

PB161 Programování v jazyce C++

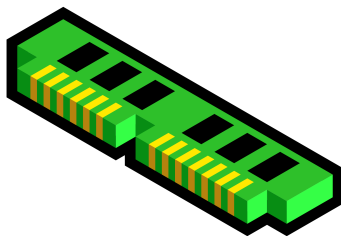
Přednáška 4

Dynamická alokace
Kopírovací konstruktor
Přetypování v C++

Nikola Beneš

12. října 2015

Dynamická alokace



Statická vs. dynamická alokace – znáte z C

Statická alokace na zásobníku

- lokální proměnné
- automatické uvolnění paměti
- krátkodobé, menší objekty

Statická alokace globálních proměnných

- alokace před spuštěním programu
- uvolnění paměti při ukončení programu
- statické proměnné ve funkcích

Dynamická alokace na haldě

- explicitní žádost o paměť
- explicitní uvolnění paměti
- dlouhodobé, větší objekty

Dynamická alokace v C

`malloc / free`

- je třeba spočítat velikost paměti
- žádná typová kontrola
- NULL, pokud se alokace nepodaří
- paměť není inicializovaná

Alokaci ve stylu C nebudeme v C++ používat

Dynamická alokace v C++

`new / delete / delete []`

Dva základní typy alokace

- jeden objekt
- pole

```
int * ptr    = new int;
```

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

new a konstruktory

Inicializace paměti

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

```
class A {
    int value;
public:
    A() : value(42) {}
    A(int v) : value(v) {}
    int getValue const () : { return value; }
};
```

```
A * ptrA1 = new A; // volá bezparametrický konstruktor A()
cout << ptrA1->getValue() << endl; // 42
```

```
A * ptrA2 = new A(200); // volá konstruktor A(int)
cout << ptrA2->getValue() << endl; // 200
```

new a konstruktory – alokace polí

Primitivní typy: neinicializovaná paměť

Objekty: bezparametrický konstruktor

```
A * arrayA = new A[17];  
for (int i = 0; i < 17; ++i) {  
    cout << arrayA[i].getValue() << endl;  
} // vypíše sedmáctkrát 42
```

```
int * array = new int[200]; // neinicializováno
```

```
class B {  
    int value;  
public:  
    B(int v) : value(v) {}  
};  
B * arrayB = new B[17]; // CHYBA!
```

Když se alokace nezdaří

- `new` vyhazuje výjimku `std::bad_alloc`
- `new (std::nothrow)` vrací `nullptr` (v C++03 `NULL`)

```
int * array = new (std::nothrow) int[1000000000L];  
if (array) {  
    // ...  
}
```


Dealokace

`delete`

- pro samostatné objekty

`delete []`

- pro pole

```
int * ptr = new int(17);  
// ...
```

```
delete ptr;
```

```
int * array = new int[200];  
// ...
```

```
delete [] array;
```

Nehlídá kompilátor! Proč?

Dynamická alokace – problémy

- memory leak (nedealokovaná paměť, na kterou již nemáme ukazatel)
- zápis do nealokované paměti
- volání `delete` místo `delete []` a naopak
- volání `delete` na špatný ukazatel

Detekce problémů: valgrind

```
int main() {  
    int * array = new int[100];  
    array += 42;  
    delete array; // špatný ukazatel  
}
```

Dealokace a destruktory

- při dealokaci paměti se volá destruktory
- destruktory potomka volá na závěr automaticky destruktory předka
- statická dealokace
- dynamická dealokace (při zavolání `delete`)

Dealokace a destruktory (pokr.)

```
class IPrinter {
public:
    virtual void printDocument(std::string);
};

class MyPrinter : public IPrinter {
    std::string * printerQueue;
public:
    MyPrinter() : printerQueue(new std::string[10]) {}
    void printDocument(std::string) override;
    ~MyPrinter() {
        delete [] printerQueue;
    }
};

int main() {
    IPrinter * p = new MyPrinter;
    p.printDocument("myDocument.doc");
    delete p; // zavolá se ~IPrinter(), MEMORY LEAK!
}
```

Kdy použít?

- od třídy se bude dědit
- instance budou dealokovány skrz ukazatele na předka

Herb Sutter

Guideline #4: A base class destructor should be either public and virtual, or protected and nonvirtual.

In brief, then, you're left with one of two situations. Either:

- a) You want to allow polymorphic deletion through a base pointer, in which case the destructor must be virtual and public; or*
- b) You don't, in which case the destructor should be nonvirtual and protected, the latter to prevent the unwanted usage.*

Virtuální destruktor (pokr.)

```
class IPrinter {
public:
    // ...
    virtual ~IPrinter() {}
};

class MyPrinter : public IPrinter { /* ... */ };

int main() {
    IPrinter * p = new MyPrinter;
    // ...
    delete p; // zavolá se ~MyPrinter(), OK
}
```

Kopírování, kopírovací konstruktory



Kopírování objektů

```
class Point {  
    int x, y;  
public:  
    Point(int sx, int sy) : x(sx), y(sy) {}  
    int getX() const { return x; }  
    int getY() const { return y; }  
    void setX(int sx) { x = sx; }  
    void setY(int sy) { y = sy; }  
    void output() const {  
        std::cout << "[" << x << ", " << y << "]\n";  
    }  
};
```

[ukázka kopírování]

Kopírování objektů – jak funguje?

Kopírovací konstruktor

- pokud nedefinujeme, vyrobí se implicitní
- plytká (shallow) vs. hluboká (deep) kopie

[ukázka IntArray bez kopírovacího konstrukturu]

- plytká kopie kopíruje pouze hodnoty atributů
 - co to znamená pro ukazatele?

Kopírování objektů – jak řešit?

Řešení 1: Zakázat kopírování

- vhodné pro objekty, které nemá smysl kopírovat
- v C++11 pomocí speciální syntaxe = `delete`
- v C++03 se řešilo přesunem kopírovacího konstruktoru do sekce `private`

[ukázka]

Kopírování objektů – jak řešit?

Řešení 1: Zakázat kopírování

- vhodné pro objekty, které nemá smysl kopírovat
- v C++11 pomocí speciální syntaxe = `delete`
- v C++03 se řešilo přesunem kopírovacího konstruktoru do sekce `private`

[ukázka]

Řešení 2: Vlastní kopírovací konstruktor

- hluboká kopie

[ukázka]

Kopírování objektů – jak řešit?

Řešení 1: Zakázat kopírování

- vhodné pro objekty, které nemá smysl kopírovat
- v C++11 pomocí speciální syntaxe = `delete`
- v C++03 se řešilo přesunem kopírovacího konstruktoru do sekce `private`

[ukázka]

Řešení 2: Vlastní kopírovací konstruktor

- hluboká kopie

[ukázka]

Typ kopírovacího konstruktoru

```
Object(const Object &)
```

Už jsme tím vyřešili veškeré problémy s kopírováním?

Už jsme tím vyřešili veškeré problémy s kopírováním?

Kopírování při přiřazení

- `operator=`
- typ operátoru přiřazení (i jiné možnosti):

`Object & operator=(const Object &)`

- doporučení: vracet v operátoru `=` vždy referenci na objekt

[ukázka operátoru `=`]

Drobné problémy

- sebepřiřazení (self-assignment)

[ukázka detekce sebepřiřazení]

Drobné problémy

- sebepřiřazení (self-assignment)

[ukázka detekce sebepřiřazení]

- duplikace kódu
 - *copy-and-swap* idiom

[ukázka copy-and-swap]

Výhody

- neduplikujeme kód
- prostor k optimalizaci

Drobné nevýhody (možná)

- sebepřiřazení vytváří kopii
- bez copy and swap můžeme někdy být efektivnější

[ukázka efektivnějšího kopírování]

Copy and swap

Výhody

- neduplikujeme kód
- prostor k optimalizaci

Drobné nevýhody (možná)

- sebezpřirazení vytváří kopii
- bez copy and swap můžeme někdy být efektivnější

[ukázka efektivnějšího kopírování]

Doporučení

- používejte copy-and-swap
- efektivitu řešte až na základě profilingu

Rule of three

- destruktork
- kopírovací konstruktor
- kopírovací přiřazovací operátor

„Buď všichni tři implicitní nebo ani jeden z nich.“

Pozn. v C++11 Rule of five

Kdy se volá kopírovací konstruktor?

- explicitně

```
Object a;  
Object b = a;  
Object c(a);
```

- při předávání hodnotou (možná optimalizace)

```
void f(Object o);  
f(a);
```

- při vracení objektu hodnotou (možná optimalizace)

```
Object f() {  
    Object x;  
    // ...  
    return x;  
}  
Object d = f();
```

- nevolá se při předávání referencí

Dokončení příkladu IntArray

[ukázka operátoru []]

Přetypování (casting)

Syntax (typ)hodnota

- není typově korektní
- nedoporučuje se používat v C++
- v PB161 zakázáno používat!

Přetypování v C++ – `static_cast`

`static_cast<typ>(hodnota)`

- typické použití:
 - primitivní datové typy
 - změna celočíselné hodnoty na `enum`
 - změna ukazatele na `void *` a naopak
 - přetypování v nevirtuální objektové hierarchii
 - přetypování z potomka na předka (ukazatele, reference)
- nemusí být vždy bezpečné
 - nedefinované chování

Přetypování v C++ – `dynamic_cast`

- ukazatel na potomka se umí automaticky přetypovat na ukazatel na předka

```
class A { /* ... */ };  
class B : public A { /* ... */ };
```

```
A * ptr = new B;
```

- opačný směr: `dynamic_cast`
- rozhoduje se za běhu programu
- jen pro virtuální objektovou hierarchii; proč?

```
B * ptr2 = dynamic_cast<B *>(ptr);
```

Přetypování v C++ – `dynamic_cast`

```
A * ptr1 = new B;
```

```
A * ptr2 = new A;
```

```
B * ptrB;
```

```
ptrB = dynamic_cast<B *>(ptr1); // OK
```

```
ptrB = dynamic_cast<B *>(ptr2); // nullptr (NULL)
```

```
B & ref1 = dynamic_cast<B &>(*ptr1); // OK
```

```
// tohle vyhodí výjimku std::bad_cast
```

```
B & ref2 = dynamic_cast<B &>(*ptr2);
```

- použití `dynamic_cast` se často dá vyhnout vhodnou změnou objektové hierarchie

Dynamická alokace

- `new` / `delete` / `delete []`
- při alokaci se volá konstruktor
- při dealokaci se volá destruktork

Kopírovací konstruktor

- implicitní – plytký (shallow)

Kopírovací operátor =

- copy and swap idiom

Pravidlo tří (Rule of three)

Přetypování v C++

- `static_cast`
- `dynamic_cast`