LEKSYKON PROFESJONALISTY

David Chisnall

Objective-C

Przystępna wiedza o Objective-C!

Tytuł oryginalny: Objective-C Phrasebook, Second Edition

Tłumaczenie: Mateusz Wieloch
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-4350-9

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/objclp.zip

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/objclp

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 3.

Zarządzanie pamięcią

Jeśli pochodzisz ze środowiska programistów C lub C++, prawdopodobnie we krwi masz śledzenie własności obiektów oraz ręczne alokowanie ich i niszczenie. Jeśli pochodzisz ze środowiska programistów języków podobnych do Javy, przywykłeś do automatycznego odśmiecacza pamięci, który wykonuje tę pracę za Ciebie.

Objective-C na poziomie języka nie pozwala na alokację lub dealokację obiektów. Jest to pozostawione kodowi napisanemu w C. Alokacja obiektów następuje zwykle po wysłaniu do ich klasy wiadomości `+alloc`, która z kolei wywołuje metodę `malloc()` lub podobną, by zarezerwować pamięć na obiekt. Wysłanie obiektowi wiadomości `–dealloc` czyści i usuwa jego zmienne instancyjne.

Framework Foundation rozbudowuje to proste, ręczne zarządzanie pamięcią o zliczanie referencji. Gdy tylko zrozumiesz, jak działa, Twoje życie stanie się prostsze. Nowsze kompilatory pomogą Ci, zdejmując z Ciebie trud pisania kodu zliczającego referencje.

Zajmowanie i zwalnianie pamięci

```
6  NSArray *anArray = [NSArray array];
7  anArray = [[NSArray alloc] init];
8  [anArray release];
```

Z pliku: retainRelease.m

Z każdym obiektem dziedziczącym z `NSObject` jest powiązany licznik referencji. Gdy osiągnie wartość 0, obiekt jest niszczony. Obiekt utworzony przy użyciu `+alloc` lub podobnej metody, jak np. `+new` czy `+allocWithZone:`, rozpoczyna swój żywot z licznikiem ustawionym na wartość jeden.

Licznik referencji obiektu można kontrolować, wysyłając mu wiadomości –retain i -release. Jak wskazują ich nazwy, wykorzystuje się je do pozyskiwania referencji na obiekt lub jej zwalniania. Wiadomość –retain inkrementuje licznik referencji, a wiadomość –release go dekrementuje.

By określić obecny stan licznika referencji, wyślij wiadomość –retainCount. Kuszące może być optymalizowanie kodu, polegające na wykorzystaniu specjalnego przypadku, gdy istnieje tylko jedna referencja na obiekt. Jest to bardzo zły pomysł. Jak sama nazwa wskazuje, metoda zwraca liczbę *jawnie* pozyskanych referencji, a nie referencji w ogóle. Podczas tworzenia na stosie wskaźnika do obiektu powszechne jest niezawracanie sobie głowy wysyłaniem do niego wiadomości –retain. W efekcie do obiektu mogą prowadzić dwie lub więcej referencji, choć licznik jest ustawiony na jeden.

Wskaźniki w Objective-C są podzielone na dwie kategorie: **referencje własnościowe** i **referencje niewłasnościowe**. Referencja własnościowa to taka, która modyfikuje licznik referencji obiektu. Wywołując metody takie jak +new lub –retain, otrzymuje się referencję własnościową do nowego obiektu. Większość pozostałych metod zwraca referencje niewłasnościowe. Zmienne instancyjne i globalne są przeważnie wskaźnikami własnościowymi, więc powinno się im przypisać referencje własnościowe. Trzeba także upewnić się, że przed przypisaniem nowej referencji własnościowej usunięto starą (wysyłając wiadomość –release).

Zmienne tymczasowe są zwykle referencjami niewłasnościowymi. Automatyczne zliczanie referencji i odśmiecanie pamięci, którym przyjrzymy się w dalszej części obecnego rozdziału, wprowadzają specjalne rodzaje tego typu referencji.

Przypisywanie do zmiennych instancyjnych

```
26  - (void)setStringValue: (NSString*)aString
27  {
28    id tmp = [aString retain];
29    [string release];
30    string = tmp;
31  }
```

Z pliku: ivar.m

Jest kilka rzeczy, na które trzeba uważać podczas korzystania ze zliczania referencji. Rozważmy taką oto prostą metodę ustawiającą:

```
14  - (void)setStringValue: (NSString*)aString
15  {
16    [string release];
17    string = [aString retain];
18  }
```

Z pliku: ivar.m

Kod wygląda rozsądnie. Zwalniana jest referencja do starej wartości, po czym pozyskiwana i przypisywana jest nowa wartość. Zwykle kod będzie działał, ale w kilku przypadkach zawiedzie. Czyni to debugowanie go zajęciem niezwykle trudnym.

Co stanie się, jeśli wartość `aString` i `string` będą takie same? W tym przypadku wyślesz jednemu obiektowi wiadomość `–release`, a następnie `–retain`. O ile inny kod utrzyma referencję do obiektu, wszystko zadziała poprawnie, ale jeśli nie, to pierwsza wiadomość spowoduje zniszczenie obiektu, a druga zostanie wysłana do „wiszącego" wskaźnika.

Na początku rozdziału pokazano lepszą implementację, która najpierw pozyskuje referencję do nowego obiektu. Zwróć uwagę, że powinieneś zachować wartość zwracaną z wiadomości `–retain`, bo niektóre obiekty zwrócą inny obiekt, gdy je pozyskasz. To bardzo rzadkie zachowanie, ale występuje raz na jakiś czas.

W końcu metoda nie jest bezpieczna ze względu na wątki. Jeśli chcesz, by taką była, musisz zarezerwować nowy obiekt, wykonać niepodzielną operację podmiany go i zmiennej instancyjnej, a następnie zwolnić starą wartość. Jest to fatalny pomysł, bo ciężko objąć rozumem kod, który zawiera tak dużo drobiazgowej obsługi współbieżności, a ilość zmarnowanego cache'u przewyższy zyski z równoległego wykonania. Jeśli naprawdę potrzebujesz równoległego wykonania, lepiej użyj zadeklarowanych właściwości do wygenerowania akcesoria, zamiast pisać go ręcznie.

Automatyczne zliczanie referencji

Wraz z wydaniem iOS 5 i Mac OS X 10.7 firma Apple wprowadziła **automatyczne zliczanie referencji** (Automatic Reference Counting — ARC). Koncepcyjnie polega to na tym, że kompilator sam decyduje, kiedy powinien wywołać `–retain` oraz `–release`, i robi to za Ciebie. Rzeczywista implementacja jest oczywiście nieco bardziej skomplikowana.

Zamiast wstawiać bezpośrednio do kodu wysyłanie wiadomości, kompilator wstawia wywołania funkcji takich jak `objc_retain()` i `objc_release()`. Następnie optymalizator stara się połączyć lub wyeliminować te wywołania. W prostym przypadku funkcje wykonują pracę analogiczną do wysyłanych wiadomości. Jednak w niektórych przypadkach są znacznie bardziej efektywne.

Większość przykładów w tej książce korzysta z ręcznego zliczania referencji. Jest to celowe. Pomimo że stosowanie ARC jest zalecane w nowych projektach, istnieje dużo starego kodu, który z niego nie korzysta. Nawet jeśli wykorzystujesz ARC, wciąż powinieneś rozumieć istotę wykonywanego za Ciebie pozyskiwania i zwalniania referencji. Podobnie warto znać zestaw instrukcji procesora, nawet jeśli nigdy nie pisze się kodu w asemblerze. Programiście, który rozumie ręczne zliczanie referencji, bardzo łatwo jest przestawić się na zliczanie automatyczne.

W prostych zastosowaniach wykorzystanie ARC pozwala zapomnieć o zarządzaniu pamięcią. Jeśli używasz nowej wersji środowiska XCode, ARC jest rozwiązaniem domyślnym. Jeśli kompilujesz z linii komend lub przy użyciu jednego z innych systemów budowania projektów, dodaj opcję `-fobjc-arc`. Teraz wystarczy tylko zapomnieć o konieczności pozyskiwania i zwalniania referencji.

Byłoby wspaniale, gdyby życie było tak proste. Niestety ARC ma pewne ograniczenia. Mówiąc konkretniej, formalizuje mętną granicę między pamięcią C a obiektami Objective-C, dzieląc wskaźniki na trzy kategorie. Silne wskaźniki pozyskują i zwalniają pamięć tak, jak widzieliśmy wcześniej. Słabe, którym przyjrzymy się później, to niewłasnościowe referencje zerowane automatycznie podczas niszczenia obiektu. Ostatnią grupę stanowią niebezpieczne, niepozyskane wskaźniki. Są ignorowane przez ARC: sam jesteś odpowiedzialny za czas życia przechowywanych w nich obiektów.

Domyślnie wszystkie wskaźniki w zmiennych instancyjnych i na stosie są mocne. Wskaźniki na obiekty w strukturach nie mają wartości domyślnej i muszą być jawnie oznaczone kwalifikatorem `__unsafe_unretained`. Mimo że w ich przypadku jest to jedyne dozwolone prawo własności, trzeba go jawnie użyć, by przypomnieć osobom czytającym kod, że wskaźniki te będą ignorowane przez ARC.

Zwracanie obiektów przez argumenty będące wskaźnikami

```
 3  __weak id weak;
 4  int writeBack(id *aValue)
 5  {
 6    *aValue = [NSObject new];
 7    weak = *aValue;
 8    return 0;
 9  }
10
11  int main(void)
12  {
```

```
13    @autoreleasepool
14    {
15      id object;
16      writeBack(&object);
17      NSLog(@"Obiekt: %@", object);
18      object = nil;
19      NSLog(@"Obiekt: %@", weak);
20    }
21    NSLog(@"Obiekt: %@", weak);
22    return 0;
23  }
```

Z pliku: writeback.m

W trybie ARC argumenty będące wskaźnikami do wskaźników działają w dosyć skomplikowany sposób. Ich zastosowanie obejmuje zwykle dwa przypadki: przekazywanie tablic lub zwracanie obiektów. Jeśli przekazujesz tablicę w dół stosu, to powinieneś upewnić się, że zadeklarowałeś ją jako `const`. Informuje to kompilator, że wywoływana funkcja nie przypisze niczego do tablicy, więc ARC będzie mógł przekazać ją bez żadnej skomplikowanej interwencji.

Jeśli poprzez argumenty będące wskaźnikami będziesz zwracał obiekt, to ARC utworzy dosyć skomplikowany kod. W przykładzie z początku tego podrozdziału wywołanie `writeBack()` generuje kod zbliżony do tego:

```
id tmp = [object retain];
writeBack(&tmp);
[tmp retain];
[object release];
object = tmp;
```

W funkcji `writeBack()` nowy obiekt będzie automatycznie usunięty z pamięci przed przechowaniem go w tymczasowej wartości. Oznacza to, że na końcu tego procesu `object` zawiera własnościową referencję do nowego obiektu.

Jeśli zadeklarowałeś `object` jako `_autoreleasing id`, wygenerowany kod jest znacznie prostszy. Automatycznie zwalnia wartość przechowywaną początkowo w `object`, co nie będzie miało żadnego efektu, bo wartością początkową wszystkich wskaźników — nawet tych przydzielanych automatycznie — i tak jest `nil`, a następnie bezpośrednio przekazuje wskaźnik i oczekuje, że jeśli wywoływana funkcja go zmodyfikuje, to umieści na jego miejsce wskaźnik niewłasnościowy (automatycznie zwalniany).

Jeśli uruchomisz ten przykład, to zobaczysz, że słaba referencja jest zerowana wyłącznie, gdy niszczona jest pula automatycznego przydziału. We wszystkich przypadkach funkcja `writeBack()` przechowuje zwolniony automatycznie obiekt w przekazanym wskaźniku i nigdy nie usuwa z pamięci przekazanej wartości. Wywołujący jest zawsze odpowiedzialny za upewnienie się, że do funkcji przekazuje referencję niewłasnościową, co czyni na jeden z dwóch sposobów: albo upewnia się, że wskaźnik pokazuje na automatycznie zwalniany obiekt, albo że jest kopią wskaźnika na obiekt, dla którego pozyskano referencję.

Jeśli zamiast tego oznaczysz parametr słowem `out` (dozwolone wyłącznie w przypadku parametrów metod, a nie funkcji C), to wywoływana metoda gwarantuje, że nie odczyta tej wartości. Kompilator będzie mógł więc pominąć krok, w którym wykonuje kopię wskaźnika przechowywanego w `object` przed wywołaniem.

Jeśli musisz przekazać kilka obiektów w górę stosu, to powinieneś prawdopodobnie zwrócić instancję `NSArray`. Jest kilka alternatyw, ale są wystarczająco złożone, by nie były warte wysiłku: korzystając z nich, łatwo popełnić mały błąd i spędzić wieki na debugowaniu kodu. Nawet jeśli uda Ci się zrobić wszystko dobrze, przypuszczalnie i tak metody te będą wolniejsze niż użycie `NSArray`.

Jeśli przekazujesz kilka wartości w dół stosu, to powinieneś zadeklarować typ tablicy z jawnym kwalifikatorem własności dla parametru, a nie typ wskaźnikowy. Na przykład `__unsafe_unretained id[]` zamiast `id*`. Zapewnia to wyłączenie mechanizmu zapisu zwrotnego.

Unikanie cykli pozyskanych referencji

```
19  - (void)setDelegate: (id)aDelegate
20  {
21    delegate = aDelegate;
22  }
```

Z pliku: ivar.m

Problemem w przypadku „czystego” zliczania referencji jest to, że nie wykrywa cykli. Jeśli dwa obiekty przechowują referencje do siebie nawzajem, żaden z nich nie zostanie nigdy zwolniony.

Generalnie nie jest to wielki problem. Struktury danych w Objective-C są zwykle acykliczne, ale istnieją przypadki, w których cykle są nieuniknione. Najpopularniejszym z nich jest **wzorzec delegacji**. Wzorzec polega na tym, że jeden obiekt implementuje jakiś mechanizm, ale zarządzanie pracą przekazuje innemu obiektowi. Większość frameworku AppKit działa w ten sposób.

Obiekty potrzebują do siebie nawzajem referencji, co momentalnie tworzy cykl. Popularnym sposobem rozwiązania problemu jest przyjęcie zasady, że obiekty nie pozyskują na siebie referencji. Jeśli przekażesz obiekt jako argument do metody `–setDelegate:`, będziesz musiał upewnić się, że referencja jest przechowywana przez jakiś inny obiekt lub że zostanie przedwcześnie zwolniona, a obiekt usunięty.

W przypadku ARC masz dwie możliwości: możesz oznaczyć zmienną instancyjną słowem `__weak` albo `__unsafe_unretained`. Pierwsza z opcji jest bezpieczniejsza, bo zapewnia, że wskaźnik nie będzie „wisiał w powietrzu". Gdy delagata zostanie zniszczona, zmienna instancyjna zostanie ustawiona na `nil`.

Są dwie wady używania słabych wskaźników. Pierwszą jest przenośność. Słabe wskaźniki działają na iOS 5, Mac OS X 10.7 i w GNUstep, ale nie działają na starszych wersjach iOS i Mac OS X. Drugą wadą jest wydajność. W każdej próbie dostępu do wskaźnika pośredniczy funkcja pomocnicza, sprawdzająca, czy obiekt nie jest w stanie dealokacji, i pozyskująca referencję do niego, jeśli jest wciąż „żywy", albo zwracająca `nil` w przeciwnym przypadku.

Pod nazwą niebezpiecznych, niepozyskanych wskaźników kryją się zwykłe, „tradycyjne" wskaźniki, oferujące tani dostęp do obiektu i działające na każdej platformie sprzętowej. Ich wadą jest obarczanie użytkownika odpowiedzialnością za upewnienie się, że po zwolnieniu wskazywanego obiektu użytkownik nie sięgnie już więcej do wskaźnika.

Dobrym kompromisem jest używanie słabych wskaźników w trakcie debugowania i niebezpiecznych, niepozyskanych wskaźników w trakcie normalnej pracy. Dodaj asercję sprawdzającą, czy wskaźnik nie jest równy `nil`, przed wysłaniem mu jakiejkolwiek wiadomości, a jeśli będziesz miał błędy, to skończysz z użytecznym komunikatem zamiast nagłego przerwania aplikacji.

Migracja do ARC

Jeśli zaczynasz nowy projekt w XCode, ARC będzie domyślnym wyborem i ciężko znaleźć powody, by z niego nie skorzystać. Jeśli jednak pracujesz ze starszym kodem, prawdopodobnie będziesz używał ręcznego zliczania referencji. W długiej perspektywie można oszczędzić sobie pracy, przenosząc kod do ARC, ale jest to nieco trudniejsze niż ustawienie przełącznika kompilacji.

Clang posiada narzędzie wspomagające migrację, które próbuje przepisać kod Objective-C tak, by korzystał z ARC. Wywołuje się je z linii komend przy użyciu argumentów `–ccc-arcmt-check` i `–ccc-arcmt-modify`. Pierwszy z nich informuje o wszystkich miejscach w kodzie, które nie mogą być automatycznie zmienione. O ile nie będzie błędów, drugi wykona konwersję kodu, modyfikując oryginalny plik.

W prostym kodzie Objective-C narzędzie zadziała poprawnie. Najbardziej oczywistymi zmianami będzie usunięcie wszystkich wiadomości `–retain`, `–release` i `autorelease`. Narzędzie usunie także jawne wywołanie `[super dealloc]` z metody `–dealloc`. Od teraz wywołanie jest wstawiane automatycznie przez kompilator. ARC automatycznie zwalnia wszystkie zmienne instancyjne, więc metodę `–dealloc` będziesz musiał zaimplementować, tylko jeśli musisz zwolnić pamięć zajętą przez funkcję `malloc()` lub podobne konstrukcje.

Jeśli utworzyłeś własne metody do zliczania referencji, będziesz musiał je usunąć. Popularnym powodem własnej implementacji jest chęć uniknięcia przypadkowego usunięcia singletonów. Traci to na znaczeniu w przypadku ARC, ponieważ błąd programisty ma mniejsze szanse na spowodowanie przedwczesnego usunięcia obiektu.

Do zwalniania zmiennych instancyjnych ARC tworzy metodę `–.cxx_destruct`. Metoda ta była pierwotnie utworzona, by po zniszczeniu obiektu automatycznie wywoływać destruktory z C++. Widoczną różnicą pomiędzy jej dawnym zastosowaniem a ARC w Objective-C jest to, że zmienne instancyjne są teraz dealokowane po wykonaniu funkcji `–dealloc` w klasie-korzeniu, a nie przed. W większości przypadków nie powinno mieć to znaczenia.

Największe problemy pojawiają się, jeśli próbujesz przechować wskaźniki Objective-C w strukturach C. Najprostszym rozwiązaniem jest po prostu nierobienie tego. W zamian użyj obiektów Objective-C z publicznymi zmiennymi instancyjnymi. Pozwoli to kompilatorowi zająć się zarządzaniem pamięcią obiektu i jego pól.

Jedynym przypadkiem, w którym uznaje się, że używanie struktur odnoszących się do obiektów Objective-C jest bezpieczne, jest sytuacja, gdy przekazujesz je w dół stosu. Wskaźniki na obiekty mogą posiadać kwalifikator `__unsafe_unretained` tak długo, jak długo pozostają poprawne w funkcji, w której powstały.

Po użyciu narzędzia do migracji zauważysz, że zniknęły właściwości `assign`. Zostały one przepisane jako `unsafe_unretained` lub `weak`, w zależności od tego, czy wybrana platforma docelowa kompilacji obsługuje słabe referencje. Jeśli właściwości `assign` wykorzystywałeś do przerywania prostych cykli i uznasz, że słabe referencje stwarzają problemy wydajnościowe, możesz jawnie zmienić niektóre z nich na `unsafe_unretained`.

Narzędzie do migracji spróbuje w odpowiednich miejscach powstawiać rzutowanie mostkowane `__bridge`, ale warto sprawdzić, czy zrobiło to poprawnie. Rzutowania te są wykorzystywane do dodawania i usuwania obiektów z kodu zarządzanego przez ARC. W kodzie niewykorzystującym ARC można korzystać z konstrukcji takich jak `(void*)someObject`, bo wskaźniki na obiekty są zwykłymi wskaźnikami C,

do których dodatkowo można wysyłać wiadomości. W trybie ARC rzutowanie byłoby wieloznaczne, bo kompilator nie wiedziałby, która zmienna ma być właścicielem obiektu wskazywanego przez `void*`. Z tego powodu taki kod jest zabroniony.

Narzędzie do migracji przepisze przedstawioną wyżej konstrukcję jako `(__bridge void*)someObject`, ale może to nie być to, o co Ci chodziło. Tego typu rzutowaniom bardziej szczegółowo przyjrzymy się w podrozdziale „Współpraca z C".

Pule automatycznego zwalniania pamięci

```
 3  id returnObject(void)
 4  {
 5    return [[NSObject new] autorelease];
 6  }
 7
 8  int main(void)
 9  {
10    @autorelease {
11      id object = returnObject();
12      [object retain];
13    }
14    // Tutaj obiekt traci ważność.
15    [object release];
16    return 0;
17  }
```

Z pliku: autorelease.m

Poza cyklami największym problemem w przypadku zliczania referencji są krótkie okresy, w których na obiekt nie wskazuje żadna referencja. W języku C problemem jest decyzja, kto jest odpowiedzialny za alokowanie pamięci: wywołujący czy wywoływany.

W funkcjach takich jak `sprintf()` pamięć zajmuje wywołujący. Niestety nie wie on, ile pamięci tak naprawdę jest potrzebne, więc utworzono wariant o nazwie `snprintf()`, który informuje wywołaną funkcję o ilości dostępnej pamięci. Takie rozwiązanie wciąż może sprawiać problemy, więc utworzono jeszcze jedną wersję o nazwie `asprintf()`, która pozwala wywoływanej funkcji zająć się alokowaniem pamięci.

Jeśli pamięć zajmuje wywoływana funkcja, kto jest odpowiedzialny za jej zwolnienie? Prawdopodobnie wywołujący, ale ponieważ to nie on zajął pamięć, to programy sprawdzające równowagę między wywołaniami funkcji `malloc()` i `free()` nie poradzą sobie ze spostrzeżeniem potencjalnego wycieku pamięci.

W Objective-C problem ten występuje jeszcze częściej. Wiele metod może zwracać obiekty tymczasowe, które muszą zostać zwolnione. Z kolei gdy zwracają wskaźnik na zmienną instancyjną, nie trzeba go zwalniać. Można by najpierw pozyskiwać takie wskaźniki, ale potem trzeba by pamiętać, żeby zwalniać każdy obiekt zwracany z metody. Szybko stałoby się to męczące.

Rozwiązaniem problemu jest **pula automatycznego zwalniania pamięci**. Po wysłaniu do obiektu wiadomości `–autorelease` jest on dodawany do aktywnej w danym momencie puli `NSAutoreleasePool`. Gdy instancja puli jest niszczona, każdemu dodanemu do niej obiektowi jest wysyłana wiadomość `–release`.

`-autorelease` to opóźniona wiadomość `–release`. Wysyła się ją do obiektu, gdy samemu nie potrzebuje się już referencji do niego, ale komuś innemu może jeszcze być potrzebna.

Jeśli używasz pętli wykonania `NSRunLoop`, pula automatycznego zwalniania pamięci będzie tworzona na początku, a niszczona na końcu każdej iteracji pętli. Oznacza to, że żaden tymczasowy obiekt nie zostanie zniszczony do końca obecnej iteracji. Jeśli robisz coś, co wymaga dużej ilości takich obiektów, możesz chcieć utworzyć nową pulę automatycznego zwalniania. Robi się to w następujący sposób:

```
id pool = [NSAutoreleasePool new];
[anObject doSomethingThatCreatesObjects];
[pool drain];
```

Zwróć uwagę, że aby zniszczyć pulę, trzeba jej wysłać wiadomość `–drain`, a nie `–release`. Jest tak, bo w trybie odśmiecania pamięci środowisko uruchomieniowe Objective-C ignoruje wiadomości `–release`. Wiadomość `–drain` jest wskazówką dla odśmiecacza, ale w trybie odśmiecania pamięci nie powoduje natychmiastowego zniszczenia puli.

W OS X 10.7 firma Apple uczyniła pulę automatycznego zwalniania pamięci częścią języka. Programy jawnie używające `NSAutoreleasePool` są uważane za niepoprawne w trybie ARC i zostaną odrzucone przez kompilator. Trzeba je zastąpić konstrukcją `@autoreleasepool`, definiującą region ważności puli. W trybach innych niż ARC konstrukcja ta spowoduje wstawienie identycznego kodu jak w listingu powyżej. W trybie ARC wstawione zostaną wywołania funkcji `objc_autoreleasePoolPush()` i `objc_autoreleasePoolPop()`, które wykonają podobną pracę.

Konstruktory automatycznie zwalniające pamięć

```
4  + (id)object
5  {
6    return [[[self alloc] init] autorelease];
7  }
```

Z pliku: namedConstructor.m

W poprzednim podrozdziale powiedziałem, że obiekty utworzone przy użyciu `+alloc` mają licznik referencji ustawiony na jeden. W rzeczywistości wszystkie obiekty są tworzone z licznikiem ustawionym na jeden, ale obiekty tworzone przy użyciu nazwanego konstruktora, np. `+stringWithFormat:` lub `+array`, są także automatycznie usuwane z pamięci.

By zachować obiekt tworzony przy użyciu jednego z tych mechanizmów, trzeba wysłać mu wiadomość `–retain`. W przeciwnym wypadku zostanie zniszczony w trakcie usuwania puli automatycznego zwalniania pamięci.

Jest to konwencja, którą warto naśladować we własnych klasach. Jeśli ktoś utworzy instancję jednej z Twoich klas, korzystając z nazwanego konstruktora, będzie oczekiwał, że nie musi jej zwalniać. Typowy nazwany konstruktor przypominałby ten z początku tego podrozdziału.

Zwróć uwagę, że ponieważ jest to metoda klasowa, obiekt `self` będzie klasą. Wysyłając wiadomość `+alloc` do `self` zamiast do klasy o określonej nazwie, metoda może działać poprawnie również w przypadku podklas.

W trybie ARC te konwencje zostały sformalizowane w **rodzinach metod**. Metody, które zaczynają się od słów `alloc`, `new`, `copy` lub `mutableCopy`, zwracają **referencję własnościową** — referencję, która musi zostać zwolniona, jeśli nie będzie przechowywana. Inne metody zwracają referencje niewłasnościowe, które muszą zostać automatycznie zwolnione lub przechowane w miejscu, co do którego gwarantuje się, że nie będzie zwolnione.

Automatyczne zwalnianie obiektów w akcesorach

```
34  - (NSString*)stringValue
35  {
36    return [[string retain] autorelease];
37  }
```

Z pliku: ivar.m

Innym powszechnym problemem ze zliczaniem referencji w formie zaimplementowanej przez Foundation jest częste niepozyskiwanie obiektów, które znajdują się na stosie. Rozważmy następujący fragment kodu:

```
NSString *oldString = [anObject stringValue];
[anObject setStringValue: newString];
```

Jeśli metoda –setStringValue: została zaimplementowana tak, jak sugerowałem wcześniej, to wykonanie kodu skończy się przerwaniem aplikacji, bo obiekt, do którego odnosi się oldString, zostanie usunięty przed ustawieniem nowej wartości ciągu znakowego. Jest to problem mający dwa możliwe rozwiązania, obydwa korzystające z puli automatycznego zwalniania pamięci. Pierwsze polega na automatycznym zwolnieniu starej wartości podczas ustawiania nowej. Drugi to takie zdefiniowanie metody –stringValue, jak pokazano na początku tego podrozdziału.

Zapewnia to, że ciąg znakowy nie zostanie przez przypadek zniszczony w trakcie normalnej pracy obiektu. Innym popularnym rozwiązaniem jest zastąpienie wiadomości –copy wiadomością –retain. Jest to użyteczne, jeśli zmienna instancyjna może być modyfikowana. Jeśli nie można jej modyfikować, –copy będzie odpowiednikiem –retain. Jeśli można ją modyfikować, wywołujący otrzyma obiekt, który nie będzie ulegał zmianom na skutek wysłanych mu wiadomości.

Wsparcie dla automatycznego odśmiecania kolekcji

```
0  $ gcc -c -framework Cocoa -fobjc-gc-only
       collected.m
1  $ gcc -c -framework Cocoa -fobjc-gc collected.m
```

Od wersji OS X 10.5 firma Apple wprowadziła do Objective-C automatyczne odśmiecanie kolekcji. Choć może to upraszczać życie programistom, w większości przypadków zmniejsza wydajność. Odśmiecacz firmy Apple do śledzenia istniejących referencji zużywa dużo pamięci i z tego powodu nie jest dostępny na iPhone'a. Nie jest też obsługiwany w starszych wersjach OS X, a jego obsługa w GNUstep jest ograniczona. Jeśli chcesz pisać przenośny kod, powinieneś unikać odśmiecania kolekcji.

Jeśli skompilujesz kod w trybie odśmiecania pamięci, wszystkie wiadomości –retain, –release i –autorelease zostaną zignorowane. Kompilator automatycznie wstawi wywołania odpowiednich funkcji przy każdej operacji zapisu do pamięci na stercie.

Żeby można było używać odśmiecania kolekcji, kod musi być skompilowany w trybie odśmiecania pamięci. Przy każdej operacji przypisania do pamięci na stercie wstawiane są wtedy wywołania funkcji, które w środowisku uruchomieniowym systemu Mac są zadeklarowane w pliku `objc-auto.h`.

Funkcje te upewniają się, że odśmiecacz kolekcji jest świadomy wykonanego przypisania. Są wymagane, bo odśmiecacz działa w tle, w osobnym wątku, usuwając obiekty, do których nie może znaleźć referencji. Odśmiecacz musi być informowany o przestawianiu wskaźników; mógłby przez przypadek usunąć obiekt, do którego przed momentem utworzyłeś referencję.

Podczas kompilacji z włączonym odśmiecaniem pamięci dostępne są dwie opcje. Po skompilowaniu z flagą `–fobjc-gc-only` kod będzie wspierał wyłącznie odśmiecanie kolekcji. Po skompilowaniu z flagą `–fobjc-gc` kod będzie obsługiwał zarówno zliczanie referencji, jak i automatyczne odśmiecanie kolekcji. Jest to użyteczne, gdy kompilujesz framework. Wciąż musisz pamiętać, by w odpowiednich miejscach dodać wywołania `–retain` i `–release`, ale użytkownicy frameworku będą mogli używać go z włączonym lub wyłączonym odśmiecaniem.

```
110  OBJC_EXPORT BOOL objc_atomicCompareAndSwapGlobal(
          id predicate, id replacement, volatile id *
          objectLocation)
111       __OSX_AVAILABLE_STARTING(__MAC_10_6,
               __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
112  OBJC_EXPORT BOOL
          objc_atomicCompareAndSwapGlobalBarrier(id
          predicate, id replacement, volatile id *
          objectLocation)
113       __OSX_AVAILABLE_STARTING(__MAC_10_6,
               __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
114  // automatyczna aktualizacja zmiennej instancyjnej
115  OBJC_EXPORT BOOL
          objc_atomicCompareAndSwapInstanceVariable(id
          predicate, id replacement, volatile id *
          objectLocation)
116       __OSX_AVAILABLE_STARTING(__MAC_10_6,
               __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
117  OBJC_EXPORT BOOL
          objc_atomicCompareAndSwapInstanceVariableBarrier
         (id predicate, id replacement, volatile id *
         objectLocation)
```

Współpraca z C

W trybie odśmiecania pamięci nie jest skanowana cała pamięć. Dane zaalokowane przy użyciu `malloc()` są niewidoczne dla odśmiecacza. Jeśli przekażesz do funkcji z C wskaźnik na obiekt w parametrze `void*`, a następnie funkcja przechowa go w pamięci zarezerwowanej `malloc()`, stanie się niewidoczny dla odśmiecacza i może zostać zwolniony, choć wciąż istnieją do niego referencje.

W trybie ARC kompilator automatycznie zajmie się wskaźnikami na stosie i zmiennymi instancyjnymi, ale nie będzie śledził wskaźników w strukturach i wszystkich innych jawnie zadeklarowanych jako `__unsafe_uretained`.

Normalnie przed wstawieniem obiektu na stertę wysłałbyś mu wiadomość `–retain`. Nie działa to jednak w trybie odśmiecania pamięci i jest zabronione w trybie ARC. W zamian musisz użyć funkcji `CFRetain()`, inkrementującej licznik referencji obiektu niezależnie od obecności odśmiecacza. Odśmiecacz zwolni obiekt jedynie wtedy, gdy licznik referencji osiągnie wartość zero i nie będzie można znaleźć do niego żadnej referencji w skanowanej pamięci.

Gdy skończysz pracę z referencją znajdującą się poza obszarem śledzonym przez odśmiecacz, będziesz musiał wywołać `CFRelease()`.

ARC posiada bogatszy model pamięci do tego typu operacji. Jawne rzutowania obiektów na nieobiektowe wskaźniki nie są już dozwolone. Muszą być zastąpione **rzutowaniami mostkowanymi**. Rozważmy następujący fragment kodu działający w trybie innym niż ARC:

```
void *aPointer = (void*)someObject;
```

W trybie ARC rezultatem wykonania go byłoby utworzenie ze wskaźnika śledzonego wskaźnika niesledzonego. Nie jest to zamiana, którą chciałbyś wykonać bez zastanowienia. Masz trzy podstawowe możliwości. Pierwsza jest często wykorzystywana w przypadku zmiennych na stosie lub wskaźników pokazujących na obiekty, co do których ma się pewność, że wskazują na nie inne referencje:

```
void *aPointer = (__bridge void*)someObject;
```

Powyżej wykonywane jest rzutowanie bez przekazania własności. Jeśli wszystkie pozostałe referencje na `someObject` przestaną istnieć, `aPointer` zostanie wiszącym wskaźnikiem.

Jeśli wskaźnik `void*` będzie umieszczony na stosie z zamiarem przechowania własnościowej referencji do obiektu, powinieneś wykonać rzutowanie mostkowane:

```
void *aPointer = (__bridge_retained void*)
    someObject;
```

Spowoduje to wyjęcie jednej referencji własnościowej spod kontroli ARC. Jest to z grubsza odpowiednik wysłania do `someObject` wiadomości `–retain` przed przechowaniem go we wskaźniku. Jeśli napiszesz `(__bridge_retained void*)someObject` bez operacji przypisania, będzie to informacja dla kompilatora, żeby pozyskał referencję na obiekt. Tego typu postępowanie uznawane jest za bardzo zły styl programowania. Podczas rzutowania z powrotem na wskaźnik na obiekt powinieneś wykonać odwrotną operację:

```
id anotherObjectPointer = (__bridge_transfer
    id)aPointer;
aPointer = NULL;
```

Kod ten powoduje przekazanie własnościowej referencji pod kontrolę ARC. ARC jest teraz odpowiedzialny za zwalnianie obiektu, więc ważne jest pamiętanie o wyzerowaniu wskaźnika z C. Jeśli nie przejmujesz własności nad wskaźnikiem, powinieneś użyć prostego rzutowania `__bridge`.

Niszczenie obiektów

```
 3  @interface Example : NSObject
 4  {
 5    void *cPointer;
 6    id objectPointer;
 7  }
 8  @end
 9  @implementation Example
10  - (void)finalize
11  {
12    if (NULL != cPointer) { free(cPointer); }
13    [super finalize];
14  }
15  - (void)dealloc
16  {
17    if (NULL != cPointer) { free(cPointer); }
18  #if !__has_feature(objc_arc)
19    [objectPointer release];
20    [super dealloc];
21  #endif
22  }
23 @end
```

Z pliku: dealloc.m

Istnieją trzy metody wywoływane podczas niszczenia obiektu zależnie od wykorzystywanego trybu. Jedna jest uruchamiana za każdym razem, ale nie może być napisana przez Ciebie. Metoda `–.cxx_destruct` jest zawsze wywoływana przez

środowisko uruchomieniowe Objective-C i zajmuje się niszczeniem pól, za których czyszczenie jest odpowiedzialny kompilator. Do kategorii tej należą obiekty C++ w trybie Objective-C++ i wskaźniki na obiekty Objective-C w trybie ARC.

Dwie pozostałe metody to `–finalize` i `–dealloc`. W trybie odśmiecania pamięci nie musisz nic robić, by usunąć referencje do obiektów Objective-C, ale wciąż musisz wykonać czyszczenie zasobów, którymi nie zarządza odśmiecacz pamięci. Do tej kategorii należy zamykanie uchwytów do plików, zwalnianie pamięci zaalokowanej przy użyciu `malloc()` itd. Jeśli klasa ma zmienne instancyjne, które wymagają ręcznego czyszczenia, powinieneś zadeklarować metodę `–finalize`.

Odśmiecacz pamięci zwykle wywoła metody `–finalize` w specjalnym wątku. Oznacza to, że metody `–finalize` muszą być bezpieczne ze względu na wątki. Jeśli korzystają z globalnych zasobów, muszą upewnić się, że nie spowoduje to problemów.

Jeśli nie używasz odśmiecania pamięci, sprzątanie powinieneś wykonać w metodzie `–dealloc`. Zawartość metody zależy od tego, czy używasz ARC. Dawniej metody `–dealloc` były wypełnione wiadomościami `–release` wysyłanymi do każdej zmiennej instancyjnej, którą deklarowała klasa. Jeśli korzystasz z ARC, nie jest to wymagane. ARC zwolni wszystkie własnościowe referencje do obiektów w `–.cxx_destruct`, więc w `–dealloc` będziesz musiał sprzątnąć jedynie zmienne instancyjne niebędące obiektami.

Zarówno w trybie odśmiecania pamięci, jak i ręcznego pozyskiwania i zwalniania referencji powinieneś przekazać wiadomość do nadklasy, wywołując `[super dealloc]` lub `[super finalize]`. W trybie ARC jawne wywołanie `–dealloc` jest niedozwolone. Zamiast tego ARC sam wstawi wywołanie `[super dealloc]` na końcu metody `–dealloc` w każdej klasie niebędącej korzeniem.

Zwróć uwagę, że kod na początku tego podrozdziału jest nadmiernie skomplikowany. W prawdziwym kodzie mało prawdopodobne jest, żebyś musiał wspierać zarówno tryb ARC, jak i tryb ręcznego pozyskiwania i zwalniania referencji. Tryby te mogą współistnieć w jednym programie, ale jedynym rozsądnym powodem wspierania jakiegokolwiek trybu poza ARC jest chęć zachowania zgodności ze starymi kompilatorami.

Słabe referencje

```
4  __weak id weak;
5
6  int main(void)
7  {
```

```
 8    id obj = [NSObject new];
 9    weak = obj;
10    obj = nil;
11    objc_collect(OBJC_FULL_COLLECTION);
12    fprintf(stderr, "Słaba referencja: %p\n", weak);
13    return 0;
14  }
```

Jednym z najprzyjemniejszych elementów implementacji odśmiecania pamięci firmy Apple jest istnienie *zerujących się, słabych referencji*. Niepozyskane wskaźniki są w starej dokumentacji Objective-C często określane mianem „słabych". Wskaźniki te mogą przetrwać dłużej niż obiekt, na który wskazują. Niestety nie ma automatycznego sposobu sprawdzania, czy są wciąż poprawne.

Słabe referencje okazały się tak przydatne podczas odśmiecania pamięci, że są teraz wspierane także przez ARC, choć mają odrobinę inne znaczenie. Implementacje ARC zachowujące wsteczną kompatybilność nie obsługują słabych referencji, więc wsparcie firmy Apple jest obecnie jedynie na platformach OS X 10.7 i iOS 5 lub późniejszych.

Słabe referencje w środowisku zliczania referencji są często używane do eliminowania cykli. Nie jest to konieczne w środowisku, w którym referencje są śledzone. W tym przypadku w miejscach potencjalnego tworzenia cykli oraz w trakcie tworzenia delegat możesz używać silnych referencji.

Jeśli zadeklarujesz wskaźnik na obiekt ze słowem `__weak`, utworzysz zerującą się, słabą referencję. Odśmiecacz pamięci nie weźmie jej pod uwagę podczas ustalania, czy obiekt wciąż istnieje, a w trybie ARC po przypisaniu do niej nie zostanie zwiększony licznik referencji. Jeśli wszystkie referencje do obiektu są słabe, może on być zniszczony. Po tej operacji wszystkie słabe referencje będą zwracały `nil`.

Słabe referencje są zwykle używane w połączeniu z konstrukcjami takimi jak powiadomienia. Słabą referencję do obiektu można przechowywać i wysyłać mu wiadomości tak długo, jak długo ktoś inny posiada do niego referencję. Gdy referencja przestanie istnieć, obiekt można automatycznie usunąć z mechanizmu powiadamiania.

Cocoa posiada teraz kolekcje pozwalające na przechowywanie słabych referencji. Starsze wersje frameworku Foundation zawierały typy utworzone w języku C: `NSMapTable` oraz `NSHashTable` wraz z zestawem funkcji C, które można z nimi wykorzystać. Interfejsy te są wciąż dostępne, ale w wersji 10.5 Apple zamieniło te typy na klasy.

Typ `NSMapTable` jest ogólniejszą formą `NSDictionary`. Można w nim przechowywać dowolne typy o rozmiarze wskaźnika zarówno w kluczach, jak i wartościach. W trybie odśmiecania pamięci możesz wykorzystywać tę klasę do przechowywania mapowań z i do silnych lub słabych wskaźników. Jest to przydatne np. w `NSNotificationCenter`, w którym przechowuje się obiekty mogące otrzymywać powiadomienia. Gdy obiekty te przestaną istnieć, są automatycznie usuwane z kolekcji, a powiadomienia nie są już więcej wysyłane.

Przykład na początku podrozdziału pokazuje ważną różnicę między trybami ARC i odśmiecania pamięci. Jeśli skompilujesz i uruchomisz go z włączonym odśmiecaniem pamięci, wypisany zostanie przypuszczalnie adres obiektu. Jest tak, gdyż odśmiecacz pamięci podczas skanowania stosu wciąż widzi stare wartości tymczasowe.

W odróżnieniu od odśmiecacza pamięci w trybie ARC zawsze zostanie wypisana wartość 0. ARC jest zupełnie deterministyczny. Przypisanie `nil` do silnego wskaźnika dekrementuje licznik referencji obiektu i powoduje dealokację. Słaby wskaźnik jest zerowany, nim zacznie się dealokacja, więc gwarantowane jest, że przyjmie wartość zero, nim wywołana zostanie funkcja `fprintf()`.

Alokowanie skanowanej pamięci

```
15  id *buffer =
16    NSAllocateCollectable(
17        10 * sizeof(id),
18        NSScannedOption);
```

Z pliku: gc.m

Pamięć zajęta przy użyciu funkcji `malloc()` jest niewidoczna dla odśmiecacza pamięci. Może to być problemem, np. jeśli potrzebujesz tablicy C zawierającej obiekty. Widzieliśmy już rozwiązanie tego problemu. Możesz wywołać `CFRetain()` dla obiektu, który masz zamiar przechować w tablicy, oraz `CFRelease()` dla znajdującej się w niej starej wartości, a następnie je zamienić.

Nie jest to idealne rozwiązanie, ale działa. Inną opcją jest alokacja fragmentu pamięci tak, by był widoczny dla odśmiecacza. Funkcja `NSAllocateCollectable()` jest podobna do `malloc()` z dwoma istotnymi różnicami.

Po pierwsze: zwracana przez nią pamięć będzie odśmiecana. Nie istnieje jej odpowiednik w postaci funkcji `NSFreeCollectable()`. Gdy ostatni wskaźnik na bufor zniknie, bufor zostanie usunięty.

Odśmiecacz firmy Apple nie wspiera wskaźników do wnętrza bufora, więc musisz upewnić się, że utrzymujesz wskaźnik na początek regionu. Inne wskaźniki nie powstrzymają procesu usunięcia go.

Drugą różnicą jest istnienie w tej funkcji drugiego parametru, definiującego pożądany typ pamięci. Jeśli chcesz używać bufora do przechowywania typów C, możesz wstawić tu zero. Jeśli przekażesz `NSScannedOption`, zwrócony bufor będzie przeszukiwany jako możliwe źródło wskaźników na obiekty oraz wskaźników na inne obszary pamięci zwracane przez `NSAllocateCollectable()`.

Skorowidz

U

W

Y

Z

Objective-C to nowoczesny język programowania, dzięki któremu możesz tworzyć zaawansowane aplikacje dla produktów firmy Apple. Urządzenia takie jak iPad, iPhone czy laptopy z systemem operacyjnym Mac OS podbiły serca użytkowników na całym świecie. Co ważne, ich pozycja wydaje się niezagrożona! Dlatego inwestycja w wiedzę na temat tego języka jest w pełni uzasadniona.

Z tą książką błyskawicznie poznasz możliwości języka Objective-C. Dzięki przystępnemu wprowadzeniu zapoznasz się z podstawami języka, a w kolejnych rozdziałach poszerzysz wiedzę o bardziej zaawansowane zagadnienia. Podręcznik wypełniony ponad setką listingów z kodem źródłowym programów sprawi, że będziesz mógł stworzyć działający kod w języku Objective-C praktycznie w każdej sytuacji. W trakcie lektury dowiesz się, jak zarządzać pamięcią, korzystać ze wzorców oraz wykonywać operacje na ciągach znaków, liczbach i kolekcjach. Ponadto sprawdzisz, jak w Objective-C korzystać z plików, wątków i dostępu do sieci. Poświęć chwilę tej książce, a już wkrótce zaczniesz tworzyć zaawansowane oprogramowanie w Objective-C!

Sprawdź:

- jak rozpocząć przygodę z Objective-C
- jak zarządzać pamięcią
- jak tworzyć aplikacje wielowątkowe
- jak debugować kod

Twój przewodnik do pierwszej aplikacji w Objective-C!

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-4350-9