Semantyka przenoszenia obiektów

dr inż. Ireneusz Szcześniak

jesień 2017 roku

Problem: kopiowanie obiektów

- Obiekt może być kopiowany, kiedy:
  - jest przekazywany przez wartość do funkcji,
  - jest zwracany przez wartość z funkcji,
  - jest wsadzany do kontenera,
  - jego wartość jest zamieniana z wartością innego obiektu.
- Jeżeli obiekty są duże, to częste kopiowanie obiektu doprowadza do spowolnienia programu.
- Kopiowanie obiektów jest realizowane przez konstruktor kopiujący lub kopiujący operator przypisania.
- Przy kopiowaniu, obiekt źródłowy i obiekt docelowy mogą znajdować się w różnych miejscach na stosie albo stercie.
- Obiekty lokalne tworzone są na stosie, ponieważ alokacja pamięci na stosie jest znacznie szybsza niż alokacja pamięci na stercie.

Semantyka przenoszenia obiektów

- Cecha języka C++11, która pozwala na *uniknięcie kopiowania obiektów* poprzez przenoszenie obiektów.
- Kopiowanie nie jest potrzebne i może być uniknione, jeżeli obiekt źródłowy nie będzie potem potrzebny.
- Przenoszenie obiektów jest realizowane przez konstruktor przenoszący lub przenoszący operator przypisania.
- Obiekt może być przeniesiony tylko wtedy, jeżeli:
  - jest użyty w wyrażeniu kategorii r-wartość,
  - ma zaimplementowaną semantykę przeniesienia.
- Implementacja przenoszenia obiektów danej klasy może być domyślna (dostarczona przez kompilator) albo dostarczona przez programistę.

Jak to działa?

- Nic magicznego się nie dzieje! Obiekty nie są przenoszone bit-po-bicie między różnymi miejscami pamięci!
- Przenoszenie obiektu polega na przenoszeniu danych z obiektu źródłowego do obiektu docelowego. Obiekt źródłowy i obiekt docelowy pozostają w pamięci tam, gdzie były i będą w normalny sposób niszczone.
- Programista może zaimplementować przenoszenie obiektów danej klasy przez zdefiniowanie:
  - konstruktora przenoszącego,
  - przenoszącego operatora przypisania.
- Obiekt, który jest źródłem przeniesienia, po przeniesieniu ma być spójny (bo obiekt będzie potem normalnie niszczony), ale jego stan nie musi być zdefiniowany.

Konstruktor: kopiujący i przenoszący

```
struct T
{
  // Konstruktor kopiujący.
  T(const T &t)
  {
    // Skopiuj dane z obiektu t do *this.
  }

  // Konstruktor przenoszący.
  T(T &&t)
  {
    // Przenieś dane z obiektu t do *this.
  }
};
```

Operator przypisania: kopiujący i przenoszący

```
struct T
{
  // Kopiujący operator przypisania.
  T &operator = (const T &t)
  {
    // Skopiuj dane z obiektu t do *this.
    return *this;
  }
  // Przenoszący operator przypisania.
  T &operator = (T &&t)
  {
    // Przenieś dane z obiektu t do *this.
    return *this;
  }
};
```

Wybór przeciążenia

Wybór przeciążenia (kopiującego lub przenoszącego) konstruktora czy operatora przypisania zależy od kategorii wartości wyrażenia, które jest argumentem wywołania i także od dostępności przeciążenia.

Jeżeli dostępne są oba przeciążenia, kompilator wybierze przeciążenie kopiujące dla l-wartości i przeciążenie przenoszące dla r-wartości.

Jeżeli dostępne jest tylko przeciążenie kopiujące, kompilator wybierze przeciążenie kopiujące także dla r-wartości, bo stała l-referencja może wskazać r-wartość.

Jeżeli dostępne jest tylko przeciążenie przenoszące, kompilator zgłosi błąd dla l-wartości, bo r-referencja nie może wskazać l-wartości.

Domyślne implementacje składowych

Kompilator dołączy domyślne implementacje:

- konstruktora: kopiującego i przenoszącego,
- operatora przypisania: kopiującego i przenoszącego,
- destruktora,

jeżeli programista nie dostarczył żadnej z nich. W przeciwnym razie kompilator ich nie dołączy, ale możemy jakąś jawnie dołączyć:

```
struct A
{
  ~A() {}
  A(const A &) = default;
};
```

Implementacja konstruktora przenoszącego

W liście inicjalizacji argumentami konstruktorów klasy bazowej i pól składowych powinny być r-wartości, żeby zostały wybrane konstruktory przenoszące obiektów, dlatego używamy funkcji `std::move`.

```
#include <string>
#include <utility>

struct A {};

struct B: A
{
  std::string m_s;

  B(B &&b): A(std::move(b)), m_s(std::move(b.m_s))
  {
  }
};
```

Implementacja przenoszącego operatora przypisania

Żeby kompilator wybrał przenoszące (a nie kopiujące) operatory przypisania (jeżeli istnieją) dla obiektu bazowego i obiektów składowych, używamy funkcji `std::move`.

```
#include <string>
#include <utility>

struct A {};

struct B: A
{
  std::string m_s;

  B & operator=(B &&b)
  {
    A::operator=(std::move(b));
    m_s = std::move(b.m_s);
    return *this;
  }
};
```

Usuwanie składowych

Możemy usunąć składowe przez zadeklarowanie ich jako `delete`:

```
struct A
{
  // Konstruktor kopiujący.
  A(const A &) = delete;

  // Kopiujący operator przypisania.
  void
  operator =(const A &) = delete;
};
```

W ten sposób usunęliśmy konstruktor kopiujący i kopiujący operator przypisania, jak robi się to w typach danych tylko do przenoszenia (ang. move-only type), którego przykładem jest `std::unique_ptr`.

Inicjalizacja parametrów funkcji

Parametr funkcji inicjalizowany jest na podstawie wyrażenia, które jest argumentem wywołania funkcji. Dla parametru typu niereferencyjnego będzie wywołany:

- konstruktor kopiujący, jeżeli argumentem jest l-wartość,
- konstruktor przenoszący, jeżeli argumentem jest r-wartość.

Jeżeli konstruktor kopiujący nie jest dostępny, a argumentem jest l-wartość, to będzie zgłoszony błąd kompilacji.
Jeżeli konstruktor przenoszący nie jest dostępny, a argumentem jest r-wartość, to będzie użyty konstruktor kopiujący.

Zwracanie wyniku funkcji przez wartość

Jeżeli wartość zwracana przez funkcję nie jest typu referencyjnego, to mówimy, że funkcja zwraca wynik przez wartość. Na przykład:

```
struct T {};

T foo() // Funkcja zwraca wynik przez wartość.
{
  T t;
  return t; // To samo co: return std::move(t);
}

int main()
{
  T t = foo(); // Wynik jest przenoszony, a nie kopiowany
}
```

Dla zwracania przez wartość, działanie instrukcji `return t;` zdefiniowano jako `return std::move(t);`, czyli będzie zwracana r-wartość, którą będzie można przenieść, żeby uniknąć kopiowania.

Funkcja `std::swap`

Funkcja `std::swap` była jednym z powodów, dla którego zaczęto pracować nad semantyką przeniesienia w języku C++. Ta funkcja pokazała, że wydajniej jest przenosić obiekty niż je kopiować.

Funkcja `std::swap(x, y)` przyjmuje przez referencję dwa obiekty `x` i `y`, których wartości zamienia. Przykładowa implementacja:

```
#include <utility>

template <typename T>
void swap(T &a, T &b)
{
  T tmp = std::move(a);
  a = std::move(b);
  b = std::move(tmp);
}
```

Podsumowanie

- Przenoszenie obiektów wprowadzono w C++11.
- Przenoszenie obiektów pozwala na uniknięcie kopiowania.
- Tylko obiekty r-wartości mogą być przenoszone.
- Kompilator może dołączyć domyślne przeciążenia przenoszące.
- Wybór przeciążenia (kopiującego lub przenoszącego) konstruktora czy operatora przypisania zależy od kategorii wartości wyrażenia, które jest argumentem wywołania i także od dostępności przeciążenia.

Dziękuję za uwagę.