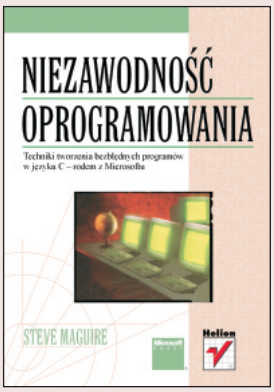
Niezawodność oprogramowania



Autor: Steve Maguire
Tłumaczenie: Andrzej Grażyński
ISBN: 83-7197-429-9
Tytuł oryginału: Writing solid code: Microsoft's techniques for developing bug-free C programs
Format: B5, stron: około 400

To właśnie programista może w znacznym stopniu przyczynić się do tego, iż wykrywanie błędów i walka z nimi staną się zadaniami łatwiejszymi i bardziej skutecznymi – tę właśnie tezę Autor stara się udowodnić w niniejszej książce, ilustrując swe wywody konkretnymi przykładami.

Niektóre ze wskazówek i zaleceń zawartych w treści niniejszej książki sprzeciwiają się wielu powszechnie przyjętym praktykom programowania i jako takie prowokować mogą do stwierdzeń w rodzaju „nikt tak nie pisze” lub „wszyscy łamią tę regułę”. Warto wówczas zastanowić się nad przyczyną – jeżeli „nikt tak nie pisze”, to dlaczego? Czy przypadkiem stare nawyki nie okazują się silniejsze od racjonalności?

Odpowiedź na te i inne pytania Czytelnik znajdzie w tej książce.

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

SPIS TREŚCI

4.

JAK WYKONUJE SIĘ TWÓJ KOD

Omawiane w poprzednich rozdziałach metody „automatycznego" wykrywania błędów — asercje, testy integralności podsystemów, itp. — stanowią narzędzia niezwykle użyteczne i znaczenie ich naprawdę trudno przecenić, jednakże w niektórych przypadkach okazują się one zupełnie „nieczułe" na błędy występujące w testowanym kodzie. Przyczyna tego stanu rzeczy jest tyleż oczywista, co banalna; wyjaśnijmy ją na (bliskim każdemu z nas) przykładzie zabezpieczenia domu czy mieszkania.

Otóż najbardziej nawet wymyślne zabezpieczenie drzwi i okien okaże się zupełnie nieprzydatne w sytuacji, gdy złodziej dostanie się do domu np. przez klapę w dachu, czy też otworzy sobie drzwi dorobionym kluczem. Podobnie, najwrażliwszy nawet czujnik wstrząsowy zamontowany skrycie w magnetowidzie czy komputerze nie uchroni przez kradzieżą np. drogocennej kolekcji obrazów. W obydwu tych przypadkach zagrożenie pojawia się bowiem poza obszarami, na monitorowanie których zorientowane są urządzenia alarmowe.

Na identycznej zasadzie, najbardziej nawet wymyślne asercje, czy jeszcze bardziej zaawansowane fragmenty kodu testujące występowanie spodziewanych warunków, są coś warte jedynie wtedy, gdy w ogóle zostają wykonane! Brak alarmu ze strony określonej asercji niekoniecznie świadczy o spełnieniu testowanego przez tę asercję warunku, ale może być także wynikiem jej pominięcia; podobnie punkt przerwania spowoduje zatrzymanie wykonywania programu jedynie wtedy, gdy wykonana zostanie instrukcja, na której punkt ten ustawiono.

Wyjaśnia to poniekąd, dlaczego niektóre błędy potrafią skutecznie wymykać się (niczym sprytne szczury) najgęstszej nawet sieci asercji czy punktów przerwań, które tym samym stanowią tylko dodatkowy kłopot dla programisty, a także powodują dodatkową komplikację i tak przeważnie już złożonego kodu.

Uciekając się do małej metafory — skoro nie potrafimy schwytać grubego zwierza w pułapkę, warto podążyć jego śladem; skoro sterowanie w naszym programie omija ustanowione punkty przerwań i asercje, spróbujmy prześledzić jego przebieg. Praca krokowa na poziomie zarówno kodu źródłowego, jak i instrukcji maszynowych jest jedną z podstawowych funkcji każdego debuggera, jest też wbudowana w znakomitą większość współczesnych środowisk projektowych.

UWIARYGODNIJ SWÓJ KOD

Opracowywałem kiedyś podprogram wykonujący specyficzną funkcję na potrzeby większego projektu (środowiska programistycznego na Macintoshu). Podczas jego rutynowego testowania znalazłem pewien błąd; jego konsekwencje dla innego fragmentu wspomnianego projektu były tak poważne, iż pozostawało dla mnie zagadką, dlaczego nie został on dotąd wykryty, skoro powinien zamanifestować się w sposób oczywisty.

Spotkałem się więc z autorem wspomnianego fragmentu i pokazałem mu błędny fragment swojego kodu. Gdy także wyraził swe zdziwienie z powodu niewykrycia widocznego jak na dłoni błędu, postanowiliśmy ustawić punkt przerwania w krytycznym miejscu kodu, a po zatrzymaniu — które naszym zdaniem musiało nastąpić — kontynuować wykonywanie w sposób krokowy.

Załadowaliśmy nasz projekt, kliknęliśmy przycisk „Run” i... ku naszemu zdumieniu program wykonał się w całości, bez zatrzymania! Wyjaśniało to skądinąd, dlaczego błąd nie został zauważony, lecz samo w sobie nadal pozostawało rzeczą zagadkową.

Ostatecznie przyczyna całego zamieszania okazała się być prozaiczna: po prostu optymalizujący kompilator wyeliminował z kodu źródłowego instrukcje, które uznał za zbędne; instrukcja, na której ustawiliśmy punkt przerwania miała nieszczęście należeć do tego zestawu. „Wykonanie” kodu źródłowego krok po kroku (czy raczej — próba takiego wykonania) uwidoczniłoby ten fakt w sposób nie budzący wątpliwości.

Jako kierownik projektu, nalegam na programistów, by „krokowe” wykonywanie tworzonego przez nich kodu stanowiło integralny element jego testowania — i, niestety, nazbyt często spotykam się ze stwierdzeniem, że przecież jest to czynność czasochłonna i jako taka spowoduje wydłużenie pracy nad projektem.

To jednak tylko mała część prawdy: po pierwsze — dodatkowy czas przeznaczony na krokowe testowanie kodu jest tylko drobnym ułamkiem czasu przeznaczonego na stworzenie tegoż kodu; po drugie — uruchomienie programu w trybie pracy krokowej nie jest w niczym trudniejsze od „normalnego” uruchomienia, bowiem różnica tkwi zazwyczaj jedynie w... naciśniętych klawiszach; po trzecie (i najważniejsze) — czas spędzony nad testowaniem programu stanowi swego rodzaju inwestycję — w przeciwieństwie do czasu spędzonego na walkę z trudnymi do wykrycia błędami, stanowiącego przykrą konieczność. W jednym z poprzednich rozdziałów, pisząc o testowaniu metodą „czarnej skrzynki”, wyjaśniałem niebagatelną rolę programowania defensywnego w walce z błędami — możliwość obserwacji zachowania się własnego kodu dodatkowo zwiększa przewagę programisty nad testerem obserwującym jedynie przetwarzanie danych przez „czarną skrzynkę”. Śledzenie stworzonego (lub zmienionego) przez programistę kodu powinno za-

tem stać się nieodłącznym elementem jego pracy i — choć może początkowo uciążliwe — z czasem będzie po prostu pożytecznym nawykiem.

Nie odkładaj testowania krokowego do czasu, gdy pojawią się błędy.

PRZETESTUJ WSZYSTKIE ROZGAŁĘZIENIA

Praca krokowa, jak wszelkie inne narzędzia, może wykazywać zróżnicowaną skuteczność w zależności od tego, jak umiejętnie jest stosowana. W szczególności — testowanie kodu zwiększa prawdopodobieństwo uniknięcia błędów tylko wtedy, jeżeli przetestuje się cały kod; niestety, w przypadku pracy krokowej sterowanie podąża ścieżką wyznaczoną przez zachodzące aktualnie warunki — mowa tu oczywiście o instrukcjach warunkowych, instrukcjach wyboru i wszelkiego rodzaju pętlach. Aby więc przetestować wszystkie możliwe rozgałęzienia, należy przeprowadzić testowanie przy np. różnych wartościach warunków instrukcji `if`, czy selektorów instrukcji `switch`.

Notabene pierwszymi ofiarami niedostatecznego testowania padają te fragmenty kodu, które wykonywane są bardzo rzadko lub wcale — do tej ostatniej kategorii należą m.in. wszelkiego rodzaju procedury obsługujące błędy. Przyjrzyjmy się poniższemu fragmentowi:

```
pbBlock = (byte *)malloc(32);
if (pbBlock == NULL)
{
     obsługa błędu
     .
     .
     .
}
```

Każda zmiana jest niebezpieczna

Programiści często pytają, jaki jest sens testowania każdej zmiany kodu spowodowanej wzbogaceniem programu w nowe możliwości. Na tak postawione pytanie można odpowiedzieć jedynie innym pytaniem — czy wprowadzone zmiany na pewno, bez żadnych wątpliwości, wolne są od jakichkolwiek błędów? To prawda, iż prześledzenie każdego nowego (lub zmodyfikowanego) fragmentu kodu wymaga trochę czasu, lecz jednocześnie fakt ten staje się nieoczekiwanie przyczyną interesującego sprzężenia zwrotnego — mianowicie programiści przywykli do konsekwentnego śledzenia własnego kodu wykazują tendencję do pisania krótkich i przemyślanych funkcji, bowiem doskonale wiedzą, jak kłopotliwe jest śledzenie funkcji rozwlekłych, pisanych bez zastanowienia.

Nie należy także zapominać o tym, by przy wprowadzaniu zmian do kodu już przetestowanego zmiany te należycie wyróżniać. Wyróżniamy w ten sposób te fragmenty, które istotnie wymagają testowania; w przeciwnym razie każda zmiana kodu może pozbawić istniejący kod wiarygodności uzyskanej drogą czasochłonnego testowania — niczym odrobina żółci zdolnej zepsuć beczkę miodu.

W prawidłowo działającym programie wywołanie funkcji `malloc` powoduje przydzielenie tu 32-bajtowego bloku pamięci i zwrócenie niezerowego wskaźnika, zatem blok uwarunkowany instrukcją `if` nie zostaje wykonany. Aby go naprawdę przetestować, należy zasymulować błędną sytuację, czyli zastąpić wartością `NULL` dopiero co przypisany wskaźnik:

```
pbBlock = (byte *)malloc(32);
pbBlock = NULL;
if (pbBlock == NULL)
{
   .
    obsługa błędu
}
```

Spowoduje to co prawda wyciek pamięci wywołany utratą wskazania na przydzielony blok, jednakże na etapie testowania zazwyczaj można sobie na to pozwolić; w ostateczności można wykonać wyzerowanie wskaźnika zamiast wywoływania funkcji `malloc`:

```
/* pbBlock = (byte *)malloc(32); */
pbBlock = NULL;
if (pbBlock == NULL)
{
   .
    obsługa błędu
}
```

Na podobnej zasadzie należy przetestować każdą ze ścieżek wyznaczonych przez instrukcje `if` z frazą `else`, instrukcje `switch`, jak również operatory `&&`, `||` i `?:`.

Pamiętaj o przetestowaniu każdego rozgałęzienia w programie.

ŻYWOTNE ZNACZENIE PRZEPŁYWU DANYCH

Pierwotna wersja stworzonej przeze mnie funkcji `memset`, prezentowanej w rozdziale 2., wyglądała następująco:

```
void *memset(void *pv, byte b, size_t size)
{
  byte *pb = (byte *)pv;
  if (size >= sizeThreshold)
  {
     unsigned long l;
     l = (b << 24) | (b << 16) | (b << 8) | b;
```

```
      pb = (byte *)longfill((long *)pb, l ,size / 4);
      size = size % 4;
   }
   while (size-- > 0)
      *pb++ = b;
   return (pv);
}
```

Sprawdziłem jej działanie w tworzonej aplikacji wyzerowując fragmenty pamięci o różnej wielkości, zarówno większej, jak i mniejszej od założonego progu `sizeThreshold`. Wszystko przebiegało zgodnie z oczekiwaniami; wiedząc jednak o tym, iż zero jest wartością w pewnym sensie wyjątkową, dla nadania testowi większej wiarygodności użyłem w charakterze „wypełniacza" innego wzorca — arbitralnie wybranej wartości `0×4E`. Dla bloków mniejszych niż `sizeThreshold` wszystko było nadal w należytym porządku, jednak dla większych bloków wartość nadana zmiennej l w linii

```
l = (b << 24) | (b << 16) | (b << 8) | b;
```

równa była `0×00004E4E` zamiast spodziewanej `0×4E4E4E4E`.

Rzut oka na asemblerową postać wygenerowanego kodu natychmiast ujawnił rzeczywistą przyczynę takiego stanu rzeczy — otóż kompilator, którego używałem, prowadził obliczenia wyrażeń całkowitoliczbowych w arytmetyce 16-bitowej, uwzględniając jedynie 16 najmniej znaczących bitów wyrażeń `(b << 16)` i `(b << 24)`, czyli po prostu wartość zero. W zmiennej `l` zapisywała się jedynie bitowa alternatywa wyrażeń `b` i `(b << 8)`.

A co z czujnością kompilatora ?

No właśnie. Kod prezentowany w niniejszej książce przetestowałem osobiście używając pięciu różnych kompilatorów; żaden z nich, mimo ustawienia diagnostyki na najwyższym możliwym poziomie, nie ostrzegł mnie, iż wspomniane instrukcje przesuwające 16-bitową wartość o 16, czy 24 bity powodują utratę wszystkich znaczących bitów. Co prawda kompilowany kod zgodny był w zupełności ze standardem ANSI C, jednakże wynik wspomnianych konstrukcji niemal zawsze odbiega od oczekiwań programisty — dlaczego więc brak jakichkolwiek ostrzeżeń?

Prezentowany przypadek wykazuje jednoznacznie konieczność nacisku na producentów kompilatorów, by tego rodzaju opcje pojawiały się w przyszłych wersjach ich produktów. Zbyt często my, jako użytkownicy, nie doceniamy siły swej argumentacji w tym względzie...

Ten subtelny błąd zostałby niewątpliwie szybko wykryty przez testerów, chociażby ze względu na widoczne konsekwencje (czyli wypełnianie dużych bloków „deseniem" `004E4E` zamiast `4E4E4E4E`), jednakże poświęcenie zaledwie kilku minut na prześledzenie kodu pozwoliło wykryć ów błąd już na etapie tworzenia funkcji.

Jak pokazuje powyższy przykład, krokowe wykonywanie kodu źródłowego może nie tylko wskazać przepływ sterowania, lecz także uwidocznić inny, niesamowicie ważny czynnik, mianowicie zmianę wartości poszczególnych zmiennych

programu w rezultacie wykonywania poszczególnych instrukcji — co nazywane bywa skrótowo *przepływem danych*. Możliwość spojrzenia na stworzony kod pod kątem przepływu danych stanowi dodatkowy oręż dla programisty — zastanów się, które z poniższych błędów mogą zostać dzięki temu wykryte i nie są możliwe do wykrycia w inny sposób:

- nadmiar lub niedomiar;
- błąd konwersji danych;
- błąd „pomyłki o jedynkę” (patrz rozdział 1.);
- adresowanie za pomocą zerowych wskaźników;
- odwołanie do nieprzydzielonych lub zwolnionych obszarów pamięci („błąd A3”, patrz rozdział 3.);
- pomyłkowe użycie operatora „=” zamiast „==”;
- błąd pierwszeństwa operatorów;
- błędy logiczne.

Przywołajmy raz jeszcze błędną instrukcję z rozdziału 1.:

```
if (ch = '\t')
  ExpandTab();
```

pomyłkowe użycie operatora = może być tu łatwo przeoczone, jednak zaobserwowana zmiana wartości zmiennej `ch` wskutek wykonania instrukcji błąd ten natychmiast demaskuje.

Podczas pracy krokowej programu
zwracaj szczególną uwagę na przepływ danych.

CZY CZEGOŚ NIE PRZEOCZYŁEŚ

Obserwując przepływ danych podczas pracy krokowej, nie jesteś jednak w stanie wykryć wszystkich błędów. Przyjrzyjmy się poniższemu fragmentowi:

```
/* jeżeli istnieje węzeł reprezentujący symbol
 * i utworzony został łańcuch zawierający jego nazwę,
 * zwolnij ten łańcuch
 */

if (psym != NULL & psym->strName != NULL)
{
  FreeMemory(psym->strName);
  psym->strName = NULL;
}
```

Odwołanie się do pola `psym->strName` ma sens jedynie wtedy, gdy wskaźnik `psym` jest niezerowy. W instrukcji `if` powinniśmy więc użyć operatora `&&` powodującego częściowe wartościowanie koniunkcji (gdy pierwszy jej argument ma wartość `false`, drugi nie jest już wartościowany), tymczasem użyty operator `&` powoduje wartościowanie kompletne, co przy zerowej wartości wskaźnika `psym` może spowodować (a w trybie chronionym — spowoduje na pewno) błąd adresowania. Pomyłkowe użycie operatora `&` nie może jednak zostać wykryte w warunkach krokowego śledzenia kodu źródłowego, postrzegającego ów kod w rozbiciu na kompletne instrukcje lub linie, nie zaś na poszczególne wyrażenia. Podobnie rzecz się ma z operatorami `||` oraz `?:`.

A co z optymalizacją przekładu?

Wzajemne „dopasowanie" instrukcji kodu źródłowego i rozkazów asemblerowych może być znacznie utrudnione, gdy kompilator dokonuje optymalizacji przekładu. Główne przejawy optymalizacji to eliminacja zbędnego kodu oraz łączne tłumaczenie kilku sąsiadujących instrukcji kodu źródłowego — czyli zjawiska jak najmniej pożądane w procesie śledzenia kodu. Znakomitą większość kompilatorów można zmusić do poniechania optymalizacji na etapie testowania programu poprzez odpowiednie ustawienie opcji kompilacji, wielu programistów dostrzega w tym jednak przejaw nadmiernego różnicowania wersji testowej i handlowej produktu. Skoro jednak podstawowym zadaniem testowania jest wyłapanie błędów, warto zastosować każdy zabieg, który może się przyczynić do pomyślnego wykonania tego zadania.

W każdym razie warto zawsze przekonać się, czy dla konkretnego programu optymalizacja istotnie utrudnia śledzenie jego kodu — być może w ogóle nie trzeba będzie jej wyłączać.

Poza wyjątkowo dokładnym sprawdzaniem składni wspomnianych instrukcji lub wyświetlaniem wartości poszczególnych wyrażeń (podczas pracy krokowej) jedynym sposobem wykrycia opisanego błędu jest prześledzenie wykonania programu na poziomie instrukcji asemblera. Wymaga to niewątpliwie kwalifikacji potrzebnych do skojarzenia rozkazów maszynowych z odpowiadającymi im instrukcjami kodu źródłowego, jednak takie błędy jak adresowanie z użyciem wyzerowanego rejestru segmentowego stają się natychmiast widoczne.

Śledzenie kodu źródłowego musi być niekiedy uzupełnione śledzeniem wygenerowanego przekładu.

SPRÓBUJ, A POLUBISZ

Niektórych programistów naprawdę trudno skłonić do systematycznego śledzenia własnych programów lub przynajmniej do spróbowania tego przez np. miesiąc. Wymawiają się brakiem czasu, bądź też niechęcią do jego tracenia. Uważam, iż

obowiązkiem każdego kierownika projektu jest przekonanie takich „opornych" programistów, iż taka oszczędność jest oszczędnością zdecydowanie źle pojętą, gdyż oznacza ryzyko tracenia (w przyszłości) znacznie większej ilości czasu na tropienie błędów w gotowym kodzie, przede wszystkim właśnie za pomocą śledzenia krokowego. Myślę, że treść niniejszego rozdziału (jak i pozostałych rozdziałów niniejszej książki) dostarcza niezbędnych ku temu argumentów.

Zresztą — jeśli po przełamaniu początkowej niechęci podejmie się próbę śledzenia stworzonego właśnie kodu i skonstatuje, że wiele tkwiących w nim błędów można z łatwością wyłapać i usunąć już w ciągu dziesięciu minut, dalsze argumenty okazują się zbyteczne. W taki oto sposób początkowa nieufność ustępuje miejsca pożytecznym nawykom...

PODSUMOWANIE

- Błędy nie pojawiają się znikąd, lecz są przyrodzonym elementem każdego nowo tworzonego kodu, bądź efektem wprowadzanych do tego kodu modyfikacji. Wyjaśnia to, dlaczego śledzenie wykonania takich nowych „kawałków" stanowi najskuteczniejszą broń w walce z błędami.
- Obawy, iż śledzenie kodu wymaga jakichś ogromnych nakładów czasu, są całkowicie nieuzasadnione. Śledzenie kodu trwa na pewno znacznie krócej od jego tworzenia, poza tym z zasady pozwala uniknąć niepotrzebnej straty czasu związanej z późniejszym wyszukiwaniem błędów w gotowym programie.
- Śledzenie kodu jest operacją w pełni skuteczną jedynie wtedy, gdy śledzeniu podlegają wszystkie rozgałęzienia w kodzie. Należy o tym pamiętać przy śledzeniu instrukcji warunkowych, pętli oraz wyrażeń zawierających operatory `&&`, `||` i `?:`.
- W niektórych przypadkach śledzenie kodu źródłowego musi być uzupełnione śledzeniem wygenerowanego przekładu w celu zorientowania się, jak w rzeczywistości przebiega wykonanie konkretnej instrukcji. Konieczność taka nie zdarza się co prawda zbyt często, lecz jeżeli faktycznie zaistnieje, nie należy jej lekceważyć.

PROJEKT: W rozdziale 1. przedstawiłem przykłady najczęściej występujących błędów programistycznych oraz pytanie o możliwość automatycznego wykrywania ich przez kompilatory. Zastanów się, w jakim stopniu krokowe śledzenie kodu może zwiększyć szansę wykrywania ich przez programistę.

PROJEKT: Sporządź listę błędów programistycznych popełnionych przez Ciebie w ciągu ostatnich sześciu miesięcy i zastanów się, ilu z nich można było zapobiec przez systematyczne śledzenie stworzonego kodu.