Inteligentny wskaźnik `std::unique_ptr`

dr inż. Ireneusz Szcześniak

jesień 2017 roku

Wskaźniki są niezbędne!

- Wskaźnik wskazuje miejsce w pamięci.
- W każdym wydajnym języku programowania jest stosowany: C, C++, Java.
- Chociaż niektóre języki, jak Java, ukrywają to w postaci referencji.
- Obsługa wskaźników może być różna:
  - ukryta przed użytkownikiem w Javie,
  - mieszana w C++,
  - surowa w C.
- W miarę możliwości należy unikać surowych wskaźników.

Motywacja: wady surowych wskaźników

- Nie wiadomo, czy wskazuje jeden obiekt czy tablice obiektów.
- Czy obiekt niszczymy przez `delete` czy `delete[]`?
- Nie mówi, kiedy wskazany obiekt powinien być zniszczony.
- Łatwo popełnić błędy używając surowych wskaźników:
  - nie niszcząc obiektu (wyciek pamięci),
  - niszcząc obiekt i potem próbować się do niego odwołać (dyndający wskaźnik),
  - niepoprawna obsługa wyjątków.
- Rozwiązaniem tych problemów są *inteligentne wskaźniki*.
- Czasy `void *` dawno minęły.

Rodzaje inteligentnych wskaźników

- `std::auto_ptr` - przestażały, lepiej unikać
- `std::unique_ptr` - stosować zamiast surowego wskaźnika
- `std::shared_ptr` - wskazuje i współdzieli obiekt
- `std::weak_ptr` - wskazuje obiekt współdzielony, ale nie współdzieli tego obiektu.
- Te klasy to opakowania (używane w czasie kompilacji) surowych wskaźników.
- Zajmują tyle samo miejsca w pamięci, co surowy wskaźnik.
- Tak wydajny pamięciowo i czasowo, jak surowy wskaźnik.
- Są odporne na wyjątki, ale nie na wątki!
- Stosujemy inteligentne wskaźniki zamiast surowych wskaźników.

Ogólnie o `std::unique_ptr`

- W większości przypadków: używać zamiast surowych wskaźników.
- Jest wyłącznym właścicielem wskazywanego obiektu.
- `#include <memory>`
- `std::unique_ptr <moja_klasa> p;`

`std::unique_ptr` jest właścicielem

- Posiada semantykę (znaczenie) wyłącznej własności.
- Posiada konstruktor przenoszący, ale nie kopiujący.
- Jest wyłącznym właścicielem wskazywanego obiektu, czyli:
  - niszczy obiekt, kiedy sam jest niszczony (np. kiedy wychodzi poza zakres),
  - niszczy obiekt, zanim będzie wskazywał następny obiekt,
  - nie da się kopiować, a jedynie przenosić (czyli nie ma dwóch `std::unique_ptr` wskazujących ten sam obiekt),
  - kiedy odda wskazywany obiekt, to sam wskazuje `nullptr`.

Przykład użycia std::unique_ptr

Funkcja zwracająca obiekt użytkownikowi:

```
auto // C++14
foo()
{
  std::unique_ptr <A> a(new A());

  // Jesteśmy bezpieczni na wypadek wyjątku.
  throw true; // To nie poezja programistyczna!

  return a;
}
```

Użycie std::unique_ptr

```
std::unique_ptr<A> p1; // OK.  Nie wskazuje na nic.
std::unique_ptr<A> p1(new A("A1")); // OK.
p1 = new A("A1'"); // Zabronione!
p1.reset(new A("A2")); // OK, ale usunie "A1".
std::unique_ptr<A> p2(p1); // Błąd! Nie można kopiować.
std::unique_ptr<A> p2 = p1; // Błąd! Nie można kopiować.
p2 = p1; // Błąd! Nie można kopiować.
p2 = std::move(p1); // OK.  Przeniesienie.
p2->foo(); // Wywyłanie A::foo(), jak przez wskaźnik.
(*p2).foo(); // Wywyłanie A::foo(), jak przez wskaźnik.
p1.release(); // p1 już nie ma obiektu, ale go nie usuwa.
p1.get(); // Wartość surowego wskaźnika.

std::deque<std::unique_ptr<A> > c;
c.push_back(std::unique_ptr<A>(new A("C1")));
```

std::make_unique

- Zdefiniowany dopiero w C++14, a nie w C++11 jak `unique_ptr`.
- Zamiast pisać tak i użyć typu A dwa razy:
  `unique_ptr<A> up(A("A1"));`
- Możemy napisać tak i użyć typu A tylko raz:
  `auto up = make_unique<A>("A1");`
- ZALETA: tworzenie obiektu zarządzanego i zarządzającego odbywa się w jedyn kroku, co jest bezpieczne pod względem obsługi wyjątków.
- FACHOWO: enkapsulacja i lokalizacja alokacji zasobów (encapsulation and localization of resource allocation).

Użycie unique_ptr

- Zamiast implementować semantykę przeniesienia, możemy posługiwać się wskaźnikiem `unique_ptr` na obiekt, którego kopiowania chcemy uniknąć.
- Wskaźnik `unique_ptr` implementuje semantykę przeniesienia.
- Przeniesienie `unique_ptr` jedynie będzie oznaczać skopiowanie surowego wskaźnika, a obiekt docelowy pozostanie nietknięty.
- Najprostsze rozwiązanie: zostaw klasę A w spokoju, używaj `unique_ptr<A>`.
- Jednak właściwym rozwiązaniem jest ciągle implementacja semantyki przeniesienia dla danej klasy.

Podsumowanie

- Unikać surowych wskaźników!
- Używać inteligentnych wskaźników!
- Najczęściej potrzebujemy `std::unique_ptr`
- `std::unique_ptr` trzeba się nauczyć używać.
- Na następnym wykładzie: `std::shared_ptr`
- Konwersja z `std::unique_ptr` do `std::shared_ptr`:
  - `std::shared_ptr<A> s = foo();`
  - `std::shared_ptr<A> s = std::move(p1);`

Dziękuję za uwagę.