C++. Styl i technika zaawansowanego programowania

Autor: James O. Coplien
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-322-6
Tytuł oryginału: Advanced C++ Programming Styles and Idioms
Format: B5, stron: 480

Zakładając znajomość podstaw języka C++ książka ta umożliwia programistom rozwinięcie zaawansowanych umiejętności programowania poprzez stosowanie styli i idiomów języka C++. Struktura książki zorganizowana jest wokół abstrakcji wspieranych przez język C++: abstrakcyjnych typów danych, kombinacji typów w strukturach dziedziczenia, programowania obiektowego i dziedziczenia wielokrotnego. W książce przedstawione zostają także te idiomy, które nie znajdują bezpośredniego wsparcia w języku C++, takie jak wirtualne konstruktory, obiekty prototypów i zaawansowane techniki odzyskiwania nieużytków.

Książka:

- Przedstawia zalety i potencjalne pułapki zaawansowanych technik programowania w języku C++.
- Sposoby efektywnego łączenia abstrakcji języka C++ ilustruje szeregiem krótkich, ale stanowiących wystarczający instruktaż przykładów.
- Dostarcza wielu praktycznych zasad wykorzystania języka C++ do implementacji rezultatów projektowania obiektowego.
- Omawia wszystkie właściwości edycji 3.0 języka C++, w tym zastosowanie szablonów w celu wielokrotnego wykorzystania kodu.
- Przedstawia istotne aspekty rozwoju złożonych systemów, w tym projektowanie bibliotek, obsługę wyjątków i przetwarzanie rozproszone.

Książka ta jest ważnym podręcznikiem dla każdego programisty aplikacji lub programisty systemowego posługującego się językiem C++.

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

Spis treści

Rozdział 9.

Emulacja języków symbolicznych w C++

Język C++ dysponuje mechanizmami umożliwiającymi definiowanie abstrakcyjnych typów danych i używanie ich w programowaniu obiektowym. Jednak elastyczność języków obiektowych wysokiego poziomu takich jak Smalltalk czy CLOS jest trudna w języku C++ tak blisko związanym z językiem C. Zarówno w języku C, jak i C++ nazwa zmiennej związana jest z *adresem* opisywanego przez nią obiektu i tym samym nie jest jedynie jego etykietą, która może zostać „odklejona" z jednego obiektu i przyporządkowana innemu. Silne powiązanie zmiennej z obiektem pozwala kompilatorowi zapewnić, że dana zmienna zawsze będzie używana z obiektem określonego typu. Właściwość ta pozwala generować bardziej efektywny kod i zapobiegać użyciu obiektów tam, gdzie nie były one spodziewane. Efektywność i zgodność typów osiąga się jednak kosztem elastyczności działania programu — na przykład zmienna zadeklarowana jako typu `float` nie może zostać użyta podczas działania programu dla obiektu typu `Complex`, chociaż oba typy są ze sobą zgodne pod względem zachowania.

Smalltalk i większość języków obiektowych bazujących na języku Lisp dysponuje dwiema właściwościami wykorzystującymi luźne powiązanie zmiennych i obiektów, które nie są bezpośrednio dostępne w języku C++, ale mogą zostać wyrażone za pomocą odpowiednich idiomów i styli programowania. Pierwszą z tych właściwości jest automatyczne zarządzanie pamięcią (zliczanie referencji lub zbieranie nieużytków). W językach symbolicznych, gdzie zmienne są jedynie etykietami obiektów, czas istnienia obiektów jest niezależny od opisujących je zmiennych. Środowiska programowania w językach symbolicznych używają specjalnych technik pozwalających odzyskać pamięć zajmowaną przez obiekty, do których nie istnieją już żadne odwołania w programie. W języku C++ możemy symulować takie rozwiązanie, adresując obiekty za pomocą wskaźników. Utworzony w ten sposób dodatkowy poziom dostępu do obiektów może zostać wykorzystany w celu automatyzacji zarządzania pamięcią, co zostało szczegółowo omówione w podrozdziałach 3.5 i 3.6.

Drugą istotną właściwością języków obiektowych wysokiego poziomu jest wysoki stopień polimorfizmu. Idiomy umożliwiające podobnie zaawansowany polimorfizm zostały omówione szczegółowo w podrozdziale 5.5. Zaawansowany polimorfizm umożliwia tworzenie bardziej elastycznych architektur systemów. Obiekty stają się słabiej powiązane i dlatego łatwiej jest nimi zarządzać.

Automatyczne zarządzanie pamięcią i zaawansowany polimorfizm stanowią o sile języków programowania opartych o Lisp, języka Smalltalk i innych języków programowania obiektowego o wywodzących się z tradycji programowania symbolicznego. Podobną elastyczność możemy także uzyskać w programach tworzonych w języku C++ w stopniu, na jaki pozwala nam emulacja wymienionych właściwości za pomocą odpowiednich idiomów. Elastyczność taka ma jednak swoja cenę — zawsze odbywa się kosztem szybkości działania programu i większego zapotrzebowania na pamięć. Również wykrywanie błędów wykonywane dotychczas przez system typów podczas kompilacji programu zostaje odroczone do momentu wykonania idiomów i w związku z tym zależy od spójności kodu kontrolującego zgodność typów w programie użytkownika, a nie od kompilatora. Doświadczenie projektanta pozwala osiągnąć w tej mierze wymagany kompromis poprzez wybór idiomów odpowiednich do potrzeb konkretnej aplikacji.

W bieżącym rozdziale omówione zostaną trzy rodzaje idiomów. Pierwszy z nich stanowi kontynuację koncepcji przedstawionych w poprzednich rozdziałach, a wspierających przyrostowy rozwój programu poprzez redukcję wpływu zmian. Przedstawiona zostanie postać kanoniczna tego idiomu, która stanowić będzie podstawę dla pozostałych dwóch rodzajów. Drugi z nich umożliwi przyrostową aktualizację programu za pomocą prostego środowiska czasu wykonania. Natomiast trzeci wykorzystywać będzie techniki automatyzacji zwalniania nieużywanych obiektów i odzyskiwania ich zasobów. Każdy z tych idiomów może być stosowany niezależnie bądź w połączeniu z pozostałymi idiomami.

Implementacja drugiego z wymienionych idiomów na dowolnej platformie wymaga sporo wiedzy i wysiłku, ponieważ zależy od szczegółów implementacji języka C++ związanych ze sposobem reprezentacji klas. Implementacje przedstawione w tym rozdziale oparte są na objaśnieniach do standardu ANSI języka C++ [1]. Zostały one uruchomione w środowisku AT&T USL C++ Compilation System Release 3 i powinny być przenośne do wielu środowisk przy wykorzystaniu kompilatorów innych producentów.

Należy zaznaczyć, że technik prezentowanych w niniejszym rozdziale nie należy traktować jako substytutów rozwiązań oferowanych w językach Smalltalk lub CLOS. Języki symboliczne oprócz elastyczności umożliwiającej przyrostowy rozwój programów posiadają również rozbudowane środowiska programowania wyposażone we własne, zaawansowane narzędzia obsługujące przyrostowy rozwój oprogramowania. Przedstawione tutaj rozwiązania pozwalają tylko w pewnym stopniu zbliżyć język C++ do tych możliwości, ale za cenę dodatkowej dyscypliny w kodowaniu i kosztem słabszej efektywności tworzonych programów. Zadaniem tego rozdziału jest wprowadzenie koncepcji wspierających możliwości przyrostowego rozwoju programów w języku C++ oraz umożliwiających elastyczną aktualizację aplikacji pracujących w trybie ciągłym. Przedstawione rozwiązania mogą również posłużyć jako model kodu generowanego

automatycznie przez narzędzia współpracujące z generatorem aplikacji lub kompilator języka wysokiego poziomu służącego do tworzenia elastycznych i interaktywnych aplikacji.

9.1. Przyrostowy rozwój programów w języku C++

Zadaniem przyrostowego rozwoju programów jest *szybkie* wprowadzanie zmian tak, by ciągłość procesu rozwoju programów nie była zakłócana procesem testowania nowych wersji. Szybkie iteracje wersji programu stanowią ważną technikę udoskonalania programu i kontrolę jego zachowań w świetle nowych wymagań. Koszt przyrostowej zmiany musi być przy tym niski, aby iteracje takie były efektywne.

Przyrostowość i projektowanie obiektowe

Idea przyrostowego rozwoju programów doskonale współgra z projektowaniem obiektowym. Hermetyzacja szczegółów implementacji wewnątrz klas sprawia, że stają się one naturalnymi jednostkami iteracji. Istnienie wspólnego protokołu umożliwiającego posługiwanie się wszystkimi klasami danej hierarchii dziedziczenia umożliwia łatwe dodawanie nowych klas. Choć wszystko to sprzyja przyrostowemu tworzeniu programów w języku C++, to jednak same iteracje ze względu na konieczność ponownej kompilacji kodu mogą okazać się zdecydowanie wolniejsze niż językach Smalltalk czy CLOS. Obecnie coraz częściej powstają zaawansowane środowiska programowania w języku C++, które, zrywając z tradycyjną technologią tworzenia oprogramowania, umożliwiają przyrostowy rozwój programów. Jednak technologia taka nadal nie jest jeszcze dostępna dla wielu platform. Na przykład elastyczność rozwoju lub aktualizacji pożądana jest najczęściej w systemach wbudowanych w pewne urządzenia pracujące poza kontekstem zawansowanych systemów operacyjnych i narzędzi programistycznych. Chociaż więc przedstawione w tym rozdziale rozwiązania w zakresie przyrostowego rozwoju programów nie będą posiadać takich możliwości jak oferowane przez zaawansowane środowiska programowania przyrostowego w języku C++, to jednak umożliwiać będą przyrostowy rozwój dla szerszego spektrum platform i systemów.

Redukcja kosztów kompilacji

Pierwszy krok na drodze do programowania przyrostowego w języku C++ musi polegać na redukcji kosztów wynikających z ponownej kompilacji kodu. Najbardziej efektywnym sposobem realizacji tego zadania będzie ograniczenie samej potrzeby ponownej kompilacji. Pomiędzy zmienną, jej typem i sposobem reprezentacji zachodzi w języku C++ silny związek ustalany w momencie kompilacji. Jeśli na skutek ewolucji programu zmieni się na przykład typ zmiennej, to kod posługujący się taką zmienną musi zostać ponownie skompilowany. Zmiana reprezentacji zmiennej na skutek zmiany jej typu może również spowodować przesunięcie adresów innych zmiennych i tym samym wymusić

ponowną kompilację jeszcze innych fragmentów kodu, które posługują się tymi zmiennymi. Na przykład jakakolwiek zmiana interfejsu klasy wymusza zawsze ponowną kompilację każdego kodu, który korzysta z *jakiegokolwiek* elementu tego interfejsu.

Większość rozwiązań służących ograniczaniu ponownej kompilacji kodu polega na tworzeniu pośredniej warstwy dostępu do symboli. Przykładami takich rozwiązań mogą być, na przykład, idiom koperty i listu (podrozdział 5.5) i jego pochodne, takie jak idiom przykładu omówiony w rozdziale 8. Idiomy przedstawione w bieżącym rozdziale bazują w znacznej mierze właśnie na idiomie przykładu.

Zapewnienie odpowiedniej elastyczności w obliczu zmian i przy minimalnym poziomie kompilacji odbywa się za cenę mniejszej efektywności działania programu wynikającej z zastosowania dodatkowych poziomów dostępu do jego symboli. Zgodnie z duchem języków symbolicznych rozwiązania takie oferują również słabszą kontrolę zgodności typów w stosunku do tradycyjnego programowania obiektowego w języku C++. Osiągnięcie odpowiedniego kompromisu możliwe jest przez wybór właściwych idiomów dla konkretnej aplikacji.

Redukcja kosztów konsolidacji i ładowania

Drugi krok na drodze ku przyrostowemu rozwojowi programów w języku C++ polega na redukcji czasu związanego z konsolidacją i ładowaniem kodu. *Konsolidacją* nazywamy etap tworzenia wykonywalnego pliku programu z relokowalnych plików wynikowych powstałych podczas kompilacji, natomiast *ładowanie* polega na umieszczeniu wykonywalnego kodu programu w pamięci w celu jego wykonania. W niektórych systemach etapy te traktuje się łącznie, a większość systemów wykonuje podczas nich wiązanie symboli z adresami.

Efektywność konsolidacji i ładowania jest szczególnie istotna w przypadku tworzenia złożonych systemów, dla którym również techniki obiektowe mają najwięcej do zaoferowania. Przyrostowa konsolidacja i ładowanie kodu nie jest silną stroną większości tradycyjnych systemów mikroprocesorowych, jednak wiele nowych wersji systemu UNIX oraz innych systemów umożliwia już przyrostową konsolidację programów. W rezultacie konsolidacji przyrostowej powstają zwykle mniejsze moduły wynikowe, a przede wszystkim umożliwia ona szybsze zmiany niż pełna konsolidacja.

Nawet wtedy, gdy konsolidacja jest wystarczająco szybka, wąskim gardłem może okazać się proces ładowania kodu. Jeśli inicjacja systemu trwa długo, to nawet przyrostowo konsolidowane zmiany wymagają sporo czasu dla każdej iteracji. Natomiast w przypadku, gdy kod może być ładowany przyrostowo do zainicjowanego i działającego programu, to wprowadzanie zmian odbywa się zdecydowanie szybciej.

Szybkie iteracje

Szybkie iteracje stanowią najefektywniejsze rozwiązanie na etapie poszukiwania docelowej architektury rozwiązania. Powstające w ten sposób tymczasowe prototypy służą głownie kontroli właściwego rozumienia aplikacji przez projektanta. Tworzenie

takich prototypów, przy założeniu pewnych ograniczeń związanych ze stabilnością powstającej struktury rozwiązania, może być samo w sobie osobną dziedziną w procesie rozwoju oprogramowania. Jeśli szybkie iteracje kodu są właściwie zarządzane, to mogą stanowić efektywną technikę rozwoju systemu. Natomiast jeśli dotyczą one za każdym razem zasadniczych interfejsów tworzonego systemu, to będą jedynie zwiększać entropię i systematycznie niszczyć strukturę systemu.

9.2. Symboliczna postać kanoniczna

Idiom symboliczny jest alternatywą ortodoksyjnej postaci kanonicznej zaprezentowanej w podrozdziale 3.1 (strona 50). Emulacja paradygmatu symbolicznego w języku C++ nie jest „ortodoksyjna", ale wymaga pewnych konwencji. Posługując się tą alternatywną postacią kanoniczną, możemy w języku C++ modelować wiele właściwości charakterystycznych dla symbolicznych języków programowania. Jednak forma ta traci nieco na zwartości wyrazu charakterystycznej dla ortodoksyjnej postaci kanonicznej.

Kiedy używać tego idiomu?

Idiom ten stosujemy, gdy pożądana jest elastyczność oraz przyrostowość charakterystyczna dla języków programowania symbolicznego. Idiom ten może być również używany jako szkielet interfejsu pomiędzy środowiskiem programowania w języku C++ a środowiskami programowania w językach symbolicznych. Idiomy i style przedstawione w tym rozdziale mogą zostać wykorzystane do budowy własnego środowiska tworzenia prototypów w języku C++, jeśli podczas tworzenia aplikacji będziemy stosować się do pewnych konwencji. Idiom ten znajduje również zastosowanie w przypadku systemów pracy ciągłej, umożliwiając ich aktualizację bez zatrzymywania oraz ewolucję w dłuższym horyzoncie czasowym.

Omawiana tu postać kanoniczna bazuje na koncepcjach w zakresie zarządzania pamięcią i polimorfizmu przedstawionych w poprzednich rozdziałach i uzupełnia je tak, by mogły obsługiwać przyrostowość. Do głównych aspektów postaci kanonicznej należą:

- Automatyczne zarządzanie pamięcią wykorzystujące klasy listu ze zliczaniem referencji (podrozdział 3.5).
- Likwidacja dostępu do obiektów za pomocą wskaźników przy jednoczesnym udostępnieniu zachowania obiektów charakterystycznego dla wskaźników (podrozdział 3.5).
- Wykorzystanie funkcji wirtualnych dla uzyskania elastyczności w zakresie ładowania i wykonania kodu.
- Wykorzystanie idiomu przykładu (rozdział 8.).

Symboliczną postać kanoniczną tworzy niewielka kolekcja klas bazowych, które wykorzystywane są do tworzenia klasy kopertowej i klasy listu dla danej aplikacji. Deklaracje klas bazowych umieszczone zostaną w globalnym pliku nagłówkowym *k.h* przedstawionym na listingu 9.1. Plik ten zawiera deklarację dwóch klas — `Top`, która jest klasą bazową dla klas kopertowych, oraz `Thing`, która służy jako klasa bazowa klas listu.

Listing 9.1. *Plik nagłówkowy k.h*

```
// plik nagłówkowy k.h
class Top {
public:
    // Obiekty tej klasy nie posiadają danych
    // oprócz __vptr dostarczanej przez kompilator.
    // Ponieważ wszystkie inne klasy powstają jako
    // pochodne klasy Top, to dla większości implementacji
    // pole __vptr będzie pierwszym elementem każdego obiektu.
    // Niektóre implementacje mogą wymagać innego mechanizmu
    // dostępu do __vptr. Dla idiomu symbolicznego od właściwości tej
    // zależy jedynie aspekt dynamicznego ładowania.
    virtual ~Top() { /* pusty */ }
    virtual Top *type() { return this; }
    // operator delete jest dostępny publicznie
    // ze względu na konieczność usuwania
    // aktualizowanych obiektów
    static void operator delete(void *p) {
        ::operator delete(p);
    }
protected:
    Top()  { /* pusty */ }
    static void *operator new(size_t l) {
        return ::operator new(l);
    }
};

typedef unsigned long REF_TYPE;

class Thing: public Top {
    // Wszystkie składowe dziedziczone po klasie Thing
    // Definiuje postać kanoniczną klas listu
public:
    Thing() : refCountVal(1), updateCountVal(0) { }
    virtual REF_TYPE deref() {       // zmniejsza licznik referencji
        return --refCountVal;
    }
    virtual REF_TYPE ref() {         // zwiększa licznik referencji
        return ++refCountVal;
    }
    virtual Thing *cutover(Thing*);  // funkcja aktualizacji klasy
    virtual ~Thing() { /* pusty */ } // destruktor
private:
    REF_TYPE refCountVal, updateCountVal;
};
```

Klasa `Thing` sama także jest klasą pochodną klasy `Top`, co pozwala zapewnić rozwiązaniu jednolitość i przypomina rozwiązanie stosowane w wielu językach symbolicznych polegające na istnieniu wspólnej klasy bazowej, od której wywodzą się wszystkie inne klasy. Zadanie klas `Top` i `Thing` przypomina pod tym względem rolę klas `Object`, `Class` i `Behavior` w języku Smalltalk, chociaż dokładne odwzorowanie pomiędzy tymi klasami nie jest ani oczywiste, ani przydatne.

Klasy te zapewniają elastyczność, wspierają zarządzanie pamięcią, aktualizację w trakcie wykonania oraz luźny model typów charakterystyczny dla języków symbolicznych. Każda z klas zostanie omówiona szczegółowo w następnych dwóch podrozdziałach.

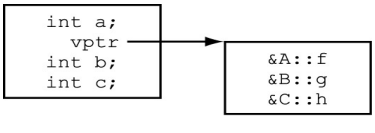
Klasa Top

Klasa `Top` znajduje się na szczycie hierarchii klas systemu. Wszystkie klasy systemu są wobec tego jej pochodnymi. Klasa `Top` nie posiada własnych, jawnych danych. Większość kompilatorów języka C++ umieszcza w jej obiektach jedynie niejawną składową wykorzystywaną do rozpoznania typu obiektu przez mechanizm funkcji wirtualnych. Składowa ta nosi nazwę `vptr` i jest wskaźnikiem elementu tablicy funkcji wirtualnych noszącej nazwę `vtbl`. Klasa Top posiada wirtualną metodę składową, aby wymusić obecność takiego wskaźnika.

Różne implementacje języka C++ mogą różnie implementować mechanizm funkcji wirtualnych, ale rozwiązania te różnią się co najwyżej w szczegółach. Rozważmy przypadek następujących trzech klas [1]:

```
class A {
public:
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};

class B : public A {
public:
    int b;
    void g(int);
};

class C : public B {
public:
    int c;
    void h(int);
};
```

W oparciu o powyższe deklaracje możemy spodziewać się, że reprezentacja obiektu klasy C w pamięci będzie wyglądać w następujący sposób:



Jeśli klasa znajdująca się na szczycie hierarchii nie posiada własnych danych, to wskaźnik `vptr` łatwo jest odnaleźć, ponieważ znajduje się na początku reprezentacji obiektu. Dysponując wskaźnikiem takiego obiektu, dowolna funkcja może uzyskać wartość wskaźnika `vptr` i użyć ją do przeglądania zawartości tablicy `vtbl` dla klasy obiektu. Możliwość ta jest kluczowa z punktu widzenia zastępowanie funkcji podczas działania programu.

Klasa `Top` posiada również domyślny (bez parametrów) konstruktor zadeklarowany w sekcji o dostępie protected, co zapobiega bezpośredniemu tworzeniu instancji tej klasy. Posiada również wirtualny destruktor, którego ciało nie zawiera żadnych instrukcji. Destruktor został zadeklarowany jako wirtualny, aby zapewnić wywoływanie destruktorów odpowiednich klas podczas wykonania programu.

Deklaracja operatora `new` również została umieszczona w sekcji `protected`, aby zapewnić, że obiekty klas kopertowych nie będą tworzone na stercie. Ograniczenie deklaracji obiektów klas kopertowych wyłącznie do obiektów lokalnych, globalnych lub składowych innych obiektów pozwala kompilatorowi całkowicie zautomatyzować ich usuwanie. Jeśli instancje klas kopertowych powinny być również tworzone na stercie, to zawsze istnieje możliwość przesłonięcia tej deklaracji w klasach pochodnych. Dynamiczny przydział i zwalnianie pamięci klas należących do hierarchii klasy listu odbywa się za pomocą operatorów klasy listu.

Funkcja składowa `type` jest przesłaniana przez klasy pochodne tak, by zwracała wskaźnik odpowiedniego przykładu. Jest on wykorzystywany w celu aktualizacji klasy w czasie działania programu, co zostanie omówione w dalszej części tego rozdziału.

Działanie klasy `Top` jest w znacznym stopniu zależne od implementacji kompilatora. Większość kompilatorów języka C++ bazuje na formacie obiektów opisanym powyżej, ale w ogólnym przypadku przeniesienie implementacji tej klasy do dowolnego środowiska może wymagać dodatkowych wysiłków.

Klasa Thing

Klasa `Thing` spełnia rolę klasy bazowej dla wszystkich klas listu. Ponieważ klasy listu zawierają zasadniczą część inteligencji danej aplikacji, to większość semantyki związanej z dynamiką obiektów znajduje się w publicznym interfejsie klasy `Thing`. W idiomie symbolicznym nawet pewna funkcjonalność związana z zarządzaniem pamięcią — która zwykle umieszczana bywa w klasie kopertowej — implementowana jest w klasach pochodnych klasy `Thing`.

Funkcje `deref` i `ref` operują na prywatnym liczniku referencji `refCountVal`. Zadeklarowane są jako funkcje wirtualne, aby klasy pochodne mogły je przesłonić. Jednak typowa aplikacja idiomu symbolicznego z reguły nie musi ich deklarować jako funkcji wirtualnych, a nawet może zadeklarować je jako funkcje rozwijane w miejscu wywołania. Funkcje te istnieją bowiem głównie dla wygody programisty. Prywatna składowa `updateCountVal` wykorzystywana jest podczas ładowania przyrostowego, a sposób jej użycia zostanie opisany w dalszej części rozdziału.

Funkcja `cutover` wykorzystywana jest do przekształcenia istniejącego obiektu danej klasy w obiekt reprezentujący inną wersję tej samej klasy. Umożliwia to konwersję danych istniejącego obiektu do nowego formatu, gdy do systemu zostaje wprowadzona nowa wersja klasy. Funkcja ta jest zwykle przesłaniana w klasach pochodnych, jeśli jest rzeczywiście używana. Jej domyślna semantyka polega jedynie na zwróceniu wskaźnika oryginalnego obiektu. Jej zastosowanie zostanie omówione w dalszej części rozdziału.

Wirtualny destruktor został zadeklarowany jedynie w celu zapewnienia, że dla obiektów klas pochodnych klasy `Thing` wykonywany będzie odpowiedni kod na skutek wywołania operatora `delete`.

Symboliczna postać kanoniczna klas aplikacji

Dysponując szkieletem złożonym z klas zadeklarowanych w pliku *k.h*, możemy scharakteryzować postać kanoniczną klas aplikacji wykorzystywaną przez idiom symboliczny. Będzie ona dotyczyć dwóch podstawowych rodzajów klas — kopert, które zarządzają tworzeniem i przypisywaniem obiektów, oraz listów, które zawierają zasadniczą semantykę aplikacji.

Klasa kopertowa może być związana z wieloma klasami listu. Załóżmy, że projekt określa typ `Number` jako typ bazowy dla typów `Double`, `BigInteger` i `Complex`. Tradycyjne rozwiązanie w języku C++ wykorzystujące dziedziczenie i funkcje wirtualne umożliwia wymienne posługiwanie się obiektami wymienionych klas za pomocą interfejsu klasy `Number`. Klasa `Number` będzie w nim abstrakcyjną klasą bazową dla pozostałych trzech klas, a instancje klasy `Number` nie będą występować. W rozwiązaniu opartym o idiom symboliczny użytkownik zachowuje możliwość wymiennego posługiwania się obiektami wymienionych trzech klas za pomocą interfejsu `Number`. Jednak klasa `Number` służy równocześnie za kopertę dla obiektów klas listu `Double`, `BigInteger` i `Complex`. Klasy te tworzone są jako klasy pochodne ogólnej klasy bazowej `NumericRep` charakteryzującej sygnaturę klas pochodnych. Klasa ta stanowi uogólnienie listu dla aplikacji numerycznej. Klasa kopertowa `Number` zawiera wskaźnik typu `NumericRep*`, który dotyczy obiektu listu. Ogólna klasa listu jest z kolei klasą pochodna klasy `Thing`, a klasa kopertowa (`Number`) jest pochodną klasy `Top`. Taka struktura tworzy odpowiednie warstwy pośrednie umożliwiające zaawansowany polimorfizm i aktualizacje podczas wykonania programu.

Zwróćmy uwagę, że ponieważ list jest klasą pochodną klasy `Thing`, a koperta klasą pochodna klasy `Top`, to nie możemy użyć wspólnej klasy bazowej dla listów i kopert, tak jak to bywało w poprzednich przykładach.

Symboliczną postać kanoniczną przedstawimy, posługując się ogólnym przykładem, w którym klasa `Envelope` będzie klasą kopertową, a klasa `Letter` będzie ogólną klasą bazową listów. Kompozyt złożony z pojedynczego obiektu klasy `Envelope` i pojedynczego obiektu klasy jednej z klas pochodnych klasy `Letter` stanowić będzie pojedynczą abstrakcję wykorzystywaną jako element programu stosującego idiom symboliczny,

Każdy program lub system może posiadać wiele klas kopertowych wykorzystujących postać kanoniczna klasy `Envelope` i wiele klas listu utworzonych na wzór klasy `Letter` i posiadających wyspecjalizowaną semantykę. Na przykład klasa `Number` może odpowiadać klasie `Envelope`, klasa `NumericRep` klasie `Letter`, a klasy `Complex` i inne będą pochodnymi klasy `NumericRep`. Ten sam program może również używać klasy `Shape` stworzonej na wzór klasy `Envelope` oraz klasy `ShapeRep` i jej pochodnych tworzących strukturę drzewiastą wzorowaną na hierarchii dziedziczenia klasy `Letter`. Chociaż klasy `Number` i `Shape` nie mają ze sobą nic wspólnego, to mogą istnieć w jednym programie, a każda z nich używa idiomu symbolicznego we własnym zakresie.

Klasa `Envelope` (listing 9.2) stanowi uogólnienie klasy zajmującej się obsługą operacji tworzenia i przypisywania obiektów (czyli w praktyce obsługuje wszelkie operacje kopiowania oraz zajmuje się kwestią przydziału pamięci). Często warto zastosować w jej przypadku ortodoksyjną postać kanoniczną (podrozdział 3.1) i uczynić ją w ten sposób konkretnym typem danych. Koperta przypomina nieco etykietę, która może być stosowana do różnych obiektów. Przypisanie oznacza wtedy powiązanie etykiety z obiektem. Usunięcie ostatniej etykiety obiektu jest równoważne z jego zwróceniem do puli obiektów nieużywanych.

Listing 9.2. *Klasa Envelope*

```
#include "k.h"        // z poprzedniego listingu

#include "9-3.h"      // funkcje składowe klasy Envelope

extern Thing *envelope, *letter; // wskaźniki przykładów

class Envelope: public Top {      // klasa Top zdefiniowana w k.h
public:
    Letter *operator->() const { // przekazuje wszystkie operacje
        return rep;              // obiektowi rep
    }
    Envelope()    { rep = letter->make(); }
    Envelope(Letter&);
    ~Envelope() {
        if (rep && rep->deref() <= 0) delete rep;
    }
    Envelope(Envelope& x) {
        (rep = x.rep)->ref();
    }
    Envelope& operator=(Envelope& x) {
        if (rep != x.rep) {
            if (rep && rep->deref() <= 0) delete rep;
            (rep = x.rep)->ref();
        }
        return *this;
    }
    Thing *type() { return envelope; }
private:
    static void *operator new(size_t) {
        Sys_Error("heap Envelope");
    }
    static void operator delete(void *) { }
    Letter *rep;
};
```

Klasa kopertowa zachowuje się jak abstrakcja o luźnym typie, a jej instancje symulują zmienne, które zachowują się jak etykiety, które nie posiadają typu, podobne do stosowanych w wielu językach symbolicznych. Na przykład funkcje składowe koperty nie przekazują szczegółowej semantyki obiektu listu, który zawiera koperta. Jednak koperta przyjmuje działanie przechowywanego obiektu klasy listu, posługując się mechanizmem operatora -> opisanym w podrozdziale 3.5 w taki sam sposób, jak zmienna języka symbolicznego przyjmuje zachowanie obiektu, do którego została „przyklejona”. W jaki sposób interfejs koperty przekazuje jednak wiedzę zawieranego obiektu klasy listu?

Odpowiedź na to pytanie tkwi w typie zwracanym przez operator ->. Typem tym jest Letter*. Zwrócony wskaźnik jest jednym z rzeczywiście niewielu wskaźników występujących w idiomie symbolicznym. Stanowi on jednak tylko przejściową wartość, która nie jest zwykle przechowywana do dalszego użytku.

Klasa kopertowa posiada konstruktory, ale zasadnicza inicjacja obiektu wykonywana jest przez funkcje wirtualne make klasy listu. Taki sposób działania jest korzystny z punktu widzenia przyrostowości i zostanie omówiony w dalszej części rozdziału. Konstruktory klasy kopertowej służą jedynie inicjacji oraz konwersji wykonywanej przez kompilator. Dwa z tych konstruktorów są charakterystyczne dla ortodoksyjnej postaci kanonicznej — konstruktor domyślny (nie posiada parametrów) oraz konstruktor kopiujący. Niezbędny jest także konstruktor tworzący nową kopertę dla instancji klas listu. Konstruktor ten dokonuje konwersji wyników wewnętrznych obliczeń klas listu na obiekty, które używane są przez klientów kompozytu złożonego z koperty i listu.

Konstruktor kopiujący, operator przypisania i destruktor modyfikują licznik referencji obiektu w sposób opisany w podrozdziale 3.5. Destruktor sprawdza, czy licznik referencji listu jest równy zero. Jeśli tak, to znaczy to, że usuwana jest ostatnia koperta posiadająca referencję tego listu i może być on usunięty.

Ostatnią z funkcji klasy Envelope, ale za to najważniejszą dla jej działania, jest operator ->, który automatyzuje przekazywanie wywołań funkcji składowych koperty do obiektu listu. Taki sam efekt uzyskalibyśmy, powielając w interfejsie koperty sygnaturę listu, a każda z funkcji składowych koperty przekazywałaby swoje działanie odpowiadającej jej funkcji listu. Jednak rozwiązanie takie wymaga dodatkowego wysiłku związanego z powieleniem funkcji listu w klasie kopertowej.

Klasa Letter (listing 9.3 i 9.4) definiuje interfejs wszystkich klas obsługiwanych za pośrednictwem interfejsu klasy Envelope. Klasa Letter sama jest klasą bazową, zwykle abstrakcyjną, dla grupy klas, których obiekty obsługiwane są za pośrednictwem interfejsu klasy Envelope. Jeden obiekt klasy Envelope może być wykorzystywany w cyklu swojego życia jako interfejs dla wielu różnych obiektów listu. Na przykład obiekt klasy Number może być początkowo interfejsem listu klasy Complex, jednak w następstwie wykonywanych obliczeń lub operacji przypisania obiekt listu może zostać zastąpiony obiektem listu innej klasy.

Listing 9.3. *Klasa Letter*

```
class Letter: public Thing {
public:
    /* deklaracje wszystkich operatorów definiowanych przez
     * użytkownika powinny znaleźć się w tym miejscu.
     * Ze względu na zastosowanie operatora ->
     * sygnatura tej klasy nie musi być powielana w klasie Envelope.
     * Jednak należy pamiętać, aby składowa rep klasy
     * Envelope była właściwego typu.
     * Operator przypisania definiowany jest w klasie Envelope.
     *
     * typ return_type zwracany przez deklarowane tutaj funkcje użytkownika powinien
     * być typem podstawowym, typem Envelope, typem Envelope&
     * lub konkretnym typem danych
     */
```

```
    virtual void send(String name, String address);
    virtual double postage();
    virtual return_type funkcja_użytkownika;
    . . . .
    virtual Envelope make();          // konstruktor
    virtual Envelope make(double);    // kolejny konstruktor
    virtual Envelope make(int days, double weight);
    virtual Thing *cutover(Thing*);   // funkcja aktualizacji podczas wykonania
    Letter() { }
    ~Letter() { }
    Thing *type();
protected:
    friend class Envelope;
    double ounces;
    static void *operator new(size_t l) {
        return ::operator new(l);
    }
    static void operator delete(void *p) {
        ::operator delete(p);
    }
    String name, address;
    . . . .
private:
    . . . .
};
```

Listing 9.4. *Funkcje składowe klasy Letter rozwijane w miejscu wywołania*

```
/*
 * Tutaj powinny zostać umieszczone wszystkie ogólne funkcje
 * rozwijane w miejscu wywołania. Jest to zgodne z konwencją
 * polegającą na umieszczaniu definicji funkcji rozwijanych w miejscu
 * wywołania poza deklaracją klasy. Rozwiązanie takie pozwala również
 * uniknąć cykli zależności pomiędzy funkcjami klas Envelope i Letter.
 */

inline double
Letter::postage() {
    if (ounces < 2) return 29.0;
    else return 29.0 + ((ounces - 1) * 23.0);
}

inline Thing *
Letter::type() {
    extern Thing *letter;     // przykład
    return letter;
}
```

Przyjmuje się, że obiekty należące do hierarchii klasy `Letter` znajdują się zawsze wewnątrz obiektu klasy `Envelope` i tylko ten obiekt widoczny jest dla użytkownika. Sygnatura klasy nie jest nigdy używana bezpośrednio przez użytkownika. Jednak funkcje składowe klasy `Letter` wykorzystywane są przez interfejs klasy `Envelope` za pośrednictwem operatora `Envelope::operator->`. Klasa `Letter` nie musi być konkretnym typem danych, ponieważ obsługiwana jest za pośrednictwem interfejsu klasy `Envelope`.

Klasa `Letter` służy jako klasa bazowa dla klas aplikacji zarządzanych przez klasę `Envelope`. Klasa `Envelope` zawiera składową `rep` wskazującą instancję klasy `Letter`. Rzeczywiste działanie kompozytu klas `Envelope` i `Letter` wykonywane jest właśnie przez obiekt jednej z klas pochodnych klasy `Letter`.

Wszystkie funkcje składowe aplikacji zostają wyspecyfikowane w interfejsie klasy listu, zwykle jako funkcje czysto wirtualne. Niektóre z funkcji, wspólne dla wszystkich klas pochodnych klasy `Letter` mogą zostać zdefiniowane już w tej klasie. Ponieważ pozostałe funkcje zdefiniowane są jako funkcje czysto wirtualne, to mamy gwarancję, że ich implementacji dostarczą klas pochodne. Jednak w dalszej części rozdziału pokażemy, że użycie funkcji czysto wirtualnych nie jest dopuszczalne w rozszerzonej formie idiomu wykorzystującej obiekty przykładów.

Funkcje definiowane przez użytkownika powinny zwracać obiekty typów wbudowanych w język lub konkretnych typów danych (czyli zgodnych z ortodoksyjną postacią kanoniczną) lub typu `Envelope` lub typu referencji `Envelope`. W sygnaturze klasy `Envelope` powinny pojawiać się co najwyżej stałe wskaźniki (`const`). Zwracanie zwykłych wskaźników do dynamicznie przydzielonych obszarów pamięci może bowiem naruszyć zaimplementowany schemat zarządzania pamięcią. Oczywiście funkcje definiowane przez użytkownika mogą być również typu `void`.

Funkcja `make` tworzy instancję klasy pochodnej klasy `Letter` i zwraca wskaźnik typu `Letter*`, co opisane zostało w rozdziale 8. W ogólnym przypadku istnieć może wiele przeciążonych funkcji `make`, przy czym każda z nich zajmuje się inicjacją nowego obiektu. Żadna operacja związana z inicjacją obiektu nie powinna być pozostawiana konstruktorowi. Na przykład funkcja `Letter::make` może inicjować obiekty klas `OverNight` i `FirstClass` w następujący sposób:

```
Envelope
Letter::make(int days, double weight) {
    Letter *retval;
    if (days < 2 && weight <= 12) {
        retval = new OverNight;
    } else {
        retval = new FirstClass;
    }
    retval->ounces = weight;
    return Envelope(*retval);
}
```

Jeśli konstruktor nie zawiera żadnej logiki związanej z inicjacją obiektów, to nie wymaga wprowadzania modyfikacji i tym samym ponownej kompilacji. Jest to istotne w środowisku przyrostowego ładowania kodu, w którym funkcje wirtualne (na przykład funkcje `make`) mogą być ładowane przyrostowo, a konstruktory nie.

W ogólnym przypadku klasy pochodne klasy `Letter` nie muszą stosować ortodoksyjnej postaci kanonicznej. Chociaż powinny posiadać konstruktor domyślny oraz destruktor, to jednak specyfikacja konstruktora kopiującego oraz operatora przypisania nie jest wymagana.

Obiekt klasy `Envelope` może zawierać obiekt dowolnej klasy pochodnej klasy `Letter`. Jeśli klasa `Envelope` jest poprawnie zaprojektowana, to dowolny obiekt tej klasy może być przypisany dowolnemu innemu obiektowi tej klasy. Listing 9.5 przedstawia przykłady prostych klas pochodnych klasy `Letter`. Każda z tych klas posiada własny konstruktor domyślny.

Listing 9.5. *Przykładowe klasy pochodne klasy Letter*

```
class FirstClass : public Letter {
public:
    FirstClass();
    ~FirstClass();
    Envelope make();
    Envelope make(double weight);
    . . . .
};

class OverNight : public Letter {
public:
    OverNight();
    ~OverNight();
    Envelope make();
    Envelope make(double weight);
    double postage() { return 8.00; }
    . . . .
};
```

Zgodnie z idiomem przykładu każda klasa kopertowa posiada pojedynczy, globalnie dostępny obiekt, który wykorzystywany jest jako przykład. Obiekt ten może być tworzony za pomocą specjalnego konstruktora w celu odróżnienia go od „zwykłych" obiektów tej klasy. Istnienie przykładów bywa często przydatne w przypadku klas listu tak, by przykład koperty mógł posiadać referencję instancji listu. Przykład listu obsługuje żądania utworzenia obiektów — wszystkie wywołania funkcji `make` przekazywane są do obiektu listu za pośrednictwem operatora `->` zdefiniowanego w klasie kopertowej. Przykład listu może być specjalną instancją ogólnej klasy bazowej listu (klasy `Letter`), jeśli nie jest ona klasą abstrakcyjną. W przeciwnym wypadku możemy utworzyć specjalną klasę pochodną, która posiadać będzie domyślne definicje funkcji czysto wirtualnych i tworzyć pojedynczy obiekt (singleton) przykładu listu.

Klasy posiadające symboliczną postać kanoniczną używane są w taki sam sposób jak zliczane wskaźniki i obiekty przykładów — czyli za pomocą operatora `->` zamiast operatora kropkowego. A oto przykład prostej aplikacji ilustrującej sposób posługiwania się naszymi wzorcowymi klasami `Envelope` i `Letter`:

```
static Envelope envelopeExemplar; // nie jest wykorzystywany bezpośrednio
Envelope *envelope = &envelopeExemplar;

int main() {
    Envelope overnighter = (*envelope)->make(1, 3.0);
    overnighter->send("Addison-Wesley", "Reading, MA");
    Envelope acrosstown = (*envelope)->make(1.0);
    overnighter = acrosstown;
    acrosstown->send("Angwantibo", "Boston Common");
    return 0;
}
```

W ten sposób omówiliśmy podstawowe aspekty symbolicznej postaci kanonicznej. Aby zaakcentować i poszerzyć przedstawione dotąd motywacje, przedstawiamy poniżej zbiór zasad związanych ze stosowaniem tego idiomu:

1. Wszystkie referencje klasy `Envelope` powinny wykorzystywać operator `->`, a nie zapis kropkowy. Operator `->` automatyzuje bowiem przekazywanie operacji do klasy `Letter`.
2. Funkcje składowe klas listu powinny być wirtualne. Wirtualne funkcje składowe mogą być łatwo ładowane przyrostowo, co zostanie opisane szczegółowo w dalszej części bieżącego rozdziału.
3. Funkcja składowa `make` wykonuje zadania konstruktora. Sam konstruktor nie wykonuje żadnych operacji związanych z inicjacją obiektów. Rozwiązanie takie jest konieczne, jeśli chcemy zachować możliwość zastępowania kodu inicjacji obiektów podczas wykonania programu, ponieważ jedynie funkcje wirtualne mogą być aktualizowane. Konstruktory są nadal obecne w klasach pochodnych klasy `Thing` (ich istnienie jest konieczne dla działania mechanizmu funkcji wirtualnych), ale nie powinny zawierać **żadnego** kodu definiowanego przez użytkownika.
4. Każda klasa powinna posiadać pojedynczy, stały obiekt przykładu. Obiekt przykładu musi być łatwy do zidentyfikowania (na przykład na skutek utworzenia go przez specjalny konstruktor). Dostęp do przykładu nie powinien odbywać się bezpośrednio, a jedynie za pomocą wyznaczonego w tym celu wskaźnika. Przyczyny takiego rozwiązania zostaną omówione w dalszej części rozdziału.
5. Funkcje składowe `cutover(Thing*)` klas pochodnych klasy `Thing` wykorzystywane są przez mechanizm dynamicznego ładowania kodu podczas wykonywania programu. Parametrem tych funkcji jest wskaźnik obiektu klasy, do której należą. Zadanie funkcji `cutover` polega na przekształceniu obiektu istniejącej klasy w obiekt nowej klasy różniącej się formatem, układem składowych i ich typem. Jeśli funkcja `cutover` nie potrafi tego dokonać, to może wykorzystać pewne sztuczki udostępniane przez środowisko, aby mimo wszystko dokonać konwersji obiektu (symulować jednokierunkową właściwość BECOMES dostępną w języku Smalltalk). Działanie funkcji `cutover` zostanie omówione szczegółowo w dalszej części rozdziału.
6. Operator `new` nie może być używany dla klasy `Envelope`, dlatego też zadeklarowany jest jako prywatny. Próba dynamicznego utworzenia obiektu klasy `Envelope` spowoduje błąd kompilacji. Koperty powinny być deklarowane jedynie jako zmienne automatyczne, składowe innych klas lub, w ostateczności, jako zmienne globalne. Wyeliminowanie wskaźników zwalnia programistę z obowiązku usuwania nieużywanych obiektów, dzięki czemu nawet elastyczne, polimorficzne typy w rodzaju klasy `Number` mogą być używane jak konkretne typy danych (czyli tym samym jak typy podstawowe wbudowane w język, np. typ `int`).

Idiom symboliczny wspomaga inżyniera oprogramowania, ponieważ zarządzanie pamięcią zostaje zautomatyzowane w znacznym stopniu przez klasę kopertową, która śledzi referencje obiektów. Dostęp do danych i funkcji odbywa się za pośrednictwem dodatkowej warstwy, dzięki czemu ich użytkownicy są mnie narażeni na wpływ pojawiających się zmian. Efekt domina wywołany przez modyfikacje klasy zostaje w znacznym stopniu ograniczony i tym samym ponowna kompilacja kodu wymagana jest na dużo mniejszą skalę. Jeśli dysponujemy odpowiednimi narzędziami konsolidacji i ładowania kodu, to technika taka może nawet zostać użyta do przeprowadzenia przyrostowych modyfikacji klas działającego programu, wymagając od niego jedynie zmiany konfiguracji uwzględniającej nową klasę.

Opisane rozwiązanie po raz kolejny zwiększa stopień polimorfizmu w programach tworzonych w języku C++, udostępniając typy parametryzowane podczas działania programu oraz pewien rodzaj ogólnej klasy. Idiom symboliczny stwarza iluzję, że charakterystyka typów może być zmieniana podczas wykonywania programu. Poniżej możliwość ta zostanie omówiona szczegółowo wraz z odpowiednimi przykładami.

9.3. Przykład — ogólna klasa kolekcji

Rozważmy przykład programu, który będzie używać zamiennie trzech różnych rodzajów kontenerów: opartego na tablicy i indeksach w postaci wartości całkowitych, wykorzystującego B-drzewa i stosującego tablice mieszające. W tym celu zdefiniujemy klasę `Collection`, która zawierać będzie wskaźnik do jednego z wymienionych obiektów wewnętrznych. Aby uzyskać dodatkową elastyczność, klasa `Collection` zdefiniowana zostanie jako szablon tak, by jego instancje mogły przechowywać obiekty dowolnego wybranego typu. Deklaracja:

```
Collection<Book, Author> library;
```

tworzy kolekcję obiektów klasy `Book` indeksowanych za pomocą obiektów klasy `Author`.

Obiekty listu w naszym przykładzie tworzone będą jako obiekty klas pochodnych klasy `CollectionRep`. Klasy te będą charakteryzować warianty (patrz podrozdział 7.7) klasy `CollectionRep`. W tym przypadku występować będą trzy warianty klasy `CollectionRep` — `Array`, `Btree` i `HashTable` — służące jako alternatywne rodzaje kontenerów wykorzystywane przez klasę `Collection`. Większość funkcji składowych klasy `CollectionRep` i jej klas pochodnych powinna być wirtualna, dzięki czemu wywołania funkcji składowych przez obiekt klasy `Collection` za pośrednictwem wskaźnika klasy `CollectionRep` będą powodować wykonanie funkcji składowych odpowiedniej klasy `Array`, `Btree` lub `HashTable`. Interfejs klasy `CollectionRep` musi zawierać deklaracje wszystkich funkcji składowych wymienionych klas. Klasa `Collection` dysponować będzie bowiem wyłącznie wiedzą o funkcjach składowych zadeklarowanych przez klasę `CollectionRep`. Ponieważ nie wszystkie klasy pochodne będą dysponować własną implementacją wszystkich tych metod, to przydatnym mechanizmem może okazać się obsługa wyjątków w sytuacji, gdy wywołana zostanie nieprawidłowa funkcja składowa. Na przykład dostęp do elementów klasy `Array` może odbywać się jedynie za pomocą indeksu przyjmującego wartości całkowite; tablice mieszające klasy `HashTable` mogą wykorzystywać

wartości całkowite podczas wyszukiwania indeksowego lub łańcuchy znakowe podczas wyszukiwania asocjacyjnego. Jeśli więc obiekt klasy `Collection` wykorzystuje obiekt klasy `Array`, to wywołanie `operator[] (S)` powinno spowodować wyrzucenie wyjątku. Jest to cena, jaką musimy zapłacić za elastyczność stylu programowania symbolicznego.

Listing 9.6 przedstawia szkielet deklaracji klasy `CollectionRep`, która jest klasą bazową dla klas listu w naszym przykładzie. Klasy należące do hierarchii dziedziczenia klasy `CollectionRep` używają dość niezwykłego sposobu implementacji funkcji składowej `type`. Ponieważ klasa `Collection` jest klasą parametryczną, to tworzenie na jej podstawie nowej klasy wymagać zawsze własnego przykładu (zazwyczaj administrowanego ręcznie), w skutek czego funkcja składowa `type` nie może zwrócić po prostu globalnej wartości wskaźnika. Zamiast tego operacja `make` danego przykładu umieszcza adres przykładu wewnątrz każdego z tworzonych obiektów (składowa `exemplarPointer`). I właśnie tę wartość zwraca funkcja składowa `type`. Na przykład:

```
class ClassDerivedFromCollectionRep<T, S>:
    public CollectionRep<T, S> {
    . . . .
    Collection<T, S> make() {
        Collection<T, S> newObject;
        . . . .
        newObject.exemplarPointer = this;
        return newObject;
    }
    . . . .
};

Thing *CollectionRep::type() { return exemplarPointer; }
```

Klasa `CollectionRep` sama w sobie stanowi użyteczną abstrakcję i może zostać bezpośrednio wykorzystana przez inny idiom. Logiczna hermetyzacja hierarchii klasy `CollectionRep` wewnątrz klasy `Collection` ma dwie zalety. Po pierwsze, pozwala klasie `Collection` zmieniać sposób reprezentacji kolekcji na żądanie podczas wykonywania programu. Możliwe jest także przypisanie kolekcji jednego typu kolekcji innego typu, dzięki czemu różne kolekcje mogą być praktycznie używane wymiennie. Kolekcje o znacznych rozmiarach mogą używać wartości progowych lub innych kryteriów, decydując się na zmianę swojego typu podczas działania programu w celu poprawy efektywności (na przykład zmieniając klasę `Array` na `HashTable`). Po drugie, klasa `Top` będąca klasą bazową klasy `Collection` używa atrybutów klasy `Thing` będącej klasą bazową klasy kopertowej w celu implementacji zarządzania pamięcią w sposób charakterystyczny dla idiomu przykładu.

Szkielet klasy `CollectionRep` przedstawiony na listingu 9.6 ilustruje omówione koncepcje. Dziedziczy on mechanizm zliczania referencji po swojej klasie bazowej `Thing`.

Listing 9.6. *Klasa bazowa klas listu wykorzystywanych przez klasę Collection*

```
#include "k.h"
#include "collection.h"

// Kolekcja elementów klasy T indeksowanych wartościami klasy S
```

```
template<class T, class S>
    class CollectionRep: public Thing {
    public:
        virtual Collection<T, S> make();
        virtual Thing* cutover(Thing *);
        virtual T& operator[](int);
        virtual T& operator[](S);
        virtual void put(const T&);
        CollectionRep() { }
        ~CollectionRep() { }
        Thing *type();
    protected:
        friend class Collection<T, S>;
        static void *operator new(size_t l) {
            return ::operator new(l);
        }
        static void operator delete(void *p) {
            ::operator delete(p);
        }
    private:
        CollectionRep<T, S> *exemplarPointer;
    };
```

Klasa `Collection` przedstawiona została na listingu 9.7. Jej zadania sprowadzają się do zarządzania pamięcią w podstawowym zakresie oraz obsługi operacji przypisania, a pozostałe operacje przekazywane są klasie listu. Klasa `Collection` może tworzyć lub wymieniać obiekty listu dowolnej klasy, opierając się na własnych kryteriach. Programista może wyposażyć ją również w dodatkowe rodzaje konstruktorów umożliwiające użytkownikowi wybór sposobu reprezentacji tworzonej kolekcji.

Listing 9.7. *Klasa Collection*

```
#include "k.h"

template<class T, class S> class CollectionRep;

template<class T, class S>
    class Collection: public Top {
    public:
        CollectionRep<T, S> *operator->() const { return rep; }
        Collection();
        Collection(CollectionRep<T, S>&);
        ~Collection();
        Collection(Collection<T, S>&);
        Collection& operator=(Collection<T, S>&);
        T& operator[](int i) { return (*rep)[i]; }
        T& operator[](S s) { return (*rep)[s]; }
    private:
        static void *operator new(size_t) { return 0; }
        static void operator delete(void *p) {
           ::operator delete(p);
        }
        CollectionRep<T, S> *rep;
    };
```

Klasy pochodne klasy `CollectionRep` przedstawione zostały na listingu 9.8. Każdy obiekt klasy `Collection` musi zawierać list klasy `Array`, `Btree` lub `HashTable`. Każda z klas pochodnych przesłania te funkcje klasy `CollectionRep`, które mają sens w jej przypadku, natomiast pozostałe operacje posiadają domyślną implementację w klasie `CollectionRep` sprowadzającą się do wygenerowania wyjątku. Zauważmy, że skoro klasy pochodne wybiórczo przesłaniają jedynie podzbiór funkcji klasy bazowej, to funkcje te nie mogą być zadeklarowane w klasie `CollectionRep` jako funkcje czysto wirtualne.

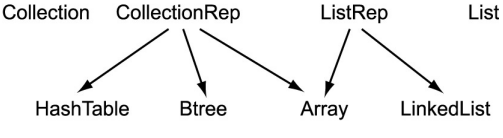
Listing 9.8. *Klasy implementujące kolekcje*

```
template<class T, class S>
    class Array: public CollectionRep<T, S> {
    public:
        Array();
        Array(Array<T, S>&);
        ~Array();
        class Collection<T, S> make();
        class Collection<T, S> make(int size);
        T& operator[](int i);
        void put(const T&);
    private:
        T *vec;
        int size;
    };

template<class T>
    struct HashTableElement {
        HashTableElement *next;
        T *element;
    };

template<class T, class S>
    class HashTable: public CollectionRep<T, S> {
    public:
        HashTable();
        HashTable(HashTable<T, S>&);
        ~HashTable();
        class Collection<T, S> make();
        class Collection<T, S> make(int);
        T& operator[](int i);
        T& operator[](S);
        void put(const T&);
    private:
        int nbuckets;
        virtual int hash(int l);
        HashTableElement<T> *buckets;
    };
```

Zarządzanie pamięcią dla obiektów tych klas wykorzystuje zwykle mechanizm zliczania referencji. Także i w tym przypadku klasy przedstawione w podrozdziale 3.5 stanowią dobry przykład zastosowania idiomu listu i koperty dla zliczania referencji. W podrozdziale tym przedstawione zostały sposoby manipulacji licznikiem referencji będącym składową obiektu listu klasy `StringRep` przez operator przypisania, konstruktor oraz destruktor oraz usuwania obiektu, gdy licznik osiąga wartość 0. Te same zasady

obowiązują w przypadku klasy `CollectionRep`. W podrozdziale 9.5 przedstawimy alternatywne rozwiązania dla mechanizmu zliczania referencji wykorzystywane podczas odzyskiwania nieużytków.

Przedstawione rozwiązanie może być efektywnie stosowane dla zwiększenia stopnia polimorfizmu — umożliwia na przykład utworzenie sumy dwóch kolekcji o dowolnym sposobie wewnętrznej reprezentacji. Jednak podczas jego stosowania na programistę czyha wiele pułapek. W szczególności, jeśli klasy używane w ten sposób zawierają operacje binarne (na przykład sumowanie dwóch kolekcji za pomocą operatora +), to muszą poradzić sobie z sytuacją, w której obiekty przekazane operacji mają różne typy (na przykład połączenie obiektów `Array` i `Btree`). Zaprojektowanie klas o takich możliwościach może być pracochłonne i wiąże się z dodatkowym narzutem związanym z koniecznością rozpoznawania typu obiektów i wywoływania odpowiednich operacji (patrz przykład klasy `Number` w podrozdziale 5.5 na stronie 140 oraz dyskusja w podrozdziale 9.7). Teoretycznie moglibyśmy zdefiniować niezbędne konwersje na poziomie klasy `Top` lub `Thing`, uzyskując w ten sposób pełen polimorfizm. Jednak definicje takich klas stałyby się bardzo skomplikowane i wymagałyby modyfikacji za każdym razem, gdy dodawana byłaby nowa operacja. Aby zdefiniować więcej niż jeden zewnętrzny interfejs, możemy użyć także dziedziczenia wielokrotnego (na przykład klasa `List` wykorzystująca reprezentację za pomocą klasy `Array` lub `LinkedList` i współdzieląca klasę `Array` z klasą `Collection`). Rysunek 9.1 przedstawia hierarchię klas dla takiego przypadku. Zwróćmy uwagę, że poziom komplikacji takiego rozwiązania również wzrasta bardzo szybko.



Rysunek 9.1. *Hierarchia klas powstała przez zastosowanie dziedziczenia wielokrotnego*

9.4. Kod i idiomy obsługujące mechanizm ładowania przyrostowego

Jeśli system dysponuje przyrostowym konsolidatorem, to często można również zaimplementować program ładujący pozwalający dodawać nowy kod do działającego programu. Listing 9.9 przedstawia prostą funkcję `load`, której parametrem jest nazwa pliku wynikowego zawierającego relokowalny kod uzyskany w procesie kompilacji. Funkcja ta ładuje ten kod i uruchamia go. Załóżmy, że plik wynikowy *incr.o* zawiera tylko jedną funkcję, a jego punkt wejściowy znajduje się na początku sekcji tekstowej. Możemy wtedy załadować i wykonać kod tej funkcji podczas działania programu za pomocą następującego wywołania:

```
int main() {
    typedef void (*PF) (...); // wskaźnik funkcji
    PF anewfunc = (PF) load("incr.o");
    (*anewfunc)();
    return 0;
}
```

Program ten będzie działał na większości platform firmy Sun Microsystems, przy czym musi zostać załadowany z opcją `-n`. Podobny kod można napisać dla większości dostępnych obecnie systemów operacyjnych.

Listing 9.9. *Funkcja ładująca plik wynikowy zawierający kod pojedynczej funkcji (platforma Sun)*

```
#include <a.out.h>
#include <fcntl.h>
#include <sys/types.h>

caddr_t load(const char *filename) {
    char buf[64];
    caddr_t oadx = (caddr_t)sbrk(0);
    caddr_t adx = ((char*)oadx) + PAGSIZ -
        (((long)oadx) % PAGSIZ);
    sprintf(buf, "ld -N -Ttext %X -A a.out %s -o a.out.new",
        adx, filename);

    system(buf);
    int fd = open(filename, O_RDONLY);
    exec Exec;
    read(fd, (char *)&Exec, sizeof(exec));
    sbrk(PAGSIZ - (((long)oadx) % PAGSIZ));
    caddr_t ldadx = (caddr_t)sbrk(Exec.a_text +
        Exec.a_data + Exec.a_bss);
    read(fd, ldadx, Exec.a_text + Exec.a_data);
    close(fd);
    return ldadx;
}
```

Kod ładowania funkcji możemy umieścić, na przykład, w interaktywnym programie w języku C++, umożliwiając użytkownikowi ładowanie nowych funkcji podczas działania programu. Gdy program znajduje się w stanie oczekiwania, możemy skompilować kod nowej funkcji, a następnie zażądać od programu załadowania jej relokowalnego kodu. Następnie program musi jeszcze wykonać dodatkowe działania w celu powiązania funkcji z istniejącym kodem, po czym może kontynuować swoje działanie.

Kiedy używać tego idiomu?

Idiomy przedstawione w tym podrozdziale opisują kod, który może być ręcznie lub półautomatycznie generowany przez środowisko umożliwiające przyrostowy rozwój oprogramowania. Styl ten może być wykorzystywany wszędzie tam, gdzie zachodzi potrzeba zmieniania programu podczas jego działania, a więc szczególnie podczas tworzenia prototypów. Idiom ten jest także przydatny do serwisowania skomplikowanych aplikacji pracujących w trybie ciągłym.

W kolejnych podrozdziałach przedstawione zostaną idiomy i struktury umożliwiające dynamiczne ładowanie kodu do działającego programu. Problem ten można podzielić w istocie na dwa podproblemy — możliwość ładowania nowych funkcji, która zostanie omówiona jako pierwsza, oraz możliwość konwersji formatów istniejących obiektów.

Ładowanie funkcji wirtualnych

Jednym z zastosowań kodu ładującego jest ładowanie nowych wersji funkcji do działającego programu. Jak pokazaliśmy przed chwilą, ładowanie funkcji jest dość łatwe pod warunkiem, że generujemy plik wynikowy zawierający wyłącznie nową funkcję. Aby powiązać istniejące wywołania funkcji z jej nową wersją, potrzebne są jednak pewne dodatkowe działania, które zostaną omówione poniżej.

W podrozdziale 9.2 omówiona zostanie tablica wskaźników funkcji wirtualnych, która dołączana jest do każdej klasy. Co więcej, obiekt przykładu dla dowolnej klasy posiada jako swój pierwszy element wskaźnik tej tablicy (`vptr`), co gwarantowane jest przez symboliczną postać kanoniczną. (Szczegóły mogą zależeć od implementacji kompilatora). Oto dlaczego wszystkie klasy kopertowe tworzone są jako pochodne klasy `Top` — klasa ta zawiera wskaźnik funkcji wirtualnej obiektu, który może zostać użyty do zaadresowania `vtbl`[1].

W środowisku języka C++ niezbędne jest podjęcie środków administracyjnych służących zapewnieniu, że program zawiera dokładnie jedną kopię tablicy funkcji wirtualnych dla każdej klasy (przy założeniu, że klasa ta nie powstała na skutek dziedziczenia wielokrotnego). Ten aspekt jest oczywisty w niektórych środowiskach języka C++, ale w innych wymaga specjalnej konfiguracji. Producent kompilatora powinien dostarczyć odpowiednich informacji umożliwiających wyjaśnienie tej kwestii.

Uchwyt tablicy deskryptorów funkcji wirtualnych to jednak ciągle za mało — musimy jeszcze ustalić, który element tej tablicy odpowiada funkcji, której kod zamierzamy zaktualizować. Nie będzie to specjalnie trudne w przypadku, gdy istnieje tablica opisująca odwzorowanie nazw funkcji na wartości indeksu tablicy `vtbl`. Niestety, dla większości kompilatorów języka C++ odwzorowanie takie nie istnieje. Dlatego też niezbędne jest wykonanie dodatkowych działań *na zewnątrz* programu, które przekażą kodowi ładującemu informacje niezbędne do zidentyfikowania właściwego elementu tablicy `vtbl`.

Jeden z możliwych sposobów polega na napisaniu prostej *funkcji pomocniczej programu ładującego*, której jedynym zadaniem jest zwrócić wartość będącą agregatem charakteryzującym funkcję, a w tym jej adres oraz indeks w tablicy deskryptorów funkcji wirtualnych. Funkcja taka może być zakodowana ręcznie lub generowana automatycznie podczas przygotowań do aktualizacji funkcji. Proces aktualizacji będzie więc odbywać się w dwóch etapach. Pierwszy polegać będzie na załadowaniu przez kod ładujący funkcji pomocniczej i wywołaniu jej w celu uzyskania indeksu tablicy `vtbl` dla aktualizowanej funkcji. Drugi etap polegać będzie na załadowaniu nowej wersji funkcji i umieszczeniu jej adresu w odpowiednim miejscu tabeli `vtbl`.

Zanim zajmiemy się samym procesem ładowania, musimy najpierw przyjrzeć się wykorzystywanym przez niego strukturom danych. Najpierw zdefiniujemy typ wskaźników funkcji:

```
typedef int (*vptp)();
```

[1] Rozwiązanie to działa jedynie dla dziedziczenia pojedynczego. Równie ogólne rozwiązanie dla dziedziczenia wielokrotnego jest trudne do opracowania i nie będzie tutaj omawiane.

Struktura `mptr` reprezentuje element tablicy funkcji wirtualnych dla większości implementacji języka C++:

```
struct mptr {
    short d;
    short i;
    vptp f;
};
```

Deklaracja taka wykorzystywana jest przez środowiska języka C++ bazujące na kompilatorze cfront i jest typowa także dla innych systemów. Jeśli jednak kompilator używa innej struktury, to kod przedstawiony w tym rozdziale musi zostać do niej dopasowany. Pierwsze dwa pola struktury posiadające typ `short` reprezentują wartości przesunięć wykorzystywane podczas dziedziczenia wielokrotnego i nie będziemy ich tutaj omawiać (patrz Ellis & Stroustrup [1]). Najbardziej interesuje nas pole `f`, które wskazuje funkcję dla danego elementu tabeli.

Implementacja funkcji pomocniczej jest prosta — jej zadanie polega za zwróceniu adresu bieżącej wersji funkcji, którą będziemy zastępować. Definicja funkcji pomocniczej może więc wyglądać w następujący sposób:

```
extern vptp functionAddress() {
    // kod zależny od platformy i kompilatora
    return (vptp)&Array::put;
}
```

Za każdym razem, gdy ładować będziemy nową funkcję, ładowana będzie najpierw nowa kopia funkcji pomocniczej `functionAddress`, która będzie nadpisywać poprzednią swoją wersję lub pozostawiać ją jako nieużytek w przypadku, gdy odzyskiwanie pamięci nie posiada wysokiego priorytetu.

Funkcja pomocnicza `functionAddress` potrafi rozwiązać niejednoznaczność związaną z ładowaniem funkcji o przeciążanym identyfikatorze, stosując odpowiednie rzutowanie. Załóżmy na przykład, że klasa `Array` posiada wiele funkcji `put`:

```
class Array {
public:
    . . . .
    void put(int, double);
    void put(int, int);
};
```

Jeśli po lewej stronie operatora przypisania umieścimy zmienną o odpowiednim typie, to operacja taka pozwoli wybrać wersję funkcji, na przykład o parametrze typu `double`:

```
extern vptp functionAddress() {
    // kod zależny od platformy i kompilatora
    typedef void ((Array::*TYPE) (int, double));
    TYPE retval = &Array::put;
    return (vptp)retval;
}
```

Następna grupa funkcji dodana do klasy `Top` będzie obsługiwać zadania przyrostowego ładowania i aktualizacji funkcji. Zadaniem pierwszej z tych funkcji, `compareFuncs` jest sprawdzenie, czy dwa deskryptory funkcji opisują tę samą funkcję:

```
int
Top::compareFuncs(int vtblindex, vptp vtblFptr, vptp fptr) {
    // kod zależny od platformy i kompilatora
    return vtblindex == (int)fptr;
}
```

Pierwsze dwa parametry tej funkcji przekazują informację o pozycji tabeli funkcji wirtualnych. Parametr typu `int` jest jej indeksem, a parametr typu `vptp` wartością wskaźnika funkcji znajdującą się na tej pozycji (wartością pola `mptr::f`). Trzeci z parametrów reprezentuje adres zastępowanej funkcji. Funkcja `compareFuncs` sprawdza, czy adres funkcji opisywanej przez dwa pierwsze parametry jest równy adresowi przekazanemu za pomocą trzeciego parametru. Jeśli tak, to funkcja zwraca wartość różną od zera. Sposób wykorzystania parametrów funkcji zależy od konkretnej platformy i kompilatora. W naszym przykładzie wiemy, że kompilator zwraca indeks tabeli funkcji wirtualnej, jeśli dokonamy rzutowania adresu funkcji wirtualnej na typ `int`. Wobec tego wystarcza jedynie porównać wartości pierwszego i trzeciego parametru funkcji.

Drugą z funkcji jest `findVtblEntry`:

```
mptr *
Top::findVtblsEntry(vptp functionAddress) {
    // kod zależny od platformy i kompilatora
    mptr **mpp = (mptrr**) this;
    register mptr *vtbl = *mpp;
    for(int i = 1; vtbl[i].f; i++) {
        if (compareFuncs(i, vtbl[i].f, functionAddress) {
            return vtbl + i;
        }
   }
   return 0;
}
```

Funkcja ta poszukuje w tablicy funkcji wirtualnych dla danego obiektu (wskazywanej przez pierwsze słowo tego obiektu, które stanowi wartość wskaźnika `vptr`) pozycji odpowiadającej funkcji, którą zamierzamy zaktualizować (przekazanej jako parametr funkcji). Funkcja `findVtblEntry` zwraca wskaźnik odpowiedniej pozycji tablicy funkcji wirtualnych (`mptr`), jeśli została ona znaleziona.

W tym momencie pozostaje nam jedynie załadować i dowiązać nową funkcję wirtualną. Zadanie to wykonuje funkcja `Top::update`:

```
extern "C" vptp load(const char *);

void
Top::update(const char *prepname, const char *loadname) {
    vptp findfunc = load(prepname);
    mptr * vtbl = findVtblEntry((*findfunc) ());
    vtbl->f = load(loadname);
}
```

Parametrami tej funkcji są nazwy plików zawierających funkcję pomocniczą oraz funkcję aktualizowaną. Używając zdefiniowanej wcześniej funkcji `load` (której typ został zmieniony z `caddr_t` na `vptp`), odnajduje ona odpowiednią pozycję tablicy funkcji

wirtualnych i zastępuje znajdującą się tam wartość wskaźnika adresem załadowanej właśnie nowej wersji funkcji. Od tego momentu wszystkie wywołania funkcji dotyczą jej nowej wersji.

Trochę więcej wysiłku wymaga przystosowanie zaprezentowanego kodu do rozszerzenia klas o zupełnie nowe funkcje wirtualne (patrz ćwiczenia na końcu tego rozdziału). Rozwiązanie takie wymaga także bardziej zaawansowanych narzędzi zarządzania konfiguracją zapewniających poprawność semantyczną oraz umożliwiających przyrostowość. Dodanie nowej funkcji wirtualnej nie sprowadza się bowiem wyłącznie do rozszerzenia tablicy `vtbl`. Problem może polegać, na przykład, na dodaniu nowej funkcji wirtualnej, której nazwa przesłaniać będzie istniejącą dotąd funkcję globalną. W jaki sposób należy zarządzać w takiej sytuacji ponowną kompilacją i ładowaniem kodu?

Aktualizacja struktury klasy i funkcja cutover

Poszerzenie technik przyrostowej aktualizacji o możliwości zmiany struktury danych zwiększa w istotny sposób elastyczność środowiska rozwoju lub serwisowania produktu programistycznego. Zagadnienie to jest jednak bardziej skomplikowane niż aktualizacja funkcji, która swą prostotę zawdzięcza istnieniu pośredniego poziomu dostępu do takich funkcji, który nie istnieje w przypadku danych. Poziomem takim dysponuje natomiast idiom listu i koperty i w związku z tym możemy go użyć w celu modyfikacji struktury danych. W bieżącym podrozdziale przedstawione zostanie rozwiązanie umożliwiające przyrostową zmianę struktury danych w klasach listu. Ponieważ większość kodu aplikacji znajduje się właśnie w klasach listu, to przedstawiona tutaj technika znajduje zastosowanie w większości przypadków, które wymagają zmiany danych składowych klasy.

Zastanówmy się, co w praktyce oznacza konieczność modyfikacji struktury klasy, załadowania nowej wersji klasy bądź załadowania nowych funkcji składowych. Ze strukturą klasy jako taką nie jest związany żaden kod, ale wiedza o strukturze klasy rozproszona jest również w kodzie operacji tej klasy. W poprzednim podrozdziale pokazaliśmy, w jaki sposób można załadować nową wersję funkcji. Jednak załadowanie nowej wersji klasy nie sprowadza się jedynie do załadowania nowych wersji funkcji składowych, ponieważ struktury istniejących obiektów muszą zostać poddane konwersji do nowego formatu.

Aby umożliwić aktualizację klas, każdy obiekt przykładu musi śledzić obiekty, które tworzone są na jego podstawie. Dzięki temu może następnie dokonać ich konwersji, gdy struktura klasy listu ulega zmianie. Do śledzenia tworzonych obiektów przez obiekt przykładu wystarcza zwykła klasa `List` dostępna w bibliotece języka C++. Obiekt klasy `List` może zostać zadeklarowany jako statyczna składowa klasy `Exemplar`.

Użytkownik musi dostarczyć funkcji składowej `cutover`, która potrafi dokonać konwersji istniejących obiektów do nowej postaci, zachowując ich semantykę. Również ta funkcja może być ładowana przyrostowo. Wywołanie funkcji `cutover` po załadowaniu nowej klasy należy do obowiązków aplikacji. Aplikacja powinna wybrać odpowiedni moment wywołania funkcji `cutover`, tak aby system znajdował się w określonym stanie — na przykład wtedy, kiedy wiadomo, że żaden z rekordów aktywacji aktualizowanych funkcji nie jest otwarty.

Istnieje wiele sposobów implementacji funkcji `cutover`. Zadaniem poniższych przykładów jest przede wszystkim ilustracja zadań, które stoją przed taką funkcją.

Dodanie nowego pola do klasy

Załóżmy, że nasze zadanie polega na dodaniu nowego pola do klasy `HashTable`:

```
template<class T, class S>
    class HashTable: public CollectionRep<T, S> {
    public:
        HashTable();
        HashTable(HashTable<T, S>&);
        . . . .
        Thing *cutover();
    private:
        int nbuckets;
        virtual int hash(int l);
        HashTableElement<T> *buckets;
        HashTableElement<T> *overflow; // nowe pole
    };
```

Zauważmy, że nowa składowa została dodana *na końcu* klasy. Zaletą takiego rozwiązania jest to, że zazwyczaj kod skompilowany dla poprzedniego interfejsu klasy może nadal używać obiektów posiadających nowy interfejs. Technika taka może również uprościć algorytm funkcji `cutover`. Zauważmy przy tym, że nowa klasa `HashTable` przesłania także funkcję `cutover`.

Jeśli dotychczasowy układ składowych nie został zmieniony przez dodanie nowego pola, to implementacja funkcji `cutover` jest oczywista:

```
Thing *
HashTable::cutover() {
    HashTable *object = (HashTable*) this;
                      *retval = new HashTable;
    // kopiuje część pochodzącą z klasy bazowej
    (*(CollectionRep *)retval) = (*(CollectionRep *)this);
    // kopiuje składowe tej klasy
    retval->buckets = object->buckets;
    // inicjuje nowe pole
    retval->overflow = 0;
    // usuwa stare
    object->buckets = 0;
    delete object;
    // zwraca nowy obiekt
    return retval;
}
```

Powyższe zmiany dotyczą szablonu `HashTable`, w związku z czym wymagane jest ponowne utworzenie jego wszystkich instancji występujących w programie. Inaczej mówiąc, każda instancja szablonu wymaga osobnej aktualizacji, chociaż konwersja kodu źródłowego wykonywana jest tylko raz.

Zasadnicze zmiany reprezentacji klasy

Jeśli wewnętrzna struktury klasy ulega zmianie w zasadniczy sposób, to aktualizacja istniejących instancji wymaga bardziej zaawansowanych środków. Nowe instancje muszą zostać utworzone na podstawie istniejących instancji i w związku z tym funkcja `cutover` wymaga dostępu zarówno do starego, jak i nowego interfejsu klasy. Kopię starego interfejsu musimy zachować pod pewną tymczasową nazwą i w ten sam sposób zmienić wszystkie wystąpienia nazwy klasy w programie. Ponieważ funkcja `cutover` dysponuje starym i nowym interfejsem klasy, to sposób tworzenia nowych obiektów zależy wyłącznie od jej implementacji.

Jako przykład rozpatrzymy klasę `Point`:

```
class Point : public ShapeRep {
public:
    Shape make(double x, double y);
    void rotate(Shape& p); // obrót dookoła punktu
private:
    double x, y;
};
```

Zadanie nasze polegać będzie na zmianie klasy `Point` tak, by wykorzystywała wartości wyrażone w radianach używane przez pewien akcelerator graficzny. Wymaganie to nie zmienia interfejsu klasy, wystarczy jedynie poddać konwersji istniejące obiekty i system może działać dalej. Nowa deklaracja klasy wygląda następująco:

```
class Point : public ShapeRep {
public:
    Shape make(double r, double theta);
    void rotate(Shape& p); // obrót dookoła punktu
    Thing *cutover();
private:
    double radius;
    Angle theta; // tworzony na podstawi wartości typu double
};
```

Możemy dokonać rzutowania zawartości starego pliku nagłówkowego w kategoriach tymczasowej nazwy klasy, uzyskując przedstawiony poniżej interfejs. Zauważmy, że klasa `Point` została zadeklarowana jako klasa zaprzyjaźniona, aby funkcja `cutover` miała bezpośredni dostęp do jej składowych. W większości przypadków funkcja `cutover` może uzyskać potrzebne dane, posługując się publiczną sygnaturą starej klasy. Jednak po jej aktualizacji stare funkcje mogą nie być już dostępne!

```
class OLDPoint : public ShapeRep {
friend Point;
public:
    Shape make(double x, double y);
    void rotate(Shape& p); // obrót dookoła punktu
private:
    double x, y;
};
```

Następnie zaimplementujemy funkcję `cutover` dla nowe klasy:

```
Thing *
Point::cutover() {
    OLDPoint *old = (OLDPoint *) this;
    Point *newPoint = new Point;
    newPoint->radius = ::sqrt(old->x*old->x + old->y*old->y);
    if (::abs(old->x) < .000001) {
        newPoint->theta = ::atan(1) * 2;
    } else {
        newPoint->theta = ::atan2(old->y, old->x);
    }
    if (y<0) newPoint->theta += ::atan(1) * 2;
    return newPoint;
}
```

Koordynacja ładowania funkcji i konwersji obiektów

Aplikacja musi sama skoordynować czynności związane z ładowaniem nowego kodu i konwersją istniejących obiektów. Przede wszystkim musi „wiedzieć", kiedy wykonanie aktualizacji jest bezpieczne (aby uniknąć jej wykonania w trakcie istotnych operacji), ale także pobrać od użytkownika pewne dane, na przykład nazwy plików zawierających nowy kod. Kod koordynujący te czynności może zostać wygenerowany automatycznie dla wielu aplikacji. W bieżącym podrozdziale omówione zostaną pewne aspekty związane z projektowaniem takiego kodu. Aktualizację kodu prześledzimy na przykładzie znanej już klasy `Point`.

Po zakończeniu konwersji kodu źródłowego klasy `Point` wszystkie funkcje składowe tej klasy wymagają ponownej kompilacji ze względu na nową strukturę klasy. Dla każdej funkcji wirtualnej klasy `Point` musi zostać napisana funkcja pomocnicza programu ładującego, a istniejący przykład klasy `Point` wywołuje funkcję składową `Top::update`, aby załadować nowe funkcje. Funkcje te ładowane są po kolei, każda wraz z funkcją pomocniczą odpowiednią dla danego wywołania funkcji `Top::update`.

Załóżmy, że wszystkie funkcje zostały załadowane. Wśród nich znajduje się funkcja `cutover` umożliwiająca konwersję istniejących instancji do nowego formatu. Ponieważ wszystkie obiekty przykładów dysponują listą istniejących instancji danej klasy, zadanie konwersji istniejących obiektów klas pochodnych klasy listu nie jest skomplikowane. Pamiętajmy, że klasy listu zagnieżdżone są na poziomie koncepcji wewnątrz klasy kopertowej, dzięki czemu wiemy, że tylko jedna klasa obiektów kopert może odwoływać się za pomocą wskaźnika `rep` do dowolnego obiektu pochodzącego z hierarchii klasy listu. Przykład tej klasy kopertowej dysponuje listą instancji klasy i może sprawdzać po kolei, jaki typ listu one zawierają. Każdy obiekt listu, dla którego funkcja `type` zwróci wskaźnik do aktualizowanego przykładu listu, jest również kandydatem do aktualizacji. Oznacza to, że może on z kolei wywołać operację dla każdego z tych obiektów, aktualizując ich pole `rep` tak, by wskazywało przekształconą wcześniej instancję.

Załóżmy na przykład, że aktualizujemy klasę `Point`, która jest klasą pochodną klasy `ShapeRep`. Przykład klasy `Shape` posiada listę wszystkich istniejących obiektów klasy `Shape`. Dodatkowo wie także, że pole `rep` każdego z tych obiektów wskazuje pewien obiekt należący do hierarchii `ShapeRep`. Przeglądając każdą instancję `s` klasy `Shape`,

algorytm aktualizacji koncentruje się na instancjach, dla których spełniony jest warunek `s.rep->type() == point`, gdzie `point` jest wskaźnikiem przykładu klasy `Point`. Dla takich obiektów `s` algorytm zastępuję wartość `s.rep` wartością `s.rep->cutover`. W ten sposób wszystkie obiekty kopert zostaną zaktualizowane tak, by posiadały referencje obiektów nowej klasy utworzonych na skutek konwersji obiektów starej klasy.

W opisanym sposobie działania niezbędna jest pewna korekta. Jeśli wiele kopert współdzieli ten sam list, to stara i nowa wersja zostaną pomieszane i konwersja się nie powiedzie. Wymagana jest bowiem tylko jedna aktualizacja współdzielonego obiektu listu. W tym celu w każdym obiekcie klasy `Thing` należy umieścić licznik. Za każdym razem, gdy algorytm aktualizacji odwiedza obiekt tej klasy, musi sprawdzić, czy wartość tego licznika równa jest 0 (wartość początkowa licznika). Jeśli tak, to licznik otrzymuje wartość licznika referencji. Następnie wartość licznika zostaje zmniejszona o 1. Jeśli na skutek tej operacji wartość licznika równa jest 0, to wywoływana jest funkcja `cutover`. W przeciwnym razie obiekt jest pomijany. W ten sposób każdy współdzielony obiekt listu zostaje poddany konwersji dopiero podczas ostatnich odwiedzin przez algorytm aktualizacji.

Wspomniane operacje możemy umieścić wewnątrz nowej funkcji składowej `docutover`:

```
int
Thing::docutover() {
    if (!updateCountVal) {
        updateCountVal = refCountVal;
    }
    return !--updateCountVal;
}
```

Funkcja składowa `Shape::dataUpdate` przedstawiona na listingu 9.10 koordynuje aktualizację związaną ze zmianą danych w klasach pochodnych klasy bazowej klasy listu `ShapeRep`. Jej parametrem jest wskaźnik przykładu aktualizowanej klasy oraz wskaźnik przykładu, który zajmie jego miejsce. Po wykonaniu opisanego wyżej algorytmu aktualizacji konieczna jest jeszcze aktualizacja obiektu przykładu. W ten sposób program aktualizacja programu zostaje skompletowana i wykorzystuje on nową wersję klasy. Funkcja `dataUpdate` zakłada, że funkcje wirtualne nowej wersji klasy zostały załadowane już wcześniej.

Listing 9.10. *Przykład kodu nadzorującego pełen cykl aktualizacji klasy*

```
typedef Thing *Thingp;

void
Shape::dataUpdate(Thingp &oldExemplar,
                  const Thingp newExemplar) {
    Thing *saveRep;
    Shape *sp;
    for (Listiter<Shape*> p = allShapes;
        p.next(sp);  p++) {
        if (sp->rep->type() == &oldExemplar) {
     if (p->rep->docutover()) {
                saveRep = sp->rep;
                sp->rep = (ShapeRep*)sp->rep->cutover();
                delete saveRep;
```

```
            }
        }
    }
    saveRep = oldExemplar;
    oldExemplar = newExemplar;
    delete saveRep;
}
```

Ładowanie przyrostowe i autonomiczny konstruktor ogólny

Ładowanie przyrostowe staje się efektywnym narzędziem, jeśli połączymy je z autonomicznymi konstruktorami ogólnymi opisanymi w podrozdziale 8.3. Autonomiczne konstruktory ogólne umożliwiają specjalizowanym obiektom przykładów (na przykład `Number`, `Name`, `Punct`) rejestrować się w bardziej ogólnym przykładzie (`Atom`), zwanym *autonomicznym przykładem ogólnym*. Specjalizowane przykłady są zwykle pochodnymi klasy autonomicznego przykładu ogólnego. Autonomiczny przykład ogólny działa jak agent wszystkich zarejestrowanych w nim przykładów. Zbiór tych przykładów może zmieniać się podczas działania programu.

Posługując się ładowaniem przyrostowym, możemy załadować nową klasę pochodną podczas działania programu, utworzyć jej przykład i zarejestrować go w przykładzie klasy bazowej. Na przykład możemy dodać klasy `BinaryOp` i `UnaryOp` do parsera działającego systemu, utworzyć ich przykłady za pomocą odpowiednich konstruktorów i zarejestrować je w przykładzie klasy `Atom`. Od tego momentu system będzie umiał parsować wyrażenia zawierające operatory unarne i binarne wszędzie tam, gdzie przykład klasy `Atom` będzie używany do analizy składni.

9.5. Odzyskiwanie nieużytków

W większości przypadków języki programowania symbolicznego zwalniają programistę z obowiązku zarządzania pamięcią. Jeśli w programie przestają istnieć wszystkie referencje pewnego obiektu, to zajmowana przez niego pamięć zostaje automatycznie odzyskana podczas działania programu, z pomocą systemu operacyjnego i (lub) platformy sprzętowej. Mechanizm ten nosi nazwę *odzyskiwania nieużytków*. Odzyskiwanie nieużytków stwarza iluzję dysponowania nieskończenie pojemną pamięcią wobec czego programiści nie muszą pamiętać o usuwaniu obiektów, które nie są już potrzebne. Jeśli system wykryje, że dany obiekt nie jest już w żaden sposób wykorzystywany, to zajmowana przez niego pamięć zostaje odzyskana i może być przydzielona innym obiektom. Działanie takiego mechanizmu jest niewidoczne dla użytkownika programu.

Idiomy zliczania referencji przedstawione w podrozdziale 3.5 stanowią słabą formę mechanizmu odzyskiwania nieużytków. W szczególności idiom zliczanych wskaźników (strona 73) oferuje przezroczystość zarządzania pamięcią charakterystyczną dla środowisk programowania symbolicznego. Jednak algorytmy odzysku pamięci bazujące na zliczaniu referencji nie potrafią odzyskiwać pamięci w przypadku występowania cyklicznie referencyjnych struktur danych, chyba że wykorzystują kosztowne

techniki skanowania rekurencyjnego. Mechanizmy odzyskiwania nieużytków wykorzystywane w środowiskach programowania w językach wysokiego poziomu nie posiadają takiego ograniczenia i rzadko używają zliczania referencji. Idiom odzyskiwania nieużytków przedstawiony w tym podrozdziale stanowi alternatywę dla technik zliczania referencji.

Techniki, które nie wykorzystują zliczania referencji, posiadają również zalety z punktu widzenia zastosowań we wbudowanych systemach czasu rzeczywistego. W systemach tych wyjątkowe zdarzenia takie jak awaria procesora mogą spowodować przejście aplikacji w stan regeneracji, w którym odzyskanie pamięci może nie być łatwe. Jeśli proces działa błędnie i musi zostać usunięty, to może on nie zdołać wywołać odpowiednich destruktorów. Jeśli proces ten współdzieli obiekty, dla których zliczane są referencje z innymi procesami, to liczniki referencji tych obiektów nigdy nie osiągną wartości zero i zajmowana przez nie pamięć nie zostanie odzyskana. W takiej sytuacji wymiatanie okazuje się bardziej elastyczną techniką niż zliczanie referencji, ponieważ potrafi odzyskać zdecydowanie więcej nieużywanych zasobów.

Większość środowisk programowania symbolicznego ukrywa szczegóły mechanizmu odzysku nieużytków w implementacji kompilatora oraz środowiska wykonywania programów. Chociaż niektóre wczesne środowiska programowania w języku Smalltalk wykorzystują technikę zliczania referencji, to jednak programista aplikacji nie musi pisać w tym celu żadnego kodu. Stanowi to oczywisty kontrast z sytuacją w języku C++ przedstawioną w podrozdziale 3.5, gdzie zarządzanie pamięcią zostało zaimplementowane jako idiom języka C++ i nie jest wobec tego ukryte *wewnątrz* implementacji kompilatora i środowiska wykonania programu. Niektóre schematy odzyskiwania użytków wykorzystują specjalizowane możliwości sprzętu polegające na możliwości uzyskania informacji, czy dane słowo pamięci reprezentuje aktywny wskaźnik obiektu czy po prostu fragment danych. Niezależnie od tego, czy wykorzystywana jest implementacja kompilatora, możliwości systemu operacyjnego lub wyspecjalizowanego sprzętu, zarządzanie pamięcią w językach symbolicznych odbywa się na poziomie niższym od kodu źródłowego tworzonego przez programistę aplikacji. Taka przezroczystość jest właściwie domniemana zawsze, gdy pojawia się termin odzyskiwanie nieużytków. Ponieważ język C++ sam w sobie działa na dość niskim poziomie, to nie jest możliwe zupełne ukrycie w nim działania mechanizmu odzyskiwania nieużytków. Jednak stosując pewne konwencje oraz dostarczając kod, który tworzy środowisko odzyskiwania nieużytków, programy w języku C++ mogą osiągnąć wysoki stopień przezroczystości mechanizmu odzyskiwania nieużytków dla wybranych klas.

Technika, która zostanie tutaj przedstawiona, przypomina pod względem stopnia przezroczystości mechanizm zliczania referencji omówiony w podrozdziale 3.5, ale oddziela proces odzyskiwania pamięci obiektu od jego „odłączenia" od ostatniej referencji. Technika ta nie pozwala odzyskiwać pamięci w przypadku istnienia cyklicznych referencji. Jej efektywność jest nieco gorsza niż w przypadku zliczania referencji, ale jej złożoność zależy w dużym stopniu od sposobu użycia oraz szczegółów implementacji. Odmiany prezentowanej techniki umożliwiają przyrostowe odzyskiwanie nieużytków, dzięki czemu zasadnicze przetwarzanie nie wymaga zawieszania na zbyt długie okresy czasu we celu odzyskiwania pamięci. W przeciwieństwie do tradycyjnych rozwiązań

odzyskiwania nieużytków omawiana technika pozwala programiście powiązać z odzyskiwaniem zasobów obiektu pewne dodatkowe czynności — mechanizm odzyskiwania nieużytków może wywoływać destruktory odzyskiwanych obiektów.

Dotychczas opracowano szereg algorytmów odzyskiwania nieużytków. Jednym z pierwszych był algorytm „znacz i zamiataj" [2]. Jego działanie polegało na analizie wszystkich istniejących obiektów w celu wykrycia zawieranych przez nie referencji innych obiektów. W procesie tym wszystkie obiekty, do których istniały referencje, zostawały odpowiednio oznaczone. W kolejnym przebiegu usuwane były wszystkie obiekty, które nie zostały oznaczone w pierwszym przebiegu.

Kopiowanie półprzestrzeni jest techniką zapewniającą całkowite odzyskanie zasobów zajmowanych przez obiekty, do których nie istnieją już żadne referencje. Implementacją tej techniki jest algorytm Bakera [3]. Pozwala on uniknąć opóźnienia charakterystycznego dla metody „znacz i zamiataj" za cenę większych wymagań odnośnie pamięci. Algorytm Bakera dzieli pamięć na dwie połowy: A i B. Przestrzeń A zwana jest zwykle docelową, a B przestrzenią źródłową z powodów, które wyjaśnią się poniżej. Nowe obiekty tworzone są w przestrzeni A. W pewnym momencie (gdy przestrzeń A zapełni się obiektami lub system nie jest obciążony) wszystkie osiągalne obiekty przestrzeni A umieszczane są w sposób ciągły w przestrzeni B, przy czym wskaźniki tych obiektów są aktualizowane. Od tego momentu przestrzeń A nie zawiera już żadnych wykorzystywanych obiektów — jej zawartość stanowią same nieużytki. W ten sposób role przestrzeni A i B ulegają zamianie z każdym cyklem algorytmu.

Jednak algorytm Bakera (i większość technik odzyskiwania nieużytków, które nie wykorzystują zliczania referencji) wymaga rozróżnienia, czy dany obszar pamięci reprezentuje pewne dane czy też referencję (wskaźnik) obiektu. W języku C++ nie jest możliwe ustalenie, czy dane słowo pamięci jest wartością typu int czy wskaźnikiem. Ostatnio opublikowano nowe techniki odzyskiwania nieużytków bazujące na algorytmie „znacz i zamiataj", które umożliwiają *prawie* zupełne odzyskiwanie pamięci. Jedyny problem w ich przypadku stanowią przypadkowe *synonimy wskaźników* czyli wartości, których reprezentacja przypadkowo równoważna jest reprezentacji adresu pewnego obiektu, chociaż w rzeczywistości reprezentują pewną wartość całkowitą lub jeszcze coś innego. Jednak algorytmy te nie pozostawiają „wiszących" referencji pod warunkiem, że programista przestrzega pewnych zasad. Przykład takiego algorytmu przedstawia Caplinger [4].

Jednak poruszając się w świecie symbolicznych kopert i listów, możemy zidentyfikować wszystkie obiekty danej klasy kopertowej (wykorzystując listę tworzoną przez przykład tej klasy), co pozwoli odnaleźć wszystkie referencje dowolnej klasy listu. Jeśli zmienimy algorytm przydziału pamięci dla klasy listu tak, by używał stałej puli (podrozdział 3.6), to wtedy typy wszystkich obiektów w puli będą znane (są tego samego typu). Znamy również wszystkie adresy, pod którymi mogą znajdować się obiekty klasy listu, istnieje więc nadzieja, że możemy zaznaczyć wszystkie używane obiekty, a pozostałe usunąć. Ponieważ wszystkie obiekty istniejące w puli są takich samych rozmiarów, to nie musimy martwić się fragmentacją pamięci, natomiast obszar pamięci używany przez obiekty każdej z klas, dla której działa mechanizm odzyskiwania nieużytków, musi zostać określony przez programistę.

Kiedy używać tego idiomu?

Zastosowanie odzyskiwania nieużytków pozwala uwolnić użytkowników od problemu zarządzania pamięcią — na przykład podczas szybkiego tworzenia prototypów. Odzyskiwanie nieużytków może być także stosowane jako technika audytu zdarzeń w systemach czasu rzeczywistego. Pomaga ona zagwarantować, że pamięć zostanie odzyskana nawet w przypadku zaistnienia sytuacji wyjątkowej.

Rozwiązanie odzyskiwania nieużytków dla idiomu listu i koperty przedstawimy na przykładzie klasy `Triangle`. W tym celu bazowa klasa listu musi zostać uzupełniona o znacznik, który będzie kasowany, jeśli obiekt jest „osiągalny" podczas wykonywania algorytmu odzyskiwania. Obiekty tej klasy muszą także dysponować znacznikiem informującym, czy obiekt jest używany. Bit ten będzie używany przez operator `new` w celu znalezienia obszaru, który można przydzielić nowemu obiektowi. Początkowo wszystkie wspomniane znaczniki będą skasowane. Podczas tworzenia obiektu ustawiony zostanie znacznik użycia. Oprócz tych znaczników każdy obiekt będzie posiadać również znaczniki A i B odpowiadające sytuacji, w której obiekt znajduje się w przestrzeniach A i B algorytmu Bakera.

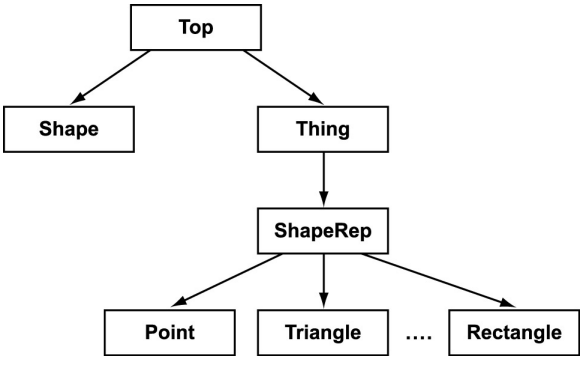
Przykład — figury geometryczne i odzyskiwanie nieużytków

W dodatku E przedstawiony został przykład pakietu umożliwiającego rysowanie figur geometrycznych, który używa idiomu symbolicznego przedstawionego w tym rozdziale. Przykład ten demonstruje możliwości ładowania przyrostowego omówione we wcześniejszej części rozdziału oraz używa zaawansowanej formy odzyskiwania nieużytków. Technika odzyskiwania nieużytków nie wykorzystuje w tym przypadku zliczania referencji, lecz utrzymana jest w duchu algorytmu „znacz i zamiataj" oraz algorytmu Bakera.

Algorytm odzyskiwania nieużytków sprawdza dla obiektów dostępnych na liście przykładu klasy `Shape`, czy ich składowa `rep` wskazuje obiekt klasy `Triangle` (przez zastosowanie funkcji składowej `type` i porównanie zwróconej przez nią wartości a adresem przykładu klasy `Triangle`). Dla każdego obiektu, który spełnia ten warunek, algorytm ustawia znacznik. Po zakończeniu tego przebiegu algorytm przegląda wektor o stałym rozmiarze (z którego przydzielane są obiekty klasy `Triangle`) w poszukiwaniu obiektów, które nie posiadają ustawionego znacznika. Jeśli obiekt taki posiada dodatkowo skasowany znacznik wykorzystania, to wywoływany jest jego destruktor i kasowany jest znacznik wykorzystania. Na końcu kasowany jest też znacznik używany przez algorytm odzyskiwania nieużytków.

Algorytm ten będzie wykonywany cyklicznie. Najpierw przetwarzać będzie wszystkie obiekty klasy `Triangle`, następnie obiekty klasy `Line`, potem obiekty klasy `Circle` i tak dalej dla wszystkich klas pochodnych klasy `ShapeRep`, po czym znowu wystartuje dla klasy `Triangle`. Na wyższym poziomie algorytm może być stosowany w ten sposób dla różnych aplikacji (hierarchii klas `Shape`, `Collection` etc.). Porządek wykonywania cykli określony jest zwykle przez środowisko wykonywania programu i może wymagać strojenia dla konkretnej aplikacji. Zastosowanie algorytmu Bakera umożliwia nawet przyrostowe odzyskiwanie nieużytków (patrz ćwiczenia na końcu tego rozdziału).

Przyjrzyjmy się szczegółom implementacji algorytmu zamieszczonej w dodatku E. Struktura klasy została już opisana powyżej, a hierarchia dziedziczenia przedstawiona jest na rysunku 9.2. Szczegóły implementacji znajdują się w klasach pochodnych klasy `ShapeRep`, których obiekty użytkownik używa jako instancje klasy `Shape`.



Rysunek 9.2. *Struktura klas Shape w idiomie symbolicznym*

Funkcja składowa `Shape::init` inicjuje globalne struktury danych. Inicjacja tych struktur odbywa się w jawny sposób w kodzie, a nie za pomocą domyślnych mechanizmów dostępnych w środowisku języka C++, co ma na celu lepszą kontrolę porządku inicjacji. Funkcja składowa `Shape::init` inicjuje najpierw dwie listy. Pierwsza z nich, `allShapes`, śledzi tworzenie instancji klasy `Shape`, a druga, `allShapeExemplars`, spełnia to samo zadanie dla obiektów klas pochodnych klasy `ShapeRep`.

Funkcja `Shape::init` wywołuje następnie operację `init` dla każdej z klas pochodnych klasy `ShapeRep`, które tworzą własne obiekty przykładów. Każdy przykład rejestruje się w klasie `Shape` za pomocą funkcji `Shape::register`, która umieszcza wskaźnik przykładu na liście `allShapeExemplars`. Właśnie dlatego istotne jest, aby listy klasy `Shape` zostały zainicjowane, zanim nastąpi inicjacja przykładów. Wadą takiego schematu inicjacji jest to, że podczas kompilacji klasy `Shape` muszą być znane wszystkie klasy pochodne klasy `ShapeRep`.

Po wykonaniu inicjacji użytkownik może zażądać utworzenia obiektu klasy `Shape`, posługując się idiomem autonomicznego konstruktora ogólnego:

```
Shape object = (*shape)->make(p1, p2, p3);
```

Operacja `make` wywoływana jest dla przykładu klasy `Shape` i zwraca obiekt utworzony na podstawie parametrów podanych przez użytkownika. Użytkownik przekazuje funkcji `make` zbiór punktów definiujących figurę geometryczną. Może również przekazać wskaźnik przykładu, jeśli sama liczba punktów nie wystarcza do określenia rodzaju figury. Para punktów może bowiem definiować prostokąt lub odcinek i w takiej sytuacji wskaźnik przykładu jest niezbędny, aby uniknąć niejednoznaczności. Domyślnym rodzajem figury tworzonym na podstawie trzech punktów jest trójkąt.

Zauważmy, że powyższe wywołanie funkcji `make` możemy również wyrazić w następujący sposób:

```
shape->operator->()->make(p1, p2, p3)
```

Przykład klasy Shape zwraca wskaźnik swojej wewnętrznej reprezentacji, który wskazuje instancję klasy ShapeRep jako obiekt, dla którego wywoływana jest metoda make. W przykładach idiomu listu i klasy przedstawionych w podrozdziale 5.5 zarządzanie pamięcią, w tym operacje make, obsługiwane było przez klasę kopertową. W obecnym przykładzie zarządzania pamięcią odbywa się z wnętrza listu. Ponieważ funkcja make jest funkcją wirtualną klasy listu, to nowe jej wersje mogą być ładowane przyrostowo w sposób opisany we wcześniejszej części tego rozdziału.

Powyższe wywołanie oznacza w efekcie wywołanie funkcji ShapeRep::make. Funkcja ShapeRep::make jest funkcją przeciążoną i posiada dwie wersje. Parametrami pierwszej z nich są współrzędne punktów reprezentujących figurę oraz wskaźnik przykładu dla danego typu obiektu. Druga wersja wykorzystuje jedynie współrzędne wierzchołków i na podstawie ich liczby zakłada określony typ figury. Implementacja tej wersji sprowadza się w rzeczywistości do wywołania pierwszej wersji ze wskaźnikiem przykładu odpowiedniego typu. W przypadku wywołania przedstawionego powyżej użyta zostanie druga z wymienionych wersji:

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2, Coordinate pp3) {
    return make(pp1, pp2, pp3, triangle);
}
```

Wersja ta wywołuje z kolei pierwszą wersję:

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2, Coordinate pp3, Thingp type) {
    return ((ShapeRep*)type)->make(pp1, pp2, pp3);
}
```

Wersja ta wywołuje następnie funkcję make klasy Triangle z parametrami opisującymi wierzchołki trójkąta:

```
Shape Triangle::make(Coordinate pp1, Coordinate pp2, Coordinate pp3) {
    Triangle *retval = new Triangle;
    retval->p1 = pp1;
    retval->p2 = pp2;
    retval->p3 = pp3;
    retval->exemplarPointer = this;
    return *retval;
}
```

Ta funkcja make tworzy najpierw nowy obiekt klasy Triangle za pomocą operatora new. Następnie składowe tego obiektu inicjowane są danymi opisującymi wierzchołki trójkąta. Składowa exemplarPointer inicjowana jest wartością this, która wskazuje w tym przypadku przykład klasy Triangle (this ma w tym kontekście taką samą wartość jak zmienna triangle, czyli globalny wskaźnik przykładu trójkąta). Kończąc swoje działanie, funkcja zwraca nowo utworzony obiekt. Instrukcja return powoduje wywołanie konstruktora Shape(ShapeRep&) w celu konwersji zwracanego obiektu do właściwego typu.

Gdy funkcja Triangle::make wywołuje operator new w celu utworzenia nowego obiektu klasy Triangle, to wywoływany jest własny operator new klasy Triangle:

```
void *Triangle::operator new(size_t nbytes) {
    if (poolInitialized - nbytes) {
        gcCommon(nbytes, poolInitialized, PoolSize, heap);
        poolInitialized = nbytes;
    }
    Triangle *tp = (Triangle *)heap;
    while (tp->inUse) {
        tp = (Triangle*)(((char*)tp) + Round(nbytes));
    }
    tp->gcmark = 0;
    tp->inUse = 1;
    return (void*) tp;
}
```

Funkcja `Triangle::operator new` zwraca wskaźnik niewykorzystywanego obiektu z puli obiektów klasy `Triangle`. Podczas pierwszego wywołania operatora zmienna `poolInitialized` posiada wartość 0 nadaną jej podczas kompilacji. W związku z tym porównanie jej ze zmienną `nbytes`, która ma wartość `sizeof(Triangle)`, spowoduje wywołanie funkcji `ShapeRep:gcCommon`. Funkcja ta używana jest do odzyskiwania nieużytków oraz w celu inicjacji pul obiektów podczas uruchamiania programu lub konwersji klasy. Funkcja ta między innymi nadaje wartości początkowe znacznikom obiektów znajdujących się w puli. Po wykonaniu funkcji `gcCommon` operator `new` przegląda pule obiektów klasy `Triangle` (wskazywaną przez składową `Triangle::heap`), poszukując obiektu posiadającego skasowany znacznik `inUse`. Obsługę sytuacji, w której w puli nie ma już takich obiektów, pozostawiamy do wykonania Czytelnikowi jako ćwiczenie.

Tworzenie wszystkich obiektów klasy `Shape` przebiega według tego samego wzorca. Inicjacja nowego obiektu klasy `Shape` na podstawie istniejącego obiektu lub przypisanie jednego obiektu innemu obiektowi powodują wywołanie odpowiedniej wersji konstruktora lub operatora przypisania:

```
Shape::Shape(Shape &x) {
    Thing tp = this;
    allShapes->put(tp);
    rep = x.rep;
}

Shape& Shape::operator=(Shape &x) {
    rep = x.rep;
    return *this;
}
```

W ten sposób wiele obiektów klasy `Shape` może dysponować wskaźnikiem tego samego, wspólnego obiektu jednej z klas pochodnych klasy `ShapeRep`.

Zgodnie z idiomem wskaźników zliczanych (strona 73) obiekty klasy `Shape` nie są tworzone na stercie. Dzięki temu nie musimy martwić się o zwalnianie zajmowanej przez nie pamięci, ponieważ każdy obiekt zadeklarowany jako zmienna automatyczna zwalnia swoją pamięć, gdy przestaje być dostępny. Podobnie każdy obiekt klasy `Shape`, który jest składową innego obiektu, zostaje automatycznie zwolniony podczas zwalniania zawierającego go obiektu. Globalne obiekty klasy `Shape` zostają usunięte podczas kończenia pracy programu, natomiast odzyskiwania nieużytków wymagają obiekty klas pochodnych klasy `ShapeRep`.

Odzyskiwanie nieużytków może zostać uruchomione w dowolnym momencie. W przykładzie zamieszczonym w Dodatku E wywoływane jest „ręcznie" w kilku punktach programu. Bardziej przezroczyste, choć zarazem bardziej kosztowne, rozwiązanie polegałoby na wywoływaniu odzysku nieużytków za każdym razem, gdy tworzony jest nowy obiekt. Jeszcze inna strategia może wywoływać odzyskiwanie nieużytków tylko w sytuacji, gdy operator `new` nie może znaleźć nieużywanych obiektów w puli. Poszczególne fazy odzyskiwania nieużytków mogą nawet zostać rozdzielone w czasie. Dotyczy to, na przykład, stosowania tego mechanizmu w celu przyrostowego odzyskiwania pamięci w systemach czasu rzeczywistego, w których zbyt długie przerwy związane z odzyskiwaniem nieużytków nie są wskazane (patrz ćwiczenia na końcu tego rozdziału). Odzyskiwanie nieużytków zarządzane jest przez funkcję `Shape::gc`, która sama wykonuje fazę oznaczenia obiektów, a fazę usuwania przekazuje poszczególnym obiektom listu:

```
void
Shape::gc() {
    Listiter<Topp> shapeIter = *allShapes;
    shapeIter.reset();
    for ( Topp tp = 0; shapeIter.next(tp);  ) {
        ((Shape*)tp)->rep->mark();
    }

    Listiter<Thingp> shapeExemplarIter = *allShapeExemplars;
    shapeExemplarIter.reset();
    for ( Thingp anExemplar = 0;
                shapeExemplarIter.next(anExemplar);  ) {
        ShapeRep *thisExemplar = (ShapeRep*)anExemplar;
        thisExemplar->gc(0);
    }
}
```

W pierwszej pętli funkcja `gc` przegląda wszystkie istniejące obiekty klasy `Shape` i ustawia ich znaczniki za pomocą funkcji `mark`. W ten sposób zaznaczone zostają wszystkie obiekty klas pochodnych klasy `ShapeRep`, dla których istnieją obiekty klasy `Shape` posiadające ich referencje. Druga z pętli odwiedza wszystkie przykłady klas listu — dla każdej klasy pochodnej klasy `ShapeRep` istnieje jeden przykład — i pozwala im wykonać fazę usuwania algorytmu odzyskiwania nieużytków.

Ponieważ każdy przykład klasy pochodnej klasy `ShapeRep` zarządza pulą obiektów, to zlokalizowanie i analiza istniejących instancji własnej klasy nie stanowi dla niego problemu. W przypadku klasy `Shape` pula jest ciągłym obszarem pamięci właśnie po to, by przykład tej klasy mógł łatwiej zarządzać jej instancjami.

Usuwanie obiektów wykonywane jest przez statyczną, wspólną operację klasy `ShapeRep` o nazwie `gcCommon` (listing 9.11). Funkcji tej przekazywana jest liczba bajtów przypadająca na obiekt danej klasy (`nbytes`), liczba bajtów zajmowanych przez każdy obiekt podczas poprzedniego przebiegu algorytmu odzyskiwania nieużytków (`poolInitialized`), liczba obiektów w puli (`PoolSize`) oraz wskaźnik puli (`heap`). Wartości tych parametrów możemy uzyskać, wywołując operacje `gc` dla klasy pochodnej. Wywołanie to może być wykonane przez dowolną operacje klasy `Shape` i nie wymaga żadnych parametrów.

Listing 9.11. *Faza usuwania obiektów określonego typu*

```
void
ShapeRep::gcCommon(size_t nbytes, const size_t poolInitialized,
        const int PoolSize, Char_p &heap) {
    size_t s = nbytes? nbytes: poolInitialized;
    size_t Sizeof = Round(s);
    ShapeRep *tp = (ShapeRep *)heap;
    for (int i = 0; i < PoolSize; i++) {
        switch (nbytes) {
        case 0:   // zwykły przypadek odzyskiwania nieużytków
            if (tp->inUse) {
                if (tp->gcmark || tp->space != FromSpace) {
                    // nie usuwa
                    tp->space = ToSpace;
                } else if (tp != tp->type()) {
                    // pamięć obiektu wymaga odzyskania
                    tp->ShapeRep::~ShapeRep();
                    tp->inUse = 0;
                    printf("ShapeRep::gcCommon ");
                    printf("Odzyskany obiekt klasy Triangle %c\n",
                     'A' + (((char *)tp-(char *)heap)/Sizeof));
                }
            }
            break;
        default:   // inicjacja
            tp->inUse = 0;
            break;
        }
        tp->gcmark = 0;
        tp = (ShapeRep*)(Char_p(tp) + Sizeof);
    }
}
```

Jak wspomnieliśmy już wcześniej, funkcja `ShapeRep` wykonuje zarówno zadanie inicjacji puli pamięci, jak i odzyskiwania pamięci. Logika mechanizmu odzyskiwania nieużytków umieszczona została w gałęzi instrukcji wyboru `switch` opatrzonej etykietą 0. Zachowuje ona zaznaczone obiekty lub obiekty znajdujące się już w docelowej przestrzeni w rozumieniu algorytmu Bakera. Sprawdzenie przynależności obiektu do określonej przestrzeni nie jest w tym wypadku zresztą istotne, ale staje się ważne w momencie, gdy odzyskiwanie pamięci działa w sposób przyrostowy. Pamięć obiektów, które nie są zaznaczone i znajdują się w przestrzeni źródłowej, jest odzyskiwana — znaczniki wykorzystania obiektów są kasowane, co przywraca je do puli używanej przez operator `new`.

Zauważmy, że w ten sposób powiązaliśmy zwalnianie zasobów obiektów (przez wywołanie destruktora) z odzyskiwaniem nieużytków. Zwalnianie zasobów obiektów jest działaniem, którego semantyka zależy od aplikacji. Natomiast odzyskiwanie nieużytków koncentruje się wyłącznie na jednym zasobie niezwykle istotnym dla większości aplikacji — pamięci. Rozdzielenie obu tych operacji jest możliwe (patrz podrozdział 3.7), ale oznacza obciążenie programistów koniecznością ręcznego usuwania niepotrzebnych obiektów, zanim zajmowana przez nie pamięć zostanie odzyskana jako nieużytek. Na przykład, jeśli obiekt przechowuje deskryptor otwartego pliku w systemie UNIX, to

samo odzyskanie pamięci zajmowanej przez taki obiekt nie wyczerpuje zagadnienia odzyskania zajmowanych przez niego zasobów. Dlatego też w naszym przykładzie obie operacje zostały połączone.

Z przedstawionym rozwiązaniem związane jest założenie, że użytkownik może pozwolić sobie na odroczenie odzyskania pamięci na dowolnie długi okres czasu. Aby zaktywizować proces odzyskiwania pamięci, użytkownik powinien wywołać funkcję odzyskiwania nieużytków natychmiast po usunięciu ostatniej referencji obiektu.

9.6. Hermetyzacja typów podstawowych

W większości środowisk programowania symbolicznego instancje wszystkich typów są obiektami, w przeciwieństwie do języka C++, gdzie pewne typy posiadające reprezentację zależną od typu maszyny — na przykład `int`, `char`, `long`, `short` i tak dalej — są typami podstawowymi wbudowanymi w język, a nie pełnoprawnymi klasami. Czasami przydatna byłaby możliwość posługiwania się wartościami tych typów jak obiektami — na przykład w celu stosowania dla nich mechanizmu automatycznego odzyskiwania nieużytków. To samo dotyczy także konkretnych typów danych czyli klas, które posiadają elastyczność użycia podobną do typów wbudowanych w język, ale nie posiadają właściwości typów symbolicznych.

Rozwiązanie tego problemu polega na obudowaniu każdego z typów podstawowych, którego zamierzamy używać w środowisku symbolicznym, za pomocą nowej klasy i tym samym utworzeniu biblioteki klas stanowiących odpowiedniki typów podstawowych. Chociaż rozwiązanie to wymaga pewnego nakładu pracy, to pod względem koncepcyjnym jest stosunkowo łatwe. Przykładem może być kolekcja klas pochodnych klasy `Number` przedstawiona w podrozdziale 3.5. Wprowadzając w tym przykładzie niewielkie zmiany związane z zastosowaniem idiomu listu i koperty — polegające głównie na utworzeniu wspomnianych klas jako pochodnych klas `Top` i `Thing` — uzyskamy w pełni polimorficzne klasy numeryczne, dla których może stosowany być mechanizm odzyskiwania nieużytków i których obiekty mogą być używane zamiast wartości typów podstawowych w przypadku bardziej tradycyjnych idiomów.

Główny problem związany z włączeniem typów podstawowych do idiomu symbolicznego polega na znalezieniu dla nich odpowiedniego miejsca w hierarchii dziedziczenia. Przykład klasy `Number` pomyślany został tak, by posiadał funkcjonalność wszystkich typów numerycznych, ale za cenę nierozróżniania wartości typu `double` od wartości typu `float`, wartości typu `char` od wartości typu `int` i tak dalej. Zastanówmy się na przykład, czy `String` powinien być samodzielnym typem symbolicznym, tak jak założyliśmy to w rozdziale 3., czy raczej powinien zostać, podobnie jak w języku Smalltalk, umieszczony w hierarchii pod klasami `ArrayedCollection`, `SequenceableCollection` i `Collection`? W rozdziale 6. podkreśliliśmy, że tego rodzaju decyzje należy podejmować wyłącznie na podstawie gruntownej analizy dziedziny aplikacji.

9.7. Wielometody i idiom symboliczny

Zastanówmy się przez chwilę, w jaki sposób działa operacja dodawania dwóch obiektów numerycznych, zwłaszcza w przypadku, gdy są one różnych typów. Tradycyjny sposób obsługi takiej sytuacji polega na zaaranżowaniu przez system typów kompilatora odpowiedniej konwersji typu. Rozwiązanie takie staje się nieaktualne, gdy typy obiektów powstają w czasie działanie programu. Dlatego też w przypadku programowania symbolicznego niezbędne jest inne rozwiązanie. Załóżmy, że każdy obiekt numeryczny dostępny jest dla programisty jako obiekt klasy `Number`, a klasy listu takie jak na przykład `Complex` czy `BigInteger` są niewidoczne. Obiekty klasy `Number` zmieniają swoją charakterystykę podczas działania programu i wobec tego kompilator nie dysponuje wystarczającą informacją, aby wygenerować kod konwersji. Konwersja taka, związana również z dodaniem odpowiednich operacji, musi zostać przeprowadzona podczas działania programu.

W jaki sposób dokonamy wyboru algorytmu dodawania dwóch wartości numerycznych? Jedno z rozwiązań polega na przekazaniu wykonania tej operacji pierwszemu jej argumentowi. Inaczej mówiąc, operator + zostanie wtedy wybrany na podstawie typu pierwszego argumentu. Sytuacja taka zachodzi zawsze w przypadku stosowania funkcji wirtualnych. Rozwiązanie takie obciąża jednak każdy typ wiedzą o wszystkich typach, których wartości mogą brać udział w dodawaniu. W ten sposób typy numeryczne tracą na swojej autonomii, zwartości i niezależności.

Zdecydowanie lepszym rozwiązaniem byłby wybór algorytmu dodawania w oparciu o typ obu argumentów. Funkcje wirtualne języka C++ nie dysponują taką możliwością, ale posiadają ją niektóre języki programowania symbolicznego. Funkcje lub operatory, których wybór odbywa się na podstawie typu wielu argumentów podczas działania programu nazywane są *wielometodami* w językach obiektowych bazujących na języku Lisp.

Symulacja wielometod w języku C++ wymaga stosowania idiomów wykorzystujących pole informujące o typie obiektu omówionych w rozdziale 4. W ogólnym przypadku stosowanie takich pól nie jest zalecane, ponieważ funkcje wirtualne stanowią preferowany mechanizm umożliwiający wybór wariantów implementacji dostępnych w klasach pochodnych. Jednak w tym przypadku funkcje wirtualne nie są odpowiednim rozwiązaniem, co zostanie pokazane poniżej.

W podrozdziale 5.6 (strona 169) przedstawiona została następująca definicja operatora klasy `LPF` reprezentującej filtry dolnoprzepustowe:

```
Value *
LPF::operator()(Value* f) {
    switch(f->type()) {
    case T_Data:
        myType = T_Data;
        cachedInput = f;
        return evaluate();
    case T_LPF:
        if (f->f1() > fa()) return this;
        else return f;
```

```
        case T_HPF:
            if (f->f1() > f1()) return new Notch(f1(), f->f1());
            else return new BPF(f1(), f->f1());
        case T_BPF:
            if (((Filter*)f)->f2() < f1()) return f;
            else return new BPF(f->f1(), f1());
        case T_Notch:
            cachedInput = f;
            return this;
        }
    }
```

Analizując implementację tego operatora, stwierdziliśmy wtedy, że mógłby on zostać zastąpiony kolekcją funkcji wirtualnych. Jednak liczba takich funkcji wzrastałaby szybko wraz z pojawianiem się nowych typów filtrów i w związku z tym uproszczenie kodu byłoby jedynie pozorne.

Semantyka funkcji `Filter::operator() (Value*)` wymaga zwrócenia przez jej implementację wartości lub obiektu reprezentującego jeden z wielu typów filtrów. Typ ten zależy od typu filtra, dla którego została wywołana ta funkcja, i od typu obiektu przekazanego jej jako parametr. Innymi słowy, o wyborze algorytmu i typie zwróconym jako wynik decyduje typ filtra wejściowego i typ filtra wyjściowego. W przykładzie przedstawionym w podrozdziale 5.6 logika tej operacji została rozproszona pomiędzy klasy pochodne klasy `Filter`. Jak pokazuje to powyższy fragment, kodu klasy `LPF` dotyczy klas reprezentujących filtry wyjściowe. Gdyby w przykładzie tym zastosować mechanizm funkcji wirtualnych, to funkcja `Filter::operator() (Value* input)` wywoływałaby po prostu funkcję składową filtra wejściowego i w związku z tym logika operacji należałaby do filtrów wejściowych.

Żadne z wymienionych rozwiązań nie jest w pełni satysfakcjonujące. Dlatego też zastosujemy rozwiązanie polegające na użyciu hybrydowych operacji, które nie należą do żadnej klasy. Rozwiązanie takie bazować będzie na funkcjach zaprzyjaźnionych i przeciążaniu globalnym omówionym w podrozdziale 3.3.

Stare rozwiązanie:

```
class Complex {
public:
    Complex operator+(Imaginary);
    operator double();
    . . . .
    . . . .
};
```

Nowe rozwiązanie:

```
Complex operator+(Complex c,
    Imaginary i) { . . . . }
class Complex {
friend Complex operator+(
    Complex, Imaginary);
public:
    operator double();
    . . . .
};
```

W przykładzie tym operator zostaje wybrany podczas kompilacji programu na podstawie *zadeklarowanych* typów argumentów. Nie istnieje bowiem bezpośredni sposób wyboru funkcji na podstawie rzeczywistych typów argumentów podczas wykonania programu. W tym celu musimy właśnie użyć wielometod.

Przyjrzymy się im najpierw na prostym przykładzie klasy `Number` dla operatora dodawania. Implementując wielometodę, `operator+` wykorzystamy przemienność operacji dodawania zmniejszając w ten sposób o połowę liczbę przypadków, które należy rozpatrzyć. Do rozpoznawania typu argumentów użyjemy funkcji wirtualnej `Top::type`. Chociaż jest to funkcja wirtualna, to jej zastosowanie odpowiada rozwiązaniu polegającemu na zastosowaniu pola informującego o typie obiektu przyjmującego wartości typu wyliczeniowego.

Pierwszą czynnością będzie dodanie nowej funkcji składowej do klasy `Number` i jej klas pochodnych: `isA(const Top *const)`. Funkcja ta będzie zwracać wartość logiczną `true`, jeśli wywołana zostanie dla obiektu klasy `A` z parametrem wskazującym obiekt klasy `B` i zachodzi związek `B` IS-A `A`. Zadaniem tej funkcji jest więc ustalenie, który z argumentów operatora dodawania jest bardziej ogólny i powinien przejąć kontrolę nad wykonaniem operacji.

Operatory konwersji występujące we wcześniejszych przykładach zastąpimy pojedynczą funkcją składową `promote`. Funkcja ta wykonywać będzie konwersje dowolnego obiektu typu `Number` do typu obiektu, dla którego została wywołana. Przy założeniu, że funkcja `promote` będzie wywoływana dla wszystkich niezbędnych konwersji, nie jest już konieczne przeciążanie operacji dodawania dla każdej klasy kopertowej. Każda klasa posiadać będzie jedną wersję operacji dodawania. (Nazwa operacji została zmieniona z `operator+` na `add`, aby uniknąć niejednoznaczności związanej z wprowadzeniem operatora + przedstawionego poniżej). Ponieważ każda koperta może przyjąć, że jej operacja `add` wywoływana jest zawsze z parametrem tej samej klasy, to funkcja `add` nie musi już korzystać z instrukcji wyboru `switch`:

```
class Number {
public:
    . . . .
    virtual Number *type() const;
    virtual int isA(const Number *const) const;
    virtual Number promote(const Number&) const;
    virtual Number add(const Number&) const;
    friend Number operator+(const Number&, const Number&) const;
    . . . .
};

class Complex : public Number {
public:
    . . . .
    int isA(const Number *const) const {
        return n->type() == complexExemplar;
    }
    Number promote(const Number& n) const {
        // zawsze zwraca obiekt typu Complex
        if (n.type() == imaginaryExemplar) {
            return Number(0, n.magnitude());
        } else if (n.type() == integerExemplar) {
            return Number(n.magnitude(), 0);
        } else . . . .
        . . . .
    }
    // poniższa operacja wykonywana jest jedynie dla obiektów typu Complex
```

```
    Number add(const Number&) const;
};

class Imaginary : public Complex {
public:
    . . . .
    int isA(const Number *const) const {
        return n->type() == imaginaryExemplar ||
            Complex::isA(n);
    }

    // brak operacji promote — żadne obiekty nie wymagają promowania
    // do klasy Imaginary

   // poniższa operacja wykonywana jest jedynie dla obiektów typu Imaginary
   Number add(const Number&) const;
};
```

Efekt odpowiadający wielometodom uzyskujemy, wprowadzając globalnie przeciążony `operator+`, który kompilator stosuje dla dowolnych argumentów klasy `Number`:

```
Number operator+(const Number &n1, const Number &n2)
    throw(NumberTypeError)
{
    if (n1.isA(&n2)) {
        Number temporary = n2.promote(n1);
        return n2.add(temporary);
    }  else if (n2.isA(&n1)) {
        Number temporary = n1.promote(n2);
        return n1.add(temporary);
    } else {
        throw NumberTypeError();
    }
}
```

Zauważmy, że rozwiązanie takie posiada szereg zalet w stosunku do podejścia przedstawionego w podrozdziale 5.5 na stronie 140. Funkcje składowe są bardziej spójne — promocje do danego typu odbywają się lokalnie w danej klasie, a wszystkie operatory binarne wykonywane są dla argumentów o homogenicznym typie (w związku z tym nie musimy przeciążać operatorów). Definicje funkcji składowych nie zawierają już instrukcji wyboru `switch`.

Przedstawione rozwiązanie przybliża sposób działania języków symbolicznych. W niektórych implementacjach języka Smalltalk klasy numeryczne bezpośrednio obsługują operacje arytmetyczne, jeśli oba argumenty są tego samego typu. Jeśli typy nie są zgodne, to wysyłany jest komunikat do klasy, aby obsłużyła wszelkie niezbędne promocje i konwersje, a następnie wywołała metodę odpowiedniej klasy pochodnej.

Rozwiązanie w języku Smalltalk odpowiada więc sytuacji, w której nasz globalnie przeciążony operator postaci:

```
operator+(const Number&, const Number&)
```

byłby zadeklarowany jako:

```
Number::operator+(const Number&)
```

Niektóre środowiska programowania w języku Lisp implementują wielometody jako kaskady wywołań funkcji wirtualnych dynamicznie wyszukujące ścieżkę do metody określonej przez typy wielu argumentów.

Ćwiczenia

1. Zmodyfikuj funkcję `Top::update` (oraz inne fragmenty kodu, jeśli to konieczne) tak, by możliwe było dodawanie nowych funkcji wirtualnych do klasy. Pamiętaj, że oryginalna tablica `vtbl` dla danej klasy jest osadzona statycznie w pamięci. Załóż, że kompilator umieszcza elementy tablicy odpowiadające nowym funkcjom wirtualnym na końcu tablicy `vtbl`. Zastanów się, jakie ograniczenia posiada takie rozwiązanie w przypadku dziedziczenia po klasie listu oraz w jaki sposób można dodać funkcje wirtualne w klasach pochodnych.
2. Dodaj funkcję składową `Thing::backout(Thing*)`, która umożliwi powrót struktury klasy i funkcji wirtualnych do wersji poprzedzającej ostatnią aktualizację.
3. Zmodyfikuj program zamieszczony w dodatku E tak, by odzyskiwanie nieużytków wykonywane było za każdym razem, gdy tworzony jest nowy obiekt.
4. Aby rozłożyć działanie mechanizmu odzyskiwania nieużytków w czasie, zmodyfikuj algorytm odzyskiwania nieużytków przedstawiony w dodatku E tak, by każde wywołanie funkcji `Shape::gc` powodowało jedynie odzyskiwanie nieużytków pojedynczej klasy pochodnej klasy `ShapeRep`.
5. Aby sposób działania mechanizmu odzyskiwania nieużytków był jeszcze bardziej stopniowy, zaimplementuj metodę `Shape::gc` tak, by w trakcie wykonywania algorytmu oddawała sterowanie do wywołującego ją kodu, a kolejne jej wywołania podejmowały wykonanie algorytmu w tym miejscu, gdzie zakończyły je poprzednie. Wskazówka: faza zaznaczania obiektów powinna być niepodzielna, natomiast wykonanie fazy zamiatania może zostać podzielone pomiędzy szereg wywołań. Kolejne wywołanie podejmować będzie zamiatanie począwszy od punktu, w którym zakończyło je poprzednie wywołanie. Zamiana roli przestrzeni A i B powinna nastąpić dopiero po zakończeniu całej fazy zamiatania. Zastanów się, jaka kombinacja wartości pól `inUse`, `gcmark` i `space` powinna powodować odzyskanie obiektu.
6. Powtórz poprzednie ćwiczenie, tym razem zakładając jednak niepodzielność fazy zamiatania.
7. Połącz dwa poprzednie ćwiczenia w jedno.
8. Napisz system przydziału pamięci, w którym klasy posiadające obiekty o tym samym rozmiarze współdzielą pule pamięci i zarządzane są przez ten sam mechanizm odzyskiwania nieużytków. Przyporządkowanie klas do puli pamięci powinno odbywać się podczas działania programu.

9. Wymiatanie generacji [5] jest techniką odzyskiwania nieużytków, która wykorzystuje osobne pule pamięci dla obiektów w różnym wieku. Pule zawierające młodsze obiekty wymiatane są częściej, w oparciu o obserwację, że prawdopodobieństwo niewykorzystywania jest wyższe dla obiektów młodszych niż dla starszych. Zaimplementuj algorytm wymiatania generacji posługujący się trzema pulami generacji dla jednej klasy. Określ wiek obiektów mierzony za pomocą cykli zaznaczania i wymiatania.
10. Przepisz klasy `Number` przedstawione w podrozdziale 5.5, posługując się idiomem symbolicznym.
11. Zaimplementuj symboliczną klasę `String`, bazując na przykładzie tej klasy zamieszczonym w podrozdziale 3.5.

Bibliografia

[1] Ellis Margaret A., i Stroustrup B., *The Annotated C++ Reference Manual*. Reading Mass.: Addison-Wesley, 1990, podrozdział 10.5.

[2] McCarthy J., „Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM 3* (1960), 184.

[3] Baker H. G., „List Processing in Real Time on a Serial Computer", A.I. Working Paper 139, MIT-AI Lab, Boston (kwiecień 1977).

[4] Caplinger Michael, „A Memory Allocator with Garbage Collection for C". *USENIX Association Winter Conference*, (luty 1988), 325 – 30

[5] Ungar David, Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm", *SIGPLAN Notices 19*, 5 (maj 1984).