Sprytne wskaźniki

Automatyczne niszczenie obiektów utworzonych na stercie w C++

Programista używając C++ musi dbać o zwalnianie obiektów dynamicznych (utworzonych na stercie). Zadanie to można automatyzować, wykorzystując obiekty pośredniczące, tak zwane sprytne wskaźniki. Narzuty czasowe i pamięciowe tego rozwiązania są pomijalne w większości zastosowań.

Dowiesz się:
- Jak zarządzać obiektami utworzonymi na stercie;
- Co to są sprytne wskaźniki;
- Jak automatycznie zwalniać obiekty, gdy występują cykliczne zależności.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++;
- Co to są obiekty dynamiczne (utworzone na stercie).

Poziom trudności

Obiekty utworzone na stercie (za pomocą operatora `new`) powinny być zwalniane przez programistę wtedy, kiedy nie są już używane (operator `delete`). Brak restrykcyjnej kontroli dotyczącej czasu życia tych obiektów powoduje wycieki pamięci, co oznacza, że aplikacja zajmuje coraz więcej pamięci operacyjnej, kończąc działanie błędem, kiedy pamięć ta zostanie wyczerpana. Poniższy tekst pokazuje wykorzystanie mechanizmów związanych z wołaniem konstruktorów i destruktorów do automatycznego zwalniania obiektów tworzonych na stercie, co zwalnia programistę z tego zadania.

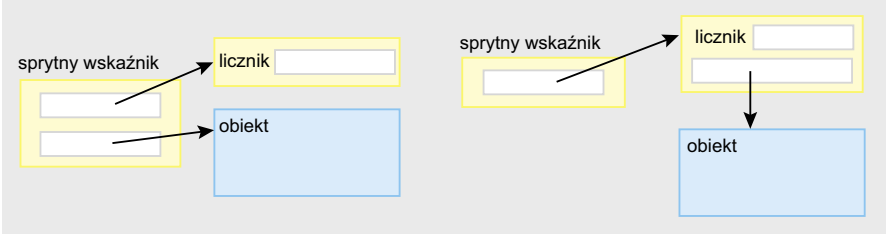
Dostęp do obiektów utworzonych na stercie jest realizowany przez wskaźnik, jeżeli taki obiekt nie jest wskazywany, to nie może być używany, więc można go zwolnić. Sprytne wskaźniki pośredniczą przy dostępie do obiektów dynamicznych (wskaźnik jest przekazywany w konstruktorze) badając, czy obiekt jest wskazywany i jeżeli nie, usuwając go. Zakładając, że obiekt dynamiczny może być wskazywany tylko przez jeden sprytny wskaźnik, to obiekt ten możemy usunąć w momencie niszczenia sprytnego wskaźnika. W tym najprostszym przypadku sprytny wskaźnik usuwa zarządzany obiekt w destruktorze, technika ta jest nazywana *zdobywanie zasobów jest inicjowaniem*. Jeżeli obiekt jest wskazywany przez wiele sprytnych wskaźników, to powinien być usunięty w destruktorze ostatniego z nich.

Sprytne wskaźniki będące wyłącznymi posiadaczami obiektów

Biblioteka standardowa C++ oraz biblioteki boost dostarczają różnych rodzajów sprytnych wskaźników. Są to szablony, ponieważ potrzebujemy wskaźników zarządzających różnymi typami obiektów. Szablony generują kod w czasie kompilacji, tworząc odpowiednie klasy, nie dodając żadnych narzutów pamięciowych i czasowych (kod generowany nie różni się od kodu tworzonego *ręcznie*), jedyną niedogodnością jest dłuższy czas kompilacji.

Sprytnym wskaźnikiem jest klasa `boost::scoped_ptr` przedstawiona na Listingu 1. Konstruktor kopiujący oraz operator przypisania jest zabroniony (prywatny), ponieważ utworzenie więcej niż jednego wskaźnika tego typu przechowującego ten sam obiekt prowadzi do błędu, polegającego na próbie powtórnego zwolnienia obiektu dynamicznego.

Sprytny wskaźnik, oprócz usuwana obiektów po wyjściu z bloku, ułatwia zarządzanie obiektami dynamicznymi, które są składowymi klasy, ponieważ nie trzeba ich jawnie zwalniać w destruktorze (patrz Listing 2). Dodatkową zaletą takiego rozwiązania jest poprawne zwalnianie zasobów, gdy są zgłaszane wyjątki. Jeżeli konstruktor klasy *Audio* (Listing 2) zgłosi wyjątek, to sprytny wskaźnik *image_* sprawi, że obiekt *Image* zostanie skasowany.



Rysunek 1. *Sprytne wskaźniki ze zliczaniem odniesień współdzielą wskaźnik do obiektu oraz licznik. Obiekt sprytnego wskaźnika może zawierać jeden lub dwa wskaźniki*

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do dowolnego kompilatora C++ oraz edytora tekstu. Większość przykładów korzysta z udogodnień dostarczanych przez biblioteki boost, warunkiem ich uruchomienia jest instalacja tych bibliotek (w wersji 1.36 lub nowszej) oraz wykorzystywanie kompilatora oficjalnie przez nie wspieranego, którymi są: msvc 7.1 lub nowszy, gcc g++ 3.4 lub nowszy, Intell C++ 8.1 lub nowszy, Sun Studio 12 lub Darvin/GNU C++ 4.x. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

Biblioteka standardowa C++ zawiera szablon `std::auto_ptr`, który jest innym typem sprytnego wskaźnika będącego wyłącznym posiadaczem obiektu. Ma on nietypową implementację konstruktora kopiującego i operatora przypisania, operacje te są przekazywaniem własności, a nie tworzeniem kopii (patrz Listing 3).

Wskaźnik ten możemy wykorzystać (oprócz automatycznego kasowania obiektu przy usuwaniu wskaźnika) do zwracania obiektu utworzonego na stercie, unikając wycieków pamięci. Listing 4 zawiera funkcję `createFoo`, która ilustruje tę technikę. Jeżeli będziemy ignorowali wartość zwracaną, to destruktor obiektu tymczasowego zwolni obiekt, jeżeli wartość zwracana zostanie przypisana do innego obiektu, to obiekt tymczasowy będzie pusty.

Nietypowa implementacja konstruktora kopiującego i operatora przypisania (przekazywanie własności, a nie tworzenie kopii) zapobiega tworzeniu więcej niż jednego wskaźnika `auto_ptr` zarządzającego tym samym obiektem. Brak możliwości tworzenia kopii ogranicza możliwość stosowania tych wskaźników, na przykład obiekty `auto_ptr` nie powinny być przechowywane w kolekcjach standardowych, ponieważ one zawsze przechowują kopię.

Sprytne wskaźniki ze zliczaniem odniesień

Przedstawione poprzednio sprytne wskaźniki nie pozwalają na współdzielenie tego samego obiektu przez kilka wskaźników. Wady tej nie mają sprytne wskaźniki ze zliczaniem odniesień pokazane na Rysunku 1. Obiekty te zajmują więcej pamięci niż zwykłe wskaźniki, ponieważ wymagają one licznika odniesień. Sprytne wskaźniki, przejmując nowy obiekt utworzony na stercie, tworzą licznik i inicjują go wartością 1, w konstruktorze kopiującym licznik ten jest zwiększany, w destruktorze zmniejszany, jeżeli osiągnie wartość 0, to licznik oraz zarządzany obiekt jest usuwany.

Wskaźnikiem ze zliczaniem odniesień jest szablon `boost::shared_ptr` (wchodzi on w skład nowego standardu C++200x). Takie sprytne wskaźniki pozwalają wygodnie manipulować obiektami dynamicznymi, możemy przechowywać je w kontenerach, przekazywać jako argument oraz zwracać jako wartość. Przykład użycia pokazano na Listingu 5.

Jeżeli występują cykliczne zależności pomiędzy obiektami, czyli obiekt A wskazuje na obiekt B, zaś obiekt B na obiekt A, to sprytne wskaźniki `shared_ptr` nie zwolnią obiektów, pomimo tego, że obiekty nie będą już używane. Ilustracją tego zjawiska jest obiekt ze składową będącą sprytnym wskaźnikiem na siebie (składowa używana zamiast `this`). Obiekt ten nie zostanie zwolniony, jeżeli składowa (spryt-

Listing 1. *Wybrane metody szablonu scoped_ptr*

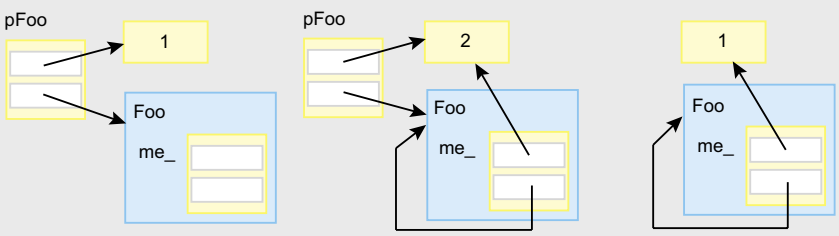
```
template<typename T> class scoped_ptr {
public:
   explicit scoped_ptr(T* p = 0) : p_(p) {}
   ~scoped_ptr(){ delete p_; } //usuwa wskazywany obiekt
   T& operator*() { return *p_; } //dostęp do zarządzanego obiektu
   T* operator->() { return p_; } //dostęp do zarządzanego obiektu
   //także inne metody, np. porównywanie wskaźników
private:
   T* p_; //Wskaźnik, którym zarządza
   scoped_ptr(scoped_ptr const &); //zabroniony konstruktor kopiujący
   scoped_ptr & operator=(scoped_ptr const &); //zabronione przypisanie
};
```

Listing 2. *Składowe przechowywane jako sprytne wskaźniki upraszczają kod*

```
class Book { //przykładowa klasa książki, która agreguje obrazek oraz nagranie
public:
   Book(const string& name, const string& image_name, const string& audio_name)
      : name_(name),
      image_(new Image(image_name) ),
      audio_(new Audio(audio_name) )
   {}
   ~Book() {} //destruktor może być pusty
private:
   string name_;
   scoped_ptr<Image> image_;
   scoped_ptr<Audio> audio_;
};
```

Listing 3. *Fragmenty szablonu std::auto_ptr*

```
template<typename T> class auto_ptr {
public:
   explicit auto_ptr(T* p = 0) : p_(p) {}
   //argument jest zwykłą (a nie stałą) referencją, ponieważ jest zmieniany
   //jest to zachowanie nietypowe i należy zwracać na to uwagę
   auto_ptr(auto_ptr& a) p_(a.p_) { a.p_ = 0L; }
   //argument nie jest const auto_ptr&, patrz konstruktor kopiujący
   auto_ptr& operator=(auto_ptr& a) {
      if(p_ ! = a.p_) { delete p_; p_ = a.p_; a.p_ = 0L; }
      return *this;
   }
   ~auto_ptr() { delete p_; } //usuwa wskazywany obiekt
   T& operator*() { return *p_; } //dostęp do zarządzanego obiektu
   T* operator->() const { return p_; } //dostęp do zarządzanego obiektu
private:
   T* p_;
};
```



Rysunek 2. *Cykliczna zależność pomiędzy obiektami. Sprytny wskaźnik nie zwalnia obiektu*

Listing 4. *Wykorzystanie std::auto_ptr do zwracania wskaźników do obiektów*

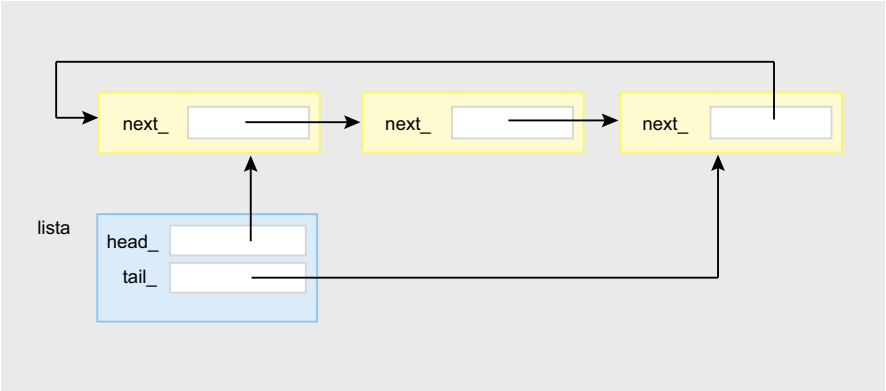
```
auto_ptr<Foo> createFoo() { //funkcja zwraca wskaźnik na obiekt
   return auto_ptr<Foo>(new Foo(n) );
}
createFoo(); //wskaźnik usuwany wtedy gdy niszczony obiekt tymczasowy
auto_ptr<Foo> v = createFoo(); //konstruktor kopiujący dla obiektu v
auto_ptr<Foo> w = v; //teraz v.p_ jest równe nullptr
//wskaźnik usuwany gdy obiekt w wyjdzie z zasięgu
```

Listing 5. *Zarządzanie obiektami dynamicznymi przez wskaźniki shared_ptr*

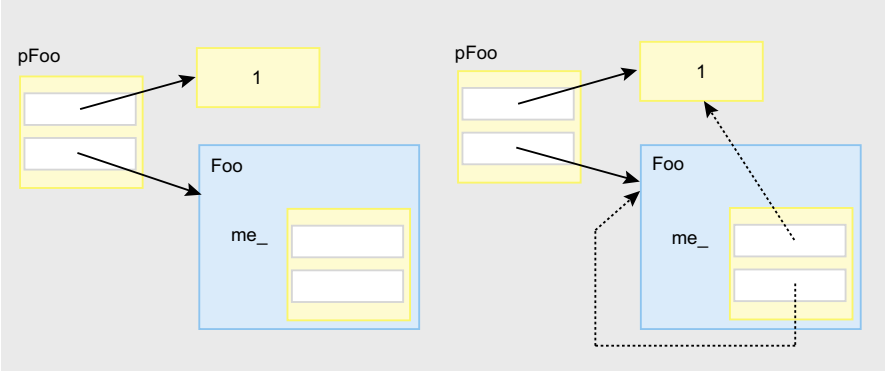
```
class Base { //klasa bazowa
   virtual ~Base(){}
};
class Derived1 : public Base { };
class Derived2 : public Base { };
//wskaźnik na klasę bazową
typedef shared_ptr<Base> PBase;
vector<PBase> v;
v.push_back( PBase(new Derived1) );
v.push_back( PBase(new Derived2) );
//destruktor wektora v usunie obiekty utworzone na stercie
```

Listing 6. *Zależność cykliczna, obiekt ma składową, która na niego wskazuje*

```
struct Foo {
   shared_ptr<Foo> me_;
};
shared_ptr<Foo> pFoo = new Foo; //licznik równy 1
pFoo->me_ = pFoo; //inicjacja składowej, licznik równy 2
//jeżeli teraz pFoo zostanie zniszczone, to licznik jest równy 1
//obiekt nie jest usuwany i nie ma do niego dostępu
```



Rysunek 3. *Lista cykliczna. Destruktor listy musi przerwać cykliczną zależność pomiędzy obiektami, ponieważ inaczej elementy listy nie będą usunięte przy niszczeniu listy*



Rysunek 4. *Eliminacja cyklicznej zależności poprzez wprowadzenie słabych sprytnych wskaźników*

ny wskaźnik) zostanie poprawnie zainicjowana, ponieważ będzie ‚*podtrzymywany*' przez tę składową (patrz Rysunek 2).

Rozwiązaniem w takim przypadku jest zwolnienie wskaźnika z obowiązku zarządzania obiektem, czyli wywołanie metody `reset`. Wystarczy przerwać zależność w jednym miejscu, patrz przykład listy cyklicznej przedstawionej na Listingu 7 oraz Rysunku 3.

Automatyczne zwalnianie obiektów, gdy występują zależności cykliczne, jest możliwe, jeżeli wykorzystamy odmianę wskaźników zwaną słabymi sprytnymi wskaźnikami. Takie obiekty przechowują odniesienie do obiektu i do licznika, ale nie modyfikują licznika odniesień, więc fakt, że słaby sprytny wskaźnik wskazuje na obiekt, nie wpływa na czas jego życia. Kod słabego sprytnego wskaźnika `boost::weak_ptr` został podany na Listingu 8.

Słaby sprytny wskaźnik wprowadzony w dowolne miejsce zależności cyklicznej sprawia, że obiekty będą zwalniane prawidłowo.

Dla obiektu zawierającego składową wskazującą na dany obiekt, składowa ta powinna być słabym wskaźnikiem (patrz Rysunek 4 oraz Listing 9).

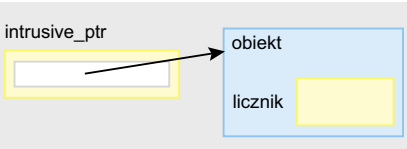
Słaby wskaźnik nie podtrzymuje zarządzanego obiektu, więc może się okazać, że obiekt wskazywany przez taki wskaźnik zostanie usunięty przez destruktor silnego sprytnego wskaźnika (`shared_ptr`). Aby uniemożliwić wystąpienie błędu polegającego na próbie odwołania do zwolnionego obiektu dynamicznego, słabe sprytne wskaźniki są powiadamiane o fakcie usunięcia zarządzanego obiektu, wtedy przechowywany wskaźnik jest zerowany i metoda `expired` będzie zwracać wartość `true`. Z tego powodu nie można inicjować składowych, które są słabymi sprytnymi wskaźnikami wskazującymi na obiekt w konstruktorze.

Sprytne wskaźniki narzucające interfejs

Umieszczenie licznika odniesień w przechowywanym obiekcie zapobiega próbom tworzenia niezależnych sprytnych wskaź-

W Sieci

- *http://www.boost.org;*
- *http://www.open-std.org.*



Rysunek 5. *Sprytne wskaźniki z licznikiem przechowywanym w obiekcie*

Listing 7. *Lista cykliczna używająca sprytnych wskaźników*

```
class List { //lista cykliczna, patrz rysunek 3
   struct Node { //węzeł dla listy
      typedef shared_ptr<Node> PNode;
      Node(PNode next) : next_(next) { }
      ~Node() { }
      PNode next_; //wskaźnik na element następny
   };
public:
   List() {}
   ~List() { //jawnie przerywa zależność cykliczną
      if( tail_ ) tail_->next_.reset();
   } //destruktory obiektów head_ oraz tail_ skasują elementy listy
private:
   Node::PNode head_, tail_;
};
```

Listing 8. *Słaby sprytny wskaźnik*

```
template<class T> class weak_ptr {
public:
   //konstruktor na podstawie silnego sprytnego wskaźnika
   template<class Y> weak_ptr(shared_ptr<Y> const & r);
   template<class Y> weak_ptr(weak_ptr<Y> const & r);
   ~weak_ptr();
   bool expired() const; //informacja o zwolnieniu zarządzanego obiektu
   shared_ptr<T> lock() const; //tworzy silny wskaźnik do danego obiektu
   //pozostałe metody
};
```

Listing 9. *Zapobieganie zależnościom cyklicznym na przykładzie składowej wskazującej na obiekt*

```
struct Foo {
   weak_ptr<Foo> me_;
};
shared_ptr<Foo> pFoo = new Foo; //licznik równy 1
pFoo->me_ = pFoo; //licznik równy 1, bo słaby wskaźnik
//gdy pFoo zostanie zniszczone, to obiekt jest usuwany
```

Listing 10. *Klasa dostarczająca interfejsu używanego przez intrusive_ptr*

```
//klasa której obiekty będą zarządzane przez sprytne wskaźniki współdzielone przez
                     różne wątki
class Foo {
   friend void intrusive_ptr_add_ref(Foo* ptr);
   friend void intrusive_ptr_release(Foo* ptr);
   boost::mutex m_; //obiekt synchronizujący
   int counter_; //licznik dla intrusive_ptr
   // pozostałe metody i składowe
};
void intrusive_ptr_add_ref(Foo* foo) {
   mutex::scoped_lock lock(foo->m_); //sekcja krytyczna
   ++(foo->counter_);
}
void intrusive_ptr_release(Foo* foo) {
   bool del = false;
   {
      mutex::scoped_lock lock(foo->m_); //sekcja krytyczna
      del = ! --(foo->counter_);
   } //flaga del pozwala kasować obiekt poza sekcją krytyczną
   if(del) delete foo;
}
```

ników zarządzających tym obiektem. Dodatkową zaletą takiego rozwiązania jest lepsze wykorzystanie pamięci, ponieważ po pierwsze, nie jest wymagany dodatkowy obiekt na stercie (licznik), a po drugie, sam sprytny wskaźnik ma wielkość taką jak zwykły wskaźnik. Wadą rozwiązania jest jego mniejsza elastyczność, sprytne wskaźniki można utworzyć tylko dla klas, które przechowują licznik (patrz Rysunek 5). Tego rodzaju sprytnym wskaźnikiem jest `boost::intrusive_ptr`.

Klasy, dla których będą tworzone wskaźniki za pomocą szablonu *intrusive_ptr* , muszą dostarczać funkcji `intrusive_ptr_add_ref(T*)` i `intrusive_ptr_release(T*)`, gdzie `T` jest nazwą klasy. Sprytne wskaźniki wykorzystujące licznik znajdujący się w obiekcie pozwalają na wygodną implementację wskaźników, które mogą być współdzielone przez różne wątki, ponieważ obiekt może dostarczać mechanizmów synchronizujących patrz (Listing 11).

Podsumowanie

Sprytne wskaźniki wykorzystuje się w C++ do zarządzania czasem życia obiektów utworzonych na stercie. Język ten nie dostarcza innego standardowego mechanizmu tego typu. Wskaźniki takie są wygodne i pozwalają na uproszczenie kodu, zwłaszcza wtedy, gdy dopuszczamy możliwość wystąpienia wyjątku, czyli przerwania sekwencyjnego ciągu instrukcji. Standard oraz biblioteki boost dostarczają tego rodzaju udogodnienia. Sprytne wskaźniki zajmują tyle samo pamięci co zwykłe wskaźniki (`scoped_ptr`, `auto_ptr` i `intrusive_ptr`), lub niewiele więcej (`shared_ptr` i `weak_ptr`), narzuty czasowe są minimalne, jedno pośrednie odwołanie dla sprytnych wskaźników będących wyłącznymi posiadaczami obiektów oraz badanie i modyfikacja licznika dla sprytnych wskaźników z licznikiem odniesień.

Sprytne wskaźniki możemy wykorzystywać w aplikacjach współbieżnych. Jeżeli ten sam obiekt dynamiczny będzie wskazywany przez sprytne wskaźniki znajdujące się w różnych wątkach, to należy użyć współbieżnej wersji wskaźników, która zapewnia synchronizację pomiędzy operacjami na liczniku i operacją zwalniania obiektu.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji dla biologii i medycyny, programuje w C++ od ponad 10 lat.

Kontakt z autorem:rno@o2.pl