nulovou režii (overhead) za běhu, na rozdíl od jiných
- použitelný pro jednotlivé objekty i pole
- základní princip:
 - objekt, který uvnitř drží ukazatel (koncept vlastnictví)
 - na konci života objektu se zavolá `delete` na vlastněný ukazatel (tedy zavolá destruktorka a dealokuje)
 - vlastnictví se nedá sdílet (proto `unique`)
 - ale může se explicitně předat

Vysokoúrovňová správa paměti (pokr.)

Základní použití `std::unique_ptr`

- alokace
 - pokud neinicializujeme, je automaticky `nullptr`

// v C++11

```
std::unique_ptr<Object> ptr(new Object(params));
```


// od C++14 - používejte raději takto

```
auto ptr = std::make_unique<Object>(params);
```

- přístup k objektu – stejně jak u klasických ukazatelů (`*`, `->`)
 - pozor, nedefinované chování, pokud by byl `ptr == nullptr`

```
ptr->method();
```

```
function(*ptr);
```

- dealokace, zavolání destruktoru objektu
 - lokální `unique_ptr`: automaticky, na konci bloku
 - `unique_ptr` jako atribut: automaticky, když skončí život hlavního objektu
 - explicitně: zavoláním metody `reset()` 

Vysokoúrovňová správa paměti (pokr.)

- zjištění, zda `unique_ptr` obsahuje nějaký ukazatel

```
if (ptr) { ... }
```

- přímý přístup ke spravovanému ukazateli (k čemu je to dobré?)

```
Object* rawPtr = ptr.get();  
// ptr je stále vlastníkem ukazatele
```

- vzdání se vlastnictví (k čemu je to dobré?)

```
// tohle raději nedělejte!  
Object* rawPtr = ptr.release();  
// ptr už není vlastníkem ukazatele, který je nyní uložen  
// v rawPtr, a je třeba jej uvolnit ručně!
```



Vysokoúrovňová správa paměti (pokr.)

- předávání vlastnictví jinému `unique_ptr`
 - `unique_ptr` se nedá kopírovat!

// tohle nebude fungovat

```
std::unique_ptr<Object> newPtr = ptr; // chyba!
```

- `unique_ptr` se umí tzv. *přesouvat* (move)
 - využívá *rvalue semantics*, mimo záběr předmětu

```
std::unique_ptr<Object> newPtr = std::move(ptr);
```

// newPtr teď vlastní ukazatel, který předtím vlastnil ptr

// ptr teď nevládní nic, je tedy ekvivalentní nullptr

- funguje nejen při inicializaci, ale i při přiřazení

```
auto ptrA = std::make_unique<Object>();
```

```
auto ptrB = std::make_unique<Object>();
```

```
ptrA = std::move(ptrB);
```

// kdo co vlastní teď?



Vysokoúrovňová správa paměti (pokr.)

- použití pro alokaci polí

// v C++11

```
std::unique_ptr<Object[]> ptr(new Object[size]);
```

// od C++14 - používejte raději takto

```
auto ptr = std::make_unique<Object[]>(size);
```

// alokuje paměť pro size objektů

// a všechny inicializuje bezparametrickým konstruktorem

- použití pro alokací polí primitivních typů

```
auto ptr = std::make_unique<int[]>(size);
```

// alokuje paměť pro size intů

// a všechny inicializuje na 0

- inicializuje tedy jako `new int[size]()`




Vysokoúrovňová správa paměti (pokr.)


Jak správně pracovat s ukazateli v moderním C++

- jasně si rozmyslete, kdo bude *vlastníkem* ukazatele
 - ten pak má `unique_ptr` ukazující na daný objekt
- ostatní (ne-vlastníci) smí mít klasický (surový, *raw*) ukazatel na tentýž objekt (nebo lépe referenci)
 - je třeba zaručit, aby vlastník ukazatele přežil všechny ne-vlastníky, kteří ukazovaný objekt používají

Jak předávat `unique_ptr` do funkce?

- hodnotou typu `unique_ptr`: volající pak musí použít `std::move` a tím se vzdává vlastnictví ukazatele
- referencí: volaná funkce může sebrat vlastnictví (nedoporučované)
- `const` referencí: OK, ale volaná funkce může modifikovat odkazovaný objekt (je to jakoby `Object* const`)
- surový ukazatel: může být `Object *` nebo `const Object *`
- úplně nejlépe – referencí na `Object`: volající musí zajistit, že není `nullptr` 

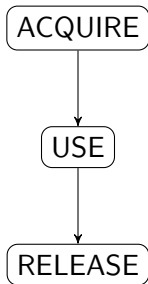
Datové struktury a `unique_ptr`

- je třeba rozmyslet strukturu vlastnictví
- není možné mít ve vlastnictví cykly (proč?)
- není možné, aby dva vlastnili stejný objekt
- oboustranně zřetězený seznam 
 - (v prvcích) `next` je `unique_ptr`, `prev` je ukazatel
 - (v seznamu) `first` je `unique_ptr`, `last` je ukazatel
 - nebo naopak
- binární strom
 - vztah *rodič vlastní potomky*
 - `left` a `right` jsou `unique_ptr`, případný `parent` je ukazatel
- apod.

Doporučení

- používejte vysokoúrovňovou správu paměti (`unique_ptr`)
- nepoužívejte `new` a `delete`
 - ale naučte se jim rozumět
 - často je uvidíte v cizím kódu
- do funkcí předávejte objekty pokud možno referencí (toto doporučení už bylo!)
 - předávejte `unique_ptr` hodnotou, pokud se chcete vzdát vlastnictví
- implementujete-li složitější datovou strukturu s ukazateli, rozmyslete si strukturu vlastnictví (kdo koho vlastní)

Správa zdrojů – princip RAIL



Co to je zdroj?

- něco, co se získá (*acquire*)
- potom se to používá (*use*)
- nakonec se to uvolní (*release*)

Příklady zdrojů

- paměť na haldě
- soubory
- síťová připojení
- zámky, mutexy, semafore, ...
- apod.

- **v C++ vše funguje na stejném principu**

Různé jazyky často implementují automatickou správu paměti, ale ne automatickou správu zdrojů (v posledních letech se to trochu zlepšuje). C++ má automatickou správu zdrojů už skoro od svého počátku.


Princip RAI – „Resource Acquisition Is Initialisation“

- někdy též známo *scope-based resource management*
- správa zdroje spjatá s životním cyklem objektu
- inicializace objektu: získání zdroje (*acquire*)
- destruktork: uvolnění zdroje (*release*)
- ideálně: jeden objekt spravuje jeden zdroj

Kde se používá RAI?

- skoro všude!
- `string`, `vector`, všechny kontejnery
- práce se soubory v C++ (později)
- chytré ukazatele: `unique_ptr` (C++11)
- zamykání, mutexy (C++11, nad rámec kurzu)

Princip RAI

Příklad – dynamické pole `intů` (pamatujete na *Rule of Three?*) 

```
class IntArray {
    size_t size;
    int* array;
public:
    IntArray(size_t size)
        : size(size), array(new int[size]) {}
    ~IntArray() { delete [] array; }
    IntArray(const IntArray& other)
        : size(other.size), array(new int[size]) {
        std::copy(other.array, other.array + size, array);
    }
    IntArray& operator=(const IntArray& other) {
        // viz další slajdy
    }
}
```

Příklad – dynamické pole intů

Jak implementovat přiřazovací operátor?

```
IntArray& operator=(const IntArray& other) {  
    delete [] array;  
    size = other.size;  
    array = new int[size];  
    std::copy(other.array, other.array + size, array);  
    return *this;  
}
```

- kde je problém?
- sebe-přiřazení (co se stane, když napíšu `x = x;`?)
- jak řešit?

Příklad – dynamické pole intů (pokr.)

Kontrola sebe-přiřazení

```
IntArray& operator=(const IntArray& other) {  
    if (&other == this)  
        return *this;  
    delete [] array;  
    size = other.size;  
    array = new int[size];  
    std::copy(other.array, other.array + size, array);  
    return *this;  
}
```

- optimalizace
 - není třeba dělat `delete []` a `new []`, pokud jsou velikosti stejné

Příklad – dynamické pole intů (pokr.)

Jiná možnost, tzv. copy-and-swap idiom

```
void swap(IntArray& other) {  
    using std::swap;  
    swap(size, other.size);  
    swap(array, other.array);  
}
```

```
IntArray& operator=(IntArray other) { // HODNOTOU!  
    swap(other);  
}
```

- co se děje?

Rule of Three and a Half

- ke třem dříve zmíněným se přidá metoda swap
- *copy-and-swap* idiom silně doporučujeme!

Jiné příklady (pro ilustraci)

- třída File
 - konstruktor otevře soubor
 - destruktork zavře soubor
 - kopírování nejspíše zakážeme
- třída Mutex
 - konstruktor zamkne mutex
 - destruktork odemkne mutex
 - kopírování opět zakážeme
- třída Texture
 - konstruktor načte texturu ze souboru do paměti GPU
 - destruktork uvolní paměť GPU
 - kopírování: vytvoření nové textury

Princip RAI v jiných jazycích

Některé jazyky mají RAI

- C++, D, Ada, Rust umí plné RAI
- tak trochu: Perl, Python (CPython), PHP mají *reference counting*

Správa zdrojů v jiných jazycích

- garbage collector: správa paměti, typicky neumožňuje správu zdrojů (není žádná záruka, kdy a jestli vůbec se zavolají destruktory)
- Java
 - `synchronized`
 - `try(Resource res = new Resource(...)) { ... } (1.7)`
- Python
 - `with get_resource() as resource: (2.5)`
- C#
 - `using(Resource res = new Resource(...)) { ... }`