Więcej niż C++. Wprowadzenie do bibliotek Boost

Autor: Björn Karlsson
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-246-0339-5
Tytuł oryginału: Beyond the C++ Standard Library: An Introduction to Boost
Format: B5, stron: 384

Język C++ znajduje coraz więcej zastosowań, w wypadku których biblioteka standardowa często okazuje się zbyt uboga. Projekt Boost powstał w celu wypełnienia luk i wyeliminowania niedoskonałości biblioteki STL. Dziś biblioteki Boost zyskują coraz większą popularność, czego dowodem jest włączenie dziesięciu z nich do przygotowywanej biblioteki standardowej języka C++0x. Twórcy kolejnej specyfikacji C++ zdecydowali się nawet na kilka modyfikacji języka w celu ułatwienia korzystania z bibliotek Boost.

Książka „Więcej niż C++. Wprowadzenie do bibliotek Boost" to przegląd 58 bibliotek projektu. Dwanaście z nich omówiono szczegółowo i zilustrowano przykładami. Analizując zaprezentowane projekty, przekonasz się, jak bardzo biblioteki Boost ułatwiają pracę i pozwalają ulepszyć aplikacje. Nauczysz się korzystać z inteligentnych wskaźników, obiektów funkcyjnych, wyrażeń regularnych i wielu innych funkcji oferowanych przez biblioteki Boost.

- Bezpieczna konwersja typów
- Stosowanie elastycznych bibliotek kontenerów
- Wyrażenia regularne
- Wywołania zwrotne
- Zarządzanie sygnałami i slotami

Wykorzystaj już teraz elementy bibliotek Boost, a nowa biblioteka standardowa nie będzie miała przed Tobą żadnych tajemnic.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 1.

Biblioteka Smart_ptr

Jak ulepszyć własne programy z użyciem biblioteki Smart_ptr?

- Poprzez automatyczne zarządzanie czasem życia obiektów za pomocą szablonu `shared_ptr`, bezpiecznie i efektywnie zarządzającego wspólnymi zasobami.
- Poprzez bezpieczne podglądanie zasobów wspólnych za pomocą szablonu `weak_ptr`, co eliminuje ryzyko charakterystyczne dla wiszących wskaźników.
- Poprzez osadzanie zasobów w zasięgach programu za pomocą szablonów `scoped_ptr` i `scoped_array`, ułatwiających konserwację kodu i pomocnych przy zabezpieczaniu przed zgubnym wpływem wyjątków.

Wskaźniki inteligentne, implementowane w bibliotece Smart_ptr, rozwiązują odwieczny problem zarządzania czasem życia zasobów (chodzi zwykle o zasoby przydzielane dynamicznie[1]). Inteligentne wskaźniki dostępne są w wielu odmianach. Wszystkie jednak mają jedną cechę wspólną: automatyzację zarządzania zasobami. Ów automatyzm manifestuje się rozmaicie, na przykład poprzez kontrolę czasu życia obiektów przydzielanych dynamicznie czy też poprzez kontrolę nad akwizycją i zwalnianiem zasobów (plików, połączeń sieciowych itp.). Wskaźniki inteligentne z biblioteki Boost implementują pierwsze z tych zastosowań, to jest przechowują wskaźniki do dynamicznie przydzielanych obiektów, dbając o ich zwalnianie w odpowiednich momentach. Można się zastanawiać, czy to nie zbytnie ograniczenie ich zadań. Czy nie można by zaimplementować również pozostałych aspektów zarządzania zasobami? Cóż, można by, ale nie za darmo. Rozwiązania ogólne wymagają często większej złożoności, a przy inteligentnych wskaźnikach biblioteki Boost główny nacisk położono nawet nie tyle na elastyczność, co na wydajność. Ale dzięki możliwości implementowania

[1] Czyli wszelkie zasoby, do których można się odwoływać za pośrednictwem typu wskaźnikowego — również inteligentnego — *przyp. aut.*

własnych mechanizmów zwalniania najbardziej inteligentne ze wskaźników z rodziny Boost (`boost::shared_ptr`) mogą obsługiwać zasoby wymagające przy zwalnianiu bardziej wyrafinowanych operacji niż proste wywołanie `delete`. Pięć implementacji wskaźników inteligentnych w bibliotece Boost.Smart_ptr odzwierciedla zaś szereg kategorii potrzeb pojawiających się w programowaniu.

Po co nam inteligentne wskaźniki?

Wskaźniki inteligentne stosuje się przy:

- manipulowaniu zasobami pozostającymi w posiadaniu wielu obiektów,
- pisaniu kodu odpornego na wyjątki,
- unikaniu typowych błędów w postaci wycieków zasobów.

Współdzielenie własności zachodzi, kiedy dany obiekt jest użytkowany przez pewną liczbę innych obiektów. Jak (albo raczej: kiedy) należy zwolnić ów używany obiekt? Aby rozpoznać odpowiedni moment zwolnienia współużytkowanego obiektu, należałoby wyposażyć każdy z obiektów użytkujących w informacje o współwłaścicielach. Tego rodzaju wiązanie obiektów nie jest pożądane z punktu widzenia poprawności projektowej, a także z uwagi na łatwość konserwacji kodu. Lepiej byłoby, aby obiekty-współwłaściciele złożyły odpowiedzialność za zarządzanie czasem życia współużytkowanego obiektu na inteligentny wskaźnik. Ten, po wykryciu, że nie ma już żadnego właściciela, może bezpiecznie zwolnić obiekt użytkowany.

Odporność na wyjątki to w najprostszym ujęciu zabezpieczenie przed wyciekami zasobów, jak również zabezpieczenie trwałości niezmienników programu w obliczu wyjątków. Obiekt przydzielony dynamicznie może w obliczu wyjątku nie zostać zwolniony. W ramach procedury zwijania stosu przy zrzucaniu wyjątku i porzucaniu bieżącego zasięgu dojdzie do utracenia wskaźników obiektów dynamicznych, co uniemożliwi zwolnienie obiektu aż do momentu zakończenia programu (a i w fazie końcowej programu język nie daje gwarancji zwolnienia zasobów). Program niezabezpieczony przed takim wpływem wyjątków może nie tylko doprowadzić do deficytu pamięci operacyjnej, ale i znaleźć się w niestabilnym stanie; zastosowanie wskaźników inteligentnych automatyzuje zwalnianie zasobów nawet w obliczu wyjątków.

Co do *unikania typowych błędów*, to najbardziej typowym jest chyba pominięcie (wynikające z przeoczenia) wywołania `delete`. Tymczasem typowy inteligentny wskaźnik nie śledzi bynajmniej ścieżek przebiegu wykonania programu — jego jedyną troską jest zwolnienie wskazywanego obiektu wywołaniem `delete` w ramach własnej destrukcji. Stosowanie inteligentnych wskaźników zwalnia więc programistę od konieczności śledzenia pożądanych momentów zwalniania obiektów. Do tego inteligentne

wskaźniki mogą ukrywać szczegóły dealokacji, dzięki czemu klienci nie muszą wie-

dzieć, kiedy wywoływać `delete`, kiedy specjalną funkcję zwalniającą, a kiedy w ogóle powstrzymać się od zwalniania zasobu.

Bezpieczne i efektywne wskaźniki inteligentne to ważna broń w arsenale programisty. Choć biblioteka standardowa języka C++ udostępnia szablon `std::auto_ptr`, jego implementacja nie spełnia wymienionych postulatów funkcjonalności inteligentnego wskaźnika. Wskaźniki `auto_ptr` nie mogą na przykład występować w roli elementów kontenerów biblioteki STL. Lukę w standardzie wypełniają z powodzeniem klasy inteligentnych wskaźników z biblioteki Boost.

W niniejszym rozdziale skupimy się na klasach `scoped_ptr`, `shared_ptr`, `intrusive_ptr` i `weak_ptr`. Uzupełniające ten zestaw klasy `scoped_array` i `shared_array`, choć też przydatne, nie są tak często potrzebne; do tego ich podobieństwo do pozostałych implementacji wskaźników uzasadnia mniejszy poziom szczegółowości omówienia.

Jak ma się biblioteka Smart_ptr do biblioteki standardowej C++?

Biblioteka Smart_ptr została zaproponowana do wcielenia do biblioteki standardowej. Propozycja jest uzasadniona trojako:

- Biblioteka standardowa języka C++ oferuje obecnie jedynie klasę `auto_ptr`, pokrywającą zaledwie wąski wycinek spektrum zastosowań inteligentnych wskaźników. Zwłaszcza w porównaniu do klasy `shared_ptr`, udostępniającej odmienne, a jakże ważne udogodnienia.
- Wskaźniki inteligentne biblioteki Boost zostały zaprojektowane jako uzupełnienie i naturalne rozszerzenie biblioteki standardowej. Na przykład przed zaproponowaniem `shared_ptr` nie istniały standardowe wskaźniki inteligentne nadające się do użycia w roli elementów standardowych kontenerów.
- Programiści ustanowili inteligentne wskaźniki biblioteki Boost standardem de facto, powszechnie i z powodzeniem wdrażając je we własnych programach.

Wymienione względy sprawiają, że biblioteka Smart_ptr stanowi bardzo pożądany dodatek do biblioteki standardowej języka C++. Klasy `shared_ptr` (i szablon pomocniczy `enable_shared_from_this`) i `weak_ptr` z Boost.Smart_ptr zostały więc zaakceptowane do najbliższego raportu technicznego biblioteki standardowej, czyli dokumentu zbierającego propozycje dla najbliższego wydania standardu opisującego bibliotekę języka C++.

scoped_ptr

Nagłówek: "boost/scoped_ptr.hpp"

Szablon boost::scoped_ptr służy do zapewniania właściwego usuwania przydzielanego dynamicznie obiektu. Cechy klasy scoped_ptr upodobniają ją do std::auto_ptr, z tą istotną różnicą, że w scoped_ptr nie zachodzi transfer prawa własności, charakterystyczny dla auto_ptr. W rzeczy samej, wskaźnik scoped_ptr nie może być kopiowany ani przypisywany! Wskaźnik scoped_ptr gwarantuje zachowanie wyłącznego posiadania obiektu wskazywanego, uniemożliwiając przypadkowe ustąpienie własności. Ta własność scoped_ptr pozwala na wyraziste rozgraniczenie tej różnicy w kodzie poprzez stosowanie raz scoped_ptr, a raz auto_ptr, zależnie od potrzeb.

Wybierając pomiędzy std::auto_ptr a boost::scoped_ptr, należy rozważyć właśnie to, czy pożądaną cechą tworzonego inteligentnego wskaźnika ma być transfer prawa własności obiektu wskazywanego. Jeśli nie, najlepiej zastosować scoped_ptr. Jego implementacja jest na tyle odchudzona, że jego wybór nie spowoduje ani rozrostu, ani spowolnienia programu — wpłynie za to korzystnie na bezpieczeństwo kodu i jego zdatność do konserwacji.

Pora na przegląd składni scoped_ptr, uzupełniony krótkim opisem poszczególnych składowych.

```
namespace boost {

   template<typename T> class scoped_ptr : noncopyable {
   public:
      explicit scoped_ptr(T* p = 0);
      ~scoped_ptr();

      void reset(T* p = 0);

      T& operator*() const;
      T* operator->() const;
      T* get() const;

      void swap(scoped_ptr& b);
   };

   template<typename T>
      void swap(scoped_ptr<T> & a, scoped_ptr<T> & b);
}
```

Metody

```
explicit scoped_ptr(T* p = 0);
```

Konstruktor przechowujący kopię p. Uwaga: p musi być przydzielone za pośrednictwem wywołania operatora new albo mieć wartość pustą (ang. *null*). T nie musi być w czasie

konstrukcji wskaźnika kompletnym typem. To przydatne, kiedy wskaźnik `p` jest wynikiem wywołania pewnej funkcji przydziału, a nie bezpośredniego wywołania `new`: skoro typ nie musi być kompletny, wystarcza deklaracja zapowiadająca `T`. Konstruktor nie zrzuca wyjątków.

```
~scoped_ptr();
```

Usuwa obiekt wskazywany. Typ `T` musi być kompletny przy usuwaniu. Jeśli wskaźnik `scoped_ptr` nie przechowuje w czasie destrukcji żadnego zasobu, destruktor nie wykonuje żadnych operacji dealokacji. Destruktor nie zrzuca wyjątków.

```
void reset(T* p = 0);
```

Wyzerowanie (ang. *reset*) wskaźnika `scoped_ptr` oznacza zwolnienie pozostającego w jego pieczy wskaźnika (o ile taki istnieje), a następnie przyjęcie na własność wskaźnika `p`. Zazwyczaj `scoped_ptr` przejmuje całkowicie zarządzanie czasem życia obiektu, ale w rzadkich sytuacjach, kiedy trzeba zwolnić zasób jeszcze przed zwolnieniem obiektu wskaźnika `scoped_ptr` albo trzeba przekazać pod jego opiekę zasób inny niż pierwotny. Jak widać, metoda `reset` może się przydać, ale należy ją stosować wstrzemięźliwie (zbyt częste stosowanie znamionuje niekiedy ułomności projektu). Metoda nie zrzuca wyjątków.

```
T& operator*() const;
```

Zwraca referencję obiektu wskazywanego przez wskaźnik przechowywany w obiekcie `scoped_ptr`. Ponieważ nie istnieje coś takiego jak referencje puste, wyłuskiwanie za pośrednictwem tego operatora obiektu `scoped_ptr` zawierającego wskaźnik pusty prowokuje niezdefiniowane zachowanie. Jeśli więc zachodzą wątpliwości co do wartości przechowywanego wskaźnika, należy skorzystać z metody `get`. Operator nie zrzuca wyjątków.

```
T* operator->() const;
```

Zwraca przechowywany wskaźnik. Wywołanie tej metody na rzecz obiektu `scoped_ptr` zawierającego wskaźnik pusty prowokuje niezdefiniowane zachowanie. Jeśli nie ma pewności, czy wskaźnik jest pusty, czy nie, należy zastosować metodę `get`. Operator nie zrzuca wyjątków.

```
T* get() const;
```

Zwraca przechowywany wskaźnik. Metodę `get` należy stosować z zachowaniem ostrożności, a to z racji ryzyka związanego z manipulowaniem „gołym” wskaźnikiem. Metoda `get` przydaje się jednak choćby do jawnego sprawdzenia, czy przechowywany wskaźnik jest pusty. Metoda nie zrzuca wyjątków. Wywołuje się ją typowo celem zaspokojenia wymogów, np. wywołania funkcji wymagającej przekazania argumentu w postaci zwykłego wskaźnika.

```
operator nieokreślony-typ-logiczny() const
```

Określa, czy `scoped_ptr` jest niepusty. Typ wartości zwracanej (bliżej nieokreślony) powinien nadawać się do stosowania w kontekście wymagającym wartości logicznych.

Tę funkcję konwersji można stosować zamiast metody `get` w instrukcjach warunkowych do testowania stanu wskaźnika `scoped_ptr`.

```
void swap(scoped_ptr& b);
```

Wymienia zawartość dwóch obiektów klasy `scoped_ptr`. Nie zrzuca wyjątków.

Funkcje zewnętrzne

```
template<typename T> void swap(scoped_ptr<T> & a, scoped_ptr<T> & b);
```

Szablon funkcji realizującej preferowaną metodę wymiany zawartości dwóch obiektów klasy `scoped_ptr`. To preferowana metoda podmiany, bo wywołanie `swap(scoped1, scoped2)` może być stosowane w sposób uogólniony (w kodzie szablonowym) dla wielu typów wskaźnikowych, w tym gołych wskaźników i inteligentnych wskaźników w implementacjach zewnętrznych[2]. Tymczasem alternatywne wywołanie `scoped1.swap(scoped2)` zadziała tylko dla odpowiednio wyposażonych wskaźników inteligentnych, ale już nie dla wskaźników zwykłych.

Stosowanie

Klasę `scoped_ptr` stosuje się jak zwykły typ wskaźnikowy, z paroma zaledwie (za to istotnymi) różnicami; najważniejsza objawia się w tym, że nie trzeba pamiętać o wywoływaniu `delete` dla takiego wskaźnika i że nie można używać go w operacjach kopiowania. Typowe operatory wyłuskania dla typów wskaźnikowych (`operator*` i `operator->`) są przeciążone dla klasy `scoped_ptr`, tak aby składniowo stosowanie wskaźników inteligentnych nie różniło się od stosowania wskaźników zwykłych. Odwołania za pośrednictwem wskaźników `scoped_ptr` są równie szybkie, jak za pośrednictwem wskaźników zwykłych, nie ma tu też żadnych dodatkowych narzutów co do rozmiaru — można więc ich używać powszechnie. Stosowanie klasy `boost::scoped_ptr` wymaga włączenia do kodu pliku nagłówkowego `"boost/scoped_ptr.hpp"`. Przy deklarowaniu wskaźnika `scoped_ptr` szablon konkretyzuje się typem obiektu wskazywanego. Oto przykładowy `scoped_ptr`, kryjący wskaźnik obiektu klasy `std::string`:

```
boost::scoped_ptr<std::string> p(new std::string("Ahoj"));
```

Przy usuwaniu obiektu `scoped_ptr` jego destruktor sam wywołuje operator `delete` dla przechowywanego wskaźnika.

Brak konieczności ręcznego usuwania obiektu

Spójrzmy na program, który wykorzystuje obiekt klasy `scoped_ptr` do zarządzania wskaźnikiem obiektu klasy `std::string`. Zauważmy brak jawnego wywołania `delete`; obiekt `scoped_ptr` jest zmienną automatyczną i jako taka podlega usuwaniu przy wychodzeniu z zasięgu.

---

[2] Dla takich zewnętrznych implementacji wskaźników inteligentnych, które nie udostępniają swojej wersji `swap`, można napisać własną funkcję wymiany — *przyp. aut.*

```
#include "boost/scoped_ptr.hpp"
#include <string>
#include <iostream>

int main()
   {
   boost::scoped_ptr<std::string>
      p(new std::string("Zawsze używaj scoped_ptr!"));

   // Wypisanie ciągu na wyjściu programu
   if (p)
   std::cout << *p << '\n';

   // Określenie długości ciągu
   size_t i=p->size();

   // Pzypisanie nowej wartości ciągu
   *p="Jak zwykły wskaźnik";

   }  // Tu usunięcie p i zwolnienie (delete) wskazywanego obiektu std::string
}
```

W powyższym kodzie wypadałoby zwrócić uwagę na kilka elementów. Przede wszystkim `scoped_ptr` można testować jak zwykłe wskaźniki, bo udostępnia funkcję niejawnej konwersji na typ zdatny do stosowania w wyrażeniach logicznych. Po drugie, wywołanie metody na rzecz obiektu wskazywanego działa identycznie jak dla zwykłych wskaźników, a to z racji obecności przeciążonego operatora `operator->`. Po trzecie wreszcie, wyłuskanie `scoped_ptr` działa również tak, jak dla wskaźników zwykłych — to za sprawą przeciążonego operatora `operator*`. Te własności czynią stosowanie obiektów `scoped_ptr` (i innych wskaźników inteligentnych) tak naturalnym, jak stosowanie najzwyklejszych gołych wskaźników. Różnice sprowadzają się więc do wewnętrznego zarządzania czasem życia obiektu wskazywanego, nie do składni odwołań.

Prawie jak auto_ptr

Zasadnicza różnica pomiędzy `scoped_ptr` a `auto_ptr` sprowadza się do traktowania prawa własności do wskazywanego obiektu. Otóż `auto_ptr` przy kopiowaniu ochoczo dokonuje transferu własności — poza źródłowy obiekt `auto_ptr`; tymczasem wskaźnika `scoped_ptr` po prostu nie można skopiować. Spójrzmy na poniższy program, porównujący zachowanie `auto_ptr` i `scoped_ptr` w kontekście kopiowania.

```
void scoped_vs_auto() {

   using boost::scoped_ptr;
   using std::auto_ptr;

   scoped_ptr<std::string> p_scoped(new std::string("Ahoj"));
   auto_ptr<std::string> p_auto(new std::string("Ahoj"));
```

```
    p_scoped->size();
    p_auto->size();

    scoped_ptr<std::string> p_another_scoped=p_scoped;
    auto_ptr<std::string> p_another_auto=p_auto;

    p_another_auto->size();
    (*p_auto).size();
}
```

Niniejszy przykład nie da się nawet skompilować, bo `scoped_ptr` nie może uczestniczyć w operacji konstrukcji kopiującej ani przypisania. Tymczasem `auto_ptr` da się i kopiować, i przypisywać kopiująco, przy czym te operacje realizują przeniesienie prawa własności ze wskaźnika źródłowego (tu `p_auto`) do docelowego (tu `p_another_auto`), zostawiając oryginał ze wskaźnikiem pustym. Może to prowadzić do nieprzyjemnych niespodzianek, na przykład przy próbie umieszczenia obiektu `auto_ptr` w kontenerze[3]. Gdyby z kodu usunąć przypisanie do `p_another_scoped`, program dałby się skompilować, ale z kolei w czasie wykonania prowokowałby niewiadome zachowanie, a to z powodu próby wyłuskania pustego wskaźnika `p_auto` (`*p_auto`).

Ponieważ metoda `scoped_ptr::get` zwraca goły wskaźnik, który może posłużyć do rzeczy haniebnych, dlatego warto od razu zapamiętać dwie rzeczy, których trzeba unikać. Po pierwsze, nie zwalniać samodzielnie wskaźnika przechowywanego w obiekcie `scoped_ptr`. Będzie on usuwany ponownie przy usuwaniu tegoż obiektu. Po drugie, nie kopiować wyciągniętego wskaźnika do innego obiektu `scoped_ptr` (ani dowolnego innego wskaźnika inteligentnego). Dwukrotne usunięcie wskaźnika, po razie przy usuwaniu każdego z zawierających go obiektów `scoped_ptr`, może sprowokować nieszczęście. Krótko mówiąc, `get` należy stosować wstrzemięźliwie i tylko tam, gdzie koniecznie trzeba posługiwać się gołymi wskaźnikami!

Wskaźniki scoped_ptr a idiom prywatnej implementacji

Wskaźniki `scoped_ptr` świetnie nadają się do stosowania tam, gdzie wcześniej zastosowania znajdowały wskaźniki zwykłe albo obiekty `auto_ptr`, a więc na przykład w implementacjach *idiomu prywatnej implementacji* (ang. *pimpl*)[4]. Stojąca za nim koncepcja sprowadza się do izolowania użytkowników klasy od wszelkich informacji o prywatnych częściach tej klasy. Ponieważ użytkownicy klasy są uzależnieni od pliku nagłówkowego tejże klasy, wszelkie zmiany w tym pliku nagłówkowym wymuszają ponowną kompilację kodu użytkowników, nawet jeśli zmiany ograniczały się do obszarów prywatnych i zabezpieczonych klasy, a więc obszarów niby niedostępnych z zewnątrz. Idiom implementacji prywatnej zakłada ukrywanie szczegółów prywatnych

[3] Nigdy, przenigdy nie wolno umieszczać wskaźników `auto_ptr` w kontenerach biblioteki standardowej. Próba taka sprowokuje zazwyczaj błąd kompilacji; jeśli nie, śmiałka czekają poważniejsze kłopoty — *przyp. aut.*

[4] Więcej o samym idiomie można przeczytać w książce *Exceptional C++* (*Wyjątkowy język C++* — *przyp. tłum.*) i pod adresem *www.gotw.ca/gotw/024.htm* — *przyp. aut.*

przez przeniesienie danych i metod prywatnych do osobnego typu definiowanego w pliku implementacji; w pliku nagłówkowym mamy jedynie deklarację zapowiadającą ów typ, a w wykorzystującej go klasie — wskaźnik tego typu. Konstruktor klasy przydziela obiekt typu właściwego dla prywatnej implementacji, a destruktor klasy go zwalnia. W ten sposób można usunąć niepożądane zależności z pliku nagłówkowego. Spróbujmy skonstruować klasę implementującą idiom za pośrednictwem inteligentnych wskaźników.

```
// pimpl_sample.hpp

#if !defined (PIMPL_SAMPLE)
#define PIMPL_SAMPLE

struct impl;

class pimpl_sample {
   impl* pimpl_;
public:
   pimpl_sample();
   ~pimpl_sample();
   void do_something();
};

#endif
```

Tak prezentuje się interfejs klasy `pimpl_sample`. Mamy tu też deklarację zapowiadającą typu `struct impl`, reprezentującego typ prywatnej implementacji klasy i obejmującego prywatne składowe i metody, definiowane w pliku implementacji klasy. W efekcie użytkownicy klasy `pimpl_sample` są zupełnie odizolowani od jej wewnętrznych szczegółów implementacji.

```
// pimpl_sample.cpp

#include "pimpl_sample.hpp"
#include <string>
#include <iostream>

struct pimpl_sample::impl {
   void do_something_() {
      std::cout << s_ << '\n';
   }
   std::string s_;
};

pimpl_sample::pimpl_sample()
   : pimpl_(new impl) {
   pimpl_->s_ = "Pimpl - idiom implementacji prywatnej";
}

pimpl_sample::~pimpl_sample() {
```

```
   delete pimpl_;
}

void pimpl_sample::do_something() {
   pimpl_->do_something_();
}
```

Z pozoru kod wygląda zupełnie poprawnie, ale nie jest niestety doskonały. Otóż taka implementacja nie jest odporna na wyjątki! Sęk w tym, że konstruktor `pimpl_sample` może zrzucić wyjątek już po skonstruowaniu `pimpl`. Zgłoszenie wyjątku z konstruktora oznacza, że konstruowany obiekt nigdy w pełni nie istniał, więc przy zwijaniu stosu nie dochodzi do wywołania jego konstruktora. To zaś oznacza, że pamięć przydzielona do wskaźnika `pimpl_` nie zostanie zwolniona i dojdzie do wycieku. Można temu jednak łatwo zaradzić: na ratunek przychodzi wskaźnik `scoped_ptr`!

```
class pimpl_sample {
   struct impl;
   boost::scoped_ptr<impl> pimpl;
   …
};
```

Kwestię odporności na wyjątki załatwiamy, przekazując zadanie zarządzania czasem życia obiektu ukrytej klasy `impl` na barki wskaźnika `scoped_ptr` i usuwając jawne zwolnienie `impl` z destruktora klasy `pimpl_sample` (za sprawą `scoped_ptr` wywołanie `delete` nie jest tam już potrzebne). Wciąż trzeba jednak pamiętać o konieczności definiowania własnego destruktora; chodzi o to, że w czasie, kiedy kompilator generowałby destruktor domyślny, typ `impl` nie byłby jeszcze znany w całości, więc nie nastąpiłoby wywołanie jego destruktora. Gdyby obiekt prywatnej implementacji był wskazywany wskaźnikiem `auto_ptr`, kod taki skompilowałby się bez błędów; użycie `scoped_ptr` prowokuje błąd kompilacji.

Kiedy `scoped_ptr` występuje w roli składowej klasy, trzeba ręcznie definiować dla tej klasy konstruktor kopiujący i kopiujący operator przypisania. Chodzi naturalnie o to, że wskaźników `scoped_ptr` nie można kopiować, więc klasa zawierająca takie wskaźniki również przestaje nadawać się do kopiowania (przynajmniej prostego kopiowania składowych).

Na koniec warto zaznaczyć, że jeśli egzemplarz implementacji prywatnej (tu `pimpl`) da się bezpiecznie dzielić pomiędzy egzemplarzami klasy z niej korzystającej (tu `pimpl_sample`), wtedy do zarządzania czasem życia obiektu implementacji należałoby wykorzystać wskaźnik klasy `boost::shared_ptr`. Zalety stosowania `shared_ptr` w takiej sytuacji przejawiają się zwolnieniem z konieczności ręcznego definiowania konstruktora kopiującego i operatora przypisania dla klasy i pustego destruktora — `shared_ptr` nadaje się do obsługi również typów niekompletnych.

scoped_ptr to nie to samo, co const auto_ptr

Uważny Czytelnik zorientował się już zapewne, że zachowanie wskaźnika `auto_ptr` można by próbować upodobnić do zachowania `scoped_ptr`, deklarując ten pierwszy ze słowem `const`:

```
const auto_ptr<A> no_transfer_of_ownership(new A);
```

To co prawda niezłe przybliżenie, ale niezupełne. Wciąż mamy taką różnicę, że wskaźnik `scoped_ptr` da się wyzerować (przestawić) wywołaniem metody `reset`, co pozwala na podstawianie nowych obiektów wskazywanych w razie potrzeby. Nie da się tego zrobić z użyciem `const auto_ptr`. Kolejna różnica, już nieco mniejsza, to różnica w wymowie nazw: `const auto_ptr` znaczy mniej więcej tyle, co `scoped_ptr`, ale przy mniej zwięzłym i jasnym zapisie. Zaś po przyswojeniu znaczenia `scoped_ptr` można stosować go w celu jasnego wyrażania intencji programisty co do zachowania wskaźnika. Jeśli trzeba gdzieś podkreślić osadzenie zasobu w zasięgu, przy równoczesnym wyrażeniu niemożności przekazywania własności obiektu wskazywanego, należy tę chęć wyrazić przez `boost::scoped_ptr`.

Podsumowanie

Gołe wskaźniki komplikują zabezpieczanie kodu przed wyjątkami i błędami wycieku zasobów. Automatyzacja kontroli czasu życia obiektów przydzielanych dynamicznie do danego zasięgu za pośrednictwem inteligentnych wskaźników to świetny sposób wyeliminowania wad zwykłych wskaźników przy równoczesnym zwiększeniu czytelności, łatwości konserwacji i ogólnie pojętej jakości kodu. Stosowanie `scoped_ptr` to jednoznaczne wyrażenie zamiaru blokowania współużytkowania i przenoszenia prawa własności obiektu wskazywanego. Przekonaliśmy się, że `std::auto_ptr` może podkradać wskazania innym obiektom tej klasy, co uważa się za największy grzech `auto_ptr`. Właśnie dlatego `scoped_ptr` tak świetnie uzupełnia `auto_ptr`. Kiedy do `scoped_ptr` przekazywany jest obiekt przydzielany dynamicznie, wskaźnik zakłada, że posiada wyłączne prawo dysponowania obiektem. Ponieważ wskaźniki `scoped_ptr` są niemal zawsze przydzielane jako obiekty (zmienne bądź składowe) automatyczne, mamy gwarancję ich usunięcia przy wychodzeniu z zasięgu, co prowadzi do pewnego zwolnienia obiektów wskazywanych, niezależnie od tego, czy opuszczenie zasięgu było planowe (instrukcją `return`), czy awaryjne, wymuszone zgłoszeniem wyjątku.

Wskaźniki `scoped_ptr` należy stosować tam, gdzie:

- w zasięgu obarczonym ryzykiem zgłoszenia wyjątku występuje wskaźnik;
- funkcja ma kilka ścieżek wykonania i kilka punktów powrotu;
- czas życia obiektu przydzielanego dynamicznie ma być ograniczony do pewnego zasięgu;
- ważna jest odporność na wyjątki (czyli prawie zawsze!).

scoped_array

Nagłówek: "boost/scoped_array.hpp"

Potrzebę tworzenia dynamicznie przydzielanych tablic elementów zwykło się zaspokajać za pomocą kontenera `std::vector`, ale w co najmniej dwóch przypadkach uzasadnione jest użycie „zwykłych" tablic: przy optymalizowaniu kodu (bo implementacja kontenera `vector` może wprowadzać narzuty czasowe i pamięciowe) i przy wyrażaniu jasnej intencji co do sztywnego rozmiaru tablicy[5]. Tablice przydzielane dynamicznie prowokują zagrożenia właściwe dla zwykłych wskaźników, z dodatkowym (zbyt częstym) ryzykiem omyłkowego zwolnienia tablicy wywołaniem `delete` zamiast `delete []`. Omyłki te zdarzyło mi się widywać w najmniej oczekiwanych miejscach, nawet w powszechnie stosowanych, komercyjnych implementacjach klas kontenerów! Klasa `scoped_array` jest dla tablic tym, czym `scoped_ptr` dla wskaźników: przejmuje odpowiedzialność za zwolnienie pamięci tablicy. Tyle że `scoped_array` robi to za pomocą operatora `delete []`.

Przyczyna, dla której `scoped_array` jest osobną klasą, a nie na przykład specjalizacją `scoped_ptr`, tkwi w niemożności rozróżnienia wskaźników pojedynczych elementów i wskaźników całych tablic za pomocą technik metaprogramowania. Mimo wysiłków nikt nie znalazł jeszcze pewnego sposobu różnicowania tych wskaźników; sęk w tym, że wskaźniki tablic dają się tak łatwo przekształcać we wskaźniki pojedynczych obiektów i nie zachowują w swoim typie żadnych informacji o tym, że wskazywały właśnie tablice. W efekcie trzeba zastosowania wskaźników zwykłych i wskaźników tablic różnicować samodzielnie, stosując dla nich jawnie `scoped_ptr` i `scoped_array` — tak jak samodzielnie trzeba pamiętać o zwalnianiu wskazywanych tablic wywołaniem `delete []` zamiast `delete`. Uciekając się do stosowania `scoped_array`, zyskujemy automatyzm zwalniania tablicy i jasność intencji: wiadomo od razu, że chodzi o wskaźnik tablicy, a nie pojedynczego elementu.

Wskaźnik `scoped_array` przypomina bardzo `scoped_ptr`; jedną z niewielu różnic jest przeciążanie (w tym pierwszym) operatora indeksowania `operator[]`, umożliwiającego stosowanie naturalnej składni odwołań do elementów tablicy.

Wskaźnik `scoped_array` należy uznać za pod każdym względem lepszą alternatywę dla klasycznych tablic przydzielanych dynamicznie. Przejmuje on zarządzanie czasem życia tablic tak, jak `scoped_ptr` przejmuje zarządzenie czasem życia obiektów wskazywanych. Trzeba jednak pamiętać, że w bardzo wielu przypadkach jest alternatywą gorszą niż kontener `std::vector` (elastyczniejszy i dający większe możliwości). Wskaźnik `scoped_array` zdaje się przewyższać kontenery `std::vector` jedynie tam, gdzie trzeba jasno wyrazić chęć sztywnego ograniczenia rozmiaru tablicy.

[5] W rzeczy samej, w zdecydowanej większości przypadków lepiej faktycznie skorzystać ze standardowej implementacji kontenera `vector`. Decyzja o stosowaniu `scoped_array` powinna wynikać z pomiarów wydajności — *przyp. aut.*

shared_ptr

Nagłówek: `"boost/shared_ptr.hpp"`

Chyba każdy nietrywialny program wymaga stosowania jakiejś postaci inteligentnych wskaźników ze zliczaniem odwołań do obiektów wskazywanych. Takie wskaźniki eliminują konieczność kodowania skomplikowanej logiki sterującej czasem życia obiektów współużytkowanych przez pewną liczbę innych obiektów. Kiedy wartość licznika odwołań spadnie do zera, oznacza to brak obiektów zainteresowanych użytkowaniem danego zasobu i możliwość jego zwolnienia. Inteligentne wskaźniki ze zliczaniem odwołań można podzielić na *ingerencyjne* i *nieingerencyjne* (ang. odpowiednio: *intrusive* i *non-intrusive*). Te pierwsze wymagają od klas obiektów zarządzanych udostępniania specjalnych metod albo składowych, za pomocą których realizowane jest zliczanie odwołań. Oznacza to konieczność projektowania klas zasobów współużytkowanych z uwzględnieniem wymaganej infrastruktury albo późniejszego uzdatniania takich klas przez np. pakowanie ich w klasy implementujące infrastrukturę zliczania odwołań. Z kolei wskaźniki zliczające odwołania w sposób nieingerencyjny nie wymagają niczego od typu obiektu zarządzanego. Wskaźniki zliczające odwołania zakładają wyłączność własności pamięci skojarzonej z przechowywanymi wskaźnikami. Kłopot ze współużytkowaniem obiektu bez pomocy ze strony inteligentnych wskaźników polega na tym, że choć trzeba wreszcie zwolnić taki obiekt, nie wiadomo, kto i kiedy miałby to zrobić. Bez pomocy ze strony mechanizmu zliczania odwołań trzeba arbitralnie i zewnętrznie ograniczyć czas życia obiektu współużytkowanego, co oznacza zwykle związanie jego użytkowników nadmiernie silnymi zależnościami. To z kolei niekorzystnie wpływa na prostotę kodu i jego zdatność do wielokrotnego użycia.

Klasa obiektu zarządzanego może przejawiać własności predestynujące ją do stosowania ze wskaźnikami zliczającymi odwołania. Takimi cechami mogą być kosztowność operacji kopiowania albo współużytkowanie części implementacji pomiędzy wieloma egzemplarzami klasy. Są też sytuacje, w których nie istnieje jawny właściciel współużytkowanego zasobu. Stosowanie inteligentnych wskaźników ze zliczaniem odwołań umożliwia dzielenie własności pomiędzy obiektami, które wymagają dostępu do wspólnego zasobu. Wskaźniki zliczające odwołania umożliwiają również przechowywanie wskaźników obiektów w kontenerach biblioteki standardowej bez ryzyka wycieków pamięci, zwłaszcza w obliczu wyjątków albo operacji usuwania elementów z kontenera. Z kolei przechowywanie wskaźników w kontenerach pozwala na wykorzystanie zalet polimorfizmu, zwiększenie efektywności składowania (w przypadku klas zakładających kosztowne kopiowanie) i możliwość składowania tych samych obiektów w wielu specjalizowanych kontenerach, wybranych ze względu na np. efektywność sortowania.

Po stwierdzeniu chęci zastosowania inteligentnego wskaźnika zliczającego odwołania trzeba zdecydować między implementacją ingerencyjną i nieingerencyjną. Niemal zawsze lepszym wyborem są wskaźniki nieingerencyjne, a to z racji ich uniwersalności, braku wpływu na istniejący kod i elastyczności. Wskaźniki nieingerencyjne można stosować z klasami, których z pewnych względów nie można zmieniać. Zwykle klasę adaptuje się do współpracy z inteligentnym wskaźnikiem ingerencyjnym przez

wyprowadzenie klasy pochodnej z klasy bazowej zliczającej odwołania. Taka modyfikacja klasy może być kosztowniejsza, niż się zdaje. W najlepszym przypadku trzeba ponieść koszt w postaci zacieśnienia zależności i zmniejszenia zdatności klasy do użycia w innych kontekstach[6]. Zwykle oznacza to również zwiększenie rozmiaru obiektów klasy adaptowanej, co w niektórych kontekstach ogranicza jej przydatność[7].

Obiekt klasy `shared_ptr` można skonstruować na bazie zwykłego wskaźnika, innego obiektu `shared_ptr` i obiektów klasy `std::auto_ptr` bądź `boost:weak_ptr`. Do konstruktora `shared_ptr` można też przekazać drugi argument, w postaci dealokatora (ang. *deleter*). Jest on później wykorzystywany do obsługi operacji usunięcia zasobu współużytkowanego. Przydaje się w zarządzaniu takimi zasobami, które nie były przydzielane wywołaniem `new` i nie powinny być zwalniane zwykłym `delete` (przykłady tworzenia własnych dealokatorów zostaną zaprezentowane dalej). Po skonstruowaniu obiektu `shared_ptr` można go stosować jak najzwyklejszy wskaźnik, z tym wyjątkiem, że nie trzeba jawnie zwalniać obiektu wskazywanego.

Poniżej prezentowana jest częściowa deklaracja szablonu `shared_ptr`; występują tu najważniejsze składowe i metody, które za chwilę doczekają się krótkiego omówienia.

```
namespace boost {

   template <typename T> class shared_ptr {
   public:
      template<class Y> explicit shared_ptr(Y* p);
      template<class Y, class D> shared_ptr(Y* p, D d);

      ~shared_ptr();

      shared_ptr(const shared_ptr & r);
      template <class Y> explicit shared_ptr(const weak_ptr<Y>& r);
      template <classY> explicit shared_ptr(std::auto_ptr<Y>& r);

      shared_ptr& operator=(const shared_ptr & r);

      void reset();

      T& operator*() const;
      T* operator->() const;
      T* get() const;

      bool unique() const;
      long use_count() const;

      operator nieokreślony-typ-logiczny() const;
```

[6] Weźmy choćby przypadek, kiedy jedną klasę obiektu zarządzanego trzeba by przystosować do stosowania z kilkoma klasami ingerencyjnych, inteligentnych wskaźników zliczających odwołania. Owe różne klasy bazowe infrastruktury zliczania odwołań mogłyby być niezgodne ze sobą; gdyby zaś tylko jedna z klas wskaźników była kategorii ingerencyjnej, stosowanie wskaźników nieingerencyjnych odbywałoby się ze zbędnym wtedy narzutem jednej klasy bazowej — *przyp. aut.*

[7] Z drugiej strony nieingerencyjne wskaźniki inteligentne wymagają dla siebie przydziału dodatkowej pamięci dla niezbędnej infrastruktury zliczania odwołań — *przyp. aut.*

```
      void swap(shared_ptr<T>& b);
   };

   template <class T, class U>
      shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
}
```

Metody

```
template<class Y> explicit shared_ptr(Y* p);
```

Konstruktor przejmujący w posiadanie przekazany wskaźnik `p`. Argument wywołania powinien być poprawnym wskaźnikiem `Y`. Konstrukcja powoduje ustawienie licznika odwołań na 1. Jedynym wyjątkiem, jaki może być zrzucony z konstruktora, jest `std::bad_alloc` (w mało prawdopodobnym przypadku, kiedy nie ma pamięci do przydzielenia licznika odwołań).

```
template<class Y, class D> shared_ptr(Y* p, D d);
```

Konstruktor przyjmujący parę argumentów. Pierwszy z nich to zasób, który ma przejść pod opiekę tworzonego wskaźnika `shared_ptr`, drugi to obiekt odpowiedzialny za zwolnienie zasobu wskazywanego przy usuwaniu wskaźnika. Zasób jest przekazywany do obiektu zwalniającego wywołaniem `d(p)`. Z tego względu poprawne wartości `p` zależą od `d`. Jeśli nie uda się przydzielić licznika odwołań, konstruktor zrzuci wyjątek `std::bad_alloc`.

```
shared_ptr(const shared_ptr & r);
```

Zasób wskazywany przez `r` jest współdzielony z nowo tworzonym wskaźnikiem `shared_ptr`, co powoduje zwiększenie licznika odwołań o 1. Konstruktor kopiujący nie zrzuca wyjątków.

```
template <class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

Konstruuje wskaźnik `shared_ptr` na podstawie wskaźnika `weak_ptr` (omawianego w dalszej części rozdziału). Pozwala to na zabezpieczenie zastosowania `weak_ptr` w aplikacjach wielowątkowych przez zwiększenie licznika odwołań do zasobu współużytkowanego wskazywanego przez argument typu `weak_ptr` (wskaźniki `weak_ptr` nie wpływają na stan liczników odwołań do zasobów współdzielonych). Jeśli wskaźnik `weak_ptr` jest pusty (to jest `r.use_count() == 0`), konstruktor `shared_ptr` zrzuca wyjątek typu `bad_weak_ptr`.

```
template <classY> explicit shared_ptr(std::auto_ptr<Y>& r);
```

Konstrukcja wskaźnika `shared_ptr` na bazie `auto_ptr` oznacza przejęcie własności wskaźnika przechowywanego w `r` przez utworzenie jego kopii i wywołanie na rzecz źródłowego obiektu `auto_ptr` metody `release`. Licznik odwołań po konstrukcji ma wartość 1. Oczywiście `r` jest zerowane. Niemożność przydziału pamięci dla licznika odwołań prowokuje wyjątek `std::bad_alloc`.

```
~shared_ptr();
```

Destruktor klasy `shared_ptr` zmniejsza o 1 licznik odwołań do zasobu wskazywanego. Jeśli wartość licznika spadnie w wyniku zmniejszenia do zera, destruktor zwolni obiekt wskazywany. Polega to na wywołaniu dlań operatora `delete` albo przekazanego w konstruktorze obiektu zwalniającego; w tym ostatnim przypadku jedynym argumentem wywołania obiektu zwalniającego jest wskaźnik przechowywany w `shared_ptr`. Destruktor nie zrzuca wyjątków.

```
shared_ptr& operator=(const shared_ptr & r);
```

Operator przypisania inicjuje współużytkowanie zasobu z `r`, z zaniechaniem współużytkowania zasobu bieżącego. Nie zrzuca wyjątków.

```
void reset();
```

Metoda `reset` służy do rezygnacji ze współdzielenia zasobu wskazywanego przechowywanym wskaźnikiem; wywołanie zmniejsza licznik odwołań do zasobu.

```
T& operator*() const;
```

Przeciążony operator zwracający referencję obiektu (zasobu) wskazywanego. Zachowanie operatora dla pustego wskaźnika przechowywanego jest niezdefiniowane. Operator nie zrzuca wyjątków.

```
T* operator->() const;
```

Przeciążony operator zwracający przechowywany wskaźnik. Razem z przeciążonym operatorem wyłuskania `operator*` upodobnia zachowanie `shared_ptr` do zwykłych wskaźników. Operator nie zrzuca wyjątków.

```
T* get() const;
```

Metoda `get` to zalecany sposób odwoływania się do wskaźnika przechowywanego w obiekcie `shared_ptr`, kiedy istnieje podejrzenie, że wskaźnik ten ma wartość pustą (kiedy to wywołania operatorów `operator*` i `operator->` prowokują niezdefiniowane zachowanie). Zauważmy, że stan wskaźnika w wyrażeniach logicznych można testować również za pośrednictwem funkcji niejawnej konwersji `shared_ptr` na typ logiczny. Metoda nie zrzuca wyjątków.

```
bool unique() const;
```

Metoda zwraca `true`, jeśli obiekt `shared_ptr`, na rzecz którego nastąpiło wywołanie, jest jedynym właścicielem przechowywanego wskaźnika. W pozostałych przypadkach zwraca `false`. Metoda nie zrzuca wyjątków.

```
long use_count() const;
```

Metoda `use_count` zwraca wartość licznika odwołań do wskaźnika przechowywanego w obiekcie `shared_ptr`. Przydaje się w diagnostyce, bo pozwala na zdejmowanie migawek wartości licznika odwołań w krytycznych punktach wykonania programu. Stosować

wstrzemięźliwie; dla niektórych możliwych implementacji interfejsu shared_ptr ustalenie liczby odwołań może być kosztowne obliczeniowo albo nawet niemożliwe. Metoda nie zrzuca wyjątków.

```
operator nieokreślony-typ-logiczny() const;
```

Niejawna konwersja na typ logiczny, umożliwiająca stosowanie obiektów shared_ptr w wyrażeniach logicznych (i tam, gdzie oczekiwane są wyrażenia logiczne). Zwracana wartość to true, jeśli shared_ptr przechowuje obecnie jakiś ustawiony wskaźnik, bądź false w pozostałych przypadkach. Zauważmy, że typ wartości zwracanej nie jest określony; zastosowanie typu bool pozwalałoby na angażowanie wskaźników shared_ptr w niektórych bezsensownych dla jego semantyki operacjach, stąd w implementacji najczęściej stosowany jest idiom *bezpiecznej wartości logicznej*[8], który ogranicza zastosowania wartości zwracanej do odpowiednich testów logicznych. Metoda nie zrzuca wyjątków.

```
void swap(shared_ptr<T>& b);
```

Niekiedy trzeba wymienić wskaźniki pomiędzy dwoma obiektami shared_ptr. Metoda swap wymienia przechowywane wskaźniki oraz liczniki odwołań. Nie zrzuca wyjątków.

Funkcje zewnętrzne

```
template <class T, class U>
   shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
```

Wykonanie statycznego rzutowania wskaźnika przechowywanego w shared_ptr można zrealizować poprzez wydobycie wskaźnika i wywołanie static_cast, ale wyniku nie można by skutecznie umieścić w innym wskaźniku shared_ptr: nowy wskaźnik shared_ptr uznałby, że jest pierwszym posiadaczem zasobu wskazywanego. Problem eliminuje funkcja static_pointer_cast. Gwarantuje ona zachowanie poprawnej wartości licznika odwołań. Funkcja nie zrzuca wyjątków.

Stosowanie

Zasadniczym celem stosowania shared_ptr jest eliminowanie konieczności kodowania logiki wykrywającej właściwy moment zwolnienia zasobu współużytkowanego przez wielu użytkowników. Poniżej mamy prosty przykład, gdzie dwie klasy — A i B — korzystają ze wspólnego egzemplarza wartości typu int. Uwaga: korzystanie ze wskaźników shared_ptr wymaga włączenia do kodu pliku nagłówkowego "boost/shared_ptr.hpp".

```
#include "boost/shared_ptr.hpp"
#include <cassert>

class A {
```

[8] Autorstwa Petera Dimowa — *przyp. aut.*

```
   boost::shared_ptr<int> no_;
public:
   A(boost::shared_ptr<int> no) : no_(no) {}
   void value(int i) {
      *no_ = i;
   }
};

class B {
   boost::shared_ptr<int> no_;
public:
   B(boost::shared_ptr<int> no) : no_(no) {}
   void value() const {
      return *no_;
   }
};

int main() {
   boost::shared_ptr<int> temp(new int(14));
   A a(temp);
   B b(temp);
   a.value(28);
   assert(b.value()==28);
}
```

Klasy `A` i `B` z powyższego przykładu zawierają składową typu `shared_ptr<int>`. Przy tworzeniu egzemplarzy `A` i `B` przekazujemy do obu konstruktorów ten sam egzemplarz obiektu `shared_ptr`: `temp`. Oznacza to, że mamy trzy wskaźniki odnoszące się do tej samej zmiennej typu `int`: jeden w postaci obiektu `temp` i dwa zaszyte w składowych `no_` klas `A` i `B`. Gdyby współużytkowanie zmiennej było realizowane za pomocą zwykłych wskaźników, klasy `A` i `B` miałyby ciężki orzech do zgryzienia w postaci wytypowania właściwego momentu zwolnienia zmiennej. W tym przykładzie licznik odwołań do zmiennej ma aż do końca funkcji `main` wartość 3; u kresu zasięgu funkcji wszystkie obiekty `shared_ptr` zostaną usunięte, co będzie powodować zmniejszanie licznika aż do zera. Wyzerowanie sprowokuje zaś usunięcie przydzielonej zmiennej typu `int`.

Jeszcze o idiomie implementacji prywatnej

Idiom prywatnej implementacji rozpatrywaliśmy wcześniej w aspekcie wskaźników `scoped_ptr`, znakomicie nadających się do przechowywania dynamicznie przydzielanych egzemplarzy implementacji prywatnej tam, gdzie wcielenie idiomu nie zakłada kopiowania i współdzielenia implementacji pomiędzy egzemplarzami. Nie wszystkie klasy, w których można by zastosować prywatną implementację, spełniają te warunki (co nie znaczy, że w ogóle nie można zastosować w nich wskaźników `scoped_ptr`, wymaga to jednak ręcznej implementacji operacji kopiowania i przypisywania implementacji). Dla takich klas, które zakładają dzielenie szczegółów implementacji pomiędzy wieloma egzemplarzami, należałoby zarządzać wspólnymi implementacjami za pomocą wskaźników `shared_ptr`. Kiedy wskaźnikowi `shared_ptr` przekazany zostanie w posiadanie egzemplarz implementacji prywatnej, zyskamy „za darmo” możliwość

kopiowania i przypisywania implementacji. W przypadku `scoped_ptr` kopiowanie i przypisywanie było niedozwolone, bo semantyka wskaźników `scoped_ptr` nie dopuszcza kopiowania. Z tego względu obsługa kopiowania i przypisań w klasach stosujących `scoped_ptr` i utrzymujących prywatne implementacje wspólne wymagałaby ręcznego definiowania operatora przypisania i konstruktora kopiującego. Kiedy w roli wskaźnika prywatnej implementacji klasy występuje `shared_ptr`, nie trzeba udostępniać własnego konstruktora kopiującego. Egzemplarz implementacji będzie współużytkowany przez obiekty klasy; jedynie w przypadku, kiedy któryś z egzemplarzy klasy ma być wyróżniony osobnym stanem, trzeba będzie zdefiniować stosowny konstruktor kopiujący. Sama obsługa implementacji prywatnej nie różni się poza tym od przypadku, kiedy wskazywał ją wskaźnik `scoped_ptr`: wystarczy zamienić w kodzie `scoped_ptr` na `shared_ptr`.

shared_ptr a kontenery biblioteki standardowej języka C++

Przechowywanie obiektów wprost w kontenerze jest niekiedy kłopotliwe. Przechowywanie przez wartość oznacza, że użytkownicy kontenera wydobywający zeń obiekty otrzymują ich kopie, co w przypadku obiektów cechujących się wysokim kosztem kopiowania może znacząco wpływać na wydajność programu. Co więcej, niektóre kontenery, zwłaszcza `std::vector`, podejmują operację kopiowania przechowywanych elementów w obliczu konieczności zmiany rozmiaru kontenera, co znów stanowi dodatkowy, potencjalnie ważki koszt. Wreszcie semantyka typowa dla wartości oznacza brak zachowania polimorficznego. Jeśli obiekty przechowywane w kontenerach mają przejawiać zachowanie polimorficzne, należałoby skorzystać ze wskaźników. Jeśli będą to wskaźniki zwykłe, staniemy w obliczu poważnego problemu zarządzania spójnością wskazań w poszczególnych elementach kontenera. Choćby przy usuwaniu elementów z kontenera trzeba będzie sprawdzić, czy istnieją jeszcze użytkownicy korzystający z kopii wskaźników wydobytych wcześniej z kontenera; trzeba też będzie koordynować odwołania do tego samego elementu inicjowane przez różnych użytkowników. Wszystkie te złożone zadania może z powodzeniem przejąć `shared_ptr`.

Poniższy przykład ilustruje sposób przechowywania współdzielonych wskaźników w kontenerze biblioteki standardowej.

```
#include "boost/shared_ptr.hpp"
#include <vector>
#include <iostream>

class A {
public:
   virtual void sing()=0;
protected:
  virtual ~A() {};
};

class B : public A {
public:
  virtual void sing() {
      std::cout << "do re mi fa sol la";
   }
```

```
};

boost::shared_ptr<A> createA() {
   boost::shared_ptr<A> p(new B());
   return p;
}

int main() {
   typedef std::vector<boost::shared_ptr<A> > container_type;
   typedef container_type::iterator iterator;

   container_type container;
   for (int i=0;i<10;++i) {
      container.push_back(createA());
   }

   std::cout << "Próba chóru: \n";
   iterator end=container.end();
   for (iterator it=container.begin();it!=end;++it) {
      (*it)->sing();
   }
}
```

Dwie widniejące powyżej klasy `A` i `B` zawierają po jednej metodzie wirtualnej `sing`. `B` dziedziczy publicznie po `A`, a funkcja wytwórcza `createA` zwraca dynamicznie przydzielany egzemplarz `B` ujęty w obiekcie `shared_ptr<A>`. W funkcji `main` dochodzi do utworzenia wektora elementów typu `shared_ptr<A>` i wypełnienia go dziesięcioma takimi elementami; następnie w pętli następują wywołania metody `sing` na rzecz każdego elementu kontenera. Gdybyśmy stosowali tu wskaźniki zwykłe, musielibyśmy ręcznie zwalniać zawartość kontenera. W tym przykładzie zwalniany jest automatyczne; licznik odwołań danego elementu ma wartość równą co najmniej 1 tak długo, jak długo istnieje obiekt kontenera. Kiedy ten jest usuwany, liczniki odwołań elementów są zerowane, co wymusza zwolnienie obiektów wskazywanych. Warto odnotować, że nawet gdyby destruktor `A` nie został zadeklarowany jako wirtualny, `shared_ptr` poprawnie wywołałby przy zwalnianiu obiektu wskazywanego destruktor `B`!

Przykład ilustruje efektywną technikę angażującą chroniony destruktor klasy `A`. Ponieważ funkcja `createA` zwraca obiekt klasy `shared_ptr<A>`, nie byłoby możliwe proste wywołanie `delete` dla wskaźnika wyłuskanego wywołaniem metody `shared_ptr::get`. Oznacza to, że gdyby pozyskać wskaźnik z `shared_ptr` — na przykład w celu przekazania go do funkcji wymagającej argumentu prostego typu wskaźnikowego — nie istniałaby możliwość ryzykownego i niebezpiecznego zwolnienia obiektu wskazywanego. W jaki więc sposób `shared_ptr` radzi sobie z prawidłowym zwolnieniem obiektu wskazywanego? Otóż z pomocą przychodzi właściwy typ wskaźnika, czyli `B`; zaś destruktor `B` nie jest zabezpieczony przed dostępem z zewnątrz. To bardzo dobry sposób dodatkowego zabezpieczania obiektów przechowywanych za pomocą wskaźników `shared_ptr`.

Wskaźniki shared_ptr a nietypowe zasoby

Niekiedy pojawia się potrzeba zastosowania wskaźnika `shared_ptr` z zasobem takiego typu, że jego zwolnienie nie sprowadza się do prostego wywołania `delete`. Takie przypadki obsługuje się za pośrednictwem *własnych dealokatorów*. Rzecz dotyczy na przykład uchwytów zasobów systemowych, jak `FILE*`, które należałoby zwalniać przy użyciu mechanizmów systemu operacyjnego, np. wywołaniem funkcji `fclose`. Gdyby więc wskaźnik `shared_ptr` miał przechowywać obiekty `FILE*`, należałoby zdefiniować klasę obiektów wykorzystywanych do zwalniania `FILE*`, jak poniżej.

```
class FileCloser {
public:
   void operator()(FILE* file) {
      std::cout << "Wywołanie FileCloser uchwytu dla FILE* -- "
         "plik zostanie zamknięty. \n";
      if (file!=0)
         fclose(file);
   }
};
```

Powyższy obiekt funkcyjny służy do realizacji specyficznego trybu zwalniania obiektu wskazywanego, polegającego na wywołaniu funkcji `fclose`. Oto program przykładowy, wykorzystujący taki obiekt zwalniający.

```
int main() {
   std::cout << "wskaźnik shared_ptr z własnym dealokatorem.\n";
   {
      FILE* f=fopen("test.txt", "r");
      if (f==0) {
         std::cout << "Nie można otworzyć pliku\n";
         throw "Nie można otworzyć pliku";
      }

      boost::shared_ptr<FILE>
         my_shared_file(f, FileCloser());

      // Pozycjonowanie kursora pliku
      fseek(my_shared_file.get(), 42, SEEK_SET);
   }
   std::cout << "Plik został przed chwilą zamknięty!\n";
}
```

Zauważmy, że pozyskanie zasobu ze wskaźnika wymagało zastosowania składni `&*`, metody `get` albo `get_pointer` klasy `shared_ptr` (oczywiście protestuję przeciwko stosowaniu wyjątkowo nieczytelnego zapisu `&*`; mniej oczywisty jest wybór pomiędzy dwoma pozostałymi sposobami). Przykład mógłby być jeszcze prostszy — skoro przy dealokacji zasobu wystarczyło wywołanie funkcji jednoargumentowej, nie trzeba w ogóle definiować własnej klasy dealokatorów. Uproszczony przykład mógłby wyglądać tak:

```
{
   FILE* f=fopen("test.txt", "r");
   if (f==0) {
      std::cout << "Nie można otworzyć pliku\n";
      throw file_exceptio();
   }

   boost::shared_ptr<FILE>
      my_shared_file(f, &fclose);

   // Pozycjonowanie kursora pliku
   fseek(my_shared_file.get(), 42, SEEK_SET);
}
std::cout << "Plik został przed chwilą zamknięty!\n";
```

Własne dealokatory są nieodzowne w przypadku zasobów, których zwalnianie wymaga wdrożenia specjalnej procedury. Ponieważ taki obiekt nie stanowi części typu `shared_ptr`, użytkownicy nie muszą posiadać żadnej wiedzy o zasobie pozostającym w posiadaniu inteligentnego wskaźnika (muszą, rzecz jasna, jedynie wiedzieć, jak tego wskaźnika używać!). W przykładowym przypadku puli obiektów-zasobów dealokator zwracałby po prostu zasób do puli. Z kolei w przypadku zasobu-jedynaka (ang. *singleton*) dealokator powstrzymywałby się od jakichkolwiek operacji, zapewniając podtrzymanie obecności jedynego egzemplarza zasobu.

Dealokatory a bezpieczeństwo

Wiemy już, że zabezpieczanie destruktora klasy zwiększa jej bezpieczeństwo w połączeniu ze wskaźnikami `shared_ptr`. Innym sposobem osiągnięcia podobnego poziomu bezpieczeństwa jest zadeklarowanie destruktora jako zabezpieczonego (bądź prywatnego) i wykorzystanie własnego dealokatora. Musi on zostać zaprzyjaźniony z klasą, której obiekty ma usuwać. Elegancki sposób implementacji takiego dealokatora polega na zagnieżdżeniu jego klasy w prywatnej części klasy obiektów usuwanych, jak tutaj:

```
#include "boost/shared_ptr.hpp"
#include <iostream>

class A {
   class deleter {
   public:
      void operator()(A* p) {
         delete p;
      }
   };
   friend class deleter;
public:
   virtual void sing() {
      std::cout << "Lalalalalalalalalalala";
   }

   static boost::shared_ptr<A> createA() {
      boost::shared_ptr<A> p(new A(), A::deleter());
      return p;
   }
```

```
protected:
  virtual ~A() {};
};

int main() {
   boost::shared_ptr<A> p=A::createA();
}
```

Zauważmy, że tym razem nie możemy tworzyć obiektów `shared_ptr<A>` za pomocą zewnętrznej funkcji wytwórczej, bo zagnieżdżona klasa dealokatora jest prywatna względem `A`. Taka implementacja uniemożliwia użytkownikom tworzenie obiektów `A` na stosie i nie pozwala na wywoływanie `delete` ze wskaźnikiem `A`.

Tworzenie wskaźnika shared_ptr ze wskaźnika this

Niekiedy trzeba utworzyć wskaźnik inteligentny `shared_ptr` ze wskaźnika `this` — co oznacza założenie, że klasa będzie zarządzana za pośrednictwem wskaźnika inteligentnego i trzeba w jakiś sposób przekonwertować wskaźnik obiektu klasy na taki `shared_ptr`. Brzmi jak niemożliwe? Cóż, rozwiązaniem jest kolejne narzędzie z zestawu wskaźników inteligentnych, które jeszcze nie doczekało się szerszego omówienia — chodzi o `boost::weak_ptr`. Otóż wskaźnik `weak_ptr` jest obserwatorem wskaźników `shared_ptr`; pozwala na podglądanie wskazania bez ingerowania w wartość licznika odwołań. Zachowanie w klasie składowej typu `weak_ptr` ze wskaźnikiem `this` pozwala na późniejsze pozyskiwanie `shared_ptr` do `this` wedle potrzeb. Aby nie trzeba było za każdym razem ręcznie pisać kodu składującego `this` w `weak_ptr` i potem pozyskiwać zeń wskaźnik `shared_ptr`, w bibliotece Boost.Smart_ptr przewidziano klasę pomocniczą o nazwie `enable_shared_from_this`. Wystarczy więc własną klasę wyprowadzić jako pochodną `enable_shared_from_this`. Oto przykład ilustrujący zastosowanie klasy pomocniczej:

```
#include "boost/shared_ptr.hpp"
#include "boost/enable_shared_from_this.hpp"

class A;

void do_stuff(boost::shared_ptr<A> p) {
   …
}

class A : public boost::enable_shared_from_this<A> {
public:
   void call_do_stuff() {
      do_stuff(shared_from_this());
   }
};

int main() {
   boost::shared_ptr<A> p(new A());
   p->call_do_stuff();
}
```

Przykład pokazuje też przypadek, w którym do zarządzania `this` potrzebny jest inteligentny wskaźnik `shared_ptr`. Otóż klasa `A` posiada metodę `call_do_stuff`, która ma wywołać funkcję zewnętrzną (wobec klasy) `do_stuff`, która z kolei oczekuje przekazania argumentu typu `boost::shared_ptr<A>`. W obrębie metody `A::call_do_stuff` wskaźnik `this` jest najzwyklejszym wskaźnikiem `A`, ale ponieważ `A` dziedziczy po `enable_shared_from_this`, wywołanie `shared_from_this` zwraca wskaźnik `this` opakowany w `shared_ptr`. W `shared_from_this`, metodzie klasy pomocniczej `enable_shared_from_this`, wewnętrznie przechowywany `weak_ptr` jest konwertowany na `shared_ptr`, co polega na zwiększeniu licznika referencji, niezbędnego w celu zachowania istnienia obiektu.

Podsumowanie

Wskaźniki inteligentne ze zliczaniem odwołań to niezwykle cenne narzędzia. Implementacja `shared_ptr` z biblioteki Boost to implementacja solidna i elastyczna, która swojej przydatności i jakości dowiodła w wyczerpujących testach praktycznych, w niezliczonych aplikacjach, środowiskach i okolicznościach. Potrzeba współużytkowania zasobów przez wielu użytkowników jest bowiem dość powszechna i najczęściej wiąże się z niemożnością albo trudnością ustalenia właściwego momentu bezpiecznego zwolnienia zasobu. Wskaźnik `shared_ptr` zwalnia użytkowników z tej troski, automatycznie opóźniając zwalnianie obiektu do momentu zlikwidowania ostatniego odwołania. Z tego względu klasę `shared_ptr` należałoby uznać za najważniejszą kategorię wskaźników inteligentnych dostępnych w bibliotekach Boost. Oczywiście pozostałe klasy wskaźników inteligentnych również należy poznać, ale ta jedna jest z pewnością najbardziej użyteczna i najbardziej warta przyswojenia i wdrażania. Dodatkowa możliwość stosowania własnych procedur usuwania zasobów wskazywanych czyni klasę `shared_ptr` uniwersalnym narzędziem obsługi aspektów zarządzania zasobami. Wskaźniki `shared_ptr` cechują się niewielkim narzutem rozmiaru względem wskaźników zwykłych. Nie spotkałem się jeszcze osobiście z sytuacją, w której ten narzut zmuszałby programistę do rezygnacji z wdrożenia wskaźników `shared_ptr`. Doprawdy, nie warto ręcznie implementować własnych klas wskaźników zliczających odwołania — praktycznie zawsze lepiej skorzystać z gotowca w postaci `shared_ptr`; nie bardzo jest jak go udoskonalić.

Wskaźniki `shared_ptr` można skutecznie stosować:

- tam, gdzie jest wielu użytkowników obiektu, ale nie ma jednego jawnego właściciela;
- tam, gdzie trzeba przechowywać wskaźniki w kontenerach biblioteki standardowej;
- tam, gdzie trzeba przekazywać wskaźniki do i z bibliotek, a nie ma jawnego wyrażenia transferu własności;
- tam, gdzie zarządzanie zasobami wymaga specjalnych procedur zwalniających[9].

[9] Przy pomocy własnych klas obiektów usuwających — *przyp. aut.*

shared_array

Nagłówek: "boost/shared_array.hpp"

Klasa shared_array to klasa inteligentnego wskaźnika umożliwiająca współużytkowanie tablic. Ma się do shared_ptr tak, jak scoped_array do scoped_ptr. Klasa shared_array różni się od shared_ptr głównie tym, że służy do zarządzania nie pojedynczymi obiektami, a całymi tablicami obiektów. Przy omawianiu scoped_array wspominałem, że w zdecydowanej większości przypadków lepszym od niego wyborem jest klasyczny kontener biblioteki standardowej std::vector. W tym przypadku przewaga jest po stronie shared_array, bo klasa ta pozwala na dzielenie *prawa własności* tablic. Interfejs shared_array przypomina interfejs shared_ptr; jedynym istotnym dodatkiem jest operator indeksowania i brak obsługi własnych dealokatorów.

Ponieważ kombinacja wskaźnika shared_ptr i kontenera std::vector (a konkretnie wskaźnika shared_ptr na wektor std::vector) cechuje się większą elastycznością niż shared_array, nie będziemy ilustrować zastosowań shared_array przykładami. Kto musi skorzystać z tej klasy, powinien sięgnąć do dokumentacji biblioteki Boost.

intrusive_ptr

Nagłówek: "boost/intrusive_ptr.hpp"

Klasa intrusive_ptr jest ingerencyjnym odpowiednikiem inteligentnego wskaźnika shared_ptr. Niekiedy nie ma wyjścia i trzeba zastosować właśnie ingerencyjny inteligentny wskaźnik zliczający odwołania. Typowym scenariuszem takiej sytuacji jest obecność gotowego kodu z wewnętrznym licznikiem odwołań, którego z braku czasu (albo innych względów) nie można przepisać. Może też chodzić o przypadek, kiedy rozmiar inteligentnego wskaźnika musi dokładnie zgadzać się z rozmiarem wskaźnika gołego albo kiedy operacje przydziału liczników odwołań wskaźników shared_ptr degradują wydajność (to bardzo rzadki przypadek!). Konieczność zastosowania ingerencyjnego wskaźnika inteligentnego jest oczywista, kiedy metoda klasy wskazywanej ma zwracać this tak, aby dało się tego wskaźnika użyć w innym wskaźniku inteligentnym (co prawda wiemy już, że takie zadanie można też rozwiązać przy użyciu wskaźników nieingerencyjnych). Klasa intrusive_ptr różni się od pozostałych wskaźników inteligentnych tym, że to użytkownik podaje licznik odwołań, którym wskaźnik będzie manipulował.

Kiedy wskaźnik intrusive_ptr zwiększa bądź zmniejsza licznik odwołań dla wskaźnika niepustego, realizuje niekwalifikowane wywołania funkcji intrusive_ptr_add_ref i intrusive_ptr_release. Funkcje te mają zapewnić poprawność wartości licznika odwołań i ewentualne usunięcie obiektu wskazywanego, kiedy licznik zostanie wyzerowany. Trzeba więc przeciążyć te funkcje dla własnego typu.

Oto częściowa deklaracja szablonu `intrusive_ptr`, prezentująca najważniejsze metody i funkcje zewnętrzne:

```
namespace boost {

   template<class T> class intrusive_ptr {
   public:
      intrusive_ptr(T* p, bool add_ref=true);

      intrusive_ptr(const intrusive_ptr& r);

      ~intrusive_ptr();

      T& operator*() const;
      T* operator->() const;
      T* get() const;

      operator nieokreślony-typ-logiczny() const;
   };

   template <class T> T* get_pointer(const intrusive_ptr<T>& p);

   template <class T,class U> intrusive_ptr<T>
      static_pointer_cast(const intrusive_ptr<U>& r);
}
```

Metody

`intrusive_ptr(T* p, bool add_ref=true);`

Konstruktor zachowujący wskaźnik `p` w `*this`. Jeśli wskaźnik `p` jest niepusty i jeśli `add_ref` ma wartość `true`, konstruktor inicjuje niekwalifikowane wywołanie `intrusive_ptr_add_ref(p)`. Jeśli `add_ref` ma wartość `false`, konstruktor rezygnuje z wywołania funkcji `intrusive_ptr_add_ref`. Zrzucanie wyjątków przez konstruktor jest uzależnione od `intrusive_ptr_add_ref` — konstruktor może zrzucić wyjątek, jeśli `intrusive_ptr_add_ref` może zrzucić wyjątek.

`intrusive_ptr(const intrusive_ptr& r);`

Konstruktor kopiujący zachowuje kopię wskaźnika zwracanego wywołaniem `r.get()` i — jeśli jest to wskaźnik niepusty — wywołuje dla niego funkcję `intrusive_ptr_add_ref`. Konstruktor nie zrzuca wyjątków.

`~intrusive_ptr();`

Jeśli przechowywany wskaźnik jest niepusty, destruktor `intrusive_ptr` podejmuje niekwalifikowane wywołanie funkcji `intrusive_ptr_release`, przekazując przechowywany wskaźnik jako argument wywołania. Funkcja `intrusive_ptr_add_ref` jest odpowiedzialna za zmniejszenie licznika odwołań i ewentualne zwolnienie obiektu wskazywanego (jeśli licznik zostanie wyzerowany). Funkcja nie zrzuca wyjątków.

```
T& operator*() const;
```

Operator wyłuskania `operator*` zwraca referencję obiektu wskazywanego przez przechowywany wskaźnik. Jeśli wskaźnik jest pusty, wywołanie operatora prowokuje niezdefiniowane zachowanie. Kiedy więc są wątpliwości co do stanu wskazania, należy się wcześniej upewnić, czy `intrusive_ptr` zawiera niepusty wskaźnik. Można to zrobić albo za pośrednictwem metody `get`, albo poprzez testowanie obiektu `intrusive_ptr` w wyrażeniu logicznym. Operator wyłuskania nie zrzuca wyjątków.

```
T* operator->() const;
```

Operator zwraca przechowywany wskaźnik. Wywołanie operatora, jeśli wskaźnik jest pusty, prowokuje niezdefiniowane zachowanie. Operator nie zrzuca wyjątków.

```
T* get() const;
```

Metoda zwraca przechowywany wskaźnik. Może być skutecznie wywoływana wszędzie tam, gdzie potrzebny jest goły wskaźnik, nawet jeśli wskaźnik przechowywany jest wskaźnikiem pustym. Metoda `get` nie zrzuca wyjątków.

```
operator nieokreślony-typ-logiczny() const;
```

Niejawna konwersja na typ zdatny do stosowania w wyrażeniach logicznych. Typ wartości zwracanej to nie `bool`, bo taki typ pozwalałby na angażowanie wskaźników `intrusive_ptr` w niektórych bezsensownych dla jego semantyki operacjach. Konwersja pozwala na testowanie wartości wskaźnika `intrusive_ptr` w kontekstach wyrażeń logicznych, np. `if (p)` (gdzie `p` to egzemplarz `intrusive_ptr`). Wartość zwracana to `true`, kiedy `intrusive_ptr` zawiera wskaźnik niepusty, i `false`, kiedy wskazanie jest puste. Konwersja nie prowokuje wyjątków.

Funkcje zewnętrzne

```
template <class T> T* get_pointer(const intrusive_ptr<T>& p);
```

Funkcja zwraca wartość `p.get()`, a jej rolą jest głównie wsparcie dla programowania uogólnionego[10]. Może też być wykorzystywana w konwencjach kodowania zakładających możliwość przeciążenia jej dla wskaźników gołych i zewnętrznych klas wskaźników inteligentnych. Niektórzy zaś po prostu preferują wywołania funkcji zewnętrznych przed wywołaniami metod na rzecz obiektów[11]. Funkcja nie zrzuca wyjątków.

[10] Takim funkcjom nadano miano *podkładek* (ang. *shims*) — zobacz 12. pozycję bibliografii — *przyp. aut.*

[11] Uzasadnienie sprowadza się do zaciemnienia rozróżnienia pomiędzy operacjami na wskaźnikach inteligentnych a operacjami na obiektach wskazywanych. Na przykład wywołania `p.get()` i `p->get()` mają w przypadku wskaźników inteligentnych zupełnie odmienne znaczenie, a rozróżnienie jest na pierwszy rzut oka mało wyraźne; dla porównania, wywołań `get_pointer(p)` i `p->get()` nie da się mylnie zinterpretować. Rzecz sprowadza się jednak raczej do konwencji i nawyków niż faktycznej wyższości jednej postaci nad drugą — *przyp. aut.*

```
template <class T,class U> intrusive_ptr<T>
   static_pointer_cast(const intrusive_ptr<U>& r);
```

Funkcja zwraca `intrusive_ptr<T>(static_cast<T*>(r.get()))`. Inaczej niż w przypadku `shared_ptr`, wskaźniki obiektów przechowywane w `intrusive_ptr` można śmiało poddawać rzutowaniu `static_cast`. Funkcja ta ma jednak ujednolicać składnię rzutowania wszystkich wskaźników inteligentnych. Funkcja `static_pointer_cast` nie zrzuca wyjątków.

Stosowanie

Stosowanie wskaźników `intrusive_ptr` różni się od stosowania wskaźników `shared_ptr` w dwóch zasadniczych aspektach. Po pierwsze, trzeba samodzielnie udostępnić mechanizm zliczania odwołań. Po drugie, wskaźnik `this` można z powodzeniem traktować jako wskaźnik inteligentny[12], co często okazuje się wygodne (o czym będziemy się mogli za chwilę przekonać). Niemniej jednak w większości przypadków właściwym wskaźnikiem inteligentnym jest nieingerencyjny wskaźnik `shared_ptr`.

Stosowanie klasy `intrusive_ptr` wymaga włączenia do kodu pliku nagłówkowego `"boost/intrusive_ptr.hpp"` i zdefiniowania pary funkcji `intrusive_ptr_add_ref` i `intrusive_ptr_release`. Powinny one przyjmować pojedynczy argument będący wskaźnikiem typu wykorzystywanego ze wskaźnikami `intrusive_ptr`. Ewentualne wartości zwracane z tych funkcji są ignorowane. Zwykle dobrze jest sparametryzować obie funkcje typem wskaźnika i w ramach ich implementacji przekazywać wywołanie do odpowiedniej metody konkretnego typu (np. metody `add_ref` czy `release` klas wskazywanych). Kiedy licznik odwołań osiągnie wartość zerową, funkcja `intrusive_ptr_release` powinna zadbać o zwolnienie obiektu wskazywanego. Oto uogólniona implementacja obu funkcji:

```
template <typename T> void intrusive_ptr_add_ref(T* t) {
   t->add_ref();
}
template <typename T> void intrusive_ptr_release(T* t) {
   if (t->release() <= 0)
      delete t;
}
```

Zauważmy, że funkcje powinny być definiowane w zasięgu odpowiednim dla typu ich argumentów. Oznacza to, że w wywołaniach funkcji z argumentami typu zagnieżdżonego w przestrzeni nazw funkcje należy zdefiniować w tejże przestrzeni nazw. Właśnie dlatego wywołania inicjowane z klasy `intrusive_ptr` są niekwalifikowane — umożliwia to dopasowywanie typów argumentów; w niektórych przypadkach może pojawić się potrzeba udostępnienia większej liczby wersji funkcji, przez co funkcji nie należy definiować w globalnej przestrzeni nazw. Przykład rozmieszczenia funkcji zobaczymy niebawem; tymczasem musimy zadbać o udostępnienie jakiegoś licznika odwołań.

[12] Nie jest to możliwe w przypadku `shared_ptr`, o ile nie stosuje się specjalnych środków, np. w postaci klasy `enable_shared_from_this` — *przyp. aut.*

Udostępnianie licznika odwołań

Po zdefiniowaniu funkcji manipulujących licznikiem odwołań wypadałoby udostępnić sam licznik. W prezentowanym przykładzie licznik będzie prywatną składową klasy licznika, inicjalizowanym zerem; klasa będzie ponadto udostępniać metody `add_ref` i `release`, operujące na liczniku. Metoda `add_ref` będzie zwiększać licznik odwołań, a metoda `release` — zmniejszać go[13]. Moglibyśmy uzupełnić definicję licznika o trzecią metodę, zwracającą bieżącą liczbę odwołań, wystarczy jednak, jeśli liczbę tę będzie zwracać metoda `release`. Poniższa klasa bazowa licznika `reference_counter` implementuje licznik i obie postulowane metody; uzupełnienie obiektów wskazywanych o liczniki polega na dziedziczeniu po tej klasie.

```
class reference_counter {
   int ref_count_;
   public:
      reference_counter() : ref_count_(0) {}

      virtual ~reference_counter() {}

      void add_ref() {
         ++ref_count_;
      }

      int release() {
         return --ref_count_;
      }

   protected:
      reference_counter& operator=(const reference_counter&) {
         // Atrapa
         return *this;
      }
   private:
      // Blokada dostępu do konstruktora kopiującego
      reference_counter(const reference_counter&);
};
```

Przyczyną oznaczenia destruktora klasy `reference_counter` słowem `virtual` jest to, że klasa ma służyć jako klasa bazowa w dziedziczeniu publicznym, co z kolei wymaga, aby wskaźnik `reference_counter` pozwalał na zwolnienie obiektu klasy pochodnej wywołaniem `delete`. Owe zwolnienie powinno wywołać destruktor na rzecz obiektu klasy pochodnej. Implementacja licznika jest wyjątkowo nieskomplikowana: metoda `add_ref` zwiększa licznik odwołań, a `release` zmniejsza go i zwraca jego nową wartość. Aby skorzystać z licznika odwołań, wystarczy wyprowadzić z niego (dziedziczeniem publicznym) klasę obiektów wskazywanych. Oto przykładowa klasa `some_class` zawierająca wewnętrzny licznik odwołań i program operujący na obiektach tej klasy za pośrednictwem wskaźników `intrusive_ptr`:

[13] W środowisku wielowątkowym wszelkie operacje na zmiennej przechowującej licznik odwołań powinny być odpowiednio synchronizowane — *przyp. aut.*

```
#include <iostream>
#include "boost/intrusive_ptr.hpp"

class some_class : public reference_counter {
public:
   some_class() {
      std::cout << "some_class::some_class()\n";
   }

   some_class(const some_class& other) {
      std::cout << "some_class(const some_class& other)\n";
   }

   ~some_class() {
      std::cout << "some_class::~some_class()\n";
   }
};

int main() {
   std::cout << "Przed wejściem do zasięgu\n";
   {
      boost::intrusive_ptr<some_class> p1(new some_class());
      boost::intrusive_ptr<some_class> p2(p1);
   }
   std::cout << "Po wyjściu z zasięgu\n";
}
```

Współdziałanie klasy `intrusive_ptr` z funkcjami `intrusive_ptr_add_ref` i `intrusive_ptr_release` ilustruje wynik uruchomienia powyższego programu:

```
Przed wejściem do zasięgu
some_class::some_class()
some_class::~some_class()
Po wyjściu z zasięgu
```

Wskaźniki `intrusive_ptr` zwalniają nas z szeregu obowiązków. Przy tworzeniu pierwszego takiego wskaźnika (`p1`) otrzymuje on nowy egzemplarz klasy `some_class`. Konstruktor `intrusive_ptr` przyjmuje faktycznie aż dwa argumenty. Drugi to wartość typu `bool`, określająca, czy przy konstrukcji ma nastąpić wywołanie `intrusive_ptr_add_ref`. Ponieważ domyślna wartość tego argumentu to `true`, konstrukcja `p1` wiąże się ze zwiększeniem licznika odwołań do tego egzemplarza `some_class` o jeden. Dalej mamy konstrukcję drugiego wskaźnika `intrusive_ptr`: `p2`. Powstaje on jako kopia `p1`; kiedy konstruktor `p2` sprawdzi, że `p1` odnosi się do niepustego wskaźnika, wywoła `intrusive_ptr_add_ref`, zwiększając licznik odwołań do 2. Potem, wraz z końcem zasięgu, kończy się czas życia obu wskaźników. Jako pierwszy usuwany jest `p2`, a jego destruktor wywołuje `intrusive_ptr_release`, zmniejszając licznik odwołań do 1. Następnie usuwany jest wskaźnik `p1` i w ramach jego destruktora znów następuje wywołanie `intrusive_ptr_release`, które tym razem zeruje licznik odwołań; wedle naszej własnej definicji `intrusive_ptr_release` następuje wtedy wywołanie `delete` dla wskaźnika przechowywanego. Wypada zaznaczyć, że implementacja `reference_counter`

nie jest zabezpieczona przed ryzykiem związanym z wielowątkowością i jako taka nie powinna być stosowana w aplikacjach wielowątkowych, chyba że w otoczce odpowiedniej synchronizacji.

Zamiast polegać na uogólnionych implementacjach funkcji `intrusive_ptr_add_ref` i `intrusive_ptr_release`, moglibyśmy operować w nich bezpośrednio na klasie bazowej licznika (tu `reference_counter`). Zaletą takiego podejścia jest to, że nawet jeśli klasy wyprowadzone z `reference_counter` będą definiowane w innych przestrzeniach nazw, wywołania `intrusive_ptr_add_ref` i `intrusive_ptr_release` będą mogły być realizowane przy użyciu reguł ADL (wyszukiwania funkcji kandydujących na bazie typów argumentów). Stosowna zmiana implementacji `reference_counter` nie byłaby wcale skomplikowana:

```
class reference_counter {
   int ref_count_;
   public:
   reference_counter() : ref_count_(0) {}

   virtual ~reference_counter() {}

   friend void intrusive_ptr_add_ref(reference_counter* p) {
      ++p->ref_count_;
   }

   friend void intrusive_ptr_release() {
      if (--p->ref_count_==0)
         delete p;
   }

   protected:
      reference_counter& operator=(const reference_counter&) {
         // Atrapa
         return *this;
      }
   private:
      // Blokada dostępu do konstruktora kopiującego
      reference_counter(const reference_counter&);
};
```

Traktowanie this jako wskaźnika inteligentnego

Wcale nie łatwo wymyślić taki scenariusz, w którym zastosowanie inteligentnego wskaźnika ingerencyjnego byłoby faktycznie *niezbędne*. Większość (jeśli nie wszystkie) problemów da się rozwiązać z użyciem wskaźników nieingerencyjnych. Jest jednak sytuacja, w której *łatwiej* zastosować zliczanie ingerencyjne: kiedy trzeba zwrócić `this` z metody, a zwrócony wskaźnik ma zostać przechwycony do innego inteligentnego wskaźnika. Zwracanie `this` z obiektu typu pozostającego w posiadaniu nieingerencyjnego wskaźnika inteligentnego prowokuje sytuację, w której dwa takie wskaźniki sądzą, że są właścicielami tego samego obiektu, przez co oba będą próbować jego zwolnienia. A wiadomo, że dwukrotne zwolnienie wskaźnika może doprowadzić do

krachu aplikacji. Trzeba jakoś powiadomić ów drugi inteligentny wskaźnik o tym, że zwracany zasób podlega już pod inny wskaźnik inteligentny — tu właśnie świetnie sprawdza się wewnętrzny licznik odwołań. Ponieważ logika wskaźnika `intrusive_ptr` operuje pośrednio na wewnętrznym liczniku odwołań do przechowywanego obiektu, nie ma mowy o ewentualnej niespójności zliczania odwołań — bo dochodzi do prostego zwiększenia licznika.

Przyjrzyjmy się potencjalnie problematycznej sytuacji, gdzie współdzielenie zasobu jest realizowane przy użyciu wskaźników `shared_ptr`. Będzie to zasadniczo powtórzenie jednego z poprzednich przykładów, ilustrującego stosowanie klasy pomocniczej `enable_shared_from_this`.

```
#include "boost/shared_ptr.hpp"

class A;

void do_stuff(boost::shared_ptr<A> p) {
   // …
}

class A {
public:
   void call_do_stuff() {
      shared_ptr<A> p(???);
      do_stuff(p);
   }
};

int main() {
   boost::shared_ptr<A> p(new A());
   p->call_do_stuff();
}
```

Klasa `A` zamierza wywołać ze swojej metody funkcję `do_stuff`, ale `do_stuff` oczekuje przekazania wskaźnika `shared_ptr<A>`, a nie zwykłego wskaźnika klasy `A`, jakim jest `this`. Jak więc utworzyć `shared_ptr` we wnętrzu `A::call_do_stuff`? Spróbujmy przepisać definicję klasy `A`, tak aby poprzez dziedziczenie po `reference_counter` uzdatnić ją do wewnętrznego zliczania odwołań (`intrusive_ptr`), a potem dodać do programu przeciążoną wersję `do_stuff`, przyjmującą argument typu `intrusive_ptr<A>`:

```
#include "boost/intrusive_ptr.hpp"

class A;

void do_stuff(boost::intrusive_ptr<A> p) {
   // …
}

void do_stuff(boost::shared_ptr<A> p) {
   // …
}
```

```
class A : public_reference_counter {
public:
   void call_do_stuff() {
      do_stuff(this);
   }
};

int main() {
   boost::intrusive_ptr<A> p(new A());
   p->call_do_stuff();
}
```

Jak widać, w tej wersji `A::call_do_stuff` możemy przekazać `this` bezpośrednio do funkcji oczekującej wskaźnika `intrusive_ptr<A>`, a to dzięki konstruktorowi konwertującemu klasy `intrusive_ptr`.

Na zakończenie coś specjalnego: teraz `A` nadaje się do stosowania ze wskaźnikami `intrusive_ptr` i możemy napisać kod, który ujałby wskaźnik `intrusive_ptr<A>` we wskaźniku `shared_ptr`, tak aby można było wywołać pierwotną wersję funkcji `do_stuff`, wymagającej argumentu typu `shared_ptr<A>`. Gdybyśmy nie mogli kontrolować kodu źródłowego funkcji `do_stuff`, byłby to faktyczny problem. Rozwiązanie ma postać własnego dealokatora, który wie o potrzebie wywołania `intrusive_ptr_release`. Oto nowa wersja `A::call_do_stuff`:

```
void call_do_stuff() {
   intrusive_ptr_add_ref(this);
   boost::shared_ptr<A> p(this, &intrusive_ptr_release<A>);
   do_stuff(p);
}
```

Doprawdy eleganckie rozwiązanie. Kiedy nie będzie już żadnego wskaźnika `shared_ptr`, wywołana zostanie funkcja usuwająca, czyli `intrusive_ptr_release`, która z kolei zmniejszy wewnętrzny licznik odwołań `A`. Gdyby z kolei funkcje `intrusive_ptr_add_ref` i `intrusive_ptr_release` odwoływały się bezpośrednio do `reference_counter` (jak w drugim wariancie implementacji), obiekt `shared_ptr` konstruowalibyśmy tak:

```
boost::shared_ptr<A> p(this, &intrusive_ptr_release);
```

Obsługa różnych liczników odwołań

Rozważaliśmy wcześniej możliwość obsługiwania różnych liczników odwołań dla różnych typów. Może to być niezbędne przy integrowaniu istniejących klas z różnymi mechanizmami zliczania odwołań (na przykład klas udostępnianych przez osoby trzecie i wdrażających własne wersje liczników odwołań). Dotyczy to również sytuacji różnych wymagań dla operacji zwalniania zasobu, np. przez zastąpienie prostego wywołania `delete` wywołaniem specjalizowanego dealokatora. Wiemy już, że wywołania `intrusive_ptr_add_ref` i `intrusive_ptr_release` są wywołaniami niekwalifikowanymi. Oznacza to, że zasięg typu argumentu (typu wskaźnikowego) ma wpływ

na dobór funkcji kandydujących do wywołania przeciążonego i że te funkcje kandydujące powinny wobec tego być definiowane w tym samym zasięgu, w którym definiowany jest typ, na którym mają operować. Implementacja uogólnionej wersji `intrusive_ptr_add_ref` i `intrusive_ptr_release` w globalnej przestrzeni nazw uniemożliwiałaby utworzenie ich uogólnionych wersji w innych przestrzeniach nazw. Jeśli na przykład dana przestrzeń nazw wymaga specjalnej wersji tych funkcji dla wszystkich definiowanych w niej typów, trzeba by udostępnić specjalizacje albo wersje przeciążone dla wszystkich tych typów. Dlatego właśnie nie jest pożądane definiowanie wersji uogólnionych obu funkcji w globalnej przestrzeni nazw; nie ma za to żadnych przeciwwskazań dla wersji uogólnionych w poszczególnych przestrzeniach nazw właściwych dla typów argumentów.

Z racji sposobu implementowania licznika odwołań (przy użyciu klasy bazowej `reference_counter`) dobrym pomysłem byłoby udostępnienie w globalnej przestrzeni nazw zwykłej funkcji akceptującej argument typu `reference_counter*`. Nie blokowałoby to możliwości przeciążania wersjami uogólnionymi w poszczególnych przestrzeniach nazw. W ramach przykładu rozważmy klasy `another_class` i `derived_class` w przestrzeni nazw `my_namespace`:

```
namespace my_namespace {
   class another_class : public reference_counter P
   public:
      void call_before_destruction() const {
         std::cout <<
            "Gotowy przed usunięciem\n";
      }
   };

   class derived_class : public another_class {};

   template <typename T> void intrusive_ptr_add_ref(T* t) {
      t->add_ref();
   }

   template <typename T> void intrusive_ptr_release(T* t) {
      if (t->release() <= 0) {
         t->call_before_destruction();
         delete t;
      }
   }
}
```

Mamy tu uogólnione wersje `intrusive_ptr_add_ref` i `intrusive_ptr_release`. Musimy więc usunąć wersje uogólnione tych funkcji z globalnej przestrzeni nazw i zastąpić je wersjami zwykłymi (nieszablonowymi), akceptującymi w roli argumentu wskaźnik klasy `reference_counter`. Można by też w ogóle pominąć te funkcje w globalnej przestrzeni nazw, unikając jej zaśmiecania. Implementacja tych funkcji z przestrzeni `my_namespace` zakłada, że dla dwóch klas `my_namespace::another_class` i `my_namespace::derived_class` przy usuwaniu ich egzemplarzy wywoływana jest specjalna funkcja `call_before_destruction`. Dla innych typów, definiowanych w innych

przestrzeniach nazw, można dalej różnicować zachowanie wersji uogólnionych albo korzystać z wersji z globalnej przestrzeni nazw, jeśli takowe tam istnieją. Oto krótki program ilustrujący tę koncepcję:

```
int main() {
   boost::intrusive_ptr<my_namespace::another_class>
      p1(new my_namespace::another_class());
   boost::intrusive_ptr<A>
      p2(new good_class());
   boost::intrusive_ptr<my_namespace::derived_class>
      p3(new my_namespace::derived_class());
}
```

Do konstruktora pierwszego wskaźnika `intrusive_ptr` przekazywany jest nowy egzemplarz klasy `my_namespace::another_class`. Przy rozstrzyganiu wywołania funkcji `intrusive_ptr_add_ref` kompilator odnajduje wersję z przestrzeni nazw `my_namespace`, czyli przestrzeni nazw właściwej dla typu argumentu: `my_namespace::another_class*`. Następuje więc jak najbardziej prawidłowe wywołanie wersji uogólnionej z tejże przestrzeni nazw. Dotyczy to również wywołania `intrusive_ptr_release`. Jako drugi tworzony jest wskaźnik `p2`; do konstruktora przekazywany jest wskaźnik klasy `A`. Klasa ta występuje w globalnej przestrzeni nazw, więc kiedy kompilator szuka najlepszego dopasowania wywołania `intrusive_ptr_add_ref`, znajduje tylko jedną wersję, akceptującą argument typu `reference_counter` (usunęliśmy przecież z globalnej przestrzeni nazw wersje uogólnione `intrusive_ptr_add_ref` i `intrusive_ptr_release`). Ponieważ `A` dziedziczy publicznie po klasie `reference_counter`, zachodzi niejawna konwersja i kompilator może pomyślnie zrealizować wywołanie. Wreszcie przy tworzeniu egzemplarza klasy `my_namespace::derived_class` wykorzystywana jest uogólniona wersja funkcji z przestrzeni nazw `my_namespace`, zupełnie jak poprzednio przy tworzeniu obiektu `my_namespace::another_class`.

Z tego wszystkiego należy zapamiętać, że przy implementowaniu funkcji `intrusive_ptr_add_ref` i `intrusive_ptr_release` powinniśmy je definiować zawsze w tych przestrzeniach nazw, z których pochodzą typy, na których te funkcje mają operować. Ma to również uzasadnienie czysto projektowe: powiązane elementy projektu należy przecież grupować; do tego wspólnota przestrzeni nazw zapewnia każdorazowo wywołanie poprawnej wersji funkcji, niezależnie od istnienia wielu alternatywnych implementacji.

Podsumowanie

W większości sytuacji korzystanie z `boost::intrusive_ptr` nie jest najlepszym pomysłem, bo do wyboru jest implementacja inteligentnych wskaźników zliczających odwołania i nieingerencyjnych, w postaci `boost::shared_ptr`. Wskaźniki nieingerencyjne są elastyczniejsze od ingerencyjnych. Jednak zdarza się, że trzeba zastosować właśnie ingerencyjne zliczanie odwołań, na przykład ze względu na zastaną w kodzie infrastrukturę albo chęć integracji z klasami zewnętrznymi. Wtedy można śmiało korzystać z klasy `intrusive_ptr`, korzystając z podobieństwa zachowania wskaźników tej klasy do pozostałych klas inteligentnych wskaźników biblioteki Boost. Korzystanie

z inteligentnych wskaźników Boost zapewnia spójność interfejsu we wszystkich przypadkach (również wskaźników ingerencyjnych). Dla klas przeznaczonych do użycia ze wskaźnikami `intrusive_ptr` należy przewidzieć licznik odwołań. Sam wskaźnik `intrusive_ptr` manipuluje udostępnionym licznikiem za pośrednictwem niekwalifikowanych wywołań dwóch funkcji: `intrusive_ptr_add_ref` i `intrusive_ptr_release`; powinny one odpowiednio manipulować wartością licznika odwołań i w razie potrzeby zwalniać obiekt wskazywany przez `intrusive_ptr`. W przypadku typów, które posiadają już gotowy własny licznik odwołań, przystosowanie tych typów do stosowania ze wskaźnikami `intrusive_ptr` sprowadza się właśnie do zaimplementowania obu wymienionych funkcji. Niekiedy funkcje te można sparametryzować (uogólnić) i stosować wspólną implementację dla całych grup typów z ingerencyjnym zliczaniem odwołań. Wersje uogólnione najlepiej osadzać w przestrzeniach nazw, z których wywodzą się owe typy.

Wskaźniki `intrusive_ptr` stosuje się:

- kiedy trzeba potraktować `this` jako wskaźnik inteligentny;
- kiedy mamy do czynienia z istniejącym już kodem używającym ingerencyjnego zliczania odwołań;
- kiedy dziedzina aplikacji wymaga koniecznie zrównania rozmiaru wskaźnika inteligentnego z rozmiarem wskaźnika zwykłego.

weak_ptr

Nagłówek: `"boost/weak_ptr.hpp"`

Klasa `weak_ptr` to klasa obserwatorów wskaźników `shared_ptr`. Nie wpływa na prawo własności obiektu pozostającego w posiadaniu `shared_ptr`. Kiedy obserwowany przez `weak_ptr` wskaźnik `shared_ptr` jest zmuszony do zwolnienia pozostającego w jego posiadaniu zasobu, ustawia wskaźnik przechowywany w obserwatorze `weak_ptr` na wartość pustą. Zapobiega to występowaniu w programie wiszących wskaźników `weak_ptr`. Po co nam taki `weak_ptr`? Otóż zdarzają się sytuacje, kiedy pożądana jest możliwość podglądania i nawet używania zasobu współużytkowanego bez przyjmowania go w posiadanie, na przykład przy zrywaniu zależności cyklicznych, przy obserwowaniu wspólnego zasobu, wreszcie właśnie przy unikaniu wiszących wskaźników. Ze wskaźnika `weak_ptr` można skonstruować wskaźnik `shared_ptr`, tym samym przejmując obserwowany zasób w posiadanie (współposiadanie).

Poniżej prezentowana jest częściowa deklaracja szablonu `weak_ptr`, z jego najważniejszymi elementami.

```
namespace boost {

   template<typename T> class weak_ptr {
   public:
      template <typename Y>
```

```
        weak_ptr(const shared_ptr<Y>& r);

      template<class Y>
         weak_ptr(weak_ptr<Y> const & r);
      ~weak_ptr();

      T* get() const;
      bool expired() const;
      shared_ptr<T> lock() const;
   };
}
```

Metody

`template <typename Y> weak_ptr(const shared_ptr<Y>& r);`

Konstruktor tworzący wskaźnik `weak_ptr` na podstawie wskaźnika `shared_ptr`, pod warunkiem że istnieje niejawna konwersja typu `Y*` na typ `T*`. Powstający obiekt `weak_ptr` jest ustawiany do obserwacji zasobu reprezentowanego przez `r`. Wartość licznika odwołań `r` pozostaje jednak bez zmian. Oznacza to, że zasób reprezentowany przez `r` może zostać zwolniony mimo istnienia odnoszącego się doń obiektu `weak_ptr`. Konstruktor nie zrzuca wyjątków.

`template<class Y> weak_ptr(weak_ptr<Y> const & r);`

Konstruktor kopiujący tworzący nowy obiekt `weak_ptr`, obserwujący zasób reprezentowany przez `r`, bez zmiany wartości licznika odwołań do zasobu. Konstruktor nie zrzuca wyjątków.

`~weak_ptr();`

Destruktor `weak_ptr` nie wpływa na wartość licznika odwołań do obserwowanego zasobu. Destruktor nie zrzuca wyjątków.

`bool expired() const;`

Zwraca `true`, jeśli obserwowany zasób „przeterminował się” albo „unieważnił się”, to znaczy został już zwolniony. Jeśli przechowywany w `weak_ptr` wskaźnik jest niepusty, wywołanie `expired` zawsze zwraca `false`. Metoda nie zrzuca wyjątków.

`shared_ptr<T> lock() const;`

Zwraca obiekt `shared_ptr` odnoszący się do zasobu obserwowanego przez wskaźnik `weak_ptr`. Jeśli `weak_ptr` zawiera wskaźnik pusty, wynikowy wskaźnik `shared_ptr` również otrzyma wskaźnik pusty. W przeciwnym przypadku dojdzie do zwykłego zwiększenia licznika odwołań do obserwowanego zasobu. Metoda nie zrzuca wyjątków.

Stosowanie

Zaczniemy od przykładu ilustrującego podstawy stosowania wskaźników `weak_ptr`, z naciskiem na fakt, że nie wpływają one na liczniki odwołań obserwowanych zasobów. Prezentowane w tym dziale przykłady będą z konieczności korzystać ze wskaźników `shared_ptr`, ponieważ `weak_ptr` mało kiedy występuje samodzielnie. Korzystanie ze wskaźników `weak_ptr` wymaga włączenia do kodu źródłowego pliku nagłówkowego `"boost/weak_ptr.hpp"`.

```
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"
#include <iostream>
#include <cassert>

class A {};

int main() {
   boost::weak_ptr<A> w;
   assert(w.expired());
   {
      boost::shared_ptr<A> p(new A());
      assert(p.use_count()==1);
      w=p;
      assert(p.use_count()==w.use_count());
      assert(p.use_count()==1);

      // Utworzenie z weak_ptr wskaźnika shared_ptr
      boost::shared_ptr<A> p2(w);
      assert(p2==p);
   }
   assert(w.expired());
   boost::shared_ptr<A> p3=w.lock();
   assert(!p3);
}
```

Wskaźnik `w` klasy `weak_ptr` jest konstruowany z użyciem konstruktora domyślnego, co oznacza, że początkowo nie obserwuje żadnego zasobu (przechowuje wskaźnik pusty). Aby sprawdzić, czy wskaźnik `weak_ptr` obserwuje istniejący obiekt, należy skorzystać z metody `expired`. Aby rozpocząć obserwację, należy przypisać do `weak_ptr` wskaźnik `shared_ptr`. W omawianym przykładzie po przypisaniu `p` klasy `shared_ptr` do `w` klasy `weak_ptr` sprawdzamy, czy liczniki odwołań obu wskaźników są faktycznie identyczne. Potem konstruujemy z `weak_ptr` wskaźnik `shared_ptr`, co jest jedną z metod pozyskania dostępu do podglądanego zasobu. Gdyby przy konstrukcji wskaźnika `shared_ptr` wskaźnik `weak_ptr` był przeterminowany, konstruktor `shared_ptr` zrzuciłby wyjątek `boost::bad_weak_ptr`. Dalej, kiedy kończy się zasięg wskaźnika `shared_ptr p`, dochodzi do unieważnienia `w`. Dlatego potem, w wyniku wywołania metody `lock` w celu pozyskania wskaźnika `shared_ptr` (to druga metoda pozyskania dostępu do podglądanego obiektu), otrzymujemy pusty wskaźnik `shared_ptr`. W przebiegu całego programu przykładowego wskaźniki `weak_ptr` nie wpływają na wartość licznika odwołań do obiektu wskazywanego.

W przeciwieństwie do pozostałych wskaźników inteligentnych, `weak_ptr` nie daje dostępu do obserwowanego wskaźnika za pośrednictwem wygodnych operatorów `operator*` i `operator->`. Chodzi o to, aby wszystkie operacje podejmowane wobec zasobu za pośrednictwem wskaźnika `weak_ptr` były bezpieczne przez sam fakt jawności. Inaczej łatwo byłoby przypadkiem nawet odwołać się do unieważnionego, pustego wskaźnika — `weak_ptr` nie wpływa na licznik odwołań i fakt prowadzenia obserwacji zasobu nie zabezpiecza przed przedwczesnym zwolnieniem tego zasobu. Dlatego też dostęp do obserwowanego zasobu wymaga utworzenia wskaźnika `shared_ptr` albo przez skorzystanie z konstruktora kopiującego, albo przez wywołanie metody `weak_ptr::lock`. Obie te czynności zwiększają licznik odwołań, co gwarantuje podtrzymanie obecności zasobu.

Odwieczne pytanie

Skoro przy porządkowaniu inteligentnych wskaźników nie ma mowy o porządkowaniu według wartości *obiektów wskazywanych*, ale według wartości samych *wskaźników*, pojawia się pytanie o sposób stosowania takich wskaźników w kontenerach biblioteki standardowej w aspekcie porządkowania elementów kontenera według wartości. Chodzi choćby o przypadek przetworzenia kontenera standardowym algorytmem `std::sort` tak, aby doszło do uporządkowania wartości wskazywanych, a nie wskaźników. Problem porządkowania wskaźników inteligentnych w kontenerach nie różni się wiele od problemu porządkowania kontenera zwykłych wskaźników, ale ten fakt łatwo przeoczyć (pewnie dlatego, że przechowywanie zwykłych wskaźników w kontenerach jest na tyle problematyczne, że zazwyczaj się tego unika). Otóż nie istnieje gotowa infrastruktura porównywania wartości wskazywanych przez inteligentne wskaźniki, ale brak bardzo łatwo uzupełnić. Zwykle polega to na udostępnieniu predykatu wyłuskującego inteligentne wskaźniki; utwórzmy więc uniwersalny predykat ułatwiający stosowanie algorytmów biblioteki standardowej języka C++ z iteratorami kontenerów inteligentnych wskaźników — tu wskaźników `weak_ptr`.

```
#include <functional>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

template <typename Func, typename T>
   struct weak_ptr_unary_t :
      public std::unary_function<boost::weak_ptr<T>, bool> {
      T t_;
      Func func_;

   weak_ptr_unary_t(const Func& func, const T& t)
      : t_(t), func_(func) {}

   bool operator()(boost::weak_ptr<T> arg) const {
      boost::shared_ptr<T> sp=arg.lock();
      if (!sp) {
         return false;
      }
      return func_(*sp, t_);
   }
```

```
};

template <typename Func, typename T> weak_ptr_unary_t<Func,T>
   weak_ptr_unary(const Func& func, const T& value) {
      return weak_ptr_unary_t<Func,T>(func, value);
}
```

Obiekt funkcyjny klasy `weak_ptr_unary_t` jest parametryzowany funkcją do wywołania i typem jej argumentu. Fakt, że funkcja przeznaczona do wywołania jest przechowywana w obiekcie funkcyjnym, ułatwia stosowanie obiektu, o czym wkrótce się przekonamy. Aby predykat nadawał się do stosowania z adapterami, klasa `weak_ptr_unary_t` jest wyprowadzana jako pochodna klasy `std::unary_function`, dzięki czemu nasza klasa obiektu funkcyjnego posiada wszystkie definicje typów wymagane przez adaptery predykatów biblioteki standardowej. Cała mokra robota jest wykonywana w przeciążonym dla klasy operatorze wywołania funkcji, gdzie najpierw następuje konstrukcja wskaźnika `shared_ptr` na bazie `weak_ptr`. To konieczne w celu podtrzymania obecności zasobu na czas wywołania właściwej funkcji predykatu. Potem następuje wywołanie funkcji (obiektu funkcyjnego) z przekazaniem argumentu (wyłuskanego, a więc w postaci referencji wskazywanego zasobu) i wartości zapamiętanej w konstruktorze `weak_ptr_unary_t`. Nasz prosty obiekt funkcyjny nadaje się teraz do stosowania z wszelkimi algorytmami. Dla wygody zdefiniowaliśmy również funkcję pomocniczą `weak_ptr_unary`, dedukującą typy argumentów i zwracającą odpowiedni dla nich obiekt funkcyjny[14]. Zobaczmy, jak całość sprawdzi się w praktyce.

```
#include <iostream>
#include <string>

#include <vector>
#include <algorithm>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

int main() {
   using std::string;
   using std::vector;
   using boost::shared_ptr;
   using boost::weak_ptr;

   vector<weak_ptr<string> > vec;

   shared_ptr<string> sp1(new string("Przykład"));
   shared_ptr<string> sp2(new string("użycia"));
   shared_ptr<string> sp3(new string("inteligentnych wskaźników z predykatami"));
   vec.push_back(weak_ptr<string>(sp1));
   vec.push_back(weak_ptr<string>(sp2));
   vec.push_back(weak_ptr<string>(sp3));

   vector<weak_ptr<string> >::iterator
      it = std::find_if(vec.begin(),vec.end(),
```

[14] Aby ten typ przystosować do całkiem uniwersalnego stosowania, trzeba by jeszcze mnóstwa kodowania — *przyp. aut.*

```
        weak_ptr_unary(std::equal_to<string>(), string("użycia")));

   if (it!=vec.end()) {
      shared_ptr<string> sp(*++it);
      std::cout << *sp << '\n';
   }
}
```

Mamy tu przykład tworzenia kontenera `vector` zawierający wskaźniki `weak_ptr`. Najciekawszym tutaj wierszem kodu jest ten długi, tworzący obiekt funkcyjny `weak_ptr_unary_t` na użytek algorytmu `find_if`:

```
vector<weak_ptr<string> >::iterator
   it = std::find_if(vec.begin(),vec.end(),
      weak_ptr_unary(std::equal_to<string>(), string("użycia")));
```

Obiekt funkcyjny jest tu tworzony przez przekazanie do funkcji pomocniczej `weak_ptr_unary` na podstawie innego obiektu funkcyjnego, konkretnie `std::equal_to`, wraz z ciągiem, który ma pełnić rolę wzorca przy porównywaniu. Ponieważ typ `weak_ptr_unary_t` jest zgodny z adapterami (bo dziedziczy po `std::unary_function`), moglibyśmy zaangażować go do kompozycji obiektu funkcyjnego dowolnego typu. Moglibyśmy na przykład szukać pierwszego ciągu, który nie pasuje do ciągu `"użycia"`:

```
vector<weak_ptr<string> >::iterator
   it = std::find_if(vec.begin(),vec.end(),
      std::not1(
         weak_ptr_unary(std::equal_to<string>(), string("użycia"))));
```

Inteligentne wskaźniki biblioteki Boost zostały solidnie przygotowane do współpracy z komponentami biblioteki standardowej. Ułatwia to tworzenie użytecznych i prostych w użyciu komponentów korzystających z takich wskaźników. Narzędzia w rodzaju pomocniczej funkcji `weak_ptr_unary` nie są potrzebne zbyt często, zwłaszcza w obliczu dostępności bibliotek uogólnionych szablonów wiązania (ang. *binders*), znacznie bardziej uniwersalnych niż `weak_ptr_unary`[15]. Biblioteki te są zazwyczaj konstruowane z uwzględnieniem semantyki inteligentnych wskaźników, przez co korzysta się z nich w sposób zupełnie przezroczysty.

Dwa sposoby tworzenia wskaźników shared_ptr ze wskaźników weak_ptr

Wiemy już, że kiedy mamy wskaźnik `weak_ptr` obserwujący pewien zasób, niekiedy zachodzi potrzeba skorzystania z tego zasobu. W tym celu trzeba skonwertować wskaźnik `weak_ptr` na `shared_ptr`, bo `weak_ptr` sam w sobie nie pozwala na korzystanie z obserwowanego zasobu (a przynajmniej nie gwarantuje podtrzymania obecności

[15] Przykładem takiej biblioteki jest Boost.Bind — *przyp. aut.*

zasobu na czas użycia). Wskaźnik shared_ptr można utworzyć z weak_ptr na dwa sposoby: przez przekazanie wskaźnika weak_ptr do konstruktora nowego egzemplarza shared_ptr albo przez wywołanie na rzecz wskaźnika weak_ptr metody lock, zwracającej gotowy wskaźnik shared_ptr. Wybór metody należy uzależnić od tego, czy pusty wskaźnik zaszyty w weak_ptr ma być traktowany jako niepoprawny. Konstruktor shared_ptr akceptujący argument typu weak_ptr dla wskaźnika pustego zrzuci wyjątek bad_weak_ptr. Należałoby więc z niego korzystać tylko wtedy, kiedy brak inicjalizacji wskaźnika weak_ptr ma być uznawany za błąd. Z kolei metoda lock zwróci dla pustego wskaźnika weak_ptr pusty wskaźnik shared_ptr i korzysta się z niej wtedy, kiedy wskazanie puste jest dopuszczalne, na przykład do wykorzystania w teście. Co więcej, metodę lock stosuje się typowo z równoczesną inicjalizacją i testowaniem zasobu, jak tu:

```
#include <iostream>
#include <string>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

int main() {
   boost::shared_ptr<std::string> sp(new std::string("Pewien zasób"));
   boost::weak_ptr<std::string> wp(sp);
   // …
   if (boost::shared_ptr<std::string> p=wp.lock())
      std::cout << "Mamy go: " << *p << '\n';
   else
      std::cout << "Ech, wskaźnik shared_ptr jest pusty\n";
}
```

Jak widać, wskaźnik shared_ptr jest inicjalizowany wartością zwracaną przez metodę lock wywołaną na rzecz wskaźnika wp (weak_ptr). Otrzymany wskaźnik jest testowany pod kątem zawierania wskazania pustego. Ponieważ skonstruowany tu shared_ptr jest dostępny jedynie w obrębie ograniczonego zasięgu, nie ma możliwości przypadkowego nadużycia go poza zasięgiem. Inny scenariusz mielibyśmy w sytuacji, w której wskaźnik weak_ptr powinien być niepusty. Wtedy łatwo przeoczyć testowanie wskaźnika shared_ptr i aby zabezpieczyć się przed niezdefiniowanym zachowaniem (w wyniku wyłuskania pustego wskaźnika shared_ptr), najlepiej korzystać z konwersji przy pomocy konstruktora konwertującego weak_ptr na shared_ptr, który dla pustego wskaźnika zrzuci wyjątek:

```
#include <iostream>
#include <string>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

void access_the_resource(boost::weak_ptr<std::string> wp) {
   boost::shared_ptr<std::string> sp(wp);
   std::cout << *sp << '\n';
}
```

```
int main() {
   boost::shared_ptr<std::string> sp(new std::string("Pewien zasób"));
   boost::weak_ptr<std::string> wp(sp);
   // …
   access_the_resource(wp);
}
```

Funkcja access_the_resource konstruuje wskaźnik shared_ptr z przekazanego wskaźnika weak_ptr. Nie musi w ogóle testować utworzonego wskaźnika, bo gdyby przekazany w wywołaniu konstruktora argument weak_ptr reprezentował wskaźnik pusty, konstruktor sam zrzuciłby wyjątek typu bad_weak_ptr, co wymusiłoby natychmiastowe przekazanie sterowania poza zasięg funkcji i uniemożliwiło realizację odwołania. Wyjątek należałoby oczywiście odpowiednio przechwycić i obsłużyć tam, gdzie to możliwe i zasadne. Takie zabezpieczenie sprawdza się lepiej niż jawne testowanie stanu wskaźnika shared_ptr i ewentualne zwracanie sterowania do wywołującego. Niniejszym poznaliśmy dwie metody pozyskiwania wskaźnika share_ptr na podstawie weak_ptr.

Podsumowanie

Klasa weak_ptr to ostatni element układanki tworzącej obraz implementacji inteligentnych wskaźników w bibliotece Boost. Abstrakcja reprezentowana przez weak_ptr znakomicie uzupełnia shared_ptr, pozwalając choćby na przerywanie zależności cyklicznych. Klasa weak_ptr rozwiązuje też powszechny problem „wiszących wskaźników”. Przy użytkowaniu wspólnego zasobu zdarza się, że niektórzy jego użytkownicy nie powinni brać odpowiedzialności za zarządzanie czasem jego życia. Zastosowanie wtedy wskaźników zwykłych nie jest rozwiązaniem, bo po usunięciu ostatniego wskaźnika shared_ptr wspólnego zasobu zostanie on zwolniony. Zwykły wskaźnik nie daje możliwości stwierdzenia, czy wskazywany zasób wciąż istnieje, czy może już nie. W tym drugim przypadku próba skorzystania z zasobu może mieć katastrofalne skutki. Tymczasem obserwacja zasobu za pośrednictwem wskaźnika weak_ptr gwarantuje przekazanie do wskaźnika informacji o zwolnieniu zasobu i unieważnienie wskaźnika, co skutecznie blokuje ryzykowne odwołania. Mamy tu do czynienia ze specjalnym wcieleniem wzorca projektowego Observer: kiedy zasób jest zwalniany, użytkownicy, którzy wyrazili zainteresowanie istnieniem zasobu, są o tym fakcie informowani.

Wskaźniki weak_ptr stosuje się:

- do zrywania cyklicznych zależności,
- do współużytkowania zasobu bez przejmowania odpowiedzialności za zarządzanie nim,
- do eliminowania ryzykownych operacji na wiszących wskaźnikach.

Smart_ptr — podsumowanie

Niniejszy rozdział prezentował implementacje inteligentnych wskaźników z biblioteki Boost — tak dla społeczności programistów C++ znaczące, że nie sposób ich znaczenia przecenić. Aby biblioteka inteligentnych wskaźników skutecznie spełniała swoje zadania, powinna uwzględniać i poprawnie obsługiwać cały szereg czynników. Czytelnik z pewnością miał okazję korzystać z mnóstwa inteligentnych wskaźników, być może też niektóre implementacje sam tworzył lub współtworzył; ma więc świadomość rozmiaru wysiłku niezbędnego do dopięcia implementacji na ostatni guzik. Nie wszystkie inteligentne wskaźniki są tak bystre, jak by się chciało, co tylko zwiększa wartość biblioteki Boost.Smart_ptr, która dowiodła swojej sprawności na tylu polach.

Inteligentne wskaźniki z biblioteki Boost, jako tak nieodzowny komponent inżynierii oprogramowania, w oczywisty sposób cieszyły się wielkim zainteresowaniem programistów i testerów. Przez to trudno należycie docenić wszystkich, którzy przyczynili się do sukcesu tej implementacji. Na ostateczny jej kształt wpływało mnóstwo wnikliwych opinii i cennych wskazówek tylu osób, że nie sposób ich tu wymienić. Nie można jednak pominąć kilku najważniejszych osób, których wysiłek był dla tego sukcesu decydujący:

- Grega Colvina, ojca wskaźników `auto_ptr`, który zaproponował implementację `counted_ptr`, która z czasem przerodziła się w dzisiejszą klasę `shared_ptr`;
- Bemana Dawesa, który wzniecił na nowo dyskusję o inteligentnych wskaźnikach i zaproponował uwzględnienie pierwotnej ich semantyki, wedle pomysłu Grega Colvina;
- Petera Dimova, który przemodelował klasy inteligentnych wskaźników, uzdatniając je do środowisk wielowątkowych, i dodał klasy `intrusive_ptr` i `weak_ptr`.

Co ciekawe, koncepcja inteligentnych wskaźników, choć tak dobrze rozpoznana, wciąż ewoluuje. Można się spodziewać postępu w dziedzinie inteligentnych wskaźników, a może już inteligentnych zasobów, ale i *współczesne* implementacje są implementacjami wysokiej jakości. Powszechność użycia biblioteki Smart_ptr wynika właśnie z tej jakości i dostosowania do powszechnych potrzeb — wygrywają najlepiej przystosowani. Trudno o lepsze narzędzia niż te z biblioteki Boost, stale goszczące w kuchniach znakomitych zespołów programistycznych, z których korzystam również osobiście (i polecam Czytelnikom!). A skoro zostały przyjęte do raportu Library Technical Report, niedługo wejdą zapewne (choćby częściowo) do specyfikacji biblioteki standardowej języka C++.