

PB161 – 3. přednáška (12. října 2015)

Poznámky k domácímu úkolu

Dynamická alokace

Statická vs. dynamická alokace

- statická alokace na zásobníku (stack)
 - deklarace uvnitř bloku (lokální proměnné)
 - automatické uvolnění paměti na konci bloku
 - rychlé (zásobník v cache)
 - zásobník má omezenou velikost
 - využití pro krátkodobé a menší objekty
- dynamická alokace na haldě
 - explicitní žádost o kus paměti
 - explicitní pokyn k uvolnění paměti
 - pro dlouhodobé nebo větší objekty
- statická alokace globálních proměnných (global scope)
 - alokace před spuštěním programu
 - uvolnění paměti po skončení programu
 - týká se i statických proměnných ve funkcích
 - globální proměnné je lepší moc nepoužívat
- toto vše byste měli už znát z C

Dynamická alokace v C

- alokace pomocí `malloc`, dealokace pomocí `free`
- je třeba správně spočítat potřebný počet bajtů
 - typicky něco jako *počet prvků* * `sizeof(prvek)`
- není typová kontrola (`malloc` vrací `void *`)
- pokud se alokace nepodaří, `malloc` vrací `NULL`
- alokovaná paměť není inicializovaná
- můžeme použít i v C++, ale v naprosté většině případů to není vůbec vhodné

Dynamická alokace v C++

- operátory `new`, `delete`, `delete []`
- dva základní typy alokace
 - alokace jednoho objektu / jedné hodnoty primitivního datového typu
 - alokace pole objektů / pole hodnot
- zadáváme typ alokovaného objektu a případně počet
 - potřebné místo je automaticky spočítáno

```
// alokuje paměť o velikosti sizeof(int)
int * ptr = new int;
```

```
// alokuje paměť o velikosti 20 * sizeof(int)
int * array = new int[20];
```

- při volání `new` je po alokaci volán konstruktor objektu
 - i primitivní datové typy se mohou inicializovat

```
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
```

- všimněte si: u primitivních datových typů je rozdíl mezi `new int` a `new int()`, u složených datových typů (objektů) to znamená totéž – není možno nechat objekt neinicializovaný
- při alokaci polí se vždy použije bezparametrický konstruktor
 - u primitivních datových typů zůstává paměť neinicializovaná

```
class A { /* ... */ }; // jako předtím
```

```
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];
// neinicializovaná paměť, před použitím je třeba nejdříve inicializovat
```

```
class B {
    int value;
public:
    B(int v) : value(v) {}
};
```

```
};
```

```
B * arrayB = new B[17]; // CHYBA! B nemá bezparametrický konstruktor
```

- pokud se alokace nepodaří, `new` vyhazuje výjimku `std::bad_alloc`
 - o výjimkách se budeme bavit později
- varianta operátoru `new` bez vyhazování výjimek: `new (nothrow)`
 - vrací `nullptr` (v C++03 `NULL`)

```
int * array = new (std::nothrow) int[1000000];  
if (array) {  
    // ...  
}
```

- dealokace pomocí `delete` / `delete []`
 - první oprátor používáme pro samostatné objekty, druhý pro pole
 - tohle kompilátor nepohlídá! proč?

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

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

- problémy s alokací a dealokací
 - 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: program `valgrind`

```
// tento program pravděpodobně spadne  
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 (na konci bloku u lokálních proměnných, na konci programu u globálních proměnných)

- dynamická dealokace (při zavolání `delete`)
 - který destruktork se zavolá?

```
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!
}
```

- pokud píšeme třídu, od které se (a) bude dědit, (b) instance potomků budou dealokovány skrze ukazatele na předka, pak je velmi vhodné používat virtuální destruktork

```
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

- ke kopírování objektů může dojít
 - při inicializaci z jiného objektu
 - při přiřazení

```
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; }
};
```

```
Point p1(11,21);
Point p2 = p1; // kopírování při inicializaci
Point p2(p1);  // ekvivalentní zápis
Point p2{p1};  // ekvivalentní zápis v C++11

p2.getX(); // vrátí 11

p2.setX(17);

p1.getX(); // vrátí opět 11, p1 a p2 jsou různé objekty

Point p3(0,0);

p3.getX(); // vrátí 0

p3 = p1; // kopírování při přiřazení

p3.getX(); // vrátí 11

p1.setX(99);

p3.getX(); // vrátí opět 11, p1 a p3 jsou různé objekty
```

- kopírování při inicializaci zajišťuje tzv. *kopírovací konstruktor*
 - i když žádný nedefinujeme, automaticky se vyrobí implicitní
 - má tedy někdy smysl definovat vlastní kopírovací konstruktor?
 - plytká (shallow) vs. hluboká (deep) kopie

// tento příklad budeme postupně rozvíjet

```
class IntArray {
```

```

    size_t size;
    int * data;
public:
    IntArray() : size(0), data(nullptr) {}
    IntArray(size_t s) : size(s), data(new int[size]) {
        for (size_t i = 0; i < size; ++i)
            data[i] = 0;
    }
    ~IntArray() {
        delete [] data;
    }
};

```

- plytká kopie kopíruje pouze hodnoty atributů
 - máme-li jako atribut ukazatel, kopíruje se jen jeho hodnota, ne to, na co ukazuje
 - to může být problém

```

int main() {
    IntArray arr1(10);
    IntArray arr2 = arr1;
} // na konci bloku se zavolá destruktory pro arr1 i arr2
// CHYBA! špatné použití delete na už uvolněný ukazatel!

```

- jedno řešení: zakázat kopírování
 - vhodné pro objekty, které nemá smysl kopírovat
 - v C++11 pomocí speciální syntaxe = **delete**
 - v C++03 se řešilo přesunem kopírovacího konstruktory do sekce **private**

```

// uvnitř třídy IntArray
IntArray (const IntArray &) = delete;

```

- druhé řešení: napsat si vlastní kopírovací konstruktor, který provede hlubokou kopii

```

// uvnitř třídy IntArray
IntArray (const IntArray & other)
    : size(other.size), data(new int[size]) {
    for (size_t i = 0; i < size; ++i)
        data[i] = other.data[i];
}

```

- všimněte si typu kopírovacího konstruktory, toto je jediná (rozumná) možnost

- kopírovací konstruktor nemůže brát vstup hodnotou (při volání hodnotou se vyrábí kopie právě pomocí kopírovacího konstruktoru)
- kdyby bral kopírovací konstruktor nekonstantní referenci, mohl by měnit kopírovaný objekt, to by bylo velmi nečekané

```
// nyní můžeme psát
IntArray arr1(10);
IntArray arr2 = arr1;
// arr1 a arr2 jsou teď dva různé objekty a různé jsou i
// jejich ukazatele data,
// každý ukazuje na jinou paměť o velikosti 10 * sizeof(int)
```

- už jsme tím vyřešili veškeré problémy s kopírováním?

```
int main() {
    IntArray arr1(10);
    IntArray arr2(20);
    arr2 = arr1; // kopírování při přiřazení
}
// dva problémy:
// * na konci je arr2.data = arr1.data
//   (stejný problém jako minule)
// * původní arr2.data se neuvolní (memory leak)
```

- je třeba ještě vyřešit kopírování při přiřazení
 - to řeší speciální metoda zvaná **operator=**
 - pozn. ve skutečnosti můžeme definovat i mnohé další operátory, podrobně to probereme až v jedné z pozdějších přednášek
- typ operátoru přiřazení může být různý, jedna možnost je

```
Object & operator=(const Object &)
– už je nám jasné, proč se parametr bere konstantní referencí
– návratový typ může být i jiný, např. void
```

- proč mít jako návratový typ přiřazovacího operátoru referenci na objekt?
 - je to konzistentní se způsobem, jak přiřazovací operátor funguje pro primitivní datové typy, což umožňuje řetězené přiřazení typu `a = b = c = 1`
 - některé části standardní knihovny toto chování očekávají
 - doporučení: pokud nemáte nějaký zvláštní důvod to dělat jinak, vraťte v operátoru přiřazení vždy referenci na objekt

```
// uvnitř třídy IntArray
IntArray & operator=(const IntArray & other) {
    delete [] data;
    size = other.size;
```

```

        data = new int[size];
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
        return *this; // vracíme referenci
                       // na tento objekt
    }
    // tento kód má několik drobných problémů

```

- problém se sebezpřirazením (self-assignment)
 - nemusí to nastat často, ale může se stát, že někdo napíše `a = a`
 - v tom případě dojde ve výše uvedeném kódu k chybě, proč?
- bývá zvykem detekovat sebezpřirazení

```

// uvnitř operátoru= třídy IntArray
// detekce sebezpřirazení
if (this == &other) return *this;

```

- kód v operátoru = částečně duplikuje kód z kopírovacího konstrukturu
 - nešlo by toho nějak využít?
 - odpovědí je tzv. *copy-and-swap* idiom

```

// uvnitř třídy IntArray, alternativní možnost operátoru =
void swap(IntArray & other) {
    // prohodí "vnitřnosti" tohoto objektu a other
    using std::swap;
    swap(size, other.size);
    swap(data, other.data);
}

// všimněte si volání hodnotou, které způsobí kopírování
IntArray & operator=(IntArray other) {
    swap(other);
    return *this;
}

```

- *copy-and-swap* idiom má různé výhody
 - neduplikujeme kód
 - dává prostor kompilátoru k optimalizaci (za jistých podmínek se kopie nemusí vůbec provést)
- ale může mít i drobné nevýhody
 - při sebezpřirazení se zbytečně dělá kopie (to nám ale nemusí moc vadit, sebezpřirazení je vzácné)
 - v původním kódu jsme mohli využít situace, kdy je velikost obou polí stejná


```

// uvnitř třídy IntArray, alternativní možnost operátoru =
IntArray & operator=(const IntArray & other) {
    if (this == &other) return *this;
    if (size != other.size) {
        delete [] data;
        size = other.size;
        data = new int[size];
    }
    // pokud jsou velikosti stejné, nemusíme
    // nic znova alokovat, prostě přepíšeme hodnoty
    for (size_t i = 0; i < size; ++i) {
        data[i] = other.data[i];
    }
    return *this;
}

```

- obecné doporučení (co je lepší?)
 - používejte copy-and-swap (výhody výrazně ve většině případů výrazně převažují nad nevýhodami)
 - až pokud zjistíte (profiling), že kopírování vás stojí moc času a je úzkým hrdlem vašeho programu, pak teprve přemýšlejte o tom, že byste to přepsali jinak
- abychom dokončili implementaci třídy IntArray, hodil by se nám ještě způsob, jak přistupovat k prvkům pole
 - proto si zde ukážeme implementaci ještě jednoho operátoru
 - podrobnější komentář k tomu, proč to děláme takhle, přijde jindy
 - ale můžete o tom zkusit přemýšlet už teď

```

// uvnitř třídy IntArray
// přístup pro čtení i zápis
int & operator[](size_t index) {
    return data[index];
}
// přístup jen pro čtení
const int & operator[](size_t index) const {
    return data[index];
}

```

Rule of three

- ještě se vrátíme ke kopírovacím konstruktorům
- pravidlo tří: pokud vaše třída definuje jednu z následujících věcí, pravděpodobně chcete ve skutečnosti definovat všechny tři:
 - destruktory
 - kopírovací konstruktor
 - kopírovací přiřazovací operátor

- zdůvodnění: třída, která definuje jednu z těchto tří věcí, zřejmě spravuje nějaký prostředek (paměť, soubor, apod.) a kompilátorem automaticky generovaný destruktorkopírovací konstruktorpřiřazovací operátorem pak zřejmě nedělá to, co chcete
- v C++11 Rule of five (C++11 má kromě kopírování i přesunování – *move semantics*, více o tom ve zvláštní přednášce)

Přetypování (casting) v C++

Přetypování v C

- jistě si vzpomínáte na přetypování v C (syntax `(typ)hodnota`)
- to je sice možné použít v C++, ale je to *velmi nedoporučované*
 - zejména proto, že to není typově korektní
- máte striktně zakázáno používat přetypování ve stylu C

Přetypování v C++ – `static_cast`

- syntaxe `static_cast<typ>(hodnota)`
- přetypování, které je možno provést staticky (během kompilace)
- typické použití:
 - primitivní datové typy (často ale funguje implicitně a není třeba uvádět `static_cast`)
 - změna celočíselné hodnoty na `enum`
 - změna ukazatele na `void *` a naopak
 - přetypování v nevirtuální objektové hierarchii
 - v tomto předmětu se nejspíše setkáte jen s prvními dvěma body
- nemusí být vždy bezpečné
 - nedefinované chování, např. pokud přetypujete hodnotu 3 na `enum`, který má jen dvě konstanty

Přetypování v C++ – `dynamic_cast`

- víme, že ukazatel na potomka se umí automaticky přetypovat na ukazatel na předka (a obdobně s referencemi)

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

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

- `dynamic_cast` umožňuje bezpečně přetypovat opačným směrem
 - funguje jen pokud je objektová hierarchie virtuální (tj. třída předka má některou metodu virtuální)
 - proč myslíte, že to tak je?

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

- rozhoduje se až za běhu programu

- pokud `ptr` ve skutečnosti ukazuje na objekt typu `B`, přetypování se provede
- pokud `ptr` ukazuje na něco jiného, výsledkem je `nullptr` (v C++03 `NULL`)
- funguje i s referencemi
 - protože nemáme žádnou null referenci, hází v druhém případě výjimku `std::bad_cast`
- použití `dynamic_cast` se často dá vyhnout vhodnou změnou objektové hierarchie, ale někdy se hodí
 - např. při řešení úkolu na cvičení tento týden

Další způsoby přetypování v C++

- existují i další operátory přetypování v C++, ale záměrně o nich nebudeme mluvit
- mají svá specifická použití, ale narušují typovou bezpečnost, což typicky nechceme