# Kopírovacia a presúvacia sémantika v C++

Martin Drozda

4. novembra 2019

## 1 Úvod

Presúvacia (move) sémantika je súčasťou jazyka C++ od štandardu C++11. Cieľom presúvacej sémantiky je nahradenie neefektívneho kopírovania objektov ich presunom, pričom po presune z objektu `obj0` do objektu `obj1` ostane objekt `obj0` prázdny. Pozrime si nasledujúci príklad:

```
std::vector<Token> v;
Token t;
v.push_back(t);
```

Pri použití kopírovania je objekt`t` prekopírovaný do vektora `v`. Takéto kopírovanie môže byť neefektívne, ak trieda obsahuje veľké množstvo členov, ktoré je potrebné kopírovať. Uvažujme nasledovnú definíciu triedy `Token`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;
};
```

V tomto prípade pri kopírovaní objektu typu `Token` je potrebné prekopírovanie obsahu vektora `vec`. Lepšie riešenie sa javí presunutie vektora `vec` bez jeho kopírovania (teda bez kopírovania prvkov, ktoré obsahuje). Kopírovaniu premennej `a` nie je možné zabrániť, pretože jej "presun" pomocou smerníkov je rovnako náročný ako jej prekopírovanie.

Pri kopírovaní je zavolaný kopírovací konštruktor, ktorý môže byť implementovaný nasledovne:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;

  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
};
```

V C++ existuje "nepísané" pravidlo, ktoré v prípade, že trieda definuje kopírovací konštruktor, operátor priradenia alebo deštruktor odporúča definovať všetky tieto tzv. špeciálne metódy. Dôvod je taký, že priraďovanie

by malo odzrkadľovať kopírovanie a deštrukcia objektu v tomto prípade by mala byť špecifická.[1] V našom prípade z dôvodu stručnosti definujeme len kopírovací konštruktor.

Pri presúvaní je zavolaný presúvací konštruktor, ktorý môže byť implementovaný nasledovne:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;

  Token(const Token& t0): vec(t0.vec), a(t0.a) {}

  Token(Token&& t0): vec(std::move(t0.vec)), a(t0.a) {}
};
```

Podobne ako v prípade kopírovania je odporúčané definovať presúvací konštruktor, ako aj operátor priradenia pre presúvanie.[2]

## 2 Presúvanie

V prípade presúvania je potrebné, aby kompilátor pochopil náš zámer, teda využitie presúvania namiesto kopírovania. Uvažujme nasledujúci kód:

```
std::vector<Token> v;
Token t;
v.push_back(std::move(t));
```

V tomto prípade sme pretypovali `t` tak, aby nastalo zavolanie presúvacieho konštruktora a nie kopírovacieho konštruktora. Ak by sme naše riešenie otestovali, zistili by sme, že presúvanie napriek našej snahe nenastalo. Presúvanie totiž predpokladá, že presúvací konštruktor, ako aj operátor priradenia pre presúvanie nemôžu generovať výnimku. Tento stav dosiahneme použitím `noexcept`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;

  Token(const Token& t0): vec(t0.vec), a(t0.a) {}

  Token(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
  }
```

[1] https://en.wikipedia.org/wiki/Rule_of_three_(C++_programming)
[2] https://cpppatterns.com/patterns/rule-of-five.html

```
  Token& operator=(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
    return *this;
  }
};
```

V prípade, že nastane výnimka pri kopírovaní je situácia zvládnuteľná, pretože stále existujú pôvodné objekty, z ktorých sa kopíruje. Pri presune tieto objekty nemusia existovať v pôvodnom stave a teda v prípade, že nastane výnimka nevieme pôvodný stav obnoviť. Z podobného dôvodu sú všetky deštuktory označené ako `noexcept`, pretože ak zlyhá deštrukcia nastane stav, keď ďalej nevieme pokračovať. Kompletná implementácia triedy `Token` pri dodržaní spomenutých odporúčaní je potom nasledovná:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  //default ctor
  Token() = default;

  //copy
  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
  Token& operator=(const Token& t0) {
    vec = t0.vec;
    a = t0.a;
    return *this;
  }

  //move
  Token(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
  }
  Token& operator=(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
    return *this;
  }

  //default dtor
  ~Token() = default;
};
```

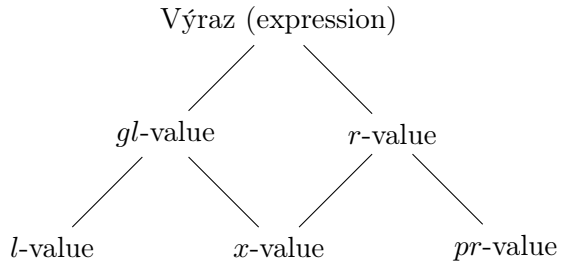
# 3 std::move

`std::move` nič nepresúva, jedná sa len o pretypovanie tak, aby bol zavolaný presúvací konštruktor. `std::move` je implementovaný nasledovne:

```
template <typename T>
typename remove_reference<T>::type&& move(T&& arg) {
  return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Je zrejmé, že `std::move` je implementovaný ako `static_cast` na univerzálnu referenciu &&. `std::move` sa využíva na pretypovanie $l$-hodnoty ($l$-value) na $r$-hodnotu ($r$-value). Pretože táto $r$-hodnota nevznikla prirodzene napr. ako konštanta 1 alebo 3.4, je takáto pretypovaná hodnota nazývaná $x$-hodnota ($x$-value).

Pred uvedením štandardu C++11 bola potrebná len $l$-hodnota a $r$-hodnotu, pričom $l$-hodnota je uložená na pamäťovom mieste, ktoré vieme určiť napr. pomocou operátora & a $r$-hodnota je všetko, čo nie je $l$-hodnota. $r$-hodnota môže tiež mať pamäťové miesto (niekedy existuje len v registri procesora), ale nevieme ho určiť, pretože operátor & nie je možné aplikovať na $r$-hodnotu napr. &3.4 nie je možné skompilovať.




Obr. 1: Druhy hodnôt zavedené v C++11.

Obr. 1 zobrazuje rôzne druhy hodnôt, ktoré boli zavedené v C++11, pričom $gl$-value je generalized $l$-value a $pr$-value je pure $l$-value. $pr$-value je hodnota, ktorá vznikla ako $r$-hodnota, teda napr. už spomínané konštanty 1, 3.4 a tiež dočasné objekty:

```
Token t0, t1;
Token t = t0 + t1;
```

Pri sčítaní objektov `t0` a `t1`, za predpokladu, že sme definovali operátor + pre triedu `Token`, vznikne dočasný objekt, ktorý je prekopírovaný resp. presunutý do `t`.

# 4 Presúvanie pri zmene veľkosti vektora

Kopírovanie vzniká bohužiaľ častejšie ako predpokladáme, a preto je použitie presúvacej sémantiky kľúčové. Majme nasledujúci vektor, ktorý obsahuje dva objekty typu `Token`:

```
std::vector<Token> v(0);
Token t0, t1;
v.push_back(std::move(t0));
v.push_back(std::move(t1));
```

V prípade, že do vektoru `v` pridáme ďalší (tretí) objekt nastane zmena (alokovanej) veľkosti vektora tak, aby sa tam tento objekt zmestil (pre jednoduchosť predpokladajme, že zmena alokovanej veľkosti v našom prípade naozaj nastane aj pri zmene veľkosti z 2 na 3). Vznikne nové pole, do ktorého sú prekopírované pôvodné dva objekty a následovne je ešte pridaný tretí objekt. Pôvodné dva objekty sú deštruované, bude teda dvakrát zavolaný deštruktor. Uvažujme teda nasledovný príklad:

```
std::vector<Token> v(0);
Token t0, t1;
v.push_back(std::move(t0));
v.push_back(std::move(t1));

Token t;
v.push_back(std::move(t));
```

Po pridaní objektu `t` by sme sa pri zmene veľkosti vektora `v` radi vyhli kopírovaniu a využili presúvanie. Z tohto dôvodu je potrebné, aby trieda `Token` obsahovala presúvací konštruktor, v opačnom prípade nastane kopírovanie.

# 5 Automatické generovanie

V prípade, že trieda neobsahuje vlastný prednastavený (default) konštruktor, kopírovací konštruktor, operátor priradenia a deštruktor sú tieto automaticky generované kompilátorom. Presnejšie povedané, sú generované ak sú potrebné. Ak napr. kód nevyžaduje priraďovanie, kompilátor v tichosti preskočí generovanie operátora priradenia, pretože nie je potrebný.

Podobne je to s presúvacím konštruktorom a operátorom priradenia pre presúvanie, ktoré sú automaticky generované s nasledujúcimi výnimkami. Ak trieda obsahuje kopírovací konštruktor, alebo operátor priradenia, potom nie sú generované. Dôvod je prechod z kompilátorov, ktoré podporujú štandard pred C++11. Pri prechode na novší kompilátor s podporou C++11 by mohlo nastať automatické generovanie presúvacieho konštruktora a operátora priradenia pre presúvanie, čo by mohlo neočakávane pozmeniť pôvodný zmysel kódu.

Všimnime si, že existencia presúvacieho konštruktora nespôsobí, že kopírovací konštruktor nebude automaticky generovaný.

V prípade, že neplánujeme využiť presúvaciu sémantiku je vhodné, aby presúvací konštruktor a operátor priradenia pre presúvanie boli označené ako `= delete`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  //default ctor
  Token() = default;
```

```
  //copy
  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
  Token& operator=(const Token& t0) {
    vec = t0.vec;
    a = t0.a;
    return *this;
  }

  //move
  Token(Token&& t0) = delete;
  Token& operator=(Token&& t0) = delete;

  //default dtor
  ~Token() = default;
};
```

# 6 Bez presúvacej sémantiky

Pred zavedením štandardu C++11 bolo možné dosiahnuť efektívnosť presúvacej sémantiky pomocou smerníkov:

```
std::vector<Token*> v;
Token* t = new Token;
v.push_back(t);
```

Objekty vznikajú dynamicky pomocou `new` a ich adresy sú prekopírované do kontajnera, v našom prípade do vektora. Výhodou tohto prístupu je, že nastane výlučne kopírovanie adries objektov, čo je možné považovať za najefektívnejší spôsob uloženia objektu do kontajnera. Uvažujme nasledovný prípad zmeny veľkosti vektora:

```
std::vector<Token*> v(0);
Token* t0 = new Token;
Token* t1 = new Token;
v.push_back(t0);
v.push_back(t1);

Token* t = new Token;
v.push_back(t);
```

V tomto prípade, pri zmene veľkosti kontajnera nastane prekopírovanie adries dvoch existujúcich objektov a pridanie ďalšej adresy do novovytvoreného poľa s dostatočnou veľkosťou.

Nevýhodou práce so smerníkmi v porovnaní s presúvacou sémantikou je možnosť, že niektorý objekt nebol vytvorený, teda môže nastať vloženie `null_ptr`. Existuje veľké množstvo kódu, ktoré bolo napísané pre zavedením C++11 a z rôznych dôvodov pokračoval jeho vývoj bez použitia presúvacej sémantiky. Z tohto dôvodu je potrebná aj znalosť návrhových vzorov, ktoré presúvaciu sémantiku nevyužívajú.

STL (Standard template library) má plnú podporu presúvacej sémantiky. Z praktického hľadiska to znamená, že ak napr. potrebujeme presunúť

objekty z jedného vektora do druhého, za predpokladu existencie presúvacieho konštruktora (hoci aj automaticky generovaného) je možné, že výsledný skompilovaný kód bude efektívnejší.