7

Inteligente wskaźniki

Hekatomby kodu i rzeki atramentu poświęcono inteligentnym wskaźnikom (ang. *smart pointers*). Ten najpopularniejszy, najsilniejszy i najbardziej frapujący z idiomów C++ jest szczególnie interesujący ze względu na połączenie w nim wielu zagadnień związanych tak ze składnią, jak i z semantyką. W niniejszym rozdziale omawiamy inteligentne wskaźniki od najprostszych do najbardziej złożonych. Przyglądamy się też błędom, jakie mogą zakraść się do ich implementacji, od najbardziej oczywistych do bardzo subtelnych.

Inteligentne wskaźniki to obiekty, które (implementując `operator->` oraz unarny `operator*`) udają, że są zwykłymi wskaźnikami. Poza naśladowaniem składni zwykłych wskaźników inteligentne wskaźniki zazwyczaj wykonują jakieś dodatkowe pożyteczne zadania, np. zarządzają pamięcią albo blokują wątki, tym samym uwalniając kod aplikacji od mozolnego zarządzania obiektem wskazywanym przez wskaźnik.

W tym rozdziale nie tylko omawiamy inteligentne wskaźniki, ale także ich implementację w postaci szablonu klasowego `SmartPtr`. Konstrukcja szablonu opiera się na wytycznych (zob. rozdz. 1), w wyniku czego otrzymujemy inteligentny wskaźnik charakteryzujący się takimi poziomami bezpieczeństwa, wydajności i poręczności, jakich żąda użytkownik.

Po lekturze tego rozdziału osiągniesz biegłość w następujących zagadnieniach:

- zalety i wady inteligentnych wskaźników,
- strategie zarządzania własnością,
- konwersje niejawne,
- porównania,
- zagadnienia wielowątkowości.

Każde zagadnienie implementacyjne omówimy w innym podrozdziale. Wyjaśnimy, dlaczego przy projektowaniu szablonu przyjęto konkretne rozwiązania. Pokażemy, w jaki sposób szablon `SmartPtr` można rozszerzać i dostosowywać do własnych potrzeb.

7.1. 1001 drobiazgów na temat inteligentnych wskaźników

Czymże więc jest ten inteligentny wskaźnik? Jest to klasa naśladująca zwykły wskaźnik składniowo, a w pewnych aspektach także semantycznie. Inteligentne wskaźniki do obiektów różnych typów mają zazwyczaj wiele wspólnego kodu, stąd prawie wszystkie dobre ich implementacje są sparametryzowane typem wskazywanego obiektu, jak w następującym kodzie:

```
template <class T>
class SmartPtr
{
public:
  explicit SmartPtr(T* pointee) : pointee_(pointee);
  SmartPtr& operator=(const SmartPtr& other);
  ~SmartPtr();
  T& operator*() const
  {
       ...
       return *pointee_;
  }
  T* operator->() const
  {
       ...
       return pointee_;
  }
private:
  T* pointee_;
  ...
};
```

Klasa `SmartPtr<T>` zawiera wskaźnik do `T` przechowywany w zmiennej składowej `pointee_`. Tak czyni większość inteligentnych wskaźników. Jednak czasami mogą one przechowywać jakieś inne informacje o danych i wyliczać wskaźnik „w biegu”.

Dzięki operatorom `operator->` i `operator*` inteligentny wskaźnik ma składnię podobną do składni zwykłych wskaźników. Możemy na przykład napisać

```
class Widget
{
public:
  void Fun();
};

SmartPtr<Widget> sp(new Widget);
sp->Fun();
(*sp).Fun();
```

Jedynie definicja zmiennej `sp` ujawnia, że nie mamy do czynienia ze zwykłym wskaźnikiem. Możemy więc bez dużych zmian w kodzie aplikacji zastąpić zwykłe wskaźniki inteligentnymi wskaźnikami. Dzięki temu można tanim kosztem osiągać dodatkowe korzyści. Jest to bardzo ważne przy wprowadzaniu inteligentnych wskaźników do kodu dużych istniejących aplikacji. Jak się jednak przekonamy, nic za darmo.

7.2. Zysk

Ale co właściwie zyskujemy – zapytacie – po wprowadzeniu do programu inteligentnych wskaźników? Odpowiedź jest prosta. Inteligentne wskaźniki mają semantykę wartości, a zwykłe nie.

Obiekt o semantyce wartości to taki obiekt, który można *kopiować* oraz do którego można *przypisywać*. O tego typu obiektach można myśleć jak o wartościach, np. typu `int`. Wartości typu `int` można swobodnie tworzyć, kopiować i zmieniać. Wskaźnik użyty do przeglądania elementów bufora też ma semantykę wartości – ustawiamy go na początek bufora i przesuwamy na kolejne elementy, aż do końca bufora. Po drodze możemy skopiować jego wartość do innych zmiennych i w nich przechowywać.

Wskaźniki przechowujące wartości zwrócone przez `new`, to jednak zupełnie inna historia. Po wykonaniu

```
Widget* p = new Widget;
```

zmienna `p` nie tylko wskazuje na początek obszaru pamięci zajętego przez nowy obiekt klasy `Widget`, ale także staje się jego *właścicielem*. Gdzieś dalej trzeba wykonać `delete p`, by zniszczyć obiekt `Widget` i zwolnić zajmowaną przezeń pamięć. Jeśli w następnym wierszu napiszemy

```
p = 0; // Przypisz do p coś innego
```

to tracimy własność obiektu, na który uprzednio wskazywał wskaźnik `p`, i nie mamy szans na jej odzyskanie. Nastąpił wyciek zasobów, a to nigdy nie jest powód do radości.

Co więcej, gdy skopiujemy wartość `p` do innej zmiennej, to kompilator nie skopiuje wskazywanego obiektu automatycznie. Otrzymujemy po prostu dwa gołe wskaźniki wskazujące na ten sam obiekt i trzeba pilnować się jeszcze bardziej, bo dwukrotne usunięcie obiektu skutkuje o wiele poważniejszą katastrofą niż nieusunięcie go. Wskaźniki do obiektów dynamicznych *nie mają* semantyki wartości[1] – nie można ich swobodnie kopiować ani przypisywać.

[1] To stwierdzenie jest mylące. Wskaźniki w C++ mają semantykę wartości – można je kopiować i przypisywać. Problem polega na tym, że pewne zmienne wskaźnikowe są w programie uznawane za właścicieli wskazywanych obiektów. W wypadku surowych wskaźników ta właściwość nie jest w żaden czytelny dla kompilatora sposób reprezentowana w kodzie, a stwierdzenie, czy dany wskaźnik jest właścicielem, pozostaje w gestii programisty. Kopiowanie surowych wskaźników jest zawsze płytkie, stąd, z punktu widzenia poprawności programu, surowych wskaźników-właścicieli nie należy kopiować do surowych wskaźników-właścicieli. Surowe wskaźniki, którym programista nadał status właścicieli, nie mają semantyki wartości – nie można ich swobodnie kopiować ani przypisywać bez pogwałcenia logicznej spójności programu. (Przyp. tłum.)

Pomocne mogą okazać się inteligentne wskaźniki. Większość z nich poza udawaniem zwykłych wskaźników, może także *zarządzać własnością* (ang. *ownership*). Inteligentne wskaźniki potrafią zorientować się, w jaki sposób własność się zmienia, a ich destruktory zwalniają pamięć według określonych strategii. Wiele z nich przechowuje informacje umożliwiające im całkowite przejęcie inicjatywy w zwalnianiu wskazywanych obiektów.

Inteligentne wskaźniki mogą zarządzać własnością na wiele sposobów, odpowiednich dla konkretnego zastosowania. Niektóre przenoszą własność automatycznie: przy kopiowaniu wskaźnik źródłowy staje się wskaźnikiem pustym (ang. *null pointer*), a wskaźnik docelowy wskazuje na obiekt i jest jego (nowym) właścicielem. Tak zachowuje się zdefiniowany przez standard C++ wskaźnik `std::auto_ptr`. Inne inteligentne wskaźniki implementują liczniki referencji, tzn. śledzą liczbę wszystkich wskaźników wskazujących na dany obiekt. Kiedy liczba ta spada do zera, niszczą obiekt (za pomocą `delete`)[1]. Jeszcze inne kopiują obiekt razem ze wskaźnikiem[2].

Krótko mówiąc, w świecie inteligentnych wskaźników własność to rzecz święta. Dzięki zarządzaniu własnością zapewniają one poprawność oraz mają prawdziwą semantykę wartości. Własność ma wiele wspólnego z tworzeniem, kopiowaniem i niszczeniem wskaźnika, łatwo się zatem zorientować, że są to najistotniejsze funkcjonalności inteligentnych wskaźników.

W następnych podrozdziałach omawiamy różne aspekty architektury i implementacji inteligentnych wskaźników. Chcemy upodobnić je do zwykłych wskaźników, na ile się da, ale nie bardziej. Jest w tym sprzeczność: jeśli inteligentne wskaźniki będą zachowywać się *dokładnie* tak, jak zwykłe, to *będą* zwykłymi wskaźnikami.

Przy implementacji zgodności między inteligentnymi a zwykłymi wskaźnikami zarysowuje się cienka granica między zakresem zgodności a drogą do chaosu. Przekonasz się, że dodanie pozornie pożytecznych cech może narazić użytkownika na kosztowne ryzyko. Sztuka projektowania dobrych inteligentnych wskaźników polega na umiejętnym wyważeniu zbioru ich właściwości.

7.3. Dowiązanie

Zacznijmy od podstawowego pytania. Czy zawsze dowiązanie do wskazywanego obiektu musi być reprezentowane za pomocą `T*`? Jeśli nie, to jak? Uprawiając programowanie uogólnione, zawsze należy zadawać sobie tego typu pytania. Każdy typ zaszyty w kodzie generycznym zmniejsza jego uniwersalność. Typy zaszyte są dla kodu generycznego tym, czym stałe magiczne[3] (ang. *magic constants*) dla zwykłego kodu.

Czasami warto dopuścić inne wskaźniki, np. ze względu na niestandardowe modyfikatory typów wskaźnikowych. W procesorach szesnastobitowych (Intel 80x86) wskaźniki miewały modyfikatory `__near`, `__far` i `__huge`. Inne architektury o pamięci segmentowanej korzystają z analogicznych modyfikatorów.

[1] Tak zachowuje się implementacja wskaźnika `boost::shared_ptr` z biblioteki Boost. (Przyp. tłum.)

[2] Dokładny opis implementacji takiego wskaźnika zamieszczono w serii artykułów B. Moo i A. Koeniga *C++ Made Easier*, publikowanej z przerwami w *C/C++ Users Journal* począwszy od sierpnia 2002 roku. (Przyp. tłum.)

[3] Stałe magiczne to wszystkie nienazwane stałe różne od 0 lub 1, występujące w kodzie (zob. Stephen C. Dewhurst, *C++ Gotchas*, AWL 2003, punkt 2). (Przyp. tłum.)

Warto też dopuścić nawarstwianie (ang. *layering*) inteligentnych wskaźników. Co zrobić, gdy chcemy rozszerzyć funkcjonalność występującego już w kodzie inteligentnego wskaźnika `LegacySmartPtr<T>`? Dziedziczyć po nim? To ryzykowna decyzja. Lepiej zapakować stary inteligentny wskaźnik w nowy. Jest to możliwe, ponieważ stary wskaźnik ma wskaźnikową składnię. Z punktu widzenia nowego wskaźnika dowiązanie do obiektu byłoby realizowane za pomocą typu `LegacySmartPtr<T>`, a nie `T*`.

Istnieją interesujące zastosowania nawarstwiania inteligentnych wskaźników, głównie dzięki mechanizmom działania operatora `operator->`. Użycie tego operatora do wartości typu, który nie jest zwykłym wskaźnikiem, powoduje ciekawy ciąg zdarzeń. Kompilator wywołuje `operator->` zdefiniowany przez użytkownika dla tego typu, po czym wraca do punktu wyjścia, stosując `operator->` do uzyskanego wyniku. Cała procedura powtarza się aż do uzyskania zwykłego wskaźnika, dla którego ostatnie użycie operatora `operator->` oznacza po prostu dostęp do składowej. Wynika stąd, że `operator->` inteligentnego wskaźnika wcale nie musi zwracać zwykłego wskaźnika. Może zwrócić inny obiekt, dla którego zdefiniowano `operator->`. Pozostaje to bez wpływu na konteksty składniowe, w jakich można używać inteligentnego wskaźnika.

Prowadzi to do bardzo ciekawego idiomu: funkcji wykonywanych *przed* i *po* dostępie do składowej (Stroustrup 2000). Gdy `operator->` zwraca przez wartość obiekt typu `PointerType`, to ciąg wywołań jest następujący:

1. konstruktor `PointerType`;
2. `PointerType::operator->`; zapewne zwraca wskaźnik do obiektu typu `PointeeType`;
3. dostęp do składowej obiektu typu `PointeeType` – zapewne wywołanie funkcji składowej;
4. destruktor `PointerType`.

Krótko mówiąc, mamy elegancki sposób zaimplementowania wywołań funkcji z blokowaniem. Ten idiom jest powszechnie stosowany w wielowątkowości i blokowanym dostępie do zasobów. Konstruktor klasy `PointerType` może blokować zasób, po czym następuje rzeczywisty dostęp i odblokowanie zasobu w destruktorze klasy `PointerType`.

Uogólnienia na tym się nie kończą. Czasami składniowe cechy „wskaźnika" bledną w porównaniu z zaimplementowanymi przez inteligentny wskaźnik silnymi technikami zarządzania własnością. Wynika stąd, że czasami inteligentne wskaźniki mogą nawet porzucić wskaźnikową składnię. Typ, dla którego nie zdefiniowano operatorów `operator->` i `operator*`, nie spełnia definicji inteligentnego wskaźnika, ale są typy, które mimo wszystko należy tak traktować.

Spójrzmy na interfejsy API (ang. *Application Programming Interfaces*) i programy aplikacyjne. Wiele systemów operacyjnych udostępnia zasoby wewnętrzne, np. okna, muteksy czy urządzenia, za pomocą uchwytów (ang. *handles*). Uchwyty to wskaźniki, które z premedytacją zamaskowano, m.in. po to, by użytkownicy nie mieli dostępu do krytycznych zasobów systemu operacyjnego. Zazwyczaj uchwyty są całkowitoliczbowymi indeksami do ukrytych w systemie tablic wskaźników. Tablice są dodatkowym poziomem pośredniości chroniącym wnętrze systemu przed programistami aplikacji. Uchwyty nie implementują operatora `operator->`, lecz mimo wszystko przypominają wskaźniki w aspektach semantycznych i w sposobie, w jaki należy się z nimi obchodzić.

Dla „inteligentnych uchwytów" nie ma sensu definiowanie operatorów `operator->` i `operator*`, ale rozsądne wydaje się użycie charakterystycznych dla inteligentnych wskaźników technik zarządzania własnością zasobów.

By pokazać różnorodność inteligentnych wskaźników wyróżnimy trzy związane z nimi typy:

- *Typ dowiązania* (ang. *storage type*). Jest to typ składowej `pointee_`. „Domyślnie", w popularnych inteligentnych wskaźnikach, jest to zwykły wskaźnik.
- *Typ wskaźnikowy* (ang. *pointer type*). Jest to typ zwracany przez operator `operator->`. Może on się różnić od typu dowiązania, jeśli zamiast wskaźnika chcemy zwracać pełnomocnika (ang. *proxy*). (W dalszej części tego rozdziału pokazujemy przykład zastosowania pełnomocnika).
- *Typ referencyjny* (ang. *reference type*). Jest to typ zwracany przez `operator*`.

Będziemy chcieli zaprojektować szablon `SmartPtr` na podstawie podanych definicji. Z tego powodu wymienione typy wyabstrahujemy w postaci wytycznej `Storage`.

Reasumując, inteligentne wskaźniki mogą i powinny być sparametryzowane typem dowiązania. W tym celu projekt szablonu `SmartPtr` za pomocą wytycznej `Storage` wyodrębnia: typ dowiązania do wskazywanego obiektu, typ wskaźnikowy oraz typ referencyjny. Dla pewnych konkretyzacji szablonu `SmartPtr` niektóre z tych typów mogą nie być określone. Niekiedy (uchwyty) klasa wytycznej może zablokować operator `operator->` i/lub `operator*`.

7.4. Funkcje składowe inteligentnych wskaźników

Wiele istniejących implementacji inteligentnych wskaźników dopuszcza operowanie na nich za pomocą funkcji składowych, np. `Get` (umożliwiającej dostęp do wskazywanego obiektu), `Set` (do modyfikacji) czy `Release` (do przejmowania własności). Jest to oczywisty i naturalny sposób zakapsułkowania funkcjonalności inteligentnego wskaźnika.

Doświadczenie jednak dowodzi, że funkcje składowe zastosowane do inteligentnych wskaźników nie są odpowiednim rozwiązaniem, a to dlatego że bardzo łatwo pomylić funkcje składowe wskaźnika i obiektu *wskazywanego*.

Przypuśćmy, że mamy klasę `Printer` (ang. *drukarka*) o funkcjach składowych `Acquire` i `Release`. Funkcja `Acquire` przejmuje dostęp do drukarki, na skutek czego inne aplikacje nie mogą na niej drukować. Jeśli korzystamy z inteligentnego wskaźnika do obiektu typu `Printer`, nietrudno zauważyć niebezpieczne podobieństwo składniowe, przy jednoczesnym zupełnym braku podobieństwa semantycznego.

```
SmartPtr<Printer> spRes = ...;
spRes->Acquire(); // Przejmij dostęp do drukarki
... drukuj ...
spRes->Release(); // Oddaj dostęp do drukarki
spRes.Release();  // Oddaj własność obiektu drukarki
```

Użytkownik szablonu `SmartPtr` wchodzi do dwóch zupełnie odmiennych światów: świata funkcji składowych obiektów wskazywanych oraz świata funkcji składowych inteligent-

nych wskaźników. Granica między nimi jest cienka na tyle, na ile jest nieznaczna różnica między kropką a strzałką.

C++ zmusza nas jednak do zauważania pewnych niewielkich różnic składniowych. Programista programujący w Pascalu a uczący się C++ może uważać, że konieczność rozróżniania między & i && to bzdura, podczas gdy programista korzystający z C++ jest przyzwyczajony rozróżniać takie niuanse.

Podczas stosowania inteligentnych wskaźników okazuje się jednak, że brakuje nam odpowiednich przyzwyczajeń. Zwykłe wskaźniki nie mają funkcji składowych, zatem programiści nie muszą rozróżniać między kropką i strzałką w tym kontekście. Tutaj bardzo pomaga im kompilator. Jeśli po zwykłym wskaźniku użyje się kropki, kompilator zgłosi błąd. Łatwo sobie zatem wyobrazić, a potwierdza to doświadczenie, że nawet programistów doświadczonych w C++ często może zmylić fakt, że zarówno `sp.Release()`, jak i `sp->Release()` kompilują się bezbłędnie, ale mają przy tym bardzo różne skutki. Rozwiązanie jest proste: inteligentne wskaźniki nie powinny korzystać z funkcji składowych. W szablonie `SmartPtr` używa się wyłącznie funkcji wolnych, które implementuje się jako funkcje zaprzyjaźnione z konkretyzacjami szablonu.

Funkcje przeciążone mogą być równie mylące co funkcje składowe inteligentnych wskaźników, rysuje się tu jednak istotna różnica. Przeciążanie występuje w C++ od dawna. Jest ważną częścią języka i jest rutynowo stosowane tak w bibliotekach, jak i w programach aplikacyjnych. Oznacza to, że programiści piszący w C++ przyzwyczaili się zwracać uwagę na różnice w wywołaniach funkcji, np. `Release(*sp)` i `Release(sp)`.

Jedyne funkcje, które siłą rzeczy muszą pozostać składowymi szablonu `SmartPtr`, to konstruktory, destruktor, `operator=`, `operator->` i unarny `operator*`. Wszystkie inne operacje na obiektach `SmartPtr` implementuje się jako funkcje wolne.

Dla jasności, `SmartPtr` nie ma żadnych nazwanych funkcji składowych. Jedynymi funkcjami umożliwiającymi dostęp do wskazywanego obiektu są funkcje wolne `GetImpl`, `GetImplRef`, `Reset` i `Release`.

```
template <class T> T* GetImpl(SmartPtr<T>& sp);
template <class T> T*&GetImplRef(SmartPtr<T>& sp);
template <class T> void Reset(SmartPtr<T>& sp, T* source);
template <class T> void Release(SmartPtr<T>& sp, T*& destination);
```

- `GetImpl` zwraca wskaźnik do obiektu wskazywanego przez `SmartPtr`.
- `GetImplRef` zwraca *referencję* do wskaźnika przechowywanego przez obiekt klasy `SmartPtr`. Umożliwia bezpośrednią manipulacje wskaźnikiem reprezentującym dowiązanie, zatem posługiwanie się nią wymaga największej ostrożności.
- `Reset` przypisuje nową wartość do wskaźnika reprezentującego dowiązanie, zwalniając zasób reprezentowany przez poprzednią wartość.
- `Release` oddaje własność, przenosząc tym samym na użytkownika odpowiedzialność związaną z zarządzaniem zasobem.

Deklaracje tych funkcji w bibliotece Loki są nieco bardziej dopracowane. Nie zakłada się w nich, że typem dowiązania jest `T*`. Jak powiedziano w podrozdziale 7.3, typ dowiązania jest określony wytyczną `Storage`. Zazwyczaj jest to zwykły wskaźnik, poza rzadkimi przypadkami uchwytów lub jakichś skomplikowanych typów[1].

[1] Mogą to być na przykład inne inteligentne wskaźniki. (Przyp. tłum.)

7.5. Strategie zarządzania własnością

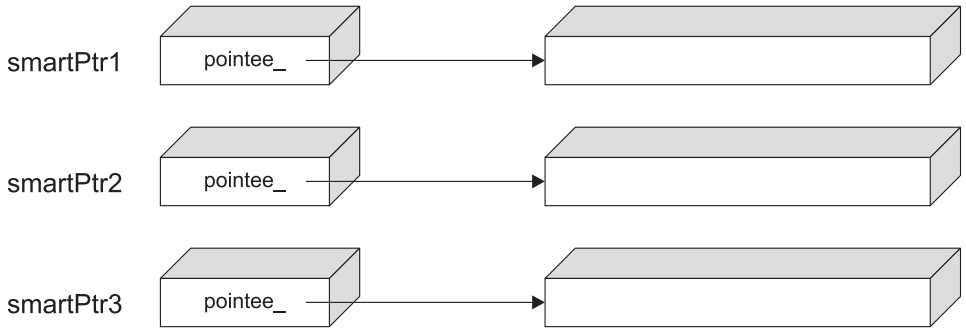
Zarządzanie własnością jest często najistotniejszą *raison d'être* inteligentnego wskaźnika. Zazwyczaj, z punktu widzenia klienta, inteligentny wskaźnik jest właścicielem wskazywanego obiektu. Z typologicznego punktu widzenia jest obiektem pierwszej kategorii (tj. ma semantykę wartości – przyp. tłum.) odpowiedzialnym za niejawne zniszczenie wskazywanego obiektu. Klient może zmienić schemat zarządzania własnością, korzystając z pomocniczych funkcji inteligentnego wskaźnika.

Implementacja takiego samodzielnego zarządzania własnością wymaga, by inteligentny wskaźnik uważnie śledził wskazywany obiekt, przede wszystkim przy kopiowaniu, przypisywaniu i niszczeniu. Wiąże się to z pewnym narzutem czasowym i/lub pamięciowym. Programista powinien wybrać strategię, która najlepiej odpowiada konkretnym zastosowaniom, a przy tym nie kosztuje zbyt wiele.

W następnych punktach omawiamy najczęściej spotykane strategie zarządzania własnością oraz sposobami ich implementacji w szablonie `SmartPtr`.

7.5.1. Głębokie kopiowanie

Najprostsza strategia polega na kopiowaniu wskazywanego obiektu razem ze wskaźnikiem. Taka implementacja zapewnia, że dla wskazywanego obiektu istnieje tylko jeden inteligentny wskaźnik. Dzięki temu destruktor inteligentnego wskaźnika może bezpiecznie zniszczyć wskazywany obiekt. Rysunek 7.1 ilustruje użycie inteligentnych wskaźników implementujących głębokie kopiowanie.



Rys. 7.1. Reprezentacja inteligentnych wskaźników z głębokim kopiowaniem

Na pierwszy rzut oka głębokie kopiowanie wydaje się trochę bez sensu. Z pozoru inteligentny wskaźnik nie wnosi nic ponad zwykłą semantykę wartości języka C++. Dlaczego mielibyśmy wkładać wysiłek w posługiwanie się nim, skoro wystarczy zwykłe przekazywanie wskazywanego obiektu przez wartość?

Odpowiedź brzmi: *polimorfizm*. Inteligentne wskaźniki są bezpiecznymi nośnikami obiektów z polimorficznych hierarchii klas. Inteligentny wskaźnik do klasy podstawowej może wskazywać na obiekt klasy pochodnej. Kopiując taki wskaźnik, chcielibyśmy skopiować także zachowanie konkretnego obiektu. Ciekawe jest to, że zawczasu nie wiemy, z jakim zachowaniem i stanem mamy do czynienia, ale wiemy na pewno, że chcemy je skopiować.

Głębokie kopiowanie jest najczęściej stosowane do typów polimorficznych, dlatego następująca, naiwna implementacja konstruktora kopiującego jest błędna:

```
template <class T>
class SmartPtr
{
public:
  template <class U>
  SmartPtr(const SmartPtr<U>& other)
  : pointee_(new T(*other.pointee_))
  {
  }
  ...
};
```

Przypuśćmy, że kopiujemy obiekt typu `SmartPtr<Widget>`. Jeśli `other` wskazuje na obiekt klasy `ExtendedWidget` pochodzącej od `Widget`, to podany konstruktor kopiujący kopiuje jedynie podobiekt `Widget` obiektu `ExtendedWidget`. Zjawisko to jest znane jako *odcinanie* (ang. *slicing*) – „nadbudowa" pochodząca od obiektu `ExtendedWidget` jest „odcinana" przy kopiowaniu. Odcinanie jest najczęściej niepożądane. Szkoda, że w C++ tak bezkrytycznie dopuszcza się odcinanie – zwykłe przekazanie obiektu przez wartość powoduje odcinanie bez żadnego ostrzeżenia.

W rozdziale 8 wyczerpująco omawiamy klonowanie. Pokazujemy tam, że klasyczną metodą tworzenia klonu obiektu z hierarchii polimorficznej jest zdefiniowanie funkcji wirtualnej `Clone` zaimplementowanej następująco:

```
class AbstractBase
{
  ...
  virtual AbstractBase* Clone() = 0;
};

class Concrete : public AbstractBase
{
  ...
  virtual AbstractBase* Clone()
  {
    return new Concrete(*this);
  }
};
```

Implementacje funkcji składowej `Clone` we wszystkich klasach pochodnych muszą być podobne. Niestety, ten schemat nie daje się sensownie zautomatyzować (pomijając makrodefinicje).

Nie możemy liczyć na to, że generyczny inteligentny wskaźnik będzie znał nazwę funkcji składowej implementującej klonowanie. Może będzie to `Clone`, a może `MakeCopy`. Z tego

powodu najbardziej ogólne podejście polega na sparametryzowaniu szablonu `SmartPtr` wytyczną określającą sposób klonowania.

7.5.2. Kopiowanie przy zapisie

Kopiowanie przy zapisie (ang. *copy on write*) czy, jak pieszczotliwie mówią jego zwolennicy, COW (*krowa*) jest techniką optymalizacji, polegającą na unikaniu zbędnego kopiowania wskazywanego obiektu. Pomysł polega na klonowaniu obiektu dopiero przy pierwszej próbie modyfikacji. Zanim to nastąpi, wiele wskaźników może współdzielić ten sam wskazywany obiekt.

Inteligentne wskaźniki to jednak nie jest najlepszy punkt implementacji techniki COW, ponieważ nie odróżniają one wywołań modyfikujących (zadeklarowanych bez `const`) i niemodyfikujących (zadeklarowanych z `const`) funkcji składowych wskazywanego obiektu. Oto przykład:

```
template <class T>
class SmartPtr
{
public:
  T* operator->() { return pointee_; }
  ...
};

class Foo
{
public:
  void ConstFun() const;
  void NonConstFun();
};

...
SmartPtr<Foo> sp;
sp->ConstFun();    // Wywołuje operator->, następnie ConstFun
sp->NonConstFun(); // Wywołuje operator->, następnie NonConstFun
```

W obu przypadkach jest wywoływany ten sam `operator->`. Z tej przyczyny inteligentny wskaźnik nie wie, czy powinien wykonać kopiowanie. Wywołania funkcji składowych wskazywanego obiektu mają miejsce poza zasięgiem jego poznania. (W podrozdziale 7.11 wyjaśniono zagadnienia związane z zastosowaniem modyfikatora `const` do inteligentnych wskaźników i wskazywanych przez nie obiektów).

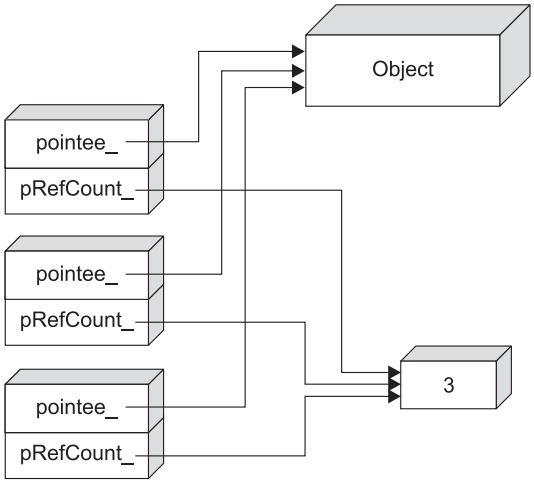
Reasumując, technika COW jest najczęściej użyteczna jako optymalizacja implementacji samych klas. Inteligentne wskaźniki są mechanizmem zbyt niskiego poziomu by dało się za ich pomocą rozsądnie zaimplementować tę technikę. Oczywiście inteligentne wskaźniki mogą okazać się pożyteczne jako składniki implementacji COW w złożonych klasach.

Przedstawiona tu implementacja szablonu `SmartPtr` nie obejmuje techniki COW.

7.5.3. Zliczanie referencji

Zliczanie referencji to strategia zarządzania własnością najczęściej stosowana w implementacjach inteligentnych wskaźników. Licznik referencji śledzi liczbę inteligentnych wskaźników wskazujących ten sam obiekt. Gdy liczba ta osiąga zero, wskazywany obiekt jest niszczony. Ta strategia bardzo dobrze się sprawdza, pod warunkiem że przestrzegamy pewnych zasad. Nie należy na przykład przechowywać zwykłych wskaźników do obiektów wskazywanych przez inteligentne wskaźniki.

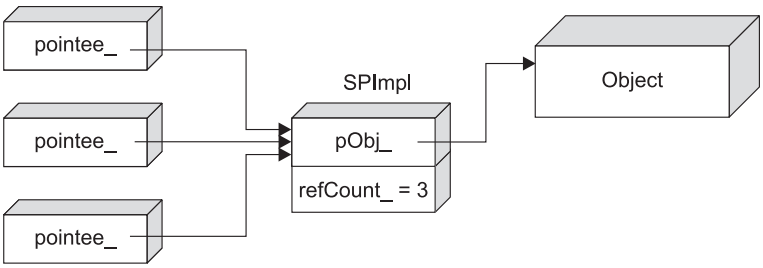
Sam licznik musi być wspólny dla wszystkich inteligentnych wskaźników wskazujących na ten sam obiekt. Prowadzi to do struktury pokazanej na rysunku 7.2. Każdy inteligentny wskaźnik oprócz samego dowiązania do wskazywanego obiektu, przechowuje wskaźnik do licznika referencji (`pRefCount_` na rysunku 7.2). Zazwyczaj powoduje to dwukrotny wzrost rozmiaru inteligentnego wskaźnika, co w konkretnej sytuacji może być niedopuszczalne.



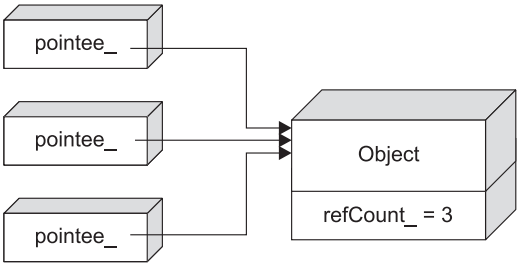
Rys. 7.2. Trzy inteligentne wskaźniki z licznikiem referencji wskazujące na ten sam obiekt

Istnieje jeszcze jeden, bardziej delikatny problem, problem wydajności. Inteligentny wskaźnik z licznikiem referencji musi przechowywać licznik na stercie. Problem polega na tym, że w wielu implementacjach domyślny przydzielacz pamięci jest w wypadku małych obiektów powolny i rozrzutny (co omówiliśmy w rozdziale 4). (Zajmujący zazwyczaj 4 bajty licznik referencji jest bezsprzecznie małym obiektem). Problemem staje się narzut pochodzący od powolnych algorytmów wyszukujących wolne miejsce na stercie, a także dodatkowa pamięć, której przydzielacz potrzebuje na księgowanie bloków pamięci.

Narzut pamięciowy można zmniejszyć, przechowując wskaźnik i licznik razem, jak to pokazano na rysunku 7.3. Zastosowanie przedstawionej tam struktury zmniejsza rozmiar inteligentnego wskaźnika z powrotem do rozmiaru zwykłego wskaźnika, dzieje się to jednak kosztem szybkości dostępu: obiekt wskazywany znajduje się teraz o jedno wyłuskanie dalej. Jest to istotna wada, ponieważ zazwyczaj wskaźnik wyłuskuje się wielokrotnie, natomiast tworzy i niszczy tylko raz.



Rys. 7.3. Alternatywna struktura wskaźników z licznikiem referencji



Rys. 7.4. Intruzyjne zliczanie referencji

Najbardziej wydajne rozwiązanie polega na przechowywaniu licznika referencji bezpośrednio we wskazywanym obiekcie, jak na rysunku 7.4. Dzięki temu `SmartPtr` ma rozmiar zwykłego wskaźnika i nie występuje żaden dodatkowy narzut. Ta technika jest znana pod nazwą *intruzyjnego zliczania referencji* (ang. *intrusive reference counting*), ponieważ licznik referencji jest „intruzem" we wskazywanym obiekcie – z semantycznego punktu widzenia należy wszak do inteligentnego wskaźnika. Ta nazwa wskazuje także na wadę omawianego rozwiązania: klasa obiektu wskazywanego musi być tak zaprojektowana, by przechowywać licznik referencji.

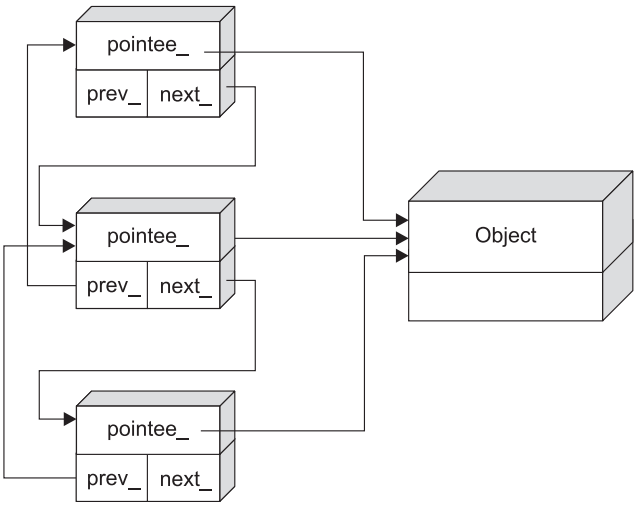
Generyczny wskaźnik inteligentny powinien korzystać z intruzyjnego licznika referencji tam, gdzie to możliwe, a stosować rozwiązania nieintruzyjne w pozostałych przypadkach. Przydzielacz małych obiektów, zaprezentowany w rozdziale 4, może okazać się bardzo pomocny przy implementacji nieintruzyjnej. Implementacja nieintruzyjnego zliczania referencji w szablonie `SmartPtr` korzysta z przydzielacza małych obiektów, zmniejszając tym samym narzuty z nią związane.

7.5.4. Listy referencji

Technika list referencji opiera się na obserwacji, że nieistotna jest wartość licznika referencji, a jedynie fakt, że spadła ona do zera. Prowadzi to do pomysłu, by utrzymywać „listę właścicieli", jak na rysunku 7.5[1].

Wszystkie inteligentne wskaźniki wskazujące na dany obiekt tworzą listę dwukierunkową. Utworzenie nowego inteligentnego wskaźnika na podstawie wskaźnika już istniejącego powoduje dołączenie go do istniejącej listy. Destruktor inteligentnego wskaźnika dba

[1] Listy referencji opisał w roku 1995 na Usenecie Risto Lankinen.



Rys. 7.5. Lista referencji

o usunięcie go z jego listy. Wskazywany obiekt jest usuwany w chwili niszczenia ostatniego wskaźnika na liście.

Lista dwukierunkowa pasuje tu jak ulał. Nie możemy posłużyć się listą jednokierunkową, ponieważ usuwanie z niej elementów wymaga czasu liniowego. Nie możemy też posłużyć się wektorem, ponieważ obiekty inteligentnych wskaźników nie tworzą spójnego obszaru (zresztą usuwanie elementów z wektora i tak wymaga czasu liniowego). Potrzebujemy struktury zapewniającej wstawianie i usuwanie elementu w czasie stałym. Jedyną strukturą danych spełniającą te wymagania jest lista dwukierunkowa.

W implementacji inteligentnych wskaźników opartej na listach referencji każdy inteligentny wskaźnik przechowuje dwa dodatkowe wskaźniki – do poprzedniego i następnego elementu listy.

Zaletą list referencji w porównaniu z licznikami jest to, że listy nie korzystają z dodatkowej pamięci sterty, co sprawia, że są bardziej niezawodne: tworzenie inteligentnego wskaźnika zaimplementowanego za pomocą list nie może się nie powieść. Wadą jest większe zapotrzebowanie na pamięć (trzy wskaźniki zamiast jednego wskaźnika i jednej liczby całkowitej). Liczniki referencji są też nieco szybsze – narzut przy kopiowaniu sprowadza się do jednego przejścia po wskaźniku i jednego zwiększenia liczby całkowitej w pamięci. Utrzymywanie listy referencji jest nieco bardziej złożone. Reasumując, powinniśmy korzystać z list referencji jedynie wtedy, gdy przestrzeń na stercie jest kosztowna. W przeciwnym razie lepsze wydają się liczniki referencji.

Podsumowując omówienie strategii śledzenia referencji zauważmy, że mają one jedną istotną wadę. Śledzenie referencji za pomocą liczników czy list powoduje wyciek zasobów, znany jako *cykl referencji* (ang. *cyclic reference*). Wyobraźmy sobie, że obiekt `A` zawiera inteligentny wskaźnik do obiektu `B`, a obiekt `B` zawiera inteligentny wskaźnik do obiektu `A`. Te dwa obiekty tworzą cykl referencji. Jeśli nawet nikt nie wskazuje na nie, to wskazują one na siebie wzajemnie. Śledzenie referencji nie jest w stanie wykryć takiego cyklu, zatem obiekty te pozostaną w pamięci na zawsze. Takie cykle mogą wieść przez wiele obiektów, znajdując sobie drogę w dowolnych zakamarkach.

Śledzenie referencji jest jednak wydajną strategią zarządzania własnością. Korzystanie z niego, przy zachowaniu należytych środków ostrożności, bardzo ułatwia tworzenie kodu.

7.5.5. Kopiowanie niszczące

Kopiowanie niszczące robi dokładnie to, co sugeruje jego nazwa: podczas kopiowania niszczy oryginał. W wypadku inteligentnych wskaźników kopiowanie niszczące niszczy wskaźnik źródłowy, odbierając mu wskazywany obiekt i przekazując go do wskaźnika docelowego. Tak zachowuje się szablon `std::auto_ptr`.

Nazwa tej techniki zawiera także ostrzeżenie. Przymiotnik „niszczący" trafnie przywodzi na myśl niebezpieczeństwa związane z jej użyciem. Niewłaściwe użycie kopiowania niszczącego może mieć zgubny wpływ na dane, poprawność programu, a także szare komórki programisty.

Inteligentne wskaźniki korzystają z kopiowania niszczącego, by zapewnić, że zawsze tylko jeden inteligentny wskaźnik wskazuje na dany obiekt. Podczas kopiowania lub przypisania do docelowego inteligentnego wskaźnika jest przekazywany „żywy" wskaźnik, a składowa `pointee_` źródłowego inteligentnego wskaźnika przyjmuje wartość wskaźnika pustego. Oto kod konstruktora kopiującego i operatora przypisania prostego inteligentnego wskaźnika implementującego kopiowanie niszczące.

```
template <class T>
class SmartPtr
{
public:
  SmartPtr(SmartPtr& src)
  {
    pointee_ = src.pointee_;
    src.pointee_ = 0;
  }
  SmartPtr& operator=(SmartPtr& src)
  {
    if (this != & src)
    {
      delete pointee_;
      pointee_ = src.pointee_;
      src.pointee_ = 0;
    }
    return *this;
 }
 ...
};
```

Etykieta języka C++ wymaga, by argument konstruktora kopiującego oraz prawy argument operatora przypisania miały typ referencji do obiektu stałego (`const`). Klasy posługujące się kopiowaniem niszczącym z oczywistych przyczyn łamią tę konwencję. Etykieta

istnieje nie bez powodu, łamiąc ją powinniśmy spodziewać się negatywnych konsekwencji. Oto one[1]:

```
void Display(SmartPtr<Something> sp);
...
SmartPtr<Something> sp(new Something);
Display(sp); // Pożera sp
```

Choć funkcja `Display` nie zamierza niszczyć swojego argumentu (przekazywanego przez wartość), to zachowuje się jak czarna dziura: pożera każdy przekazany do niej wskaźnik. Po wywołaniu `Display(sp)` inteligentny wskaźnik `sp` jest pusty.

Inteligentne wskaźniki z kopiowaniem niszczącym nie mają semantyki wartości, nie można ich zatem przechowywać w pojemnikach[2] i należy się z nimi obchodzić niemal tak ostrożnie, jak ze zwykłymi wskaźnikami. Możliwość przechowywania inteligentnego wskaźnika w pojemniku standardowym jest bardzo istotna. Pojemniki zawierające zwykłe wskaźniki bardzo utrudniają zarządzanie własnością, użycie do tego inteligentnych wskaźników niesie zazwyczaj duży pożytek. Niemniej jednak inteligentne wskaźniki z kopiowaniem niszczącym nie idą w parze z pojemnikami.

Inteligentne wskaźniki z kopiowaniem niszczącym mają też istotne zalety:

- Praktycznie nie wprowadzają narzutów.
- Dobrze się spisują, jeśli chodzi o wymuszanie semantyki przenoszenia własności. Dobrze sprawdza się tutaj opisany wcześniej „efekt czarnej dziury": definicja funkcji jasno określa, że funkcja przejmuje przekazywany jej wskaźnik.
- Nadają się do zwracania wskaźników jako wyników funkcji. Jeśli implementacja inteligentnego wskaźnika stosuje pewną sztuczkę[3], to inteligentne wskaźniki z kopiowaniem niszczącym można zwracać jako wyniki funkcji. Dzięki temu można mieć pewność, że wskazywany obiekt zostanie zniszczony, jeśli klient nie przejmie wyniku wywołania.
- Są wspaniałe jako zmienne automatyczne w funkcjach o wielu ścieżkach wykonania. Nie trzeba pamiętać o konieczności zniszczenia wskazywanego obiektu – robi to inteligentny wskaźnik.

Ze strategii kopiowania niszczącego korzysta zdefiniowany przez standard C++ inteligentny wskaźnik `std::auto_ptr`. To sprawia, że kopiowanie niszczące ma jeszcze jedną istotną zaletę:

- Inteligentne wskaźniki z kopiowaniem niszczącym są jedynymi inteligentnymi wskaźnikami zdefiniowanymi przez standard, co oznacza, że wielu programistów wcześniej czy później przyzwyczai się do ich zachowania.

Z podanych powodów implementacja szablonu `SmartPtr` powinna umożliwiać stosowanie kopiowania niszczącego.

[1] Rozwiązaniem alternatywnym do kopiowania niszczącego jest *konstrukcja przenosząca* (ang. *moving construction*), opisana w artykule autora z lutego 2003 w *C/C++ Users Journal*. Artykuł jest dostępny także w internetowym dziale *C++ Experts Forum* pod adresem www.cuj.com. (Przyp. tłum.)

[2] Chodzi oczywiście o pojemniki wymagające, by ich elementy miały semantykę wartości, jak pojemniki standardowe. (Przyp. tłum.)

[3] Wymyśloną przez Grega Colvina i Billa Gibbonsa na użytek `std::auto_ptr`.

Inteligentne wskaźniki korzystają z różnych strategii zarządzania własnością, z których każda stanowi kompromis innego rodzaju. Najważniejszymi strategiami są: głębokie kopiowanie, zliczanie referencji, listy referencji i kopiowanie niszczące. Szablon `SmartPtr` implementuje wszystkie te strategie za pomocą wytycznej `Ownership`, umożliwiającej użytkownikowi wybór tej, która najlepiej odpowiada jego konkretnym potrzebom. Strategią domyślną jest zliczanie referencji.

7.6. Operator pobrania adresu

Podczas próby upodobnienia wskaźników inteligentnych do zwykłych wskaźników projektanci na liście operatorów podlegających przeciążaniu natrafiają na niejasny unarny `operator&` – *operator pobrania adresu*[1] (ang. *address-of operator*).

Programista implementujący inteligentny wskaźnik może zdecydować się na przeciążenie operatora pobrania adresu:

```
template <class T>
class SmartPtr
{
public:
  T** operator&()
  {
    return &pointee_;
  }
  ...
};
```

Jeśli inteligentny wskaźnik ma symulować zwykły wskaźnik, to trzeba zadbać i o to, by jego adres można było podstawić za adres zwykłego wskaźnika. Dzięki temu działa następujący kod:

```
void Fun(Widget** pWidget);
...
SmartPtr<Widget> spWidget(...);
Fun(&spWidget); // OK, wywołuje operator&, otrzymując
                // wskaźnik do wskaźnika do Widget
```

Tak bliska zgodność między inteligentnymi a zwykłymi wskaźnikami wydaje się bardzo pożądana, jednak przeciążanie unarnego operatora `operator&` jest jedną z tych sztuczek, które przynoszą więcej szkody niż pożytku. Są dwa powody, dla których przeciążanie unarnego operatora `operator&` nie jest dobrym pomysłem.

Po pierwsze, obnażanie adresu wskazywanego obiektu oznacza utratę jakiegokolwiek automatycznego zarządzania własnością. Jeśli użytkownik ma swobodny dostęp do adresu wskaźnika przechowującego dowiązanie do wskazywanego obiektu, to wszystkie pomocnicze struktury danych utrzymywane przez inteligentny wskaźnik, np. liczniki refe-

[1] *Unarny* `operator&` różni się od binarnego operatora `operator&`, czyli koniunkcji bitowej (AND)

rencji, stają się bezużyteczne. Użytkownik może dokonywać dowolnych operacji na składowej `pointee_`, a inteligentny wskaźnik nic o tym nie wie.

Po drugie, przeciążenie unarnego operatora `operator&` czyni go niezdatnym do zastosowania w pojemnikach biblioteki STL. W istocie przeciążenie unarnego operatora `operator&` dla jakiegoś typu powoduje, że ten typ staje się praktycznie bezużyteczny w kodzie generycznym, ponieważ adres obiektu jest własnością zbyt podstawową, by można się nią było beztrosko bawić. Na ogół w kodzie generycznym zakłada się, że użycie `&` do obiektu typu `T` zwraca obiekt typu `T*` – pobranie adresu jest po prostu operacją podstawową. Jeśli nie zadbamy o spełnienie tego założenia, to kod generyczny zacznie się dziwnie zachowywać podczas kompilacji albo, co gorsza, podczas wykonania[1].

Z podanych przyczyn nie zaleca się przeciążania unarnego operatora `operator&` przede wszystkim dla inteligentnych wskaźników. Szablon `SmartPtr` nie przeciąża unarnego operatora `operator&`.

7.7. Niejawna konwersja do opakowanego wskaźnika

Rozważmy następujący kod:

```
void Fun(Something* p);
...
SmartPtr<Something> sp(new Something);
Fun(sp); // Poprawne czy błędne?
```

Czy ten kod powinien się skompilować? Jeśli hołdujemy zasadzie „maksymalnej zgodności”, odpowiedź brzmi „tak”.

Z technicznego punktu widzenia bardzo łatwo jest sprawić, by ten kod był poprawny. Wystarczy zdefiniować konwersję

```
template <class T>
class SmartPtr
{
public:
  operator T*() // Konwersja do T*
  {
    return pointee_;
  }
  ...
};
```

Nie koniec jednak na tym.

Definiowane przez użytkownika konwersje typów w C++ mają interesującą historię. Kiedy wprowadzono je w latach osiemdziesiątych XX wieku wielu programistom wydawało się, że to fantastyczny wynalazek. Zapowiadały bardziej spójny system typów, dużą

[1] W bibliotece Boost (www.boost.org) zdefiniowano szablon funkcyjny `adressof` implementujący pobranie adresu nawet dla obiektów tych klas, dla których przedefiniowano `operator&`. (Przyp. tłum.)

siłę wyrazu, możliwość definiowania nowych typów nieodróżnialnych od typów wbudowanych. Jednak z upływem czasu okazało się, że są toporne i mogą być niebezpieczne. Mogą stać się szczególnie groźne, gdy odsłaniają uchwyty danych prywatnych (Meyers 1998a, p. 29), a tak się właśnie dzieje w wypadku operatora `operator T*` w podanym kodzie. Z tego powodu należy poważnie się zastanowić, zanim do swoich inteligentnych wskaźników zastosuje się automatyczne konwersje.

Jedno z zagrożeń wynika z udostępnienia użytkownikowi nieskrępowanego dostępu do wskaźnika zamkniętego we wnętrzu implementacji. Możliwość „wyjęcia" tego wskaźnika niweczy całą mechanikę inteligentnego wskaźnika. Wskaźnik, który wymknie się z wnętrza inteligentnego wskaźnika, stwarza te zagrożenia dla poprawności programu, których chcieliśmy uniknąć przez wprowadzenie inteligentnych wskaźników.

Inne niebezpieczeństwo polega na tym, że konwersje zdefiniowane przez użytkownika pojawiają się niespodziewanie, nawet gdy ich nie potrzebujemy:

```
SmartPtr<Something> sp;
...
// Poważny błąd semantyczny
// Kompilator jednak tego nie wykrywa
delete sp;
```

Kompilator zastosuje niejawną konwersję `sp` do wartości typu `T*`, po czym wykona na niej `delete`. Zdecydowanie nie tego oczekujemy – inteligentny wskaźnik sam miał zarządzać niszczeniem wskazywanego obiektu. Zatem będzie próbował ponownie zniszczyć wskazywany obiekt – cała misterna konstrukcja, mająca zarządzać własnością obiektów, wali się.

Istnieje kilka sposobów na wymuszenie zerwania kompilacji kodu zawierającego takie użycie `delete`. Niektóre są bardzo pomysłowe (Meyers 1996). Jeden z nich, bardzo wydajny i prosty w implementacji, polega na celowym *uniejednoznacznieniu* wywołania operatora `delete`. Można to osiągnąć przez zastosowanie dodatkowo automatycznej konwersji do innego typu, który pasowałby w operatorze `delete`. Oprócz konwersji do `T*` dodajemy na przykład konwersję do `void*`.

```
template <class T>
class SmartPtr
{
public:
  operator T*() // Konwersja do T*
  {
    return pointee_;
  }
  operator void*() // Dodatkowa konwersja do void*
  {
    return pointee_;
  }
  ...
};
```

Wywołanie `delete` na takim inteligentnym wskaźniku jest wieloznaczne. Kompilator nie potrafi zdecydować się na żadną z konwersji, a my z tego korzystamy.

Nie zapominajmy jednak, że zablokowanie operatora `delete` rozwiązuje tylko część problemów. Decyzja o implementacji automatycznej konwersji do zwykłego wskaźnika jest ważną decyzją projektową. Bezkrytyczne dopuszczenie automatycznej konwersji jest niebezpieczne, ale z drugiej strony jest ona zbyt wygodna, by z góry ją wykluczyć. Implementacja szablonu `SmartPtr` pozostawia wybór użytkownikowi.

Ponadto wykluczenie niejawnej konwersji nie oznacza, że wykluczamy jakikolwiek dostęp do opakowanego wskaźnika; zbyt często tego potrzebujemy. Z tej przyczyny wszystkie konkretyzacje szablonu `SmartPtr` zezwalają na jawną konwersję inteligentnego wskaźnika za pomocą następującej funkcji:

```
void Fun(Something* p);
...
SmartPtr<Something> sp;
Fun(GetImpl(sp)); // OK, zawsze dopuszczamy konwersję jawną
```

Nie chodzi o to, czy można uzyskać dostęp do opakowanego wskaźnika, ale o to, jak łatwo. Może się wydawać, że to bez znaczenia, ale tak nie jest. Konwersja niejawna odbywa się bez woli i wiedzy programisty. Konwersja jawna, taka jak wywołanie funkcji `GetImpl`, jest decyzją programisty, ponadto jest wyraźnie udokumentowana w kodzie.

Niejawna konwersja ze wskaźnika inteligentnego jest pożądana, ale często niebezpieczna. Szablon `SmartPtr` umożliwia włączenie konwersji niejawnej, ale tego nie narzuca. Domyślnie jest włączony wariant bezpieczniejszy, tj. brak konwersji niejawnej. Konwersja jawna jest dostępna zawsze za pomocą funkcji `GetImpl`.

7.8. Równość i nierówność

C++ uczy programistów, że każda sztuczka, jak ta z poprzedniego rozdziału (umyślna niejednoznaczność), tworzy nowy kontekst, który ma swoje nowe chropowatości.

Zastanówmy się nad sprawdzaniem równości inteligentnych wskaźników. Powinny one tutaj dopuszczać taką składnię, jak wskaźniki zwykłe. Zacznijmy od porównania z zerem. Programiści spodziewają się, że tak jak w wypadku zwykłych wskaźników, skompiluje się i zadziała następujący kod:

```
SmartPtr<Something> sp1, sp2;
Something* p;
...
if (sp1)       // Test 1: bezpośrednie sprawdzenie, czy wskaźnik nie jest pusty
  ...
if (!sp1)      // Test 2: bezpośrednie sprawdzenie, czy wskaźnik jest pusty
  ...
if (sp1 == 0)  // Test 3: jawne porównanie z zerem
  ...
```

```
if (sp1 == sp2)  // Test 4: porównanie inteligentnych wskaźników
  ...
if (sp1 == p)    // Test 5: porównanie ze zwykłym wskaźnikiem
  ...
```

To nie są jeszcze wszystkie możliwe porównania, pominęliśmy przypadki symetryczne oraz `operator!=`. Jeśli jednak rozwiążemy problem dla tych warunków, to reszta będzie już prosta.

Istnieje nieszczęśliwy związek między rozwiązaniem problemu z poprzedniego podrozdziału (zablokowanie `delete`) a rozwiązaniem problemu z tego rozdziału. Jeśli zdefiniujemy tylko jedną konwersję do zwykłego wskaźnika, to większość warunków skompiluje się i zadziała poprawnie. Wadą jest to, że odblokujemy operator `delete` na inteligentnych wskaźnikach. Definiując dwie konwersje (umyślna niejednoznaczność), blokujemy błędne użycie `delete`, ale żaden z warunków się już nie skompiluje – w nich też pojawi się niejednoznaczność.

Dodatkowa konwersja do typu `bool` pomaga, ale (to już nikogo nie dziwi) sprawia nowe kłopoty. Rozważmy definicję:

```
template <class T>
class SmartPtr
{
public:
  operator bool() const
  {
    return pointee_ != 0;
  }
  ...
};
```

Teraz nasze warunki się kompilują, a wraz z nimi pozbawione sensu następujące operacje:

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;  // Orange i Apple nie mają nic wspólnego
if (sp1 == sp2)        // Konwertuje oba wskaźniki do bool
                       // i porównuje
  ...
if (sp1 != sp2)        // Jak wyżej
  ...
bool b = sp1;          // Konwersja to umożliwia
if (sp1 * 5 == 200)    // O rany! SmartPtr zachowuje się jak
                       // typ całkowitoliczbowy!
  ...
```

Albo nic, albo zbyt wiele. Dodanie konwersji do typu `bool` oznacza przyzwolenie na to, by inteligentny wskaźnik zachowywał się jak `bool` nie tylko tam, gdzie chcieliśmy, ale także tam, gdzie sobie tego nie życzymy. Z wielu praktycznych powodów definiowanie

operatora `operator bool` dla inteligentnych wskaźników nie jest wcale inteligentnym posunięciem.

Autentyczne, kompletne i solidne rozwiązanie tego problemu prowadzi długą drogą przeciążania każdego operatora z osobna. W ten sposób każda operacja poprawna dla wskaźnika staje się poprawna dla wskaźnika inteligentnego, ale nic ponadto. Oto kod implementujący ten pomysł.

```
template <class T>
class SmartPtr
{
public:
  bool operator!() const // Umożliwia "if (!sp) ..."
  {
    return pointee_ == 0;
  }
  inline friend bool operator==(const SmartPtr& lhs,
    const T* rhs)
  {
    return lhs.pointee_ == rhs;
  }
  inline friend bool operator==(const T* lhs,
    const SmartPtr& rhs)
  {
    return lhs == rhs.pointee_;
  }
  inline friend bool operator!=(const SmartPtr& lhs,
    const T* rhs)
  {
    return lhs.pointee_ != rhs;
  }
  inline friend bool operator!=(const T* lhs,
    const SmartPtr& rhs)
  {
    return lhs != rhs.pointee_;
  }
  ...
};
```

To podejście rozwiązuje problem prawie wszystkich porównań, również porównania z zerem. Przedstawiona implementacja polega na zepchnięciu implementacji operatorów porównania do tych samych operacji na składowych `pointee_` inteligentnych wskaźników.

Wciąż jednak nie jest to pełne rozwiązanie. Zdefiniowanie automatycznej konwersji do zwykłego wskaźnika naraża nas na niejednoznaczności. Przypuśćmy, że mamy klasę podstawową `Base` i wyprowadzoną z niej klasę `Derived`. Następujący kod jest zupełnie rozsądny, jednak nie jest poprawny z powodu niejednoznaczności.

```
SmartPtr<Base> sp;
Derived* p;
...
if (sp == p) {} // Błąd! Niejednoznaczność:
                // '(Base*)sp == (Base*)p'
                // czy 'operator==(sp, (Base*)p)' ?
```

Zaiste pisanie inteligentnych wskaźników nie jest zadaniem dla słabych duchem.

Ale mamy jeszcze coś w zanadrzu. Do istniejących definicji operatorów `operator==` i `operator!=` możemy dołożyć ich wersje *szablonowe*:

```
template <class T>
class SmartPtr
{
public:
  ... jak poprzednio ...
  template <class U>
  inline friend bool operator==(const SmartPtr& lhs,
    const U* rhs)
  {
    return lhs.pointee_ == rhs;
  }
  template <class U>
  inline friend bool operator==(const U* lhs,
    const SmartPtr& rhs)
  {
    return lhs == rhs.pointee_;
  }
  ... podobnie operator!= ...
};
```

Operatory szablonowe są „zachłanne" w tym sensie, że pasują do porównania z dowolnym wskaźnikiem, a tym samym eliminują niejednoznaczność.

Skoro tak, to czy jest sens trzymać operatory nieszablonowe? Przecież one nigdy nie mają szansy być użyte, ponieważ operatory szablonowe będą zawsze pasowały do dowolnego wskaźnika, również tego występującego w definicji operatorów nieszablonowych.

Tutaj „nigdy" znaczy „prawie nigdy" . W porównaniu `if (sp == 0)` kompilator próbuje następujących dopasowań:

- *Operatory szablonowe.* Nie pasują, ponieważ stała zero nie ma typu wskaźnikowego. Stała zero może być niejawnie skonwertowana do typu wskaźnikowego, ale konwersje niejawne nie są brane pod uwagę przy dopasowywaniu szablonów.
- *Operatory nieszablonowe.* Po wyeliminowaniu operatorów szablonowych kompilator próbuje dopasować operatory nieszablonowe. Jeden z nich działa dzięki niejawnej konwersji zera do typu wskaźnikowego. Porównanie nie skompilowałoby się, gdyby nie zadeklarowano operatorów nieszablonowych.

Potrzebujemy zatem zarówno szablonowych, jak i nieszablonowych operatorów porównania.

Zobaczmy teraz, co się stanie, jeśli porównamy inteligentne wskaźniki otrzymane z szablonu SmartPtr skonkretyzowanego różnymi typami dowiązania.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)
  ...
```

Kompilator nie radzi sobie z porównaniem z powodu niejednoznaczności: każda konkretyzacja szablonu `SmartPtr` definiuje swój `operator==`, a kompilator nie wie, który wybrać. Możemy ominąć ten problem, definiując „pogromcę niejednoznaczności”:

```
template <class T>
class SmartPtr
{
public:
  // Pogromca niejednoznaczności
  template <class U>
  bool operator==(const SmartPtr<U>& rhs) const
  {
    return pointee_ == rhs.pointee_;
  }
  // Podobnie dla operatora !=
  ...
};
```

Ten nowy operator jest funkcją składową specjalizującą się w porównywaniu obiektów różnych konkretyzacji szablonu `SmartPtr<...>`. Piękno pogromcy polega na tym, że dzięki niemu porównania inteligentnych wskaźników zachowują się jak porównania zwykłych wskaźników. Próba porównania inteligentnego wskaźnika do `Apple` i inteligentnego wskaźnika do `Orange` skończy się tak jak próba porównania `Apple*` do `Orange*`. Jeśli porównanie ma sens, to kod się skompiluje, jeśli nie, to kompilator zgłosi błąd.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)  // Semantycznie równoważne porównaniu
                 // sp1.pointee_ == sp2.pointee_
  ...
```

Pozostał jeszcze jeden kontekst syntaktyczny: `if (sp)`. Dopiero tu akcja staje się ciekawa. Instrukcja `if` ma zastosowanie tylko do typów arytmetycznych i wskaźnikowych. Jeśli więc `if (sp)` ma się kompilować, to musimy zdefiniować automatyczną konwersję albo do typu arytmetycznego, albo do typu wskaźnikowego.

Jak pokazują wcześniejsze doświadczenia z konwersją do typu `bool`, konwersja do typu arytmetycznego nie jest dobrym pomysłem. Wskaźnik to nie typ arytmetyczny i już. Konwersja do typu wskaźnikowego ma o wiele większy sens i tu stajemy na rozdrożu.

Jeśli zdefiniujemy konwersję do zwykłego wskaźnika na obiekt wskazywany (zob. poprzedni podrozdział), to mamy dwie możliwości: albo ryzykujemy i nie zabezpieczamy się przed operatorem `delete`, albo rezygnujemy z `if (sp)`[1]. Niewygoda albo ryzyko. Wygrywa bezpieczeństwo, zatem nie będziemy mogli pisać `if (sp)`. Mamy za to `if(sp != 0)` albo stare `if (!!sp)`. Koniec.

Jeśli nie definiujemy automatycznej konwersji do zwykłego wskaźnika na obiekt wskazywany, to implementacja `if (sp)` jest możliwa. Wewnątrz szablonu `SmartPtr` definiujemy prywatną klasę `Tester` i definiujemy konwersję do typu `Tester*`:

```
template <class T>
class SmartPtr
{
  class Tester
  {
    void operator delete(void*);
  };
public:
  operator Tester*() const
  {
    if (!pointee_) return 0;
    static Tester test;
    return &test;
  }
  ...
};
```

Teraz przy kompilacji `if (sp)` działa `operator Tester*`. Zwraca on wskaźnik pusty wtedy i tylko wtedy, gdy `pointee_` jest wskaźnikiem pustym. Sama klasa `Tester` blokuje operator `delete`, zatem jeśli ktoś spróbuje wywołać `delete sp`, to nastąpi błąd kompilacji. Co ciekawe, definicja klasy `Tester` jest prywatna w klasie `SmartPtr`, zatem kod klienta nie może z nią nic zrobić.

Szablon `SmartPtr` rozwiązuje problem porównań następująco:

- Definiuje `operator==` i `operator!=` w dwóch wersjach (szablonowej i nieszablonowej).
- Definiuje `operator!`.
- Jeśli użytkownik dopuszcza automatyczną konwersję do zwykłego wskaźnika, to `SmartPtr` definiuje dodatkową konwersję do `void*`, by uniejednoznacznić wywołania operatora `delete`. W przeciwnym razie definiuje prywatną klasę wewnętrzną `Tester`,

[1] Implementacja inteligentnego wskaźnika z biblioteki Boost obchodzi ten problem, implementując konwersję nie do typu opakowanego wskaźnika, lecz to typu wskaźnika do (dowolnej) funkcji składowej. (Przyp. tłum.)

która deklaruje prywatny operator delete i definiuje konwersję ze SmartPtr do Tester*, zwracającą wskaźnik pusty wtedy i tylko wtedy, gdy pointee_ jest wskaźnikiem pustym.

7.9. Porównania porządkujące

Do operatorów porównań porządkujących należą: operator<, operator<=, operator> i operator>=. Wszystkie można zaimplementować, posługując się tylko implementacją operator<.

Interesujące jest pytanie, czy dopuścić korzystanie z operatorów porównań porządkujących na inteligentnych wskaźnikach? Wiąże się ono z dualną naturą wskaźników, która wciąż wprowadza w błąd programistów. Wskaźniki zawierają w sobie dwie idee: iteratorów i pośredników. Iteracyjna natura wskaźników umożliwia przeglądanie tablic obiektów. O tej części ich natury decyduje arytmetyka na wskaźnikach wraz z porównaniami. Jednocześnie wskaźniki są pośrednikami (ang. *monikers*) – tanimi do skopiowania reprezentantami umożliwiającymi błyskawiczny dostęp do obiektu. Za tę część odpowiadają operatory: operator* i operator->.

Te dwie natury czasami się mylą, zazwyczaj wtedy, gdy chcemy stosować tylko jedną z nich. Przeglądając wektor, korzystamy tak z iteracji, jak i z wyłuskania, ale przeglądając listę wskaźnikową lub działając na odosobnionych obiektach, używamy wyłącznie wyłuskania.

Porównania porządkujące na wskaźnikach są zdefiniowane tylko wtedy, gdy wskaźniki pokazują na obiekty w jednym spójnym obszarze pamięci. Innymi słowy, w sposób uporządkowany można porównywać tylko wskaźniki pokazujące na elementy tej samej tablicy.

Zdefiniowanie porównań porządkujących dla inteligentnych wskaźników sprowadza się do pytania: czy inteligentne wskaźniki do obiektów w jednej tablicy mają sens? W sumie nie. Ich główną cechą jest umiejętność zarządzania własnością obiektów, a obiekty mające różnych właścicieli zazwyczaj nie należą do tej samej tablicy. Z tego powodu byłoby niebezpiecznie dawać użytkownikom do ręki bezsensowne operatory porównania.

Jeśli naprawdę potrzebujesz porównań porządkujących, możesz zawsze skorzystać z jawnej konwersji do zwykłego wskaźnika. Wówczas problem polega na znalezieniu rozwiązania bezpiecznego i wygodnego w większości przypadków – ale niekoniecznie we wszystkich.

Poprzedni podrozdział zakończyliśmy stwierdzeniem, że niejawna konwersja do zwykłego wskaźnika jest opcjonalna. Jeśli użytkownik szablonu SmartPtr zdecyduje się dopuścić tę konwersję, to następujący kod się skompiluje:

```
SmartPtr<Something> sp1, sp2;
if (sp1 < sp2) // Niejawnie konwertuje sp1 i sp2 do zwykłych wskaźników,
               // po czym wykonuje porównanie
...
```

Oznacza to, że jeśli chcemy zablokować porównania porządkujące, to musimy podjąć jawne działanie. Jednym ze sposobów jest ich zadeklarowanie, ale nie zdefiniowanie, co oznacza, że próba ich użycia spowoduje błąd kompilacji.

```
template <class T>
class SmartPtr
{ ... };

template <class T, class U>
bool operator<(const SmartPtr<T>&, const U&); // Brak definicji
template <class T, class U>
bool operator<(const T&, const SmartPtr<U>&); // Brak definicji
```

Zamiast w podobny sposób blokować pozostałe operatory, mądrzej jest zdefiniować je za pomocą operatora `operator<`. Jeśli użytkownik zdecyduje, że jednak operatory porządkujące na inteligentnych wskaźnikach są potrzebne, to będzie mógł je włączyć, definiując `operator<`.

```
// Pogromca niejednoznaczności
template <class T, class U>
bool operator<(const SmartPtr<T>& lhs, const SmartPtr<U>& rhs)
{
  return lhs < GetImpl(rhs);
}
// Wszystkie inne operatory
template <class T, class U>
bool operator>(SmartPtr<T>& lhs, const U& rhs)
{
  return rhs < lhs;
}
... podobnie dla pozostałych ...
```

Zauważmy, że znów pojawia się pogromca niejednoznaczności. Jeśli użytkownik zdecyduje, że wskaźniki typu `SmartPtr<Widget>` powinny być uporządkowane, to następujący kod rozwiązuje problem:

```
inline bool operator<(const SmartPtr<Widget>& lhs,
  const Widget* rhs)
{
  return GetImpl(lhs) < rhs;
}

inline bool operator<(const Widget* lhs,
  const SmartPtr<Widget>& rhs)
{
  return lhs < GetImpl(rhs);
}
```

Szkoda, że zmuszamy użytkownika do definiowania dwóch operatorów, a nie jednego, ale to i tak lepiej niż osiem.

Powstał jeszcze pewien ciekawy szczegół. Czasami warto dysponować uporządkowaniem dowolnie rozmieszczonych obiektów, a nie tylko obiektów znajdujących się w tej samej tablicy. Czasami musimy związać z obiektami jakieś dodatkowe informacje i mieć do nich szybki dostęp. Słownik, którego kluczami są adresy obiektów, może być wydajnym rozwiązaniem.

Standard C++ umożliwia implementację tego schematu. Chociaż porównanie wskaźników do dowolnie rozmieszczonych obiektów ma wynik nieokreślony, to standard zapewnia, że funktor `std::less` zwraca sensowny wynik dla dwóch wskaźników tego samego typu. Standardowe pojemniki asocjacyjne korzystają z `std::less` jako domyślnego funktora porządkującego, zatem bezpiecznie można używać wskaźników jako ich kluczy.

Szablon `SmartPtr` też powinien umożliwiać posługiwanie się tym idiomem. Z tego powodu implementacja szablonu `SmartPtr` zawiera specjalizację funkcji `std::less`. Ta specjalizacja po prostu deleguje zadanie do wersji funktora `std::less` dla opakowanych wskaźników:

```
namespace std
{
  template <class T>
  struct less<SmartPtr<T> >
    : public binary_function<SmartPtr<T>, SmartPtr<T>, bool>
  {
    bool operator()(const SmartPtr<T>& lhs,
      const SmartPtr<T>& rhs) const
    {
      return less<T*>()(GetImpl(lhs), GetImpl(rhs));
    }
  };
}
```

Podsumowując, szablon `SmartPtr` nie definiuje domyślnie operatorów porównań porządkujących dla inteligentnych wskaźników. Deklaruje, ale bez implementacji, dwa generyczne operatory `operator<`, a za ich pomocą implementuje inne operatory porównań porządkujących. Użytkownik może zdefiniować wyspecjalizowane lub generyczne wersje operatora `operator<`.

Implementacja szablonu `SmartPtr` zawiera specjalizację funktora `std::less`, implementującą porównanie dowolnych inteligentnych wskaźników.

7.10. Kontrola i zgłaszanie błędów

Różne programy wymagają od inteligentnych wskaźników różnych poziomów bezpieczeństwa. Niektóre wykonują głównie obliczenia i muszą być zoptymalizowane pod względem czasu działania, inne (większość) wykonują głównie operacje wejścia–wyjścia, co umożliwia im dogłębną kontrolę poprawności wykonania bez istotnego wpływu na wydajność.

Najczęściej jednak w kodzie programu są potrzebne oba rozwiązania: ryzyko i wysoka wydajność w pewnych obszarach krytycznych, bezpieczeństwo i mniejsza wydajność poza nim.

Zagadnienia kontroli błędów związanych z użyciem inteligentnych wskaźników możemy podzielić na dwie kategorie: kontrolę inicjowania oraz kontrolę wyłuskania.

7.10.1. Kontrola inicjowania

Czy inteligentny wskaźnik powinien przyjmować wartość pustą (NULL)?

Bardzo łatwo zapewnić niepustość inteligentnego wskaźnika. Może to być bardzo przydatne w praktyce. Oznacza to, że inteligentny wskaźnik zawsze można wyłuskać (jeśli użytkownik nie robi sztuczek za pomocą `GetImplRef`). Rozwiązanie to implementujemy, zmuszając konstruktor do zgłoszenia wyjątku w razie przekazania mu wskaźnika pustego.

```
template <class T>
class SmartPtr
{
public:
  SmartPtr(T* p) : pointee_(p)
  {
    if (!p) throw NullPointerException();
  }
  ...
};
```

Wartość pusta jest jednak wygodnym znacznikiem niepoprawności wskaźnika.

Dopuszczalność pustości ma także związek z konstruktorem domyślnym. Jeśli inteligentny wskaźnik nie dopuszcza wartości pustej, to w jaki sposób konstruktor domyślny ma zainicjować składową `pointee_`? Moglibyśmy opuścić konstruktor domyślny, ale wtedy inteligentny wskaźnik staje się o wiele mniej wygodny w użyciu[1]. Co na przykład powinniśmy zrobić, jeśli w kodzie jest zmienna o typie inteligentnego wskaźnika, ale podczas jej tworzenia nie dysponujemy odpowiednim inicjatorem? Inicjowanie wymaga istnienia wartości domyślnej inteligentnego wskaźnika.

7.10.2. Kontrola wyłuskania

Kontrola wyłuskania jest ważna, ponieważ wyłuskanie wskaźnika pustego powoduje efekt nieokreślony. W wielu programach nie może pozwolić na zachowanie nieokreślone, zatem kontrola wyłuskania jest jak znalazł. Ma ona miejsce w operatorach `operator->` i `operator*`.

W odróżnieniu od kontroli inicjowania kontrola wyłuskania może stać się wąskim gardłem, jeśli chodzi o wydajność, ponieważ typowy program wyłuskuje inteligentny wskaźnik o wiele częściej niż go inicjuje. Trzeba zatem znaleźć kompromis między szybkością a bezpieczeństwem. Dobrą strategią jest wstawianie wszędzie bezpiecznych wskaźników

[1] Zabranie konstruktora domyślnego powoduje utratę kanonicznej semantyki wartości. (Przyp. tłum.)

i zmienianie ich później na mniej bezpieczne tam, gdzie profilowanie wykaże taką potrzebę.

Czy kontrolę inicjowania i wyłuskania można pojęciowo oddzielić? Nie, ponieważ są między nimi silne powiązania. Jeśli wymusimy rygorystyczną kontrolę inicjowania, to kontrola wyłuskania jest niepotrzebna, bo wskaźnik nigdy nie może być pusty[1].

7.10.3. Zgłaszanie błędów

Jedynym sensownym sposobem zgłoszenia błędu jest zgłoszenie wyjątku.

Można próbować uniknąć błędu. Jeśli podczas wyłuskania okazuje się, że wskaźnik jest pusty, to można go w biegu zainicjować. Jest to poprawna i wartościowa strategia, zwana *leniwym inicjowaniem* – konstruujemy obiekt tylko wtedy, gdy po raz pierwszy okaże się potrzebny.

Jeśli chcemy kontrolować poprawność tylko podczas uruchamiania kodu, to możemy posłużyć się standardowym makrem `assert` (lub podobnym, bardziej wyszukanym mechanizmem). Kompilator je ignoruje przy generowaniu kodu produkcyjnego, zatem (zakładając, że wszystkie błędy wyłuskania pustych wskaźników usunięto podczas uruchamiania) mamy i kontrolę, i prędkość.

Szablon `SmartPtr` deleguje kontrolę do specjalnej wytycznej `Checking`, która odpowiada za implementację funkcji kontrolujących (one też mogą implementować leniwe inicjowanie) oraz za implementację strategii zgłaszania błędów.

7.11. Stałe inteligentne wskaźniki i inteligentne wskaźniki do stałych

Przy zwykłych wskaźnikach mamy do czynienia z dwoma rodzajami stałości (ang. *constness*): stałością wskazywanego obiektu oraz stałością samego wskaźnika. Następujący kod ilustruje oba:

```
const Something* pc = new Something; // Wskazuje do obiektu stałego
pc->ConstMemberFunction();   // Ok
pc->NonConstMemberFunction(); // Błąd
delete pc; // Ok (niespodziewanie)[2]
Something* const cp = new Something; // Wskaźnik stały
cp->NonConstMemberFunction(); // Ok
cp = new Something; // Błąd, nie można przypisać do stałego wskaźnika
const Something* const cpc = new Something; // Stały wskaźnik, stały obiekt
cpc->ConstMemberFunction(); // Ok
cpc->NonConstMemberFunction(); // Błąd
cpc = new Something; // Błąd, nie można przypisać do stałego wskaźnika
```

[1] Mimo kontroli inicjowania wskaźnik może stać się pusty w wyniku przypisania `GetImplRef(p)=0`. (Przyp. tłum.)

[2] Co jakiś czas w grupie dyskusyjnej comp.std.c++ pojawia się pytanie: „Dlaczego można zastosować operator `delete` do wskaźnika do obiektu stałego?", po którym następuje burzliwa wymiana zdań. Czy tego chcemy, czy nie, język na to aktualnie zezwala.

Analogiczne wcielenia inteligentnego wskaźnika wyglądają następująco:

```
// Inteligentny wskaźnik do obiektu stałego
SmartPtr<const Something> spc(new Something);
// Stały inteligentny wskaźnik
const SmartPtr<Something> scp(new Something);
// Stały inteligentny wskaźnik do obiektu stałego
const SmartPtr<const Something> scpc(new Something);
```

Szablon `SmartPtr` potrafi wykryć stałość wskazywanego obiektu za pomocą specjalizacji częściowej lub cechowania (szablon `TypeTraits` z rozdz. 2). Ta druga metoda jest lepsza, gdyż nie powoduje powielania kodu, charakterystycznego dla specjalizacji częściowej.

Implementacja szablonu `SmartPtr` naśladuje semantykę wskaźników do obiektów stałych, wskaźników stałych oraz ich kombinacji.

7.12. Tablice

Często zamiast borykać się z tablicami na stercie za pomocą `new[]` i `delete[]`, lepiej posłużyć się szablonem `std::vector`. Zdefiniowany w standardzie szablon `std::vector` udostępnia to wszystko co dynamicznie przydzielane tablice i dużo więcej. Dodatkowe narzuty są przy tym na ogół zaniedbywalne.

Większość przypadków to jednak nie wszystkie. Czasami nie potrzebujemy i nie chcemy posługiwać się wektorem. Chcemy sami przydzielić pamięć dynamicznie. Niemożność posłużenia się w takiej sytuacji inteligentnymi wskaźnikami wydaje się dziwna. Między dynamicznie przydzielanymi tablicami a szablonem `std::vector` jest mimo wszystko duża luka, którą mogą wypełnić inteligentne wskaźniki o semantyce tablic.

Z punktu widzenia inteligentnego wskaźnika jedyną istotną sprawą jest wywołanie w destruktorze operatora `delete[]`, a nie `delete` na składowej `pointee_`. Tę sprawę zresztą powierzyliśmy już wytycznej `Ownership`.

Sprawą drugorzędną jest umożliwienie dostępu indeksowanego przez przeciążenie operatora `operator[]` dla inteligentnych wskaźników. Jest to technicznie możliwe; pierwsze wersje szablonu `SmartPtr` zawierały oddzielną wytyczną implementującą semantykę tablicy. Okazuje się jednak, że w praktyce inteligentne wskaźniki stosunkowo rzadko wskazują na tablice, ale nawet w tych rzadkich przypadkach dysponujemy już mechanizmem indeksowania:

```
SmartPtr<Widget> sp = ...;
// Dostęp do szóstego elementu tablicy sp
Widget& obj = GetImpl(sp)[5];
```

Implementacja dodatkowej wygody syntaktycznej kosztem wprowadzania kolejnej wytycznej wydaje się złą decyzją.

Szablon `SmartPtr` umożliwia określenie sposobu niszczenia wskazywanego obiektu za pomocą wytycznej `Ownership`. Możemy zatem zapewnić poprawne niszczenie tablicy (za pomocą `delete[]`). Szablon `SmartPtr` nie udostępnia jednak arytmetyki na inteligentnych wskaźnikach.

7.13. Inteligentne wskaźniki i wielowątkowość

Najczęściej inteligentne wskaźniki umożliwiają zarządzanie obiektami współdzielonymi. Wielowątkowość ma istotny wpływ na współdzielenie obiektów. Zatem wielowątkowość ma wpływ na funkcjonowanie inteligentnych wskaźników.

Interakcje między inteligentnymi wskaźnikami a wielowątkowością zachodzą na dwóch poziomach: na poziomie wskazywanego obiektu i na poziomie struktur pomocniczych.

7.13.1. Wielowątkowość na poziomie wskazywanego obiektu

Jeśli wiele wątków próbuje uzyskać dostęp do tego samego obiektu za pomocą inteligentnego wskaźnika, to być może warto zablokować ten obiekt podczas działania funkcji wywoływanych przez `operator->`. Można to zrealizować, zwracając pełnomocnika (ang. *proxy*) zamiast zwykłego wskaźnika. Konstruktor pełnomocnika blokuje obiekt wskazywany, a jego destruktor go odblokowuje. Technikę tę opisał Stroustrup (2000b). Pokazujemy ją również tutaj.

Po pierwsze, rozważmy klasę `Widget` o dwóch funkcjach składowych, realizujących blokowanie: `Lock` i `Unlock`. Po wywołaniu `Lock` możemy bezpiecznie korzystać z obiektu. Wywołanie `Unlock` umożliwia innym wątkom zablokowanie obiektu.

```
class Widget
{
  ...
  void Lock();
  void Unlock();
};
```

Teraz definiujemy szablon `LockingProxy`. Jego zadanie polega na zablokowaniu danego obiektu (za pomocą `Lock`/`Unlock`) na czas życia obiektu `LockingProxy`.

```
template <class T>
class LockingProxy
{
public:
  LockingProxy(T* pObj) : pointee_ (pObj)
  { pointee_->Lock(); }
  ~LockingProxy()
  { pointee_->Unlock(); }
  T* operator->() const
  { return pointee_; }
private:
  LockingProxy& operator=(const LockingProxy&);
  T* pointee_;
};
```

Poza konstruktorem i destruktorem szablon `LockingProxy` definiuje `operator->` zwracający wskaźnik do wskazywanego obiektu.

Chociaż LockingProxy już wygląda jak inteligentny wskaźnik, dokładamy jeszcze jedną warstwę – szablon SmartPtr.

```
template <class T>
class SmartPtr
{
  ...
  LockingProxy<T> operator->() const
  { return LockingProxy<T>(pointee_); }
private:
  T* pointee_;
};
```

Przypomnijmy, że w podrozdziale 7.3 omawialiśmy działanie operatora operator-> stosowanego przez kompilator wielokrotnie, aż do otrzymania zwykłego typu wskaźnikowego. Przyjmijmy teraz, że klasa Widget definiuje funkcję składową DoSomething i rozważmy następujący kod:

```
SmartPtr<Widget> sp = ...;
sp->DoSomething();
```

SmartPtr<Widget>::operator-> zwraca tymczasowy obiekt typu LockingProxy<T>. Kompilator znowu stosuje operator->. LockingProxy<T>::operator-> zwraca Widget*. Kompilator, posługując się tym wskaźnikiem, realizuje wywołanie funkcji składowej Widget::DoSomething. Podczas realizacji tego wywołania żywy jest obiekt tymczasowy klasy LockingProxy<Widget>, co oznacza, że obiekt klasy Widget jest bezpiecznie zablokowany. Zaraz po powrocie z funkcji składowej DoSomething obiekt tymczasowy LockingProxy<Widget> jest niszczony, skutkiem czego obiekt Widget jest odblokowywany.

Automatyczne blokowanie jest dobrym przykładem zastosowania nawarstwiania inteligentnych wskaźników. W szablonie SmartPtr inteligentne wskaźniki można nawarstwiać przez modyfikację wytycznej Storage.

7.13.2. Wielowątkowość na poziomie struktur pomocniczych

Czasami inteligentne wskaźniki poza obiektem wskazywanym utrzymują pewne struktury pomocnicze. Zgodnie z podrozdziałem 7.5 inteligentne wskaźniki z licznikami referencji niejawnie współdzielą pewne dane – liczniki referencji. Kopiowanie takiego inteligentnego wskaźnika między wątkami sprawia, że mamy dwa wskaźniki korzystające z tego samego licznika referencji. Oczywiście współdzielą one także wskazywany obiekt, ale obiekt ten jest dostępny dla użytkownika, który może go zablokować. Liczniki referencji są jednak przed użytkownikiem ukryte. Mogą nimi zarządzać wyłącznie inteligentne wskaźniki.

Zresztą nie tylko wskaźniki z licznikami referencji są narażone na niebezpieczeństwa związane z wielowątkowością. Inteligentne wskaźniki z listami referencji (p. 7.5.4) przechowują wskaźniki do siebie nawzajem i to też są dane dzielone. Listy referencji prowadzą do utworzenia zbiorowości wskaźników, ale członkostwo tych zbiorowości nie musi pokrywać się z przynależnością do wątków. Z tego powodu każda operacja kopiowania,

przypisywania i niszczenia inteligentnych wskaźników z listami referencji wymaga odpowiedniego blokowania. W przeciwnym razie struktura listy dwukierunkowej może zostać zniszczona. Wielowątkowość w istotny sposób wpływa na implementację inteligentnych wskaźników. Zobaczmy, jak radzić sobie z wielowątkowością w implementacjach inteligentnych wskaźników z licznikami i listami referencji.

7.13.2.1. Wielowątkowe zliczanie referencji

Kopiowanie inteligentnego wskaźnika między wątkami oznacza perspektywę zwiększania licznika referencji przez różne wątki w nieprzewidywalnym czasie.

Jak wyjaśniono w Dodatku, zwiększanie wartości o jeden nie jest operacją atomową. W celu atomowego zwiększenia/zmniejszenia wartości całkowitych o jeden musimy korzystać z typu `ThreadingModel<T>::IntType` oraz funkcji `AtomicIncrement` i `AtomicDecrement`.

Tutaj wszystko się nieco komplikuje. Ściślej mówiąc, komplikuje się, gdy chcemy oddzielić zliczanie referencji od wątkowości.

W architekturze opartej na wytycznych zalecono, by rozkładać klasy na podstawowe elementy funkcjonalne i zamykać każdy z nich w oddzielnym parametrze szablonu. Byłoby ideałem, gdyby szablon `SmartPtr` określał wytyczne `Ownership` oraz `ThreadingModel` i za ich pomocą implementował swoją funkcjonalność.

Jednak w wielowątkowym zliczaniu referencji te aspekty są o wiele bardziej powiązane. Licznik musi być typu `ThreadingModel<T>::IntType`. Zamiast operatorów `operator++` i `operator--` musimy korzystać z funkcji `AtomicIncrement` i `AtomicDecrement`. Wątkowość i zliczanie referencji stapiają się w jedno, a rozdzielenie ich okazuje się bardzo kosztowne.

Najlepszym wyjściem jest włączenie wątkowości do wytycznej `Ownership`. Można wtedy utworzyć jej dwie implementacje: `RefCounting` i `RefCountingMT`.

7.13.2.2. Wielowątkowe listy referencji

Rozważmy destruktor inteligentnego wskaźnika z listą referencji. Wygląda zapewne tak:

```
template <class T>
class SmartPtr
{
public:
  ~SmartPtr()
  {
    if (prev_ == this)
    {
      delete pointee_;
    }
    else
    {
      prev_->next_ = next_;
      next_->prev_ = prev_;
    }
  }
  ...
```

```
private:
  T* pointee_;
  SmartPtr* prev_;
  SmartPtr* next_;
};
```

Kod destruktora wykonuje klasyczną operację usuwania elementu z listy dwukierunkowej. By uprościć i przyspieszyć implementację, posługujemy się listą cykliczną – ostatni element wskazuje na pierwszy. Dzięki temu nie musimy sprawdzać pustości wskaźników `prev_` i `next_`. W jednoelementowej liście cyklicznej wskaźniki `prev_` i `next_` są równe `this`.

Jeśli wiele wątków niszczy inteligentne wskaźniki połączone w jedną listę, to destruktor musi być atomowy (nieprzerywalny przez inne wątki). W przeciwnym razie inny wątek może przerwać wykonanie destruktora, na przykład między modyfikacją `prev_->next_` a modyfikacją `next_->prev_`. Wówczas ten wątek wykonywałby operacje na rozspójnionej liście.

Podobne rozumowanie dotyczy konstruktora kopiującego i operatora przypisania. Te funkcje muszą być atomowe, ponieważ wykonują operacje na liście właścicieli.

Co ciekawe, nie można się tu posłużyć blokowaniem na poziomie obiektów. W Dodatku wymieniono *blokowanie na poziomie klas* i *na poziomie obiektów*. Blokowanie na poziomie klas blokuje na czas operacji wszystkie obiekty danej klasy. Blokowanie na poziomie obiektów blokuje wyłącznie obiekt, którego dotyczy operacja. Pierwsza z tych strategii prowadzi do mniejszej zajętości pamięci (tylko jeden muteks na klasę), ale może być wąskim gardłem, jeśli chodzi o wydajność. Druga potrzebuje więcej pamięci (jeden muteks na obiekt), ale może okazać się szybsza.

Dla inteligentnych wskaźników z listami referencji nie można skorzystać z blokowania na poziomie obiektów, ponieważ operacja może dotyczyć nawet trzech obiektów jednocześnie: bieżącego (który jest dodawany lub usuwany z listy) oraz jego poprzednika i następnika na liście właścicieli.

Jeśli chcielibyśmy zaimplementować blokowanie na poziomie obiektów, to musimy pamiętać, że potrzebny jest jeden muteks na każdy blokowany obiekt, ponieważ na każdy wskazywany obiekt przypada jedna lista. Możemy dynamicznie przydzielać muteksy dla każdego obiektu, ale to niweczy główną zaletę list referencji w porównaniu z licznikami referencji. Listy referencji były atrakcyjne właśnie dlatego, że pozwalały uniknąć przydziału pamięci na stercie.

Możemy też zdecydować się na rozwiązanie intruzyjne: obiekt wskazywany przechowuje muteks, a inteligentny wskaźnik się nim posługuje. Istnienie solidnego i wydajnego rozwiązania alternatywnego – inteligentnych wskaźników z licznikami referencji – sprawia, że nie warto zajmować się rozwiązaniem intruzyjnym.

Reasumując, wielowątkowość ma istotny wpływ na inteligentne wskaźniki z licznikami referencji lub listami referencji. Bezpieczna wątkowo implementacja zliczania referencji wymaga atomowych operacji całkowitoliczbowych, a bezpieczna wątkowo implementacja list referencji – muteksów. Szablon `SmartPtr` udostępnia jedynie wielowątkowe zliczanie referencji.

7.14. Składamy generyczny inteligentny wskaźnik

To już prawie koniec! Zaczyna się najlepsza zabawa. Do tej pory traktowaliśmy każde zagadnienie osobno. Nadszedł czas, by zamknąć wszystkie decyzje w implementacji jednego szablonu `SmartPtr`.

Posłużymy się strategią opisaną w rozdziale 1: projektowaniem klas parametryzowanych wytycznymi. Każdy aspekt architektoniczny, który nie ma jednoznacznego rozwiązania, odsuwamy do wytycznej. Wytyczne stają się parametrami szablonu `SmartPtr`, który dziedziczy po swoich parametrach, tym samym umożliwia im przechowywanie stanu.

Wymieńmy stopnie swobody szablonu `SmartPtr`. Każdy stopień przekłada się na jedną wytyczną.

- *Wytyczna `Storage`* (podrozdz. 7.3). Domyślnie typem dowiązania jest `T*` (`T` jest pierwszym parametrem szablonu `SmartPtr`), typem wskaźnika jest również `T*`, a typem referencyjnym `T&`. Domyślnym sposobem niszczenia wskazywanego obiektu jest wywołanie operatora `delete`.
- *Wytyczna `Ownership`* (podrozdz. 7.5). Powszechnymi implementacjami są: głębokie kopiowanie, zliczanie referencji, listy referencji i kopiowanie niszczące. Zauważmy, że wytyczna `Ownership` nie zajmuje się samym niszczeniem wskazywanego obiektu; to leży w gestii wytycznej `Storage`. Wytyczna `Ownership` wyznacza jedynie *moment* niszczenia obiektu.
- *Wytyczna `Conversion`* (podrozdz. 7.7). W niektórych programach przydaje się automatyczna konwersja inteligentnego wskaźnika do wskaźnika weń opakowanego.
- *Wytyczna `Checking`* (podrozdz. 7.10). Określa, czy inicjator inteligentnego wskaźnika jest poprawny oraz czy wyłuskanie inteligentnego wskaźnika jest poprawne.

Innych zagadnień nie warto przesuwać do odrębnych wytycznych, gdyż istnieją dla nich rozwiązania optymalne:

- Operator pobrania adresu (podrozdz. 7.6) – najlepiej go nie przeciążać.
- Sprawdzanie równości i nierówności implementujemy w sposób opisany w podrozdziale 7.8.
- Porównania porządkujące (podrozdz. 7.9) pozostawiamy bez implementacji; Loki definiuje jednak specjalizację funktora `std::less` dla konkretyzacji `SmartPtr`. Użytkownik może zdefiniować `operator<`, wówczas biblioteka Loki za jego pomocą automatycznie zdefiniuje pozostałe operatory porównań porządkujących.
- Implementacja szablonu `SmartPtr` jest poprawna pod względem stałości (ang. *const-correct*) ze względu na stałość wskaźnika, stałość wskazywanego obiektu oraz ich kombinację.
- Nie zaimplementowano bezpośredniego wsparcia dla tablic, ale jedna z gotowych implementacji wytycznej `Storage` potrafi poprawnie niszczyć tablice za pomocą operatora `delete[]`.

Omówienie zagadnień projektowych związanych z inteligentnymi wskaźnikami sprawiło, że łatwiej je zrozumieć i łatwiej nimi zarządzać, ponieważ każde z zagadnień opracowaliśmy osobno. Byłoby dobrze zachować w implementacji ten rozkład i traktować po-

szczególne zagadnienia osobno, zamiast walczyć ze wszystkimi przejawami złożoności na raz.

Dziel i rządź – ta stara zasada pochodząca od Juliusza Cezara może okazać się pomocna nawet dziś przy implementacji inteligentnych wskaźników. (Założę się, że tego nie przewidział). Dzielimy problem na mniejsze klasy, zwane *wytycznymi*. Każda wytyczna implementuje tylko jedno zagadnienie. Szablon `SmartPtr` dziedziczy po wszystkich tych klasach, tym samym dziedziczy wszystkie ich cechy. To takie proste, a jednocześnie bardzo elastyczne rozwiązanie. Każda wytyczna jest argumentem szablonu, co oznacza, że można łączyć gotowe wytyczne albo tworzyć własne.

Zaczynamy od typu wskazywanego obiektu, po nim przekazujemy kolejne wytyczne. Otrzymujemy taką deklarację szablonu `SmartPtr`:

```
template
<
  typename T,
  template <class> class OwnershipPolicy = RefCounted,
  class ConversionPolicy = DisallowConversion,
  template <class> class CheckingPolicy = AssertCheck,
  template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

Kolejność wytycznych w deklaracji szablonu `SmartPtr` dobrano tak, by wcześniej występowały te, które użytkownicy modyfikują najczęściej.

W następnych podrozdziałach omówiono wymagania czterech zdefiniowanych przez nas wytycznych. Przyjęto, że wszystkie wytyczne muszą mieć semantykę wartości, czyli muszą definiować odpowiedni konstruktor kopiujący oraz operator przypisania.

7.14.1. *Wytyczna* `Storage`

Wytyczna `Storage` opisuje strukturę inteligentnego wskaźnika, definiując typy oraz zmienną składową `pointee_`, przechowującą dowiązanie do obiektu wskazywanego przez inteligentny wskaźnik.

Jeśli `StorageImpl` jest implementacją wytycznej `Storage`, a `storageImpl` jest obiektem typu `StorageImpl<T>`, wyrażenia podane w tabeli 7.1 muszą mieć opisaną tam semantykę.

Oto domyślna implementacja wytycznej `Storage`:

```
template <class T>
class DefaultSPStorage
{
protected:
  typedef T* StoredType;    // Typ dowiązania
  typedef T* PointerType;   // Typ wskaźnika
  typedef T& ReferenceType; // Typ referencyjny
public:
  DefaultSPStorage() : pointee_(Default()) {}
```

Tabela 7.1. Semantyka wytycznej Storage

Wyrażenie	*Semantyka*
`StorageImpl<T>::StoredType`	Typ reprezentujący dowiązanie do wskazywanego obiektu. Domyślnie `T*`.
`StorageImpl<T>::PointerType`	Zdefiniowany przez implementację typ wskaźnika. Wartość tego typu zwraca `operator->`. Domyślnie `T*`. Może różnić się od `StorageImpl<T>::StoredType`, jeśli konkretyzacja implementuje nawarstwianie (zob. podrozdz. 7.3, p. 7.13.1).
`StorageImpl<T>::ReferenceType`	Typ referencyjny. Jego wartość zwraca `operator*`. Domyślnie `T&`.
`GetImpl(storageImpl)`	Zwraca obiekt typu `StorageImpl<T>::StoredType`.
`GetImplRef(storageImpl)`	Zwraca obiekt typu `StorageImpl<T>::StoredType&`, stały (`const`), jeśli typ `storageImpl` ma modyfikator `const`.
`storageImpl.operator->()`	Zwraca obiekt typu `StorageImpl<T>::PointerType`. Stosowane w implementacji funkcji składowej `SmartPtr::operator->`.
`storageImpl.operator*()`	Zwraca obiekt typu `StorageImpl<T>::ReferenceType`. Stosowane w implementacji funkcji składowej `SmartPtr::operator*`.
`StorageImpl<T>::StoredType p;` `p = storageImpl.Default();`	Zwraca domyślną wartość wskaźnika (zazwyczaj zero).
`storageImpl.Release()`	Niszczy wskazywany obiekt.

```
    DefaultSPStorage(const StorefType& p) : pointee_(p) {}
    PointerType operator->() const { return pointee_; }
    ReferenceType operator*() const { return *pointee_; }
    friend inline PointerType GetImpl(const DefaultSPStorage& sp)
    { return sp.pointee_; }
    friend inline const StoredType& GetImplRef(const DefaultSPStorage& sp)
    { return sp.pointee_; }
    friend inline StoredType& GetImplRef(DefaultSPStorage& sp)
    { return sp.pointee_; }
  protected:
    void Release() { delete pointee_; }
    static StoredType Default() { return 0; }
  private:
```

```
    StoredType pointee_;
  };
```

Oprócz `DefaultSPStorage` biblioteka Loki mogłaby definiować także następujące wytyczne:

- `ArrayStorage` – w funkcji `Release` wykonuje `delete[]`,
- `LockedStorage` – korzysta z nawarstwiania w celu implementacji inteligentnego wskaźnika, który blokuje obiekt przy wyłuskaniu (zob. p. 7.13.1),
- `HeapStorage` – jawnie wywołuje destruktor, po czym zwalnia pamięć za pomocą funkcji `std::free`.

7.14.2. Wytyczna `Ownership`

Wytyczna `Ownership` musi wspierać implementację intruzyjnego i nieintruzyjnego zliczania referencji. Z tego powodu wytyczna korzysta z jawnych wywołań funkcji, a nie z technik opartych na konstruktorach/destruktorach (zob. Koenig 1996). Powodem jest fakt, że funkcje składowe można wywołać zawsze, a konstruktory i destruktory są wywoływane automatycznie, i to tylko w określonym czasie.

Implementacja wytycznej jest szablonem `Ownership`, sparametryzowanym typem wskaźnika. Szablon `SmartPtr` przekazuje typ `StoragePolicy<T>::PointerType` do implementacji wytycznej `Ownership`. Zauważmy, że parametr wytycznej `OwnershipPolicy` jest typem wskaźnika, a nie typem wskazywanego obiektu.

Jeśli `OwnershipImpl` jest implementacją wytycznej `Ownership` i `ownershipImpl` jest obiektem typu `OwnershipImpl<P>`, to wyrażenia podane w tabeli 7.2 muszą mieć opisaną tam semantykę.

Tabela 7.2. Semantyka wytycznej Ownership

Wyrażenie	*Semantyka*

`P val1;`
`P val2 = OwnershipImpl.`
`  Clone(val1);`
	Klonowanie obiektu. Może zmodyfikować wartość źródłową, jeśli `OwnershipImpl` implementuje kopiowanie niszczące.

`const P val1;`
`P val2 = ownershipImpl.`
`  Clone(val1);`
	Klonuje obiekt.

`P val;`
`bool unique = ownershipImpl.`
`  Release(val);`
	Oddaje prawo własności do obiektu. Zwraca `true`, jeśli nie ma innych referencji do obiektu.

`bool dc = OwnershipImpl<P>` `  ::destructiveCopy;`	Stwierdza, czy `OwnershipImpl` implementuje kopiowanie niszczące. Jeśli tak, to `SmartPtr` implementuje sztuczkę Colvina/Gibbonsa (Meyers 1999), jak w implementacji `std::auto_ptr`.

Implementacja wytycznej `Ownership`, definiująca zliczanie referencji wygląda następująco:

```
template <class P>
class RefCounted
{
  unsigned int* pCount_;
protected:
  RefCounted() : pCount_(new unsigned int(1)) {}
  P Clone(const P & val)
  {
    ++*pCount_;
    return val;
  }
  cool Release(const P&)
  {
    if (!--*pCount_)
    {
      delete pCount_;
      return true;
    }
    return false;
  }
  enum { destructiveCopy = false }; // Zob. dalej
};
```

Implementacja wytycznej dla zliczania referencji jest bardzo prosta. Napiszmy teraz implementację wytycznej `Ownership` dla obiektów COM. Obiekty COM implementują funkcje `AddRef` i `Release`. Przy ostatnim wywołaniu `Release` obiekt zostaje zniszczony. Musimy tylko podłączyć `Clone` do `AddRef` i `Release` do `Release`:

```
template <class P>
class COMRefCounted
{
public:
  static P Clone(const P& val)
  {
    val->AddRef();
    return val;
  }
  static bool Release(const P& val)
  {
    val->Release();
    return false;
  }
  enum { destructiveCopy = false }; // Zob. dalej
};
```

W bibiliotece Loki sformułowano następujące implementacje wytycznej `Ownership`:

- `DeepCopy` – opisana w punkcie 7.5.1, implementuje głębokie kopiowanie, zakłada istnienie funkcji składowej `Clone`.
- `RefCounted` – opisana w punkcie 7.5.3 i niniejszym podrozdziale.
- `RefCountedMT` – wielowątkowa wersja implementacji `RefCounted`.
- `COMRefCounted` – wariant intruzyjnego zliczania referencji opisany w niniejszym podrozdziale.
- `RefLinked` – opisana w punkcie 7.5.4.
- `DestructiveCopy` – opisana w punkcie 7.5.5.
- `NoCopy` – nie definiuje składowej `Clone`, uniemożliwia tym samym jakiekolwiek kopiowanie.

7.14.3. Wytyczna `Conversion`

Wytyczna `Conversion` jest prosta. Definiuje logiczną stałą statyczną, która określa czy inteligentny wskaźnik dopuszcza niejawne konwersje do zwykłego wskaźnika.

Jeśli `ConversionImpl` jest implementacją wytycznej `Conversion`, to wyrażenia z tabeli 7.3 muszą mieć opisaną tam semantykę.

Typ wskaźnika jest wyznaczany przez wytyczną `Storage` jako `StorageImpl<T>::StorageType`.

Jak można się spodziewać, biblioteka Loki zawiera dwie implementacje wytycznej `Conversion`:

- `AllowConversion` – dopuszcza niejawne konwersje;
- `DisallowConversion` – zabrania niejawnych konwersji.

Tabela 7.3. Semantyka wytycznej Conversion

Wyrażenie	*Semantyka*
`bool allowConv = ConversionImpl<P>::allow;`	Wartość `allow` określa, czy `SmartPtr` dopuszcza niejawne konwersje do typu opakowanego wskaźnika.

7.14.4. Wytyczna `Checking`

Zgodnie z omówieniem z podrozdziału 7.10 kontrola integralności inteligentnego wskaźnika jest wykonywana podczas inicjowania i przed wyłuskaniem. Mechanizm kontroli może korzystać z `assert`, wyjątków, inicjowania leniwego albo po prostu nie robić nic.

Wytyczna `Checking` działa na typie dowiązania (`StoredType`), określonym przez wytyczną `Storage` (a nie na typie `PointerType`), definicja wytycznej `Storage` znajduje się w punkcie 7.14.1.

Jeśli `S` jest typem dowiązania zdefiniowanym przez implementację wytycznej `Storage`, `CheckingImpl` jest implementacją wytycznej `Checking`, a `checkingImpl` jest obiektem typu `CheckingImpl<S>`, to wyrażenia z tabeli 7.4 muszą mieć opisaną tam semantykę.

Tabela 7.4. Semantyka wytycznej Checking

Wyrażenie	*Semantyka*
`S value;` `checkingImpl.OnDefault(value);`	Konstruktor domyślny `SmartPtr` wywołuje `OnDefault`. Jeśli `CheckingImpl` nie definiuje tej funkcji, to konstruktor domyślny jest statycznie zablokowany.
`S value;` `checkingImpl.OnInit(value);`	Konstruktor `SmartPtr` wywołuje `OnInit`.

`S value;`
`checkingImpl.`
`OnDereference(value);`
	Klasa `SmartPtr` wywołuje `OnDereference` tuż przed powrotem z wywołania `operator->` lub `operator*`.

`const S value;`
`checkingImpl.`
`OnDereference(value);`
	Klasa `SmartPtr` wywołuje `OnDereference` tuż przed powrotem z wywołania stałych wersji operatorów `operator->` i `operator*`.

W bibliotece Loki zdefiniowano następujące implementacje wytycznej `Checking`:

- `AssertCheck` – kontroluje poprawność wyłuskania, korzysta z `assert`;
- `AssertCheckStrict` – kontroluje poprawność inicjowania, korzysta z `assert`;
- `RejectNullStatic` – nie definiuje `OnDefault`, a więc konstruktor domyślny `SmartPtr` jest zablokowany;
- `RejectNull` – zgłasza wyjątek przy próbie wyłuskania wskaźnika pustego;
- `RejectNullStrict` – zgłasza wyjątek przy próbie utworzenia wskaźnika pustego;
- `NoCheck` – obsługuje błędy zgodnie z uświęconą tradycją C i C++, więc nie robi nic.

7.15. Podsumowanie

Gratulacje! To był jeden z najdłuższych i najtrudniejszych rozdziałów tej książki. Wiesz teraz o wiele więcej o inteligentnych wskaźnikach, a także dysponujesz konfigurowalnym szablonem `SmartPtr`.

Inteligentne wskaźniki naśladują zwykłe wskaźniki w składni i semantyce. Dodatkowo implementują całą gamę funkcjonalności, których zwykłe wskaźniki nie mają. Może to być zarządzanie własnością albo kontrola poprawności.

Idea inteligentnych wskaźników wykracza poza same wskaźniki i może być uogólniona do inteligentnych uchwytów, takich jak pełnomocnicy (uchwyty, które nie mają składni wskaźników, ale przypominają je ze względu na sposób implementacji dostępu do zasobów).

Inteligentne wskaźniki umożliwiają wygodną automatyzację zadań, które bardzo trudno realizować ręcznie. Dzięki temu są one niezbędnymi składnikami dobrze zbudowanego programu aplikacyjnego. Mogą przesądzić o sukcesie lub porażce projektu. Często granica ich stosowania oddziela programy poprawne od programów z wyciekami zasobów.

Z podanych powodów twórca inteligentnego wskaźnika powinien włożyć w implementację maksymalną dozę wysiłku i uwagi. Ta inwestycja z pewnością się zwróci. Rów-

nież użytkownicy powinni zrozumieć konwencje posługiwania się nimi i stosować się do nich.

Zaprezentowana implementacja inteligentnego wskaźnika rozkłada jego funkcjonalność na niezależne wytyczne, których implementacje można dowolnie łączyć za pomocą głównego szablonu klasowego `SmartPtr`. Jest to możliwe dzięki istnieniu precyzyjnych definicji interfejsów poszczególnych wytycznych.

7.16. `SmartPtr` w skrócie

- Deklaracja szablonu `SmartPtr`:

```
template
<
  typename T,
  template <class> class OwnershipPolicy = RefCounted,
  class ConversionPolicy = DisallowConversion,
  template <class> class CheckingPolicy = AssertCheck,
  template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

- `T` jest typem obiektu wskazywanego przez inteligentny wskaźnik. Może być typem podstawowym lub zdefiniowanym przez użytkownika. Dopuszcza się typ `void`.
- Pozostałymi parametrami (`OwnershipPolicy`, `ConversionPolicy`, `CheckingPolicy` i `StoragePolicy`) mogą być własne implementacje wytycznych lub implementacje dostarczone przez bibliotekę. Implementacje wytycznych zdefiniowane w bibliotece Loki opisano w punktach 7.14.1–7.14.4.
- `OwnershipPolicy` definiuje strategię zarządzania własnością. Można wybrać gotową implementację spośród `DeepCopy`, `RefCounted`, `RefCountedMT`, `COMRefCounted`, `RefLinked`, `DestructiveCopy` i `NoCopy`, opisanych w punkcie 7.14.2.
- `ConversionPolicy` blokuje (dopuszcza) niejawne konwersje do typu opakowanego wskaźnika. Niezależnie od tego do wskazywanego obiektu można zawsze uzyskać dostęp za pomocą `GetImpl`. Gotowe implementacje to: `AllowConversion` i `DisallowConversion` (p. 7.14.3).
- `CheckingPolicy` definiuje strategię zgłaszania błędów. Gotowe implementacje to: `AssertCheck`, `AssertCheckStrict`, `RejectNullStatic`, `RejectNull`, `RejectNullStrict` i `NoCheck` (p. 7.14.4).
- `StoragePolicy` określa szczegóły związane z przechowywaniem i dostępem do wskazywanego obiektu. Domyślną implementacją jest szablon `DefaultSPStorage`, który po skonkretyzowaniu typem `T` definiuje typ referencyjny jako `T&`, typ dowiązania jako `T*` i typ wartości zwracanej przez `operator->` jako `T*`. Inne gotowe implementacje to: `ArrayStorage`, `LockedStorage` i `HeapStorage` (p. 7.14.1).