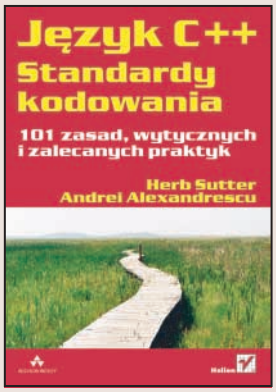
Język C++. Standardy kodowania. 101 zasad, wytycznych i zalecanych praktyk



Autorzy: Herb Sutter, Andrei Alexandrescu
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-849-X
Tytuł oryginału: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices
Format: B5, stron: 320

Czytelny i przejrzysty kod to podstawa sprawnego tworzenia aplikacji. W przypadku pracy zespołowej stosowanie wspólnego standardu kodowania to konieczność. Pisanie kodu w oparciu o określone standardy kodowania przyspiesza powstawanie programu, ułatwia komunikację pomiędzy członkami zespołu i pozwala na szybkie wdrożenie nowych programistów do projektu. Oczywiście, w każdej firmie lub zespole można ustalić własny standard kodowania — ważne jest jednak, aby opierał się na określonych regułach, wynikających ze specyfiki języka programowania.

Książka „Język C++. Standardy kodowania. 101 zasad, wytycznych i zalecanych praktyk" zawiera opis wspomnianych reguł. Przedstawia zasady pisania kodu źródłowego i standaryzowania określonych zapisów, operacji i sposobów wykorzystania elementów języka C++. Każda z zasad jest szczegółowo omówiona i poparta praktycznymi przykładami. Książka prezentuje najlepsze ze znanych praktyk — zarówno „starych", jak i tych, które całkiem niedawno uległy standaryzacji, oraz opisuje techniki, o których nie słyszeli nawet programiści z wieloletnim doświadczeniem.

- Organizacja kodu
- Styl projektowy i styl kodowania
- Skalowalność kodu
- Racjonalna i efektywna obsługa błędów
- Prawidłowe stosowanie elementów języka
- Odpowiednie korzystanie z STL
- Bezpieczeństwo typów

Usprawnij pracę, stosując standardy kodowania — gdy za parę miesięcy będziesz musiał wrócić do swoich dzisiejszych programów, przekonasz się, że było warto.

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

Spis treści

Rozdział 2.
Styl projektowy

Głupcy ignorują złożoność. Pragmatycy od niej cierpią. Niektórzy potrafią jej unikać. Geniusze zaś ją eliminują

— Alan Perlis

Ale wiedziałem też, i zapomniałem, o powiedzeniu Hoare'a o tym, że przedwczesna optymalizacja to źródło wszelakiego zła w programowaniu

— Donald Knuth, z *The Errors of Tex* [Knuth98]

Trudno w pełni rozdzielić styl kodowania od stylu projektowania. Dlatego w tym rozdziale postaraliśmy się uwzględnić te wytyczne, które umykają uwadze, kiedy mowa o właściwym kodowaniu.

Niniejszy rozdział poświęcony jest zasadom i praktykom dającym się zastosować szerzej niż do pojedynczej klasy czy funkcji. Klasycznym przykładem jest zachowanie równowagi pomiędzy prostotą a przejrzystością kodu (patrz wytyczna 6.) czy unikanie przedwczesnej optymalizacji (wytyczna 8.), a także przedwczesnej pesymizacji (wytyczna 9.). Owe trzy wytyczne można stosować nie tylko na poziomie kodowania funkcji, ale również na poziomie wyższym, obejmującym kwestie projektowania klas i modułów oraz decyzje co do architektury aplikacji (owe wytyczne obowiązują wszystkich programistów — uważający inaczej powinni zerknąć raz jeszcze na stwierdzenie Donalda Knutha i sprawdzić, kogóż on z kolei cytował).

Wiele wytycznych z tego i następnych rozdziałów odnosi się do aspektów zarządzania zależnościami — kamienia węgielnego inżynierii oprogramowania i równocześnie zagadnienia powracającego w tej książce wielokrotnie. Pomyśl przez chwilę nad dowolną dobrą techniką inżynierii oprogramowania — dowolną *dobrą* techniką. Jakakolwiek by ona była, w ten czy inny sposób polega na redukcji zależności. Dziedziczenie? Zmniejsza zależność kodu pisanego pod kątem klasy bazowej od klas pochodnych. Redukcja liczby zmiennych globalnych? To jawna redukcja rozciągłych zależności w stosunku do widocznych rozlegle danych. Abstrakcja? To eliminacja zależności pomiędzy kodem manipulującym pojęciami a kodem implementującym te pojęcia.

Ukrywanie informacji (hermetyzacja)? Czyni kod użytkownika mniej zależnym od szczegółów implementacyjnych danej jednostki. Właściwa waga przykładana do zarządzania zależnościami przejawia się też w unikaniu wspólnych danych o stanie (wytyczna 10.), zaleceniu hermetyzacji informacji (wytyczna 11.) i wielu innych.

Naszym zdaniem najcenniejszą poradę w tym rozdziale zawiera wytyczna 6. — „Przede wszystkim poprawność, prostota i przejrzystość". W istocie, nic dodać, nic ująć.

Wytyczna 5. Jednej jednostce jedno zadanie

Streszczenie

Lepiej robić jedną rzecz, a dobrze. Wedle tej zasady należałoby nadawać poszczególnym jednostkom programu (zmiennym, klasom, funkcjom, przestrzeniom nazw, modułom, bibliotekom) jasno określone i równocześnie ograniczone zadania. W miarę rozrostu jednostki zakres jej zadań w sposób naturalny się zwiększa, nie powinien jednak obejmować coraz to nowych obszarów.

Uzasadnienie

Powiada się, że dobry pomysł na biznes to taki, który można ująć w jednym zdaniu. Podobna reguła tyczy się jednostek programu, które powinny mieć konkretne i jasno określone zadania.

Jednostka odpowiadająca za więcej niż jedno zadanie jest zwykle nieproporcjonalnie trudniejsza w użyciu niż zestaw jednostek prostszych o mniejszej odpowiedzialności, ponieważ jej implementacja obejmuje więcej niż tylko sumę intelektualnego wysiłki, złożoności i błędów w stosunku do jej poszczególnych składowych funkcjonalnych. Taka jednostka jest większa (zwykle niepotrzebnie) i trudniejsza do stosowania i ponownego wykorzystania. Zwykle też jednostka taka udostępnia okrojone interfejsy każdego z zadań — okrojone z racji częściowego pokrywania się różnych obszarów zadaniowych i rozmycia wizji implementacji każdego z nich.

Jednostki o łączonych zadaniach są zwykle trudniejsze z punktu widzenia projektowego i implementacyjnego. „Mnoga odpowiedzialność" oznacza wtedy zazwyczaj „mnogą osobowość" — kombinacyjną liczbę różnych możliwych stanów i zachowań. Dlatego zalecamy stosowanie prostych i jasnych, jednozadaniowych funkcji (patrz też wytyczna 39.), prostych klas oraz modułów o ściśle ograniczonym zakresie zadań.

Abstrakcje wyższego poziomu należy konstruować z prostszych abstrakcji niższego poziomu. Nie warto w żadnym razie grupować wielu abstrakcji niskiego poziomu w większym i bardziej złożonym konglomeracie niskiego poziomu. Implementacja złożonego zachowania na bazie szeregu prostszych jest bowiem łatwiejsza niż implementacja odwrotna.

Przykłady

Przykład 1. — wywołanie `realloc()`. W standardowym języku C `realloc()` to jeden z typowych przykładów ułomnego projektu. Funkcja `realloc()` ma zdecydowanie za dużo zadań: dla wskaźnika pustego przydziela pamięć, dla zerowego argumentu rozmiaru zwalnia wskazywaną pamięć, zaś dla pozostałych wartości argumentów zmienia

rozmiar przydzielonej pamięci, przy czym nowy przydział w części pokrywa się w przestrzeni adresowej z poprzednim, a jeśli jest to niemożliwe, wykonywany jest zupełnie nowy przydział. Trudno o lepszy przykład wadliwego projektu funkcji.

Przykład 2. — `basic_string`. Klasa `std::basic_string` to w standardzie języka C++ równie niesławny przykład monolitycznego projektu klasy. Klasa ta została rozepchana zbyt wielką liczbą (nawet użytecznych i przyjemnych) dodatków — przez to, choć aspiruje do miana kontenera, nie jest nim do końca, nie może bowiem wybrać pomiędzy indeksowaniem a iteracją i równocześnie powiela wiele standardowych algorytmów, nie zostawiając za to za wiele miejsca na rozszerzenia (patrz przykład do wytycznej 44.).

Źródła

[Henney02a] ◆ [Henney02b] ◆ [McConnell93] §10.5 ◆ [Stroustrup00] §3.8, §4.9.4, §23.4.3.1 ◆ [Sutter00] §10, §12, §19, §23 ◆ [Sutter02] §1 ◆ [Sutter04] §37–40

Wytyczna 6. Przede wszystkim poprawność, prostota i przejrzystość

Streszczenie

Wedle zasady KISS (*Keep It Simple Software* — parafraza *Keep It Simple, Stupid*, czyli „jak najprościej, głupku") im prościej, tym lepiej. Proste jest niemal zawsze lepsze od złożonego. Przejrzyste zaś jest lepsze od niejasnego. No i bezpieczne jest lepsze od niebezpiecznego (patrz wytyczne 83. i 99.).

Uzasadnienie

Trudno przecenić znaczenie prostoty projektu i przejrzystości kodu. Programista tworzący kod czytelny i zrozumiały będzie cieszył się wdzięcznością ze strony przyszłego opiekuna tego kodu. Powinieneś przy tym pamiętać, że opiekę nad kodem często sprawują jego twórcy i, mając to na uwadze, dbać o swoje samopoczucie w przyszłości. Stąd klasyczne prawdy w rodzaju:

> *Programy muszą być pisane tak, aby dały się czytać przez ludzi, ewentualnie od czasu do czasu wykonywać przez maszyny*
>
> — Harold Abelson i Gerald Jay Sussman

> *Pisz programy przede wszystkim dla ludzi, potem dla komputerów*
>
> — Steve McConnell

> *Najtańszymi, najszybszymi i najbardziej niezawodnymi komponentami systemu komputerowego są te, których w nim nie ma*
>
> — Gordon Bell

> *Owe brakujące komponenty są również najdokładniejsze (nigdy się nie mylą), najbezpieczniejsze (nie da się do nich włamać) i najprostsze w projektowaniu, dokumentowaniu, testowaniu i konserwacji. Nie sposób przecenić prostoty projektowej*
>
> — Jon Bentley

Wiele wytycznych prezentowanych w tej książce ma ukierunkować czytelnika na kod i projekty łatwe w modyfikacji; przejrzystość i zrozumiałość to najbardziej pożądane cechy prostych w konserwacji i rozbudowie programów. Trudno zmienić to, czego się nie rozumie.

Najsilniejsza sprzeczność zachodzi pomiędzy przejrzystością kodu a jego optymalizacją (patrz wytyczne 7., 8. i 9.). Kiedy (a nie jeżeli!) staniesz w obliczu pokusy przedwczesnej optymalizacji kodu pod kątem wydajności, a kosztem przejrzystości, przypomnij sobie sens wytycznej 8. — dużo łatwiej jest przyspieszyć poprawny program, niż poprawić szybki.

Unikaj więc „zaułków” języka programowania i stosuj zawsze najprostsze z efektywnych technik.

Przykłady

Przykład 1. — unikaj zbędnego (choć efektownego) przeciążania operatorów. Jedna z (niepotrzebnie) udziwnionych bibliotek graficznego interfejsu użytkownika wymagała, celem dodania do widgetu `w` elementu sterującego `c`, napisania wyrażenia `w + c;` (zobacz wytyczną 26.).

Przykład 2. — w roli parametrów konstruktorów stosuj zmienne nazwane, nie tymczasowe. Pozwala to na uniknięcie niejednoznaczności deklaracji. Pozwala też na lepszą prezentację zadania realizowanego przez kod i tym samym uproszczenie konserwacji programu. Jest też niejednokrotnie bezpieczniejsze (zobacz wytyczne 13. i 31.).

Źródła

[Abelson96] ◆ [Bentley00] §4 ◆ [Cargill92] pp.91–93 ◆ [Cline99] §3.05–06 ◆ [Constantine95] §29 ◆ [Keffer95] p. 17 ◆ [Lakos96] §9.1, §10.2.4 ◆ [McConnell93] ◆ [Meyers01] §47 ◆ [Stroustrup00] §1.7, §2.1, §6.2.3, §23.4.2, §23.4.3.2 ◆ [Sutter00] §40–41, §46 ◆ [Sutter04] §29

Wytyczna 7. Jak i kiedy kodować z uwzględnieniem skalowalności

Streszczenie

Wystrzegaj się wybuchowego rozrostu kodu — unikając przedwczesnej optymalizacji, kontroluj równocześnie złożoność asymptotyczną kodu. Algorytmy działające na danych użytkownika powinny cechować się liniową złożonością, czyli liniowym przyrostem czasu wykonania przy przyroście ilości przetwarzanych danych. Tam, gdzie optymalizacja okaże się niezbędna, i zwłaszcza gdy zostanie wymuszona zwiększeniem ilości danych, skupiaj się raczej na uzyskaniu sensownej złożoności obliczeniowej algorytmu niż na urywaniu tu i ówdzie po jednej instrukcji maszynowej.

Uzasadnienie

Niniejsza wytyczna ilustruje punkt równowagi pomiędzy wytycznymi 8. i 9. („unikaj przedwczesnej optymalizacji" i „unikaj przedwczesnej pesymizacji)". Z tego względu tę wytyczną dość ciężko sformułować tak, aby nie mylić jej sensu z sensem wytycznej 8. Ale do rzeczy.

Oto tło zagadnienia: pojemności pamięci ulotnych i dysków twardych rosną wykładniczo; w latach od 1988 do 2004 pojemność dysków rosła o 112 procent rocznie (co daje w ciągu dekady wzrost blisko 1900-krotny), podczas gdy prawo Moore'a zakłada przyrost zaledwie 59-procentowy (100-krotny w ciągu dekady). Jedną z konsekwencji tej dynamiki jest to, że czynności realizowane dziś przez kod mogą jutro obejmować znacznie większe ilości danych — *znacznie* większe. Jeśli stosowane do ich przetwarzania algorytmy będą cechować się kiepską asymptotyczną złożonością obliczeniową, wcześniej czy później przestaną się nadawać do wykorzystywania nawet na najwydajniejszych systemach komputerowych — to tylko kwestia ilości danych, którymi te algorytmy będą „karmione".

Obrona przed tą wątpliwą karierą algorytmu polega na unikaniu „wbudowywania w projekt" takich elementów, które w obliczu konieczności przetwarzania plików większych niż dziś przewidywane (większych baz danych, większej liczby pikseli, większej liczby okien, większych szybkości transmisji) okażą się jego wąskimi gardłami. W przypadku biblioteki standardowej języka C++ elementami zabezpieczającymi przyszłość są choćby gwarancje co do złożoności obliczeniowej algorytmów i operacji na kontenerach.

Oto wniosek: nie powinniśmy przedwcześnie optymalizować programu przez zastosowanie w nim mniej przejrzystego algorytmu, jeśli spodziewany przyrost ilości przetwarzanych danych nie jest pewny. Ale równie niewskazana jest przedwczesna pesymizacja

algorytmu, polegająca na zamykaniu oczu na jego niekorzystną asymptotyczną złożoność obliczeniową (rozumianą jako koszt wykonania obliczeń w funkcji liczby przetwarzanych elementów).

Uzasadnienie to można by podzielić na dwie części. Po pierwsze, nawet przed poznaniem docelowego woluminu danych i oszacowaniem, czy jego rozmiar będzie miał istotny wpływ na wydajność danego kodu, należy unikać takich algorytmów operujących na danych użytkownika, które kiepsko się skalują, chyba że dzięki zastosowaniu takiego algorytmu zyskamy na czytelności albo przejrzystości kodu (patrz wytyczna 6.). Zbyt często programiści są jednak zaskakiwani — piszą kod z myślą o tym, że nigdy nie przyjdzie mu operować na olbrzymich zbiorach danych (i w dziewięciu przypadkach na dziesięć nie mylą się). Jednak w tym jednym na dziesięć przypadku wpadają w pułapkę braku wydajności — zdarzało się to nam i z pewnością zdarzy się (prędzej czy później) również czytelnikowi. Z pewnością można wtedy opracować poprawkę i dostarczyć ją klientom, ale znacznie lepiej byłoby uniknąć tego rodzaju zakłopotania i wysiłku. Więc, jeśli pozostałe cechy kodu (w tym czytelność i przejrzystość kodu) na tym nie ucierpią, od początku warto:

- *w miejsce tablic o stałych rozmiarach stosować elastyczne, przydzielane dynamicznie struktury danych.* Statyczne tablice „większe, niż kiedykolwiek będą potrzebne" to obraza dla poprawności i bezpieczeństwa programu (patrz wytyczna 77.). Są one do zaakceptowania jedynie wtedy, kiedy rozmiar danych jest faktycznie ustalony i stały!
- *znać faktyczną złożoność algorytmu* — szczególnie groźne są takie algorytmy, których złożoność obliczeniowa jest z pozoru liniowa, ale które wywołują wewnętrznie inne operacje o liniowej złożoności, dając w efekcie złożoność kwadratową (przykład w wytycznej 81.).
- *wszędzie tam, gdzie to możliwe, preferować algorytmy o złożoności liniowej i lepszej* — najlepsza byłaby stała złożoność w funkcji liczby elementów, jak w przypadku operacji `push_back` na kontenerach albo operacji wyszukiwania w tabeli haszowanej (patrz wytyczne 76. i 80.). Niezła jest złożoność logarytmiczna (*O(logN)*), osiągana między innymi w operacjach na kontenerach `set` i `map` czy operacjach `lower_bound` i `upper_bound` z iteratorami swobodnego dostępu (patrz wytyczne 76., 85. i 86.). Do zaakceptowania jest złożoność liniowa (*O(N)*), jak w operacjach `vector::insert` albo algorytmie `for_each` (zobacz wytyczne 76., 81. i 84.).
- *tam, gdzie to zasadne, unikać algorytmów o złożoności gorszej niż liniowa* — na przykład w obliczu algorytmu o złożoności rzędu *O(N log N)* albo $O(N^2)$ warto spędzić trochę czasu na poszukiwaniu rozwiązań o mniejszej złożoności obliczeniowej, celem uniknięcia pułapki wydajnościowej wynikającej z przewidywanej dynamiki wzrostu ilości przetwarzanych danych. Z tego właśnie powodu w wytycznej 81. doradzamy preferowanie metod przetwarzających całe sekwencje elementów (metody te cechują się złożonością liniową) zamiast wywołań ich odpowiedników przetwarzających pojedyncze elementy (ponieważ w przypadku wywołania w operacji o liniowej złożoności innej takiej operacji otrzymujemy złożoność kwadratową; patrz też przykład 1. w ramach wytycznej 81.).

- *nie stosować nigdy algorytmu wykładniczego, chyba że nie ma innego* — w obliczu konieczności zastosowania algorytmu o wykładniczej złożoności obliczeniowej nie wolno szczędzić wysiłku na poszukiwania alternatywy, ponieważ w tym przypadku nawet nieznaczne zwiększenie rozmiaru przetwarzanych danych stanowi istną barierę wydajnościową.

Z drugiej strony, po wykonaniu pomiarów dowodzących, że optymalizacja jest zasadna i ważna, zwłaszcza z uwagi na rosnące ilości danych do przetworzenia, powinniśmy skupić się na redukcji złożoności obliczeniowej, nie próbując szukać ratunku w pomniejszych optymalizacjach, urywających tu i ówdzie po jednej czy parę instrukcji maszynowych.

Reasumując — wszędzie tam, gdzie to możliwe, korzystaj z algorytmów o złożoności liniowej albo lepszej. Staraj się unikać algorytmów o złożoności wielomianowej; jak ognia wystrzegaj się zaś algorytmów o złożoności wykładniczej.

Źródła

[Bentley00] §6, §8, dod. A ◆ [Cormen01] ◆ [Kernighan99] §7 ◆ [Knuth97a] ◆ [Knuth97b] ◆ [Knuth98] ◆ [McConnell93] §5.1–4, §10.6 ◆ [Murray93] §9.11 ◆ [Sedgewick98] ◆ [Stroustrup00] §17.1.2

Wytyczna 8. Wystrzegaj się przedwczesnej optymalizacji

Streszczenie

Nie bodzie się chętnego wierzchowca. Przedwczesna optymalizacja jest równie uzależniająca, jak bezproduktywna, pierwsza reguła optymalizacji mówi bowiem: zaniechaj jej. Druga reguła (dla ekspertów) mówi zaś: powstrzymaj się jeszcze. Jedną optymalizację trzeba poprzedzić dwoma pomiarami dowodzącymi jej konieczności.

Uzasadnienie

We wstępie do [Stroustrup00] §6 znajdziemy świetne cytaty:

> *Przedwczesna optymalizacja to źródło wszelkiego zła*
>
> — Donald Knuth (cytujący z kolei z Hoare'a)

> *Z drugiej strony, nie możemy ignorować efektywności*
>
> — Jon Bentley

Hoare i Knuth mają (jak zwykle) rację (patrz wytyczna 6. i niniejsza). Tak jak i Bentley (wytyczna 9.).

Przedwczesną optymalizację zdefiniowalibyśmy jako zwiększanie złożoności projektu albo kodu, a przez to zmniejszenie ich czytelności, w imię wydajności, której potrzeba zwiększenia nie została jeszcze dowiedziona (na przykład pomiarami i porównaniem ich wyników z założonymi celami) — jako taka optymalizacja ta nie wnosi do projektu żadnych korzyści. Często przedwczesna i niepoparta pomiarami optymalizacja, mimo włożonego w nią wysiłku, nie daje dosłownie żadnego efektu wydajnościowego.

Warto więc zapamiętać, że:

> *Znacznie łatwiej przyspieszyć poprawny program, niż poprawić szybki!*

Nie należy więc od początku skupiać się na szybkości kodu — w pierwszej kolejności powinna nas interesować raczej jego przejrzystość i czytelność (zgodnie z wytyczną 6.). W kodzie czytelnym łatwiej o poprawność, zrozumienie jego działania, wprowadzanie poprawek i zmian, i wreszcie optymalizację. Komplikacje, nieodzowne dla optymalizacji, zawsze można wprowadzić później — i tylko wtedy, gdy są niezbędne.

Przedwczesna optymalizacja często nie daje spodziewanych efektów z dwóch głównych powodów. Po pierwsze, programiści stale mylą się w swoich szacunkach co do szybkości danego kodu i typowania jego wąskich gardeł. Dotyczy to nas, autorów tej książki, i dotyczy najprawdopodobniej również Ciebie. Współczesne komputery realizują

potwornie skomplikowany model maszyny obliczeniowej, obejmującej niekiedy kilka czy kilkanaście potoków przetwarzania obsługiwanych współbieżnie, z rozbudowaną hierarchią pamięci podręcznych, predykcją rozgałęzień programu — i wszystko to w jednym procesorze! Kompilatory, bazujące na tych możliwościach, również starają się transformować kod źródłowy tak, aby wynikowy kod maszynowy jak najlepiej wpasował się w architekturę procesora. Dopiero na bazie kompilatora operuje programista, i jeśli dla poparcia swoich decyzji ma jedynie nieścisłe szacunki i swoją intuicję, to szansa, że wprowadzane przez niego mikrooptymalizacje będą miały znaczący wpływ na program, jest prawie żadna. Jak widać, optymalizację należy koniecznie poprzedzić odpowiednimi pomiarami. Dopóki nie uda się w ten sposób dowieść istotnej potrzeby optymalizacji, należy skupić się na kwestii najważniejszej, czyli tworzeniu zrozumiałego i czytelnego kodu (jeśli ktoś zażąda optymalizacji programu, żądaj dowodów jej konieczności).

Dalej, we współczesnych programach efektywność znacznej części operacji nie jest już ograniczana wydajnością procesora. Znakomita część z nich znacznie bardziej ograniczona jest efektywnością dostępu do pamięci, szybkością transmisji w sieci, czasem dostępu do napędów pamięci masowych, czasem oczekiwania na odpowiedź serwera WWW czy serwera baz danych. Wobec tego optymalizacja kodu aplikacji wykonującej wszystkie te operacje spowoduje jedynie, że aplikacja będzie szybciej na nie czekać. A to oznaczałoby, że programista zmarnował sporo czasu, ulepszając to, co ulepszenia nie wymagało, zamiast ulepszać to, co faktycznie kuleje.

Oczywiście nadejdzie wreszcie ten dzień, kiedy kod trzeba będzie nieco zoptymalizować. W takim przypadku w pierwszej kolejności należy szukać ratunku w zmniejszeniu złożoności obliczeniowej algorytmu (wytyczna 7.) i równocześnie próbować hermetyzować i ograniczać zasięg optymalizacji (na przykład do funkcji albo klasy — patrz wytyczne 5. i 11.) oraz koniecznie opatrzyć kod stosownymi komentarzami, wyjaśniającymi bieżące potrzeby optymalizacji i odnoszącymi się do zastosowanych algorytmów.

Powszechnym błędem początkujących programistów jest pisanie — z dumą! — nowego kodu z obsesyjną myślą o jego jak największej wydajności, kosztem czytelności i zrozumiałości. Najczęściej efektem takiej pracy jest kod spaghetti, który — nawet jeśli poprawny — utrudnia analizę i ewentualne modyfikacje (wytyczna 6.)

Nie jest przedwczesną optymalizacją przekazywanie argumentów i wartości zwracanych przez referencję (patrz wytyczna 25.), preferowanie przedrostkowych wersji operatorów inkrementacji (wytyczna 28.) i tym podobne idiomy, które w naturalny sposób wpasowują się w tok pracy programisty. Nie są to optymalizacje przedwczesne, ponieważ nie komplikują kodu — pozwalają za to uniknąć przedwczesnej jego pesymizacji (patrz wytyczna 9.).

Przykłady

Przykład — ironia `inline`. Oto ilustracja ukrytego kosztu przedwczesnej mikrooptymalizacji. Otóż narzędzia profilujące służą do tego, aby na podstawie licznika wywołań funkcji informować programistę o tym, które z funkcji nadają się do rozwijania w miejscu

wywołania, a nie zostały jako takie oznaczone. Niestety, nawet najlepsze takie narzędzie nie będzie w stanie wskazać takich funkcji, które zostały oznaczone jako rozwijane w miejscu wywołania (`inline`), choć nie powinny — nie będzie bowiem żadnej możliwości określenia liczby „wywołań" tejże funkcji w kodzie wynikowym. Zbyt często programiści w imię optymalizacji decydują się na rozwijanie wielu funkcji w miejscu wywołania, co mało kiedy przynosi rzeczywiste korzyści (zakładając, że kompilator nie ignoruje zupełnie słowa `inline` — patrz [Sutter00], [Sutter02] czy [Sutter04]).

Wyjątki

Twórca kodu biblioteki ma zadanie utrudnione o tyle, że nie bardzo może przewidzieć, które z jej elementów będą w przyszłości wykorzystywane w kodzie czułym na wydajność wykonania. Ale nawet twórcy bibliotek powinni poprzedzić optymalizację testami na szerokiej bazie klientów-użytkowników biblioteki.

Źródła

[Bentley00] §6 ◆ [Cline99] §13.01–09 ◆ [Kernighan99] §7 ◆ [Lakos96] §9.1.14 ◆ [Meyers97] §33 ◆ [Murray93] §9.9–10, §9.3 ◆ [Stroustrup00] §6 (wprowadzenie) ◆ [Sutter00] §30, §46 ◆ [Sutter02] §12 ◆ [Sutter04] §25

Wytyczna 9. Wystrzegaj się przedwczesnej pesymizacji

Streszczenie

Jeśli pozostałe czynniki (jak choćby czytelność kodu czy jego złożoność) nie ucierpią na tym, to pewne wzorce projektowe, praktyki programistyczne i tym podobne idiomy programistyczne należy uznać za o tyle naturalne, że ich wprowadzenie nie wiąże się dla programisty ze zwiększonym wysiłkiem i niejako same wychodzą spod jego palców. Nie uznajemy ich za przedwczesną optymalizację, a raczej za unikanie niepotrzebnej pesymizacji.

Uzasadnienie

Unikanie przedwczesnej pesymizacji nie może oznaczać wzrostu efektywności, jeśli osiąga się go znacznym kosztem. Przedwczesną pesymizację będziemy rozumieć jako niepotrzebne potencjalne ograniczenia efektywności, takie jak:

- definiowanie parametrów jako przekazywanych przez wartość tam, gdzie można by je przekazywać przez referencję (patrz wytyczna 25.);
- stosowanie przyrostkowych wersji operatorów inkrementacji tam, gdzie można by zastosować wersje przedrostkowe (patrz wytyczna 28.);
- wykonywanie przypisań w ciele konstruktora, a nie w liście inicjalizacyjnej (patrz wytyczna 48.).

Nie jest również przedwczesną optymalizacją redukowanie liczby niepotrzebnych tymczasowych kopii obiektów, zwłaszcza w pętlach wewnętrznych i zwłaszcza wtedy, kiedy ta redukcja nie wpływa na złożoność kodu. Co prawda wytyczna 18. zachęca do deklarowania zmiennych jak najbardziej lokalnie, wskazuje jednak na wyjątki, w obliczu których lepiej deklarację zmiennej przenieść poza pętlę. W większości przypadków takie przesunięcia nie mają znaczenia dla przejrzystości kodu, a nawet pozwalają na lepsze uwidocznienie operacji wykonywanych w pętli i wyodrębnienie niezmienników tej pętli. Oczywiście najlepiej w miejsce jawnych pętli stosować algorytmy STL (patrz wytyczna 84.).

Przejrzystość godzi się z efektywnością przez stosowanie abstrakcji i bibliotek (patrz wytyczne 11. i 36.). Na przykład, korzystając ze standardowych elementów bibliotecznych języka C++ (kontenerów `vector`, `list`, `map`, algorytmów `sort` czy `find`), projektowanych i implementowanych przez światowej klasy ekspertów, nie tylko zwiększamy przejrzystość kodu, ale niejednokrotnie znacznie go przyspieszamy.

Unikanie przedwczesnej pesymizacji jest szczególnie istotne dla twórców bibliotek. Zazwyczaj nie mają oni możliwości przewidzenia wszystkich kontekstów, w których ich kod zostanie wykorzystany, powinni więc przesunąć nieco środek ciężkości w kierunku efektywności i modułowości (przydatności do ponownego wykorzystania), wystrzegając się jednak przesady w zwiększaniu efektywności, jeśli przyrost ten odczuje jedynie niewielki odsetek potencjalnych użytkowników biblioteki. Wyznaczenie punktu ciężkości to oczywiście zadanie programisty, ale zgodnie z wytyczną 7., nacisk należy położyć raczej na uzyskanie rozsądnej skalowalności, niż na mikrooptymalizacje polegające na urywaniu pojedynczych cykli procesora.

Źródła

[Keffer95] pp. 12–13 ◆ [Stroustrup00] §6 (wprowadzenie) ◆ [Sutter00] §6

Wytyczna 10. Minimalizuj ilość danych globalnych i współużytkowanych

Streszczenie

Współużytkowanie oznacza rywalizację — należy więc unikać danych wspólnych, zwłaszcza globalnych. Zwiększają one powiązanie kodu kosztem łatwości konserwacji, a niejednokrotnie i wydajności.

Uzasadnienie

Niniejsza wytyczna jest pewnym uogólnieniem wytycznej 18.

Chodzi o unikanie stosowania danych (wiązanych zewnętrznie) o zasięgu pokrywającym się z zasięgiem przestrzeni nazw albo występujących w postaci statycznych składowych klas. Komplikują one logikę programu i uściślają związki pomiędzy różnymi (i, co gorsza, odległymi) elementami programu. Współużytkowanie danych zmniejsza możliwości testowania jednostki programu, ponieważ poprawność kodu odwołującego się do takich danych jest mocno uzależniona od historii zmian tych danych i warunków wykonania dalszego, nieznanego bliżej kodu, który się później do tych danych odwołuje.

Nazwy obiektów w globalnej przestrzeni nazw zaśmiecają tę przestrzeń, zwiększając ryzyko kolizji nazw.

Jeśli już trzeba zastosować obiekt globalny, obiekt o zasięgu pokrywającym się z zasięgiem przestrzeni nazw albo statyczny obiekt klasy, należy starannie go zainicjalizować. Porządek inicjalizacji tego rodzaju obiektów w różnych jednostkach kompilacji jest niezdefiniowany i aby zapewnić jego poprawność, trzeba wdrożyć specjalne techniki (odsyłamy do źródeł). Reguły kolejności inicjalizacji są subtelne — lepiej unikać konieczności zagłębiania się w te subtelności, a jeśli jest to niemożliwe, warto je przynajmniej dobrze poznać i starannie stosować.

Obiekty o zasięgu przestrzeni nazw, składowe statyczne oraz obiekty dzielone przez wiele wątków albo procesów redukują zrównoleglenie w środowiskach wielowątkowych i wieloprocesorowych i są częstymi wąskimi gardłami wydajności i skalowalności (patrz wytyczna 7.). Optuj za zasadą „jak najmniej wspólnego” — zamiast danych wspólnych (współużytkowanych) stosuj komunikację pomiędzy użytkownikami danych (np. kolejki komunikatów).

Całość sprowadza się zaś do unikania ścisłych zależności i do minimalizacji interakcji pomiędzy klasami (patrz [Cargill92]).

Wyjątki

Za wyjątki można uznać takie mechanizmy, jak obiekty `cin`, `cout` i `cerr`, implementowane celowo jako obiekty globalne. Dalej, np. fabryka (generator obiektów według wzorca projektowego Factory) musi utrzymywać rejestr funkcji do wywołania celem utworzenia obiektu danego typu i zwykle w programie znajduje się jeden taki rejestr (powinien być on jednak obiektem wewnętrznym fabryki, a nie współużytkowanym obiektem globalnym; patrz wytyczna 11.).

Kod zakładający współużytkowanie obiektów przez wiele wątków powinien zawsze szeregować wszelkie odwołania do owych obiektów (patrz wytyczna 12. oraz [Sutter04c]).

Źródła

[Cargill92] pp.126–136, 169–173 ◆ [Dewhurst03] §3 [Lakos96] §2.3.1 ◆ [McConnell93] §5.1–4 ◆ [Stroustrup00] §C.10.1 ◆ [Sutter00] §47 ◆ [Sutter02] §16, dod. A ◆ [Sutter04c] ◆ [SuttHysl03]

Wytyczna 11. Ukrywaj informacje

Streszczenie

Nie eksponuj wewnętrznych informacji jednostki stanowiącej abstrakcję.

Uzasadnienie

Minimalizacja zależności pomiędzy wywołującym, manipulującym pewną abstrakcją, a wywoływanym, czyli implementacją tej abstrakcji, wymaga ukrywania danych wewnętrznych tej implementacji. W przeciwnym razie wywołujący może się do owych informacji odwoływać (albo, co gorsza, manipulować nimi) z pominięciem implementacji abstrakcji. Eksponować należy raczej samą abstrakcję (nawet, jeśli ma ona jedynie postać akcesorów `set-get`), a nie jej dane.

Ukrywanie informacji zmniejsza koszt projektu, skraca harmonogram realizacji lub (i) ryzyko jego przekroczenia, dzięki:

- *ograniczaniu zasięgu zmian* — ukrywanie informacji redukuje efekt domina w przypadku zmian, a więc redukuje ich koszt.
- *wzmocnieniu niezmienników* — przez ograniczanie kodu odpowiedzialnego za zachowanie (albo i złamanie) niezmienników programu (patrz wytyczna 41.).

Nie powinno się eksponować danych żadnej jednostki, która stanowi abstrakcję (patrz też wytyczna 10.), konkretne dane są bowiem charakterystyczne jedynie dla jednego z możliwych wcieleń abstrakcji, jednego z jej konceptualnych stanów. Jeśli skupić się na koncepcjach, a nie ich reprezentacjach wewnętrznych, to dla tej samej abstrakcji i wspólnego interfejsu można udostępnić całkowicie różne implementacje (na przykład obliczenia z buforowaniem wyników w jednej albo realizowane „w locie" w innej), wykorzystujące odmienne reprezentacje wewnętrznych danych (na przykład współrzędne w układzie biegunowym albo kartezjańskim).

Powszechnie uważa się, że nie należy dokonywać ekspozycji składowych danych klas przez oznaczanie ich jako publicznych (wytyczna 41.) albo przez udostępnianie ich wskaźników czy uchwytów (wytyczna 42.). Tyczy się to jednak również jednostek większych od klas, takich jak bibliotek, które również nie powinny eksponować danych implementacyjnych. Moduły i biblioteki powinny raczej udostępniać interfejsy definiujące abstrakcje i transfery między nimi — pozwala to na bezpieczniejsze komunikowanie się z wywołującym i mniej ścisłe powiązanie wywołującego z biblioteką, niż to ma miejsce przy stosowaniu danych współużytkowanych.

Wyjątki

Wyjątkiem może być kod testujący, niejednokrotnie wymagający swobodnego dostępu do danych testowanych klas i modułów.

Regule ukrywania danych nie podlegają również agregaty wartości (np. znane z języka C struktury), stanowiące jedynie zlepek danych, dla których nie przewidziano abstrakcji behawioralnej — dane te stanowią wtedy równocześnie swój własny (jedyny) interfejs (zobacz wytyczna 41.).

Źródła

[Brooks95] §19 ◆ [McConnel] §6.2 ◆ [Parnas02] ◆ [Stroustrup00] §24.4 ◆ [SuttHysl04a]

Wytyczna 12. Niepotrzebna rywalizacja to niezdrowa rywalizacja

Streszczenie

Bezpieczeństwo wątkowe to podstawa — jeśli aplikacja wykorzystuje wiele wątków czy procesów, programista musi wiedzieć, jak ma minimalizować współużytkowanie obiektów (zobacz wytyczna 10.) i jak bezpiecznie użytkować te, które muszą pozostać wspólne.

Uzasadnienie

Wątki to obszerne zagadnienie. Jego waga wymaga potwierdzenia w wydzieleniu dla niego osobnej wytycznej. W ramach jednej takiej wytycznej nie sposób jednak ująć wszystkiego, co związane z programowaniem wątków, ograniczymy się więc do podsumowania kilku kwestii zasadniczych — po szczegóły odsyłamy zaś do źródeł. Za kwestie najważniejsze uważamy zaś unikanie zakleszczeń, unikanie zawłaszczania zasobów i unikanie szkodliwej rywalizacji w dostępie do zasobów (i ich uszkodzenia w wyniku niewystarczającego blokowania).

Standard języka C++ nie poświęca wątkom ani słowa. Mimo tego język ten jest rutynowo i powszechnie wykorzystywany do pisania solidnych, wielowątkowych aplikacji. Jeśli więc Twój program dzieli dane pomiędzy wątki, niech robi to bezpiecznie:

- *Sprawdź w dokumentacji platformy docelowej dostępność elementarnych mechanizmów synchronizacji lokalnej* — od niepodzielnych maszynowych operacji na wartościach całkowitych po bariery pamięciowe i blokady wewnątrzprocesowe i międzyprocesowe.
- *Spróbuj ująć owe elementarne mechanizmy we własnych abstrakcjach* — to dobry pomysł, zwłaszcza jeśli program ma docelowo działać na wielu platformach. Alternatywnie można skorzystać z gotowych bibliotek tego rodzaju (np. biblioteki pthreads [Butenhof97]).
- *Upewnij się, że wykorzystywane typy mogą być bezpiecznie stosowane w programie wielowątkowym* — w szczególności każdy z takich typów powinien:
  - *gwarantować niezależność obiektów niewspółużytkowanych*. Dwa wątki powinny móc swobodnie korzystać z dwóch różnych obiektów.
  - *dokumentować wymagania odnośnie do wywołującego, jeśli ten chce odwoływać się do tego samego obiektu z różnych wątków*. Część typów wymaga szeregowania dostępu do tak współużytkowanych obiektów, inne obchodzą się bez takiej synchronizacji. W przypadku tych ostatnich brak konieczności blokowania i synchronizacji dostępu wynika zazwyczaj

z projektu typu, ewentualnie z zastosowania synchronizacji wewnętrznej — w którym to przypadku programista powinien być świadom ograniczeń owego wewnętrznego blokowania i znać jego szczegółowość.

Zauważ, że powyższe reguły dotyczą wszelkich typów, bez wyróżniania typów łańcuchowych, kontenerów, kontenerów STL czy jakichkolwiek innych (zauważyliśmy bowiem, że niektórzy autorzy wyróżniają tutaj kontenery STL jako w jakiś sposób szczególne, tymczasem obiekty te nie wyróżniają się niczym w tym zakresie). W szczególności, gdy zamierzamy wykorzystać w programie wielowątkowym komponenty biblioteki standardowej, powinniśmy sprawdzić w dokumentacji biblioteki, czy jej implementacja daje taką możliwość.

Tworząc własne typy przeznaczone do wykorzystywania w programach wielowątkowych, musimy zadbać o te same dwa elementy: po pierwsze, zagwarantować niezależność (niewymagającą blokowania) różnych egzemplarzy tego danego typu (podpowiedź: typ z modyfikowalną składową statyczną nie daje takiej gwarancji); po drugie, udokumentować wymagania co do użytkowników typu w zakresie stosowania wspólnego obiektu tego typu w różnych wątkach. W tej kwestii zasadnicze znaczenie ma problem rozłożenia pomiędzy klasą a jej użytkownikami odpowiedzialności za poprawne wykonanie programu. Mamy w tym zakresie trzy podstawowe możliwości:

- *Blokowanie zewnętrzne — za blokowanie odpowiedzialny jest wywołujący (użytkownik).* W tym układzie kod korzystający z obiektu jest w pełni odpowiedzialny za synchronizację odwołań do tego obiektu, jeśli jest on wykorzystywany w wielu wątkach. Z blokowania zewnętrznego korzystają zazwyczaj typy łańcuchowe (często uciekają się też do niezmienności, zobacz opis trzeciej opcji).
- *Blokowanie wewnętrzne — każdy obiekt samodzielnie szereguje odwołania do niego, zwykle przez blokady wszystkich metod publicznych, zwalniające użytkowników z odpowiedzialności za synchronizację dostępu do obiektu.* Przykładowo, w kolejkach producentów-konsumentów stosowane jest blokowanie wewnętrzne, ponieważ obiekty te są z zasady przeznaczone do współużytkowania przez wątki i ich interfejsy są zaprojektowane z uwzględnieniem blokowania niezbędnego do bezpiecznego wykonania każdej z metod. Ta opcja jest właściwa w przypadkach, kiedy z góry wiadomo, że:
  - obiekty danego typu będą wykorzystywane niemal wyłącznie jako współużytkowane w wielu wątkach — jeśli nie jest to pewne, blokowanie wewnętrzne będzie w znacznej części zbędne. Wypada zauważyć, że niewiele typów spełnia to wymaganie — znakomita większość obiektów nawet w programie silnie wielowątkowym nie podlega współużytkowaniu przez wątki (co nie jest bynajmniej zarzutem — zobacz wytyczna 10.).
  - blokowanie poszczególnych metod pozwoli osiągnąć odpowiednią szczegółowość synchronizacji, odpowiednią dla większości użytkowników. W szczególności interfejs typu powinien faworyzować operacje „gruboziarniste", samowystarczalne. Jeśli typowy użytkownik będzie musiał z zasady blokować kilka operacji zamiast jednej, to indywidualne blokowanie metod nie zda egzaminu. Będzie musiało być poparte

blokowaniem ogólniejszym (prawdopodobnie zewnętrznym, pozostającym w gestii użytkownika), pozwalającym na szeregowanie nie pojedynczych operacji, a całych transakcji. Weźmy choćby typ kontenera zwracającego iterator i problem unieważnienia iteratora przed jego użyciem. Albo typ udostępniający w postaci metody algorytm `find` zwracający wynik, którego poprawność zostanie zniesiona w czasie pomiędzy utworzeniem obiektu a wywołaniem metody. Albo kiedy użytkownik obiektu `c` pewnego typu zechce wykonać operację `if (c.empty()) c.push_back(x);` (więcej przykładów w [Sutter02]). W takich przypadkach użytkownik będzie musiał — mimo wewnętrznego szeregowania dostępu do poszczególnych metod — wdrożyć na własną rękę blokadę, której czas życia obejmuje wiele kolejnych wywołań metod. W takim układzie ich blokowanie wewnętrzne traci zupełnie sens.

Jak widać, wewnętrzne blokowanie ma ścisły związek z publicznym interfejsem typu: jest właściwe, kiedy poszczególne operacje tego interfejsu są kompletne, czyli kiedy poziom abstrakcji typu zostanie podniesiony i wyrażony bardziej precyzyjnie (na przykład „kolejka producent-konsument" zamiast ogólnego „tablica"). Interfejsy te łączą elementarne manipulacje na typie do postaci operacji znaczących i użytecznych samych w sobie. Jeśli liczba takich kombinacji jest nie do przewidzenia i nie sposób wychwycić kombinacji najczęstszych, celem wyodrębnienia ich do „większych" operacji, mamy dwie możliwości: (a) zastosować model oparty na wywołaniach zwrotnych (kiedy użytkownik wywołuje pojedynczą metodę, przekazując do niej obiekt funkcyjny bądź funkcję, która ma posłużyć do realizacji rozleglejszego zadania — patrz wytyczne od 87. do 89.) albo (b) wyeksponować mechanizmy blokowania w interfejsie.

- *Bez blokowania, za to z założeniem niezmienności (w przypadku obiektów niemodyfikowalnych).* Można tak zaprojektować typy obiektów, aby ich blokowanie nie było w ogóle potrzebne (patrz źródła). Przykładem takiego projektu są obiekty niemodyfikowalne — szeregowanie dostępu do nich jest zbędne, ponieważ nie można ich zmieniać. Przykładem może być typ niemodyfikowalnego łańcucha znaków, którego obiektu nie można zmieniać w czasie życia, a każda operacja na łańcuchu powoduje utworzenie nowego obiektu z nowym łańcuchem znaków.

Warto pamiętać, że użytkownik niekoniecznie musi dysponować wiedzą co do szczegółów implementacji danego typu (zgodnie z wytyczną 11.). Jeśli Twój typ wykorzystuje ukryte mechanizmy zarządzające współużytkowaniem obiektów (np. opóźnianie kopiowania, tzw. „kopiowanie przy zapisie"), nie musisz brać odpowiedzialności za wszelkie możliwe kwestie związane z wielowątkowością, ale nie możesz zignorować odpowiedzialności za zapewnienie bezpieczeństwa wątkowego w stopniu wystarczającym do zapewnienia poprawności odwołań do obiektu przez użytkownika, jeśli ten dopełni swoich zwykłych obowiązków — typ powinien być co najmniej tak bezpieczny, jak byłby, gdyby nie stosował utajonych mechanizmów współużytkowania (zobacz [Sutter04c]). Wszystkie prawidłowo zdefiniowane typy powinny pozwalać na manipulowanie osobnymi, niezależnymi egzemplarzami z poziomu niezależnych wątków bez potrzeby jakiejkolwiek synchronizacji pomiędzy tymi egzemplarzami.

Twórca biblioteki przeznaczonej do powszechnego użytku powinien szczególnie rozważyć zabezpieczenie obiektów biblioteki przed interakcjami w środowisku wielowątkowym. Powinien do tego podejść w sposób opisany powyżej, ale tak, aby zabezpieczenia te nie powodowały znacznych narzutów w środowiskach wielowątkowych. Jeśli na przykład piszesz bibliotekę zawierającą typ stosujący kopiowanie przy zapisie i z tego względu wykorzystujący również jakieś wewnętrzne blokady, blokady te należy tak zaaranżować, aby w kompilacjach dla środowisk jednowątkowych były niewidoczne (można uciec się wtedy do dyrektyw `#ifdef` i pustych implementacji).

Zakładając wiele blokad, powinieneś unikać zakleszczeń i układać kod tak, aby we wszystkich miejscach pozyskiwania tegoż kompletu blokad kolejność ich zakładania była zawsze taka sama (zwalnianie blokad może być wtedy realizowane w dowolnym porządku). Rozwiązaniem problemu stałej kolejności zakładania blokad może być ich zakładanie według rosnących adresów w pamięci — bazując na adresach, możesz łatwo ustalić porządek blokowania wspólny dla całej aplikacji.

Źródła

[Alexandrescu02a] ◆ [Alexandrescu04] ◆ [Butenhof97] ◆ [Henney00] ◆ [Henney01] ◆ [Meyers04] ◆ [Shmidt01] ◆ [Stroustrup] §14.9 ◆ [Sutter02] §16 ◆ [Sutter04c]

Wytyczna 13. Zagwarantuj opiekę nad zasobami przez obiekty. Stosuj RAII i inteligentne wskaźniki

Streszczenie

Nie walaj rąk, jeśli masz narzędzia — idiom „pozyskanie zasobu to jego inicjalizacja" (RAII, od *resource acquisition is initialization*) to świetne narzędzie poprawnej obsługi zasobów. RAII pozwala kompilatorowi na udostępnianie silnych i automatycznych gwarancji, które w innych językach wymagają karkołomnych sztuczek programistycznych. Przydzielając surowy zasób, bezzwłocznie przekaż go do obiektu, który ma być jego dysponentem. I nigdy nie przydzielaj więcej niż jednego zasobu w pojedynczej instrukcji.

Uzasadnienie

Język C++, wymuszający symetrię wywołań konstruktorów i destruktorów, odwzorowuje w niej symetrię charakterystyczną dla par funkcji pozyskujących i zwalniających obiekty, takich jak `fopen()` i `fclose()`, `lock()` i `unlock()` czy operatorów `new` i `delete`. Dzięki temu przydzielany w pamięci stosu (albo implementowany ze zliczaniem odwołań) obiekt z pozyskującym zasoby konstruktorem i zwalniającym je destruktorem jest znakomitym narzędziem automatyzacji zarządzania zasobami.

Automatyzacja ta jest prosta w implementacji, elegancka, mało kosztowna i w swej istocie odporna na błędy. Jej odrzucenie oznacza obciążenie samego siebie niebanalnym i angażującym zadaniem ręcznego parowania wywołań pozyskujących i zwalniających zasoby, z uwzględnieniem wyjątków i wynikających z logiki programu rozgałęzień przepływu sterowania. Tego rodzaju przywiązanie do języka C i charakterystycznego dla niego mikrozarządzania operacjami zwalniania zasobów jest nie do zaakceptowania w języku C++, w którym owe czynności są automatyzowane za pośrednictwem RAII.

Gdy mamy do czynienia z zasobem wymagającym parowania wywołań funkcji pozyskujących i zwalniających, powinniśmy ów zasób hermetyzować w obiekcie, składając zadanie zwalniania zasobu na barki jego destruktora. Na przykład w miejsce wywołań pary funkcji `OpenPort()` i `ClosePort()` wypadałoby rozważyć takie rozwiązanie:

```
class Port {
  public:
    Port(const string& destination);    // wywołanie OpenPort()…

    ~Port();                            // wywołanie ClosePort()…
    // porty nie mogą być zwykle duplikowane, trzeba więc zablokować
    // przypisania i kopiowania obiektów
};
```

```
void DoSomething() {
    Port port1("server1:80");
   //…
}  // nie sposób zapomnieć o zamknięciu portu port1 – jest on zamykany automatycznie
   // na skutek zakończenia zasięgu hermetyzującego go obiektu
shared_ptr<Port> port2 = /* … */;   // port2 jest zamykany automatycznie, kiedy
                                    // zwolniony zostanie ostatni odwołujący się
                                    // do niego wskaźnik shared_ptr…
```

Można również korzystać z gotowych bibliotek implementujących ten schemat (zobacz [Alexandrescu00c]).

Implementując idiom RAII, musimy uważać na konstruktory kopiujące i operatory przypisania (zobacz też wytyczna 49.); generowane przez kompilator wersje tych metod nie będą raczej poprawne. Jeśli kopiowanie obiektów hermetyzujących zasoby nie ma sensu semantycznego, powinniśmy jawnie zablokować możliwość korzystania z tych operacji, czyniąc je składowymi prywatnymi i niezdefiniowanymi (patrz wytyczna 53.). W pozostałych przypadkach należy zadbać o to, aby konstruktor kopiujący wykonywał duplikat zasobu (ewentualnie zwiększał licznik odwołań do niego), a operator przypisania robił to samo, po uprzednim ewentualnym zwolnieniu zasobu przetrzymywanego dotychczas. Klasycznym przeoczeniem jest zwolnienie poprzednio przetrzymywanego zasobu przed skutecznym wykonaniem duplikatu nowego (wytyczna 71.).

Upewnij się też, że wszystkie zasoby należą do odpowiednich obiektów. Zasoby przydzielane dynamicznie najlepiej przechowywać za pośrednictwem wskaźników „inteligentnych", a nie zwykłych. Warto też każdy jawny przydział zasobu (np. wywołanie operatora `new`) wyodrębniać do osobnej instrukcji, w której nowo przydzielony zasób natychmiast wędruje pod opiekę swojego dysponenta (np. wskaźnika `shared_ptr`); inaczej łatwo o wycieki zasobów spowodowane nieoczekiwanym porządkiem ewaluacji parametrów funkcji (porządek ten jest bowiem niezdefiniowany — patrz wytyczna 31.). Oto przykład:

```
void Fun(shared_ptr<Widget> sp1, shared_ptr<Widget> sp2);
//…

Fun(shared_ptr<Widget>(new Widget), shared_ptr<Widget>(new Widget));
```

Powyższy kod nie jest bezpieczny. Standard języka C++ daje twórcom kompilatorów znaczną swobodę w zakresie porządkowania wyrażeń reprezentujących argumenty wywołania funkcji. W szczególności kompilator może przeplatać obliczanie obu wyrażeń i najpierw wykonać przydział pamięci dla obu obiektów (operatorem `new`), a potem dopiero wywołać (w dowolnej kolejności) konstruktory obiektów klasy `Widget`. W takim układzie bardzo łatwo o wyciek pamięci, ponieważ jeśli jeden z konstruktorów zgłosi wyjątek, to pamięć *drugiego* z obiektów nie zostanie nigdy zwolniona (po szczegóły odsyłamy do [Sutter02])!

Ten subtelny problem ma proste rozwiązanie: wystarczy pilnować, aby w pojedynczej instrukcji nie przydzielać więcej niż jednego zasobu, a każdy przydział realizować jawnie, z natychmiastowym przekazaniem zasobu do obiektu-dysponenta (np. wskaźnika `shared_ptr`). Jak tutaj:

```
shared_ptr sp1(new Widget), sp2(new Widget);
Fun(sp1, sp2);
```

Inne zalety stosowanie takiej konwencji przedstawione zostaną w wytycznej nr 31.

Wyjątki

Łatwo o nadużycie inteligentnych wskaźników. Zwykłe wskaźniki świetnie sprawdzają się w kodzie, w którym wskazywane zasoby są widoczne jedynie w ograniczonym fragmencie kodu (np. wyłącznie wewnątrz klasy, jak w przypadku wewnętrznych wskaźników nawigacji wśród węzłów w klasie `Tree`).

Źródła

[Alexandrescu00c] ◆ [Cline99] §31.03–05 ◆ [Dewhurst03] §24, §67 ◆ [Meyers96] §9–10 ◆ [Milewski01] ◆ [Stroustrup00] § 14.3–4, §25.7, §E.3, §E.6 ◆ [Sutter00] §16 ◆ [Sutter02] §20–21 ◆ [Vandervoorde03] §20.1.4