

ŽIVOTNÍ CYKLUS OBJEKTU; KONSTRUKTORY, DESTRUKTORY, KOPÍROVÁNÍ APOD.

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

7. března 2023

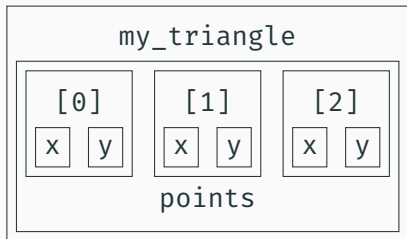
Objekty mohou obsahovat podobjekty

- položky složeného datového typu (**struct**, ntice, pole)
- úložiště podobjektu je vnořeno uvnitř úložiště objektu
- (podobjekt předka – uvidíme později)

```
struct point {  
    int x, y;  
};
```

```
struct triangle {  
    std::array<point, 3>  
        points;  
};
```

```
triangle my_triangle;
```

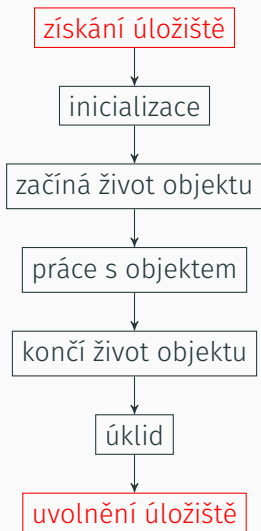


Vnitřní uložení hodnoty v objektu

- objekt je jednoduchého typu, nebo
- hodnota je přímo dána hodnotami podobjektů
 - viděli jsme: `std::tuple`, `std::array`, naše typy (s hodnotovými položkami)

Vnější uložení hodnoty

- hodnota je uložena někde *mimo objekt samotný*
- objekt se explicitně stará o její správu
 - (dnes začneme odhalovat, jak se to děje)
- dynamické kontejnery (`std::vector`, `std::map` apod.)



Automatické úložiště („zásobník“)

- lokální proměnné, parametry funkcí, dočasné objekty uvnitř funkcí
- prostor alokován na začátku bloku, dealokován na konci bloku

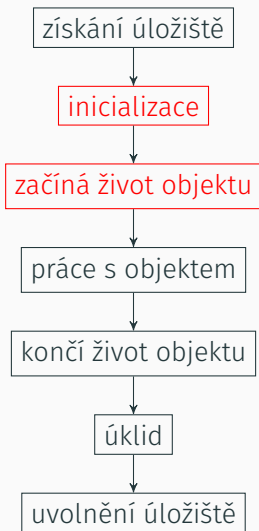
Statické úložiště

- globální proměnné, (statické proměnné)
- prostor je alokován při spuštění programu, dealokován při ukončení programu

Dynamické úložiště („halda“)

- explicitní žádost o alokaci/delokaci
- (uvidíme příště)

Mimo záběr předmětu – **úložiště pro vlákna**

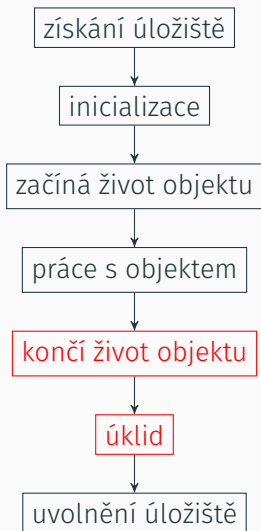


Inicializace

- podle deklarace
- součástí inicializace objektu je inicializace jeho podobjektů
 - v pořadí podle jejich deklarace
- proces inicializace můžeme ovlivnit
 - speciální metoda – *konstruktor*
 - (uvidíme za chvíli)

Život objektu začíná

- jakmile je úspěšně dokončena inicializace (tj. skončí tělo případného konstrukturu)
- všimněte si: život podobjektů začíná před životem objektu



Život objektu končí

- dočasný objekt: na konci příkazu
 - není-li svázán s referencí (viz dále)
- lokální proměnná: na konci bloku
- parametr funkce: na konci funkce
- podobjekt: po konci života objektu
- globální proměnná: na konci programu
- dynamicky alokovaný objekt: explicitně
- obecně: v pořadí opačném k začátku života

Úklid

- můžeme ovlivnit, co se stane po konci
- speciální metoda – *destruktor*
- volá se automaticky v případech výše
 - předtím než skončí život podobjektů

Materializace dočasné hodnoty – vytvoření dočasného objektu

- (pravidla od C++17, zde lehce zjednodušeno)
- přístup k položce (pomocí „tečky“, indexace pole) nebo metodě
- „zahození“ hodnoty
 - ignorování návratové hodnoty funkce apod.
- svázání hodnoty s referencí
 - v tomto případě je život dočasného objektu prodloužen do konce existence této reference
 - `const typ&`, `typ&&` (uvidíme za chvíli)
- k materializaci zejména **nedochází**
 - při vracení dočasné hodnoty z funkce
 - „na pravé straně“ inicializace – místo toho se dočasná hodnota použije přímo pro inicializaci cílového objektu (bez kopírování)

Vlastnictví

- „kdo je zodpovědný za úklid“

Syntaktické vlastnictví (implicitní)

- objekt vlastní své podobjekty, funkce vlastní lokální proměnné
- „úklid“ zajišťuje přímo sémantika jazyka

Sémantické vlastnictví (explicitní)

- příklad: dynamické kontejnery (`std::vector`, apod.) vlastní objekty, které jsou v nich uloženy
- „úklid“ musí zařídit programátor (v destrukturu)

Kolik metod má tento typ?

```
struct point {  
    int x, y;  
};
```

- šest (před C++11 čtyři)
- bezparametrický konstruktor
- kopírovací konstruktor
- přesouvací (move) konstruktor
- kopírovací operátor přiřazení
- přesouvací (move) operátor přiřazení
- destruktork

Konstruktor

- speciální metoda, stejného jména jako aktuální typ
- nemá návratový typ
- má tzv. **inicializační sekci**
 - možnost ovlivnit, jak se budou inicializovat položky
 - má přednost před inicializací u deklarace
- konstruktory můžeme přetěžovat jako jiné metody
 - navíc můžeme použít tzv. *delegování*
- v těle smíme přistupovat k podobjektům a volat metody
 - i když život aktuálního objektu ještě nezačal

```
struct rectangle {  
    int width, height;  
    rectangle(int w, int h)  
        : width{ w }, height{ h } {}  
    rectangle(int x) : rectangle{ x, x } {}  
};
```

Inicializační sekce

- preferujte před přiřazením v těle konstruktoru
- inicializace přímo hodnotou může být efektivnější než implicitní inicializace + přiřazení
- někdy ani nemáte jinou možnost
 - tělo konstruktoru **není** magické

```
struct example {  
    const int x = 0;  
  
    example(int val) {  
        x = val; // cannot assign to const  
    }  
};
```

Implicitní bezparametrický konstruktork

- inicializace všech položek dle deklarace
- není vygenerován, jakmile napíšeme nějaký vlastní konstruktork
- chceme-li jej přesto nechat vygenerovat, použijeme
= **default**;

```
struct counter {  
    std::map<char, int> freq;  
  
    counter() = default;  
  
    counter(const std::map<char, int>& f)  
        : freq{ f } {}  
};
```

Kopírovací konstruktor `typ(const typ&)`

- bere konstantní referenci na aktuální typ
 - při kopírování není slušné měnit originál
- určuje, jakým způsobem se kopíruje *při inicializaci*

Implicitní kopírovací konstruktor

- inicializace všech položek kopií
 - volají se jejich kopírovací konstruktory
- kopírování můžeme *zakázat* pomocí = **delete**;
- není vygenerován, pokud je některá položka nekopírovatelná nebo pokud definujeme *přesouvací* speciální metody (viz dále)

operator=

- musí být metoda
- použije se ve chvíli, kdy přiřazujeme do *existujícího objektu*
 - tedy *ne při inicializaci*
- konvence: přiřazovací operátor vrací referenci na ***this**
 - umožňuje řetězení `a = b = c = ...`

```
maybe_int& operator=(int v) {  
    _just = true;  
    _value = v;  
    return *this;  
}
```

Kopírovací přiřazení `typ& operator=(const typ&)`

- určuje, jakým způsobem se kopíruje *při přiřazení*

Implicitní kopírovací přiřazení

- přiřazení po jednotlivých položkách
 - volají se jejich operátory =
- opět můžeme zakázat pomocí = **delete**;
- není vygenerováno, pokud se do některé položky nedá přiřadit (**const** nebo reference) nebo pokud definujeme *přesouvací* speciální metody

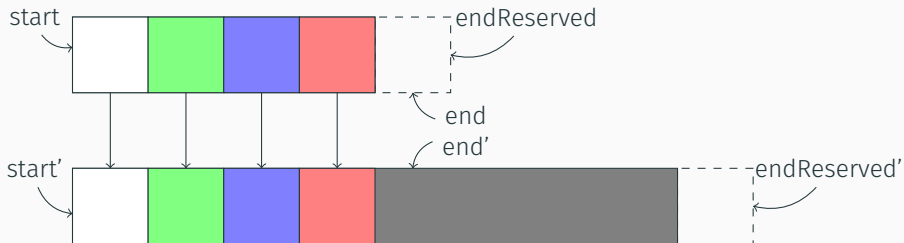
Poznámka: Správná implementace operátoru = musí bez chyb přežít i sebe-přiřazení `x = x` (hraje roli ve chvíli, kdy objekt spravuje nějaké zdroje, uvidíme později).

Destruktor

- speciální metoda, jméno `~typ`
- nemá návratový typ, nemá parametry
- určen pro úklid po konci života objektu
- pořadí volání opačné k pořadí volání konstruktorů
 - tam, kde je to dáno sémantikou jazyka
 - tj. netýká se např. objektů uvnitř `std::vector`
- *všimněte si*: destrukce objektů je *deterministická*
 - toho využívá princip RAII, uvidíme později

Typická implementace

- rezervovaná paměť, část z ní je obsazená
- `std::vector` si drží tři adresy:
 - začátek rezervované, konec obsazené, konec rezervované
- když je rezervovaná paměť plná, provede se alokace nové a prvky se do ní zkopírují (od C++11 *přesunou*, viz dále)



Motivace

- kopírujeme objekt, který vzápětí zanikne
- OK pro malé (snadno kopírovatelné) objekty
- ale co když je kopírování objektu *drahé*?
 - nebo nemožné?

```
// data is a container
results.push_back(data);
data.clear();
```

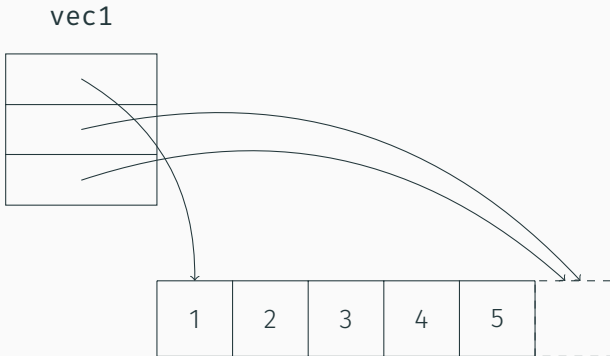
Pomíjivý objekt

- ve standardu *r-hodnota* (*rvalue*)
 - historicky podle toho, že může být jen na pravé straně přiřazení
- dočasný objekt
- jiný objekt *explicitně označený jako pomíjivý*
 - pomocí speciální funkce `std::move`
- položka pomíjivého objektu (**struct**, ntice, pole)
- (detaily pro odvážné:
https://en.cppreference.com/w/cpp/language/value_category)

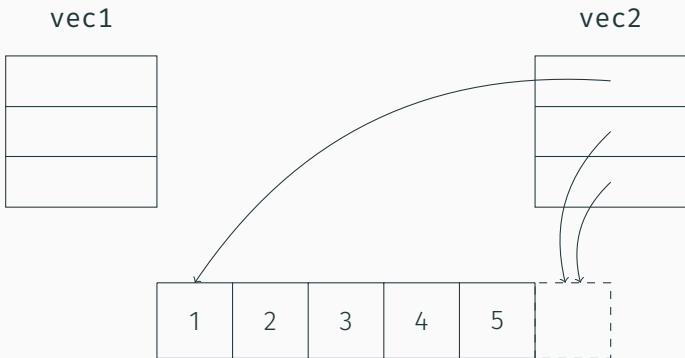
Smysl `std::move`

- „chovej se k tomuto objektu, jako by měl za chvíli zaniknout“

```
std::vector vec1 = { 1, 2, 3, 4, 5 };
```



```
std::vector vec1 = { 1, 2, 3, 4, 5 };  
std::vector vec2 = std::move(vec1);
```



R-hodnotová reference `typ&&`

- váže se pouze k *pomíjivým objektům*
 - dočasné objekty, explicitní `std::move`
- při přetěžování má přednost před klasickou referencí
- použití: situace, kdy chceme nějak využít pomíjivosti objektu
 - přesouvací konstruktor / operátor =
 - některé situace u přetěžování operátorů

Přesouvací konstruktor `typ(typ&&)`

- bere r-hodnotovou referenci, tj. referenci na *pomíjivý objekt*
- určuje, jakým způsobem se přesouvá *při inicializaci*
- má možnost „vykrást“ tomuto objektu vnitřní reprezentaci
- život „vykradeného“ objektu tím ovšem nekončí
 - až skončí, bude se volat jeho destruktork
 - je třeba zaručit, aby toto volání proběhlo bez problémů
- záruka std. knihovny: „vykradený“ objekt je ve validním stavu
 - (typicky implementováno jako prázdný kontejner)
 - *doporučení*: mít u vlastních typů stejnou záruku
- uživatelská definice přesouvacího konstruktoru by měla být označena specifikátorem **noexcept**
 - píše se za hlavičku metody, před inicializační sekci
 - přesné důvody nad rámec předmětu
 - souvisí s výjimkami a se zárukami slibovanými **std::vectorem**

Implicitní přesouvací konstruktor

- jednoduché typy: totéž, co kopírování
- složené typy: inicializace položek přesunutím
 - volají se jejich přesouvací konstruktory
- vygeneruje se pouze tehdy, nedefinujeme-li vlastní kopírovací metody, přesouvací přiřazení, destruktory

Přesouvací přiřazení `typ& operator=(typ&&)`

- určuje, jakým způsobem se přesouvá *při přiřazení*
- jinak pro ně platí totéž, co pro přesouvací konstruktor

Implicitní přesouvací přiřazení

- jednoduché typy: totéž, co kopírovací přiřazení
- jinak: přesouvací přiřazení po jednotlivých položkách
 - volají se jejich přesouvací operátory =
- vygeneruje se pouze tehdy, nedefinujeme-li vlastní kopírovací metody, přesouvací konstruktor, destruktor

VYNECHÁNÍ KOPIÍ (*COPY ELISION*)

- překladač může za určitých okolností vynechat kopírování
 - od C++17 v některých případech *musí*
- objekt je přímo vytvořen v cílovém místě
- (děje se typicky i při vypnutých optimalizacích -O0)

Povinné od C++17 (před C++17 jen optimalizace)

- inicializace objektu dočasnou hodnotou
- **return** dočasné hodnoty z funkce
 - tzv. *return value optimization* (RVO)
- dáno pravidly pro materializaci dočasných hodnot

Volitelné (překladače většinou provádějí, je-li to možné)

- **return** hodnoty lokální proměnné z funkce
 - tzv. *named return value optimization* (NRVO)

```
std::vector<int> create_vector() {
    const int size = 1 << 24;
    std::vector<int> result;
    result.resize(size);
    for (int i = 0; i < size; ++i) {
        result[i] = i;
    }
    return result;
}

int main() {
    auto vec = create_vector();
}
```

- moderní překladače zde provedou jen jednu alokaci paměti pro vektor (i bez zapnutých optimalizací)

(jen pro zajímavost)

- volající funkce (v příkladu `main`) vyhradí místo pro lokální proměnnou
- adresu tohoto místa pošle jako tajný parametr volané funkci (v příkladu `create_vector`)
- volaná funkce provede inicializaci objektu přímo v zadaném místě paměti

```
std::vector<int> create_vector(bool flag) {  
    std::vector result1 = {1, 2, 3, 4};  
    std::vector result2 = {5, 6, 7, 8};  
    return flag ? result1 : result2;  
}
```

```
int main() {  
    auto vec = create_vector(true);  
}
```

(překladač předem neví, který vektor má být v zadané paměti)

Implicitní `std::move`

- za **return** stojí hodnotový parametr funkce
 - tj. ne reference, ne složitější výraz
- sice nedojde ke *copy elision*, ale použije se přesun
 - jako by bylo jméno parametru obaleno `std::move`

Důsledek

- vzpomeňte si na implementaci operátoru `+` z minulé přednášky

Vkládání „velkých“ objektů do kontejnerů

- metody `push_back` apod. mají přetížení pro r-hodnotové reference

```
// data is a container
results.push_back(std::move(data));
data.clear(); // might not be needed
```

Kopírování hodnoty do položky v konstruktoru

- `typ(const big& b) : my_b(b) {}` vs. `typ(big b) : my_b(std::move(b)) {}`
- volání s dočasnou hodnotou: 1 kopie vs. 1 přesun
- volání s proměnnou (bez `move`): 1 kopie vs. 1 kopie + 1 přesun
- volání s proměnnou s `move`: 1 kopie vs. 2 přesuny
- jsou-li přesuny „levné“, druhá možnost je lepší

`std::array`

- není dynamický kontejner, položky drží vevnitř
- přesouvání `std::array` znamená přesun jednotlivých položek
 - může mít smysl, jsou-li položky „velké“ s levným přesunem
 - pro `std::array<int>` apod. to ale smysl nemá (efektivně to je kopie)

Rule of Five (the rule formerly known as Rule of Three)

- implementujete-li jednu z těchto věcí, implementujte *všechny*
 - kopírovací a přesouvací konstruktor
 - kopírovací a přesouvací operátor přiřazení
 - destruktor
- pokud některá z těchto metod nedává smysl, *zakažte* ji
 - konstrukce = **delete**;
- pokud některá z těchto metod může zůstat implicitní, explicitně ji uveďte s notací = **default**;
 - pravidla pro to, kdy se generování potlačí, jsou složitá
 - čitelnost: je vidět, že implicitní chování používáte schválně

Rule of Zero

- pokud nemusíte, nepište žádnou z výše uvedených metod
 - nespravujete zdroje, nepíšete datovou strukturu

Idiom „copy and swap“

- alternativní implementace přiřazovacích operátorů
- metoda / funkce **swap** pro prohození
- jediný operátor přiřazení s *hodnotovým parametrem*

```
reporter& operator=(reporter other) {  
    swap(other);  
    return *this;  
}
```

```
void swap(reporter& other) {  
    // ...  
}
```

C++ “move” semantics are simple, and unchanged since C++11. But they are still widely misunderstood, sometimes because of unclear teaching and sometimes because of a desire to view move as something else instead of what it is. (Herb Sutter)