Životní cyklus objektu; konstruktory, destruktory, kopírování apod.

PB161 Programování v jazyce C++

Nikola Beneš

12. března 2024

Podobjekty

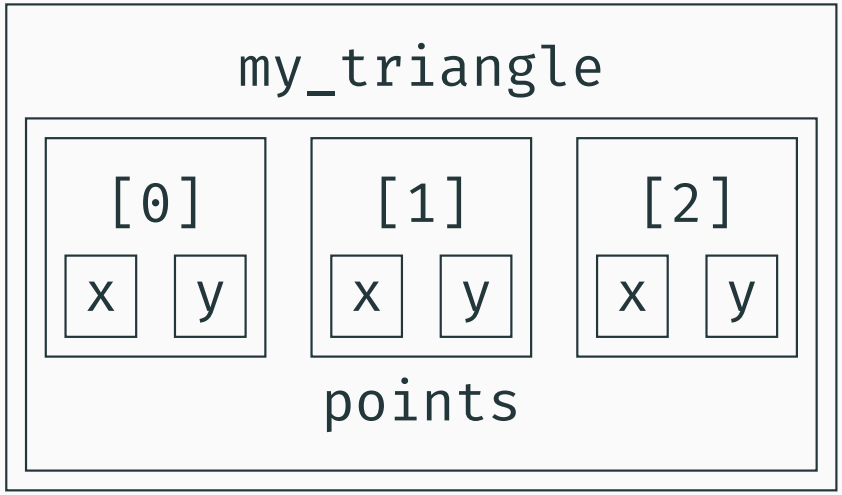
Objekty mohou obsahovat **podobjekty**

- 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 a vnější uložení hodnot

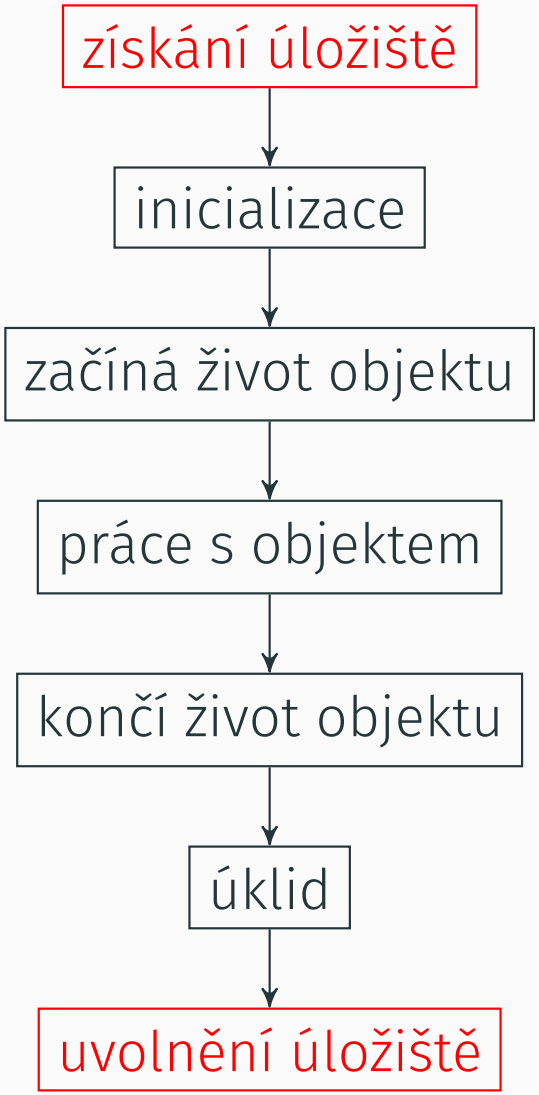
Vnitřní uložení hodnoty v objektu

- 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.)

Životní cyklus objektu



Automatické úložiště („zásobník“)

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

Statické úložiště

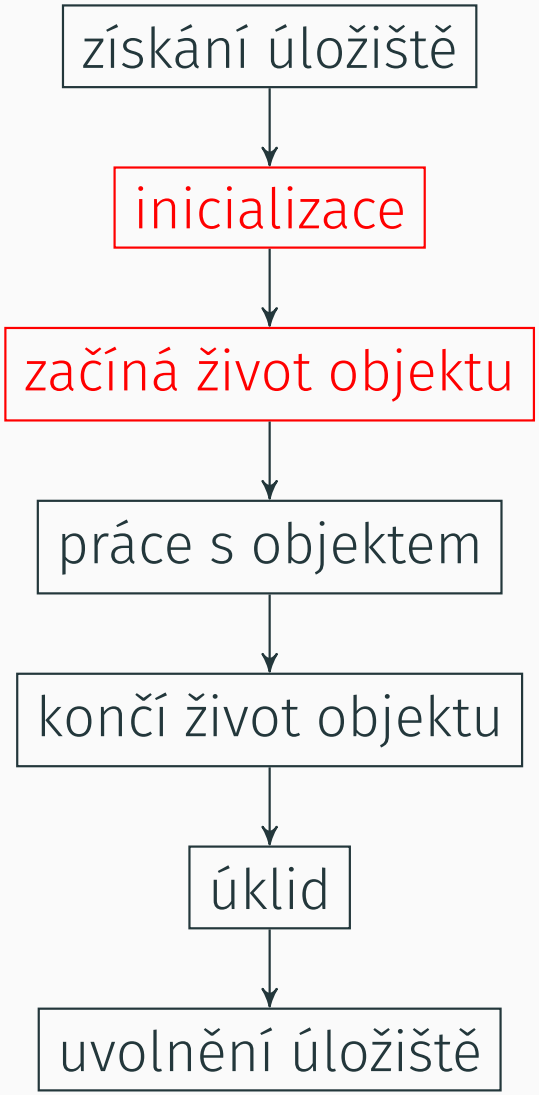
- globální proměnné, (statické proměnné)
- prostor je alokován při spuštění programu, dealokován při ukončení programu

Dynamické úložiště („halda“)

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

Mimo záběr předmětu – **úložiště pro vlákna**

Životní cyklus objektu



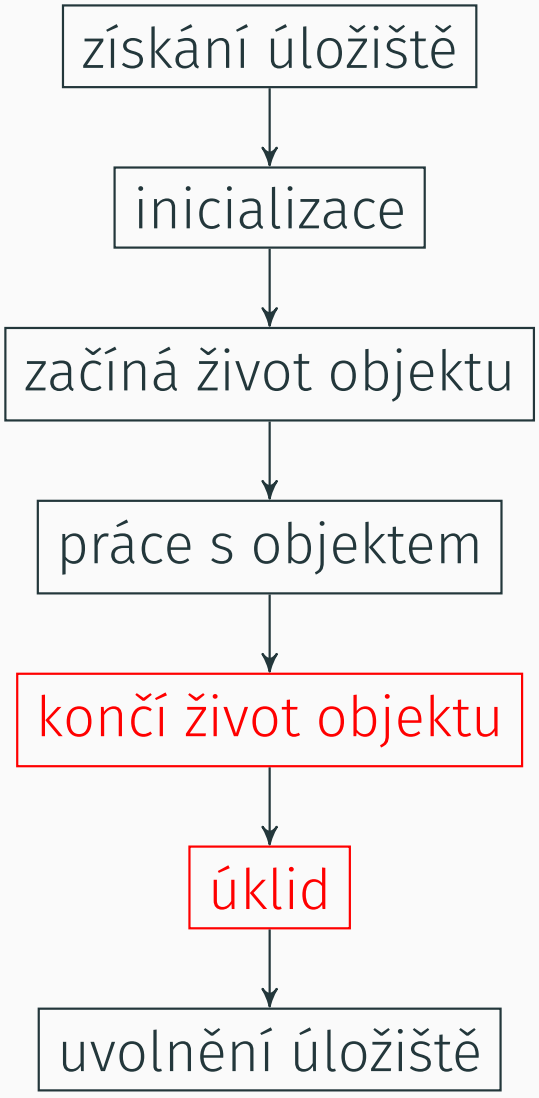
Inicializace

- 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 konstruktoru)
- všimněte si: život podobjektů začíná před životem objektu

Životní cyklus 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

Dočasné hodnoty a objekty

temp.cpp

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[1]
  - `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)

[1] neplatí pro vracení referencí z funkce

Pojem vlastnictví

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 destruktoru)

Implicitně generované metody

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
- destruktor

ctor1.cpp

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 } {}
```

init_sec.cpp

Inicializační sekce

- preferujte před přiřazením v těle konstruktoru
- inicializace přimo 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
    }
};
```

ctor2.cpp

Implicitní bezparametrický konstruktor

- inicializace všech položek dle deklarace
- není vygenerován, jakmile napíšeme nějaký vlastní konstruktor
- 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 } {}
};
```

Kopírování

copy_ctor.cpp

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)

Přiřazení

op_assign.cpp

`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;
}
```

Kopírování při přiřazení

copy_both.cpp

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).

Destrukce

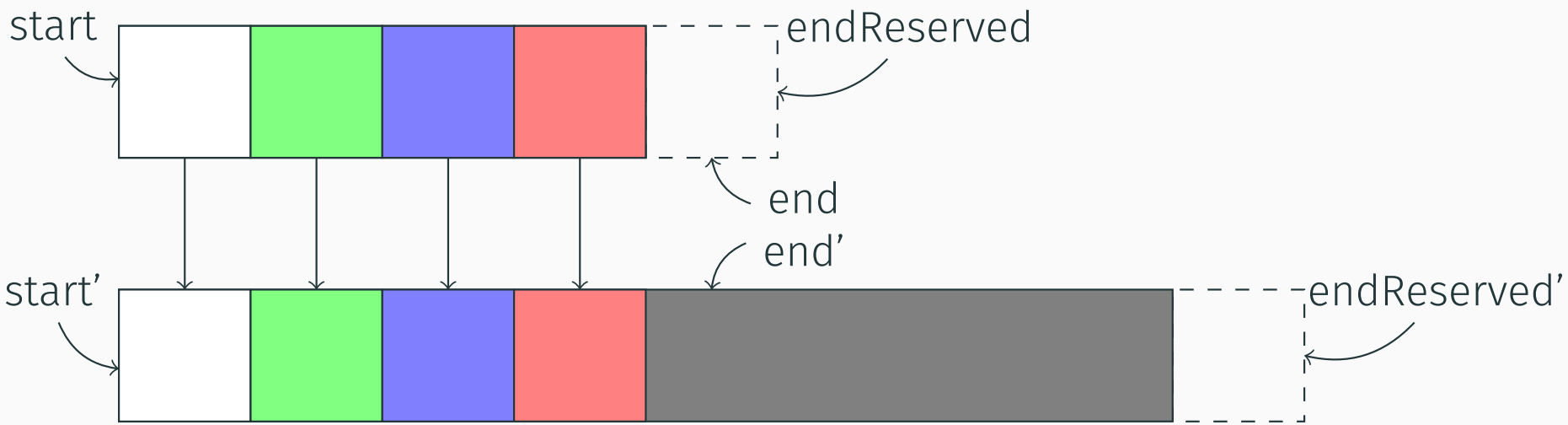
dtor.cpp

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)



Přesouvací (*move*) sémantika

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();
```

Přesouvací (move) sémantika

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)

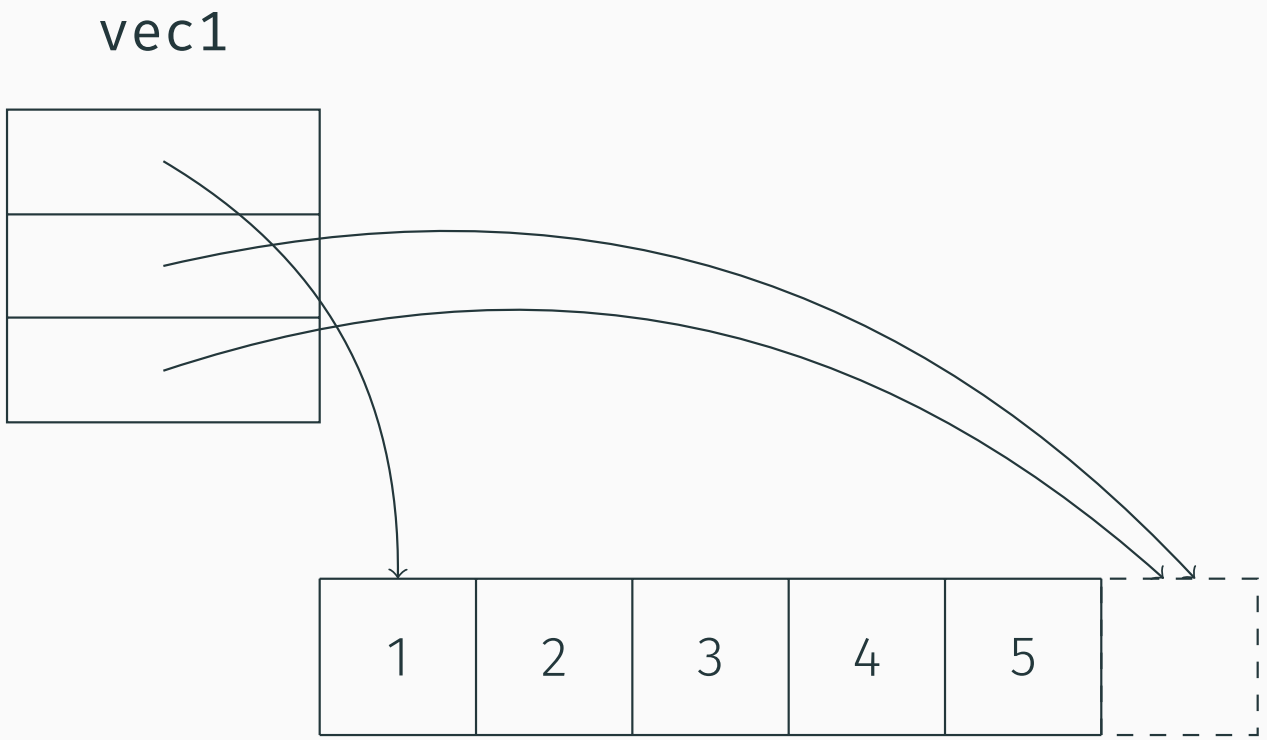
Smysl `std::move`

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

Ilustrace přesouvací sémantiky

move.cpp

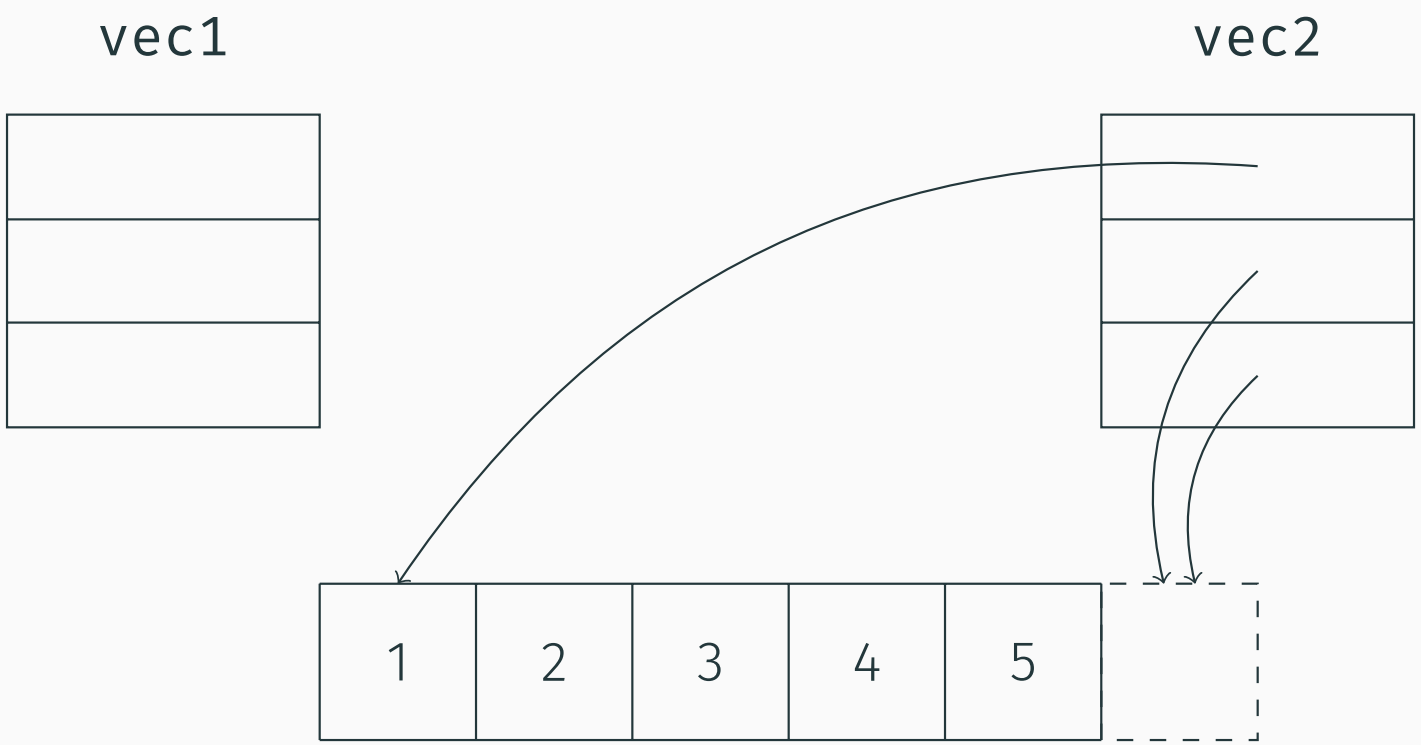
```
std::vector vec1 = { 1, 2, 3, 4, 5 };
```



Ilustrace přesouvací sémantiky

move.cpp

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



Přesouvací (*move*) sémantika

`rvalue_ref.cpp`

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

move_ctor.cpp

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 destruktor
  - je třeba zaručit, aby toto volání proběhlo bez problémů
- záruka std. knihovny: „vykradený“ objekt je ve validním stavu
  - smí se na něm zavolat destruktor, přiřazení, příp. další metody
  - (typicky implementováno jako prázdný kontejner)
  - *doporučení:* mít u vlastních typů stejnou záruku

Přesouvání

move_ctor.cpp

Přesouvací konstruktor `typ(typ&&)`

- 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::vector`em

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, destruktor

move_both.cpp

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

Kopie vs. přesun

copy_move.cpp

Jednoduché typy (`int`, `double`, atd.)

- kopie a přesun je totéž

Složené typy s vnitřním uložením hodnoty

- `std::tuple`, `std::array`, `struct` s implicitním kopírováním/přesunem
- přesun = přesun jednotlivých položek
- tedy např. pro `std::array<int>` je kopie a přesun totéž

Složené typy s vnějším uložením hodnoty

- `std::vector`, `std::set`, `std::map`, …
- přesun je „levný“
  - typicky kopie ukazatele / ukazatelů + uvedení „vykradeného“ objektu do validního (prázdného) stavu

Automatický přesun

move_implicit.cpp

Implicitní `std::move` za `return`

- lokální proměnná nebo parametr funkce
- hodnotového typu nebo typu r-hodnotové reference
- (nefunguje pro složitější výrazy nebo l-hodnotové reference)

- jméno proměnné je jakoby obaleno `std::move`
- v případě lokální proměnné může ještě dojít k vynechání přesunu úplně (*copy elision*, viz další slajd)

Důsledek

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

Vynechání kopií (*copy elision*)

elision.cpp

- překladač může za určitých okolností vynechat kopírování/přesun (pomocí konstruktoru) úplně
  - 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
- 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)

https://en.cppreference.com/w/cpp/language/copy_elision

Implementace RVO / NRVO

(jen pro zajímavost)

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

Důsledky přesouvací sémantiky (a vynechání kopií)

Vracení hodnot z funkcí

- dočasná hodnota – žádné kopie, žádné přesuny
- lokální hodnotová proměnná – buď přesun nebo nic
- hodnotový parametr – přesun
- složitější výrazy vedou vždy ke kopírování
  - OK pro malé objekty, ale ne pro velké (kontejnery apod.)

- úplné vynechání kopií/přesunů funguje jen při inicializaci
  - ne při přiřazení
  - preferujte inicializaci před přiřazením

Důsledky přesouvací sémantiky (a vynechání kopií)

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();
```

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

Pravidla

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

copy_and_swap.cpp

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)*