

Lekcja programowania

Najlepsze praktyki

TWÓRZ ZGODNIE Z TRZEMA ZASADAMI
STANOWIĄCYMI KANON DOBREGO PROGRAMOWANIA

- Prostota – czyli kod presty i łatwy w obsłudze
- Ogólność – czyli kod działający dobrze w różnych sytuacjach i adaptujący się do nowych warunków
- Przejrzystość – czyli kod łatwy do zrozumienia zarówno przez ludzi, jak i maszyny

Brian W. Kernighan
Rob Pike

» Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

Lekcja programowania. Najlepsze praktyki

Autorzy: Brian W. Kernighan, Rob Pike

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-3226-8

Tytuł oryginału: [The Practice of Programming](#)

Format: 172×245, stron: 272



Twórz zgodnie z trzema zasadami stanowiącymi kanon dobrego oprogramowania

- Prostota – czyli kod prosty i łatwy w obsłudze
- Ogólność – czyli kod działający dobrze w różnych sytuacjach i adaptujący się do nowych warunków
- Przejrzystość – czyli kod łatwy do zrozumienia zarówno przez ludzi, jak i maszyny

Czy zdarzyło Ci się kiedykolwiek... .

- pominąć oczywisty błąd w programie i spędzić cały dzień na szukaniu go?
- próbować wprowadzić sensowne zmiany w programie napisanym przez kogoś innego?
- przepisać program od nowa, bo nie dało się go zrozumieć?

Jeśli tak, w przyszłości na pewno chciałbyś tego uniknąć! Takie problemy dla zbyt wielu programistów są niestety chlebem powszednim. Dzieje się tak między innymi dlatego, że testowanie, diagnostyka, przenośność, wydajność czy styl programowania są często traktowane po macoszemu przez osoby tworzące oprogramowanie. A świat rządzony przez olbrzymie interfejsy, wciąż zmieniające się narzędzia, języki czy systemy nie sprzyja podstawowym zasadom tworzenia dobrego kodu – prostocie, ogólności i przejrzystości.

Programowanie to coś więcej niż samo pisanie kodu. W książce „Praktyka programowania” znajdziesz opis wszystkich zagadnień, z którymi styka się programista – od projektowania, poprzez usuwanie usterek, testowanie kodu czy poprawę jego wydajności, po problemy związane z poprawianiem oprogramowania napisanego przez innych. Wszystko zostało oparte na zaczerpniętych z realnych projektów przykładach, napisanych w językach C, C++, Java i innych.

Tylko tutaj znajdziesz omówienia następujących zagadnień:

- Styl: pisanie kodu, który dobrze działa i przyjemnie się czyta
- Projektowanie: wybór algorytmów i struktur danych najlepiej nadających się do określonego zadania
- Interfejsy: kontrolowanie relacji między składnikami programów
- Usuwanie błędów: szybkie i metodyczne wyszukiwanie błędów
- Testowanie: zapewnianie niezawodności i poprawności oprogramowania
- Wydajność: maksymalizowanie szybkości działania programów
- Przenośność: pisanie programów, które działają wszędzie bez żadnych zmian
- Notacja: wybór języków i narzędzi, które pozwalają maszynie zrobić więcej

Stwórz swój własny kod w najlepszym stylu!

Spis treści

Wstęp	7
1. Styl	11
1.1. Nazwy	13
1.2. Wyrażenia i instrukcje	16
1.3. Spójność i idiomy	20
1.4. Makra w roli funkcji	28
1.5. Liczby magiczne	29
1.6. Komentarze	33
1.7. Dlaczego warto dbać o styl?	38
2. Algorytmy i struktury danych	39
2.1. Przeszukiwanie	40
2.2. Sortowanie	42
2.3. Biblioteki	44
2.4. Sortowanie szybkie w Javie	47
2.5. Notacja O	50
2.6. Tablice rozszerzalne	51
2.7. Listy	54
2.8. Drzewa	59
2.9. Tablice mieszania	64
2.10. Podsumowanie	68
3. Projektowanie i implementacja	69
3.1. Algorytm łańcucha Markowa	70
3.2. Wybór struktury danych	72
3.3. Budowa struktury danych w języku C	73
3.4. Generowanie tekstu	77
3.5. Java	79
3.6. C++	83
3.7. Awk i Perl	86
3.8. Wydajność	88
3.9. Wnioski	89

4. Interfejsy	93
4.1. Wartości oddzielane przecinkami	94
4.2. Prototyp biblioteki	95
4.3. Biblioteka dla innych	99
4.4. Implementacja w języku C++	108
4.5. Zasady projektowania interfejsów	112
4.6. Zarządzanie zasobami	114
4.7. Obsługa błędów	117
4.8. Interfejsy użytkownika	121
5. Usuwanie błędów	125
5.1. Programy diagnostyczne	126
5.2. Dobre pomysły, łatwe błędy	127
5.3. Brak pomysłów, trudne błędy	131
5.4. Ostatnia deska ratunku	135
5.5. Błędy niepowtarzalne	138
5.6. Narzędzia diagnostyczne	140
5.7. Błędy popełnione przez innych	143
5.8. Podsumowanie	144
6. Testowanie	147
6.1. Testuj kod podczas jego pisania	148
6.2. Systematyczne testowanie	153
6.3. Automatyzacja testów	157
6.4. Ramy testowe	159
6.5. Testowanie przeciążeniowe	163
6.6. Porady dotyczące testowania	166
6.7. Kto zajmuje się testowaniem	167
6.8. Testowanie programu markov	168
6.9. Podsumowanie	170
7. Wydajność	171
7.1. Wąskie gardło	172
7.2. Mierzenie czasu wykonywania i profilowanie programu	177
7.3. Strategie przyspieszania	181
7.4. Regulowanie kodu	184
7.5. Oszczędzanie pamięci	188
7.6. Szacowanie	191
7.7. Podsumowanie	193
8. Przenośność	195
8.1. Język	196
8.2. Nagłówki i biblioteki	202
8.3. Organizacja programu	204
8.4. Izolacja	208
8.5. Wymiana danych	209
8.6. Kolejność bajtów	210
8.7. Przenośność a uaktualnianie	213
8.8. Internacjonalizacja	215
8.9. Podsumowanie	218

9. Notacja	221
9.1. Formatowanie danych	222
9.2. Wyrażenia regularne	228
9.3. Programowalne narzędzia	234
9.4. Interpretery, kompilatory i maszyny wirtualne	237
9.5. Programy, które piszą programy	242
9.6. Generowanie kodu za pomocą makr	246
9.7. Kompilacja w locie	247
A Epilog	253
B Zebrane zasady	255
Skorowidz	259

Usuwanie błędów

bug

b. Usterka lub błąd w maszynie, planie itp. poch. USA. 11 marca 1889 Pall Mall Gaz. 1/1: Powiadomiono mnie, że pan Edison nie śpi już od dwóch dni, próbując znaleźć usterkę (ang. bug) w swoim fonografie — wyrażenie oznaczające poszukiwanie rozwiązania problemu i sugerujące, że gdzieś wewnątrz ukrył się wymagany insekt, który powoduje trudności.

Oxford English Dictionary, wyd. 2.

W poprzednich czterech rozdziałach przedstawiliśmy sporo przykładów kodu i za każdym razem udawaliśmy, że wszystkie one od razu prawidłowo działały. Oczywiście tak nie było — w każdym z nich początkowo aż roiło się od błędów. Słowo **bug** mimo iż nie powstało w środowisku programistycznym, jest niewątpliwie jednym z najczęściej używanych słów w tej dziedzinie. Dlaczego tworzenie oprogramowania jest takie trudne?

Jednym z powodów jest to, że na złożoność programów ma wpływ liczba interakcji występujących między ich składnikami, a programy są pełne składników i relacji. Istnieje wiele technik służących do zmniejszania liczby powiązań między komponentami. Zalicza się do nich ukrywanie informacji, abstrakcję i interfejsy oraz właściwości języka, które służą do ich realizowania. Są również techniki zapewniające integralność projektów programów — dowodzenie poprawności programów, modelowanie, analiza wymagań, formalna weryfikacja — ale żadna z nich nie zmieniła sposobu, w jaki tworzy się oprogramowanie. Wszystkie okazały się skuteczne tylko w rozwiązywaniu bardzo małych problemów. Rzeczywistość jest taka, że zawsze znajdują się błędy, które będziemy wykrywać za pomocą testowania i eliminować za pomocą technik usuwania błędów (ang. *debugging*).

Dobry programista wie, że usuwanie błędów zajmuje tyle samo czasu, co pisanie kodu, i dlatego zawsze stara się wyciągać z nich wnioski. Każdy wykryty błąd jest nauką na przyszłość, jak uniknąć powtórki takiej sytuacji lub jak rozpoznać, że miała ona miejsce.

Usuwanie błędów to trudna i nieprzewidywalnie czasochłonna sztuka, dlatego należy zrobić wszystko, aby mieć z nią jak najmniej do czynienia. Sposobów na skrócenie czasu usuwania usterek jest wiele, np. staranne opracowywanie projektu, pisanie w dobrym stylu, sprawdzanie

warunków brzegowych, stosowanie asercji i testów sensowności, programowanie defensywne, projektowanie dobrych interfejsów, ograniczanie ilości danych globalnych oraz korzystanie z narzędzi diagnostycznych. Profilaktyka zawsze jest lepsza od leczenia.

Jaka jest rola języka? Największą siłą od zawsze kształtującą ewolucję języków programowania jest chęć zapobiegania występowaniu błędów poprzez odpowiednie dobranie właściwości języka. Niektóre cechy języków programowania pozwalają wyeliminować całe grupy błędów, np. sprawdzanie zakresu w operacjach indeksowania, ograniczenie lub wręcz wyłączenie możliwości stosowania wskaźników, automatyczne odzyskiwanie pamięci, łańcuchowe typy danych, kontrola typów wejścia-wyjścia i rygorystyczna kontrola typów. Z drugiej strony pewne własności języków zwiększają prawdopodobieństwo powstawania błędów: instrukcje goto, zmienne globalne, nieograniczony dostęp do wskaźników i automatyczne konwersje typów. Programiści powinni wiedzieć, które właściwości języka są potencjalnie ryzykowne, i zachować szczególną ostrożność przy ich używaniu. Ponadto powinni włączyć wszystkie narzędzia diagnostyczne kompilatora i zwracać uwagę na zgłaszane przez niego ostrzeżenia.

Właściwości językowe, które uniemożliwiają powstawanie pewnych błędów, mają swoją cenę. Jeśli język programowania wysokiego poziomu automatycznie usuwa niektóre błędy, ceną jest to, że łatwiej jest nam popełniać błędy wyższego poziomu. Żaden język nie sprawi, że całkiem przestaniemy popełniać błędy.

Chociaż wolelibyśmy, aby było inaczej, każdy programista najwięcej czasu spędza na testowaniu kodu i usuwaniu błędów. W tym rozdziale omówimy techniki produktywnego i szybkiego usuwania błędów. Do testowania wrócimy jeszcze w rozdziale 6.

5.1. Programy diagnostyczne

Kompilatory najważniejszych języków programowania są wyposażone w zaawansowane programy diagnostyczne (ang. *debugger*). Narzędzia takie wchodzi w skład wielu zintegrowanych środowisk programistycznych oferujących w jednym pakiecie narzędzia do pisania i edytowania kodu, kompilacji oraz wykonywania utworzonych programów. Programy diagnostyczne mają graficzne interfejsy, za pomocą których można wykonywać kod programu po jednej instrukcji lub funkcji albo zatrzymywać wykonywanie po wykonaniu określonych wierszy lub spełnieniu zdefiniowanych warunków. Ponadto oferują możliwość formatowania i wyświetlania bieżących wartości zmiennych.

Program diagnostyczny można uruchomić bezpośrednio, jeśli wiadomo, że wystąpił błąd. Niektóre takie programy automatycznie przejmują sterowanie, gdy wykryją, iż coś się nie powiodło w czasie wykonywania programu. Zwykle wykrycie miejsca wystąpienia błędu jest niełatwe. W tym celu należy tylko sprawdzić sekwencję funkcji, które były w tym czasie wykonywane (**stos wywołań**) oraz wyświetlić wartości zmiennych lokalnych i globalnych. Tyle informacji często wystarczy do znalezienia źródła problemu. Jeśli to zawiedzie, można skorzystać z punktów wstrzymania i funkcji wykonywania programu krok po kroku, aby znaleźć miejsce, w którym po raz pierwszy wystąpiły jakieś anomalie.

W rękach doświadczonego programisty korzystającego z dobrego środowiska program diagnostyczny może być bardzo efektywnym i wydajnym narzędziem, które pozwala zaoszczędzić mnóstwo nerwów. Skoro dostępne są tak wspaniałe narzędzia, po co ktoś miałby usuwać błędy, nie korzystając z ich pomocy? Po co usuwaniu błędów poświęcać aż cały rozdział?

Istnieje ku temu kilka dobrych powodów, zarówno obiektywnych, jak i wynikających z naszego osobistego doświadczenia. Dla niektórych języków spoza głównego nurtu nie ma żadnego programu diagnostycznego albo, jeżeli jest, jego funkcjonalność jest bardzo ograniczona.

Ponadto działanie narzędzi diagnostycznych zależy od systemu operacyjnego, a więc nie zawsze możesz mieć dostęp do swoich ulubionych programów tego rodzaju. Programy diagnostyczne słabo radzą sobie z niektórymi rodzajami programów, np. wieloprocesowymi i wielowątkowymi, systemami operacyjnymi i systemami rozproszonymi. W takich przypadkach konieczne jest użycie technik niższego poziomu. Programista jest wówczas zdany na siebie, do dyspozycji ma tylko instrukcje drukujące oraz własne doświadczenie i umiejętność analizowania kodu.

Osobiście staramy się nie nadużywać programów diagnostycznych i ograniczamy się do sprawdzenia za ich pomocą stosu wywołań oraz wartości paru zmiennych. Jednym z powodów podjęcia takiej decyzji jest to, że można bardzo łatwo pogubić się w skomplikowanej płataninie struktur danych i ścieżek wykonawczych. Naszym zdaniem wykonywanie kodu krok po kroku jest mniej produktywnie niż jego dokładniejsze przeanalizowanie oraz dodanie kilku instrukcji wyjściowych i samosprawdzającego się kodu w krytycznych miejscach. Na przejrzenie danych zwróconych przez kilka rozproszonych instrukcji drukujących potrzeba mniej czasu niż na wykonywanie kolejnych instrukcji za pomocą kliknięć myszą. Podjęcie decyzji, gdzie wstawić instrukcję drukowania, zajmuje mniej czasu niż przechodzenie do krytycznego fragmentu kodu po jednej instrukcji, nawet jeśli dokładnie wiadomo, które to miejsce. Co ważniejsze, instrukcje diagnostyczne pozostają w programie, a sesje programu diagnostycznego znikają.

Szukanie błędów po omacku za pomocą programu diagnostycznego rzadko bywa produktywnie. O wiele lepiej jest użyć go do sprawdzenia stanu programu w chwili wystąpienia usterki i na podstawie zdobytych informacji zastanowić się, jak mogło do tej sytuacji dojść. Programy diagnostyczne bywają niezwykle skomplikowane i trudne do opanowania. Zwłaszcza początkujący programista może mieć z nich sto pociech i tysiąc utrapień. Jeśli programowi diagnostycznemu zada się niewłaściwe pytanie, to zwykle zwróci on odpowiedź, ale nie wiadomo, czy poprawną.

Mimo to program diagnostyczny może być niezwykle pomocny i każdy programista powinien mieć go pod ręką. W wielu przypadkach jest to pierwsze narzędzie, z którego pomocy się korzysta. Jeśli jednak nie masz programu diagnostycznego albo napotkasz wyjątkowo trudny do rozwiązania problem, dzięki technikom opisanym w tym rozdziale i tak szybko wyjdiesz z opresji. Ponadto nauczysz się dzięki nim efektywniej korzystać z programów diagnostycznych, gdyż dotyczą tego, jak analizować błędy i szukać ich prawdopodobnych przyczyn.

5.2. Dobre pomysły, łatwe błędy

Oho! Coś jest nie tak. Mój program padł, wydrukował bzdury albo nie chce przestać działać. Co robić?

Początkujący programiści w takich sytuacjach najczęściej zrzucają winę na kompilator, bibliotekę i wszystko, tylko nie ich kod. Doświadczeni programiści też by tak chcieli, ale będąc realistami, doskonale wiedzą, że większość błędów powstaje wyłącznie z ich winy.

Na szczęście głównie robimy proste błędy, które można wyeliminować prostymi technikami. Przeanalizuj zwrócone przez program błędne dane i spróbuj wywnioskować, w jaki sposób mogły powstać. Przejrzyj dane diagnostyczne wyprodukowane przed wystąpieniem awarii. Jeśli masz taką możliwość, sprawdź stos wywołań. Po wykonaniu tych czynności będziesz już mieć jakieś pojęcie na temat tego, co i gdzie się stało. Przemyśl to. Jak mogło do tego dojść? Przeanalizuj zachowanie programu od początku i zastanów się, co mogło spowodować jego wadliwe działanie.

Diagnostyka błędów wymaga analizowania w myślach przeszłości, podobnie jak wykrywanie sprawców morderstw. Zdarzyło się coś niemożliwego, a jedyna informacja, jaką posiadamy,

to fakt, że rzeczywiście miało to miejsce. Aby odkryć przyczynę problemów, musimy się cofnąć w czasie. Po znalezieniu pełnego wyjaśnienia będziemy wiedzieć, jak naprawić program, a przy okazji prawdopodobnie odkryjemy jeszcze kilka innych rzeczy, których się nie spodziewaliśmy.

Szukaj znajomych wzorców. Odpowiedz sobie na pytanie, czy już coś takiego widziałeś. Odpowiedź typu „Gdzieś już to widziałem” zwykle stanowi pierwszy krok do zrozumienia, a niejednokrotnie oznacza nawet rozwiązanie. Często występujące błędy mają pewne cechy szczególne. Przykładowo początkujący programiści często piszą tak:

```
? int n;
? scanf("%d", n);
```

zamiast tak:

```
int n;
scanf("%d", &n);
```

co zwykle kończy się próbą odczytu danych z miejsca poza wyznaczonym obszarem pamięci przy pobieraniu wiersza danych wejściowych. Wykładowcy języka C natychmiast rozpoznają ten problem.

Niewyczerpanym źródłem prostych błędów są źle dobrane typy danych i ich konwersje w funkcjach `printf` i `scanf`:

```
? int n = 1;
? double d = PI;
? printf("%d %f\n", d, n);
```

Znakiem szczególnym tego rodzaju błędów jest czasami pojawienie się niedorzecznych wartości: wielkich liczb całkowitych albo niewiarygodnie małych lub dużych wartości zmiennoprzecinkowych. Powyższy program uruchomiony na komputerze SPARC firmy Sun zwrócił następującą astronomiczną liczbę (z konieczności podzieloną na kilka wierszy):

```
1074340347 268156158598852001534108794260233396350\
1936585971793218047714963795307788611480564140\
0796821289594743537151163524101175474084764156\
422771408323839623430144.000000
```

Kolejny pospolity błąd dotyczy wczytywania liczb typu `double` za pomocą funkcji `scanf` przy użyciu ciągu `%f` zamiast `%lf`. Niektóre kompilatory wyłapują takie błędy, ponieważ sprawdzają zgodność typów argumentów funkcji `scanf` i `printf` z łańcuchami formatu. Przy włączonych wszystkich ostrzeżeniach kompilator `gcc` w systemie GNU dla powyższego wywołania funkcji `printf` zwróci następujące informacje:

```
x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

Kolejny rodzaj błędu, który łatwo rozpoznać po znakach szczególnych, to brak inicjalizacji zmiennej lokalnej. Wynikiem tego zaniedbania jest zwykle niesłychanie duża wartość, będąca pozostałością po tym, co uprzednio znajdowało się w tym miejscu w pamięci. Niektóre kompilatory mogą przestrzegać przed takimi błędami, aczkolwiek do tego konieczne może być włą-

czenie opcji sprawdzania podczas kompilacji, a poza tym — żaden kompilator nie wychwyci wszystkiego. Także pamięć alokowana za pomocą takich funkcji, jak `malloc`, `realloc` i `new`, może być bezużyteczna, jeśli nie zostanie zainicjalizowana.

Przeanalizuj ostatnią zmianę. Jakie zmiany w programie zostały ostatnio wprowadzone? Jeśli rozwijając program, za każdym razem dodajesz do niego tylko jedną rzecz, to są wyłącznie dwie możliwości: nowy kod spowodował wystąpienie błędu albo ujawnił błąd w starym kodzie. W znalezieniu problemu pomocne jest dokładne przejrzanie ostatnich zmian. Jeśli błąd występuje w nowej wersji programu, a nie ma go w starszej, to nowy kod jest częścią problemu. Dlatego trzeba zawsze zachowywać przynajmniej poprzednią wersję programu, aby w razie kłopotów móc porównać zachowanie z najnowszą wersją. Ponadto należy prowadzić rejestr wprowadzanych zmian i naprawianych błędów, by nie musieć zdobywać tych informacji na nowo, gdy trzeba będzie naprawić kolejny błąd. Pomocne są w tym systemy kontroli kodu źródłowego i inne techniki śledzenia historii zmian.

Nie popełniaj dwukrotnie tego samego błędu. Gdy naprawisz jakiś błąd, zastanów się, czy nie mógł on wystąpić jeszcze gdzieś indziej. Taka sytuacja przydarzyła się jednemu z nas krótko przed rozpoczęciem pisania tego rozdziału. Miało to miejsce w prostym, pisanym dla kolegi prototypie przedstawiającym schemat obsługi opcjonalnych argumentów:

```
? for (i = 1; i < argc; i++) {
?     if (argv[i][0] != '-') /* Koniec opcji */
?         break;
?     switch (argv[i][1]) {
?     case 'o': /* Nazwa pliku wyjściowego */
?         outname = argv[i];
?         break;
?     case 'f':
?         from = atoi(argv[i]);
?         break;
?     case 't':
?         to = atoi(argv[i]);
?         break;
?     ...
?
```

Niedługo po wypróbowaniu programu kolega poinformował nas, że do nazwy pliku zawsze dołączany był przedrostek `-o`. Było nam wstyd, ale błąd okazał się łatwy do naprawienia. Poprawiliśmy jedną instrukcję:

```
outname = &argv[i][2];
```

Po naprawieniu tego błędu i odesłaniu programu do użytkownika niebawem przyszła kolejna wiadomość. Tym razem program niepoprawnie obsługiwał argumenty typu `-f123`: po konwersji wartość liczbową zawsze wynosiła zero. To ten sam błąd, co wcześniej. Poprawiliśmy zatem następną klauzulę `case`:

```
from = atoi(&argv[i][2]);
```

Ponieważ autor się spieszył, nie zauważył, że ten sam błąd występował jeszcze w dwóch innych miejscach, przez co zanim udało się ostatecznie oczyścić program z kilku wystąpień identycznego błędu, potrzebna była jeszcze jedna wymiana doświadczeń z naszym kolegą.

W łatwym kodzie nietrudno popełnić błąd, ponieważ widząc znany problem, przestajemy być ostrożni. Nawet jeśli kod jest tak prosty, że mógłbyś go napisać z zamkniętymi oczami, lepiej nie zamykaj oczu podczas jego pisania.

Nie odkładaj poprawiania błędów na później. Pośpiech przy wykonywaniu pracy może mieć szkodliwe skutki także w innych sytuacjach. Nigdy nie ignoruj awarii. Zawsze od razu popraw błąd, bo może się nie powtórzyć, aż będzie za późno. Słynny stał się przykład takiego niedopatrzenia w misji sondy „Pathfinder” wysłanej na Marsa. Po jej pomyślnym lądowaniu na powierzchni planety w lipcu 1997 roku komputery pokładowe resetowały się mniej więcej raz na dzień, co stanowiło wielką zagadkę dla inżynierów. Gdy znaleźli przyczynę problemów, zdali sobie sprawę, że mieli już z tym do czynienia. Takie zachowania komputerów zdarzały się już w fazie wstępnych testów, ale zostały zlekceważone, ponieważ inżynierowie pracowali wówczas nad czymś innym. Zostali więc zmuszeni do zajęcia się tym dopiero później, gdy maszyna znajdowała się miliony kilometrów od nich i znacznie trudniej było ją naprawić.

Sprawdzaj stos wywołań. Mimo iż programy diagnostyczne pozwalają badać programy podczas działania, to najczęściej są wykorzystywane do analizowania stanu programu, który przestał działać. Do najbardziej przydatnych informacji dostarczanych przez program diagnostyczny należy numer wiersza kodu źródłowego, w którym wystąpił problem. Cenną wskazówką są również nieprawdopodobne wartości argumentów (puste wskaźniki, bardzo duże wartości całkowite, podczas gdy spodziewane są małe, ujemne wartości tam, gdzie powinny być dodatnie, łańcuchy znaków nienależących do alfabetu).

Oto typowy przykład z opisu algorytmów sortowania przedstawionego w rozdziale 2. Aby posortować tablicę liczb całkowitych, należy wywołać funkcję `qsort`, przekazując jej jako argument funkcję `icmp` porównującą liczby całkowite:

```
int arr[N];
qsort(arr, N, sizeof(arr[0]), icmp);
```

Załóżmy, że pomyłkowo podano nazwę `scmp` funkcji porównującej łańcuchy:

```
? int arr[N];
? qsort(arr, N, sizeof(arr[0]), scmp);
```

Jako że kompilator w tym przypadku nie może wykryć niezgodności typów, nieuchronnie napytaliśmy sobie biedy. Program ulega awarii spowodowanej próbą dostępu do niedozwolonego miejsca w pamięci. Program diagnostyczny `dbx` zwraca następujące informacje o stosie wywołań (przeredagowane, aby zmieściły się na stronie):

```
0 strcmp(0x1a2, 0x1c2) ["strcmp.s":31]
1 scmp(p1 = 0x10001048, p2 = 0x1000105c) ["badqs.c":13]
2 qst(0x10001048, 0x10001074, 0x400b20, 0x4) ["qsort.c":147]
3 qsort(0x10001048, 0x1c2, 0x4, 0x400b20) ["qsort.c":63]
4 main() ["badqs.c":45]
5 __istart() ["crt1tinit.s":13]
```

Z tych danych wynika, że awaria nastąpiła w funkcji `strcmp`. Widać, że przekazywane do niej dwa wskaźniki są o wiele za małe, co niewątpliwie jest oznaką kłopotów. W stosie wywołań zostały podane orientacyjne numery wierszy, w których nastąpiło wywołanie każdej funkcji. Wiersz nr 13 w naszym pliku `badqs.c` zawiera takie wywołanie:

```
return strcmp(v1, v2);
```

wskazujące na źródło błędu.

Przy użyciu programu diagnostycznego można również wyświetlić wartości zmiennych lokalnych i globalnych, które także mogą naprowadzić nas na jakiś trop.

Najpierw przeczytaj, a potem poprawiaj. Jedną z najbardziej niedocenianych efektywnych technik wykrywania błędów jest uważne przeczytanie kodu i zastanowienie się nad nim bez dokonywania jakichkolwiek zmian. Pokusa, aby chwycić za klawiaturę i zacząć wprowadzać zmiany, jest bardzo duża, ale należy się jej oprzeć. Istnieje duże ryzyko, że w ten sposób nie dowiesz się, co tak naprawdę szwankuje, i zmienisz nie to, co trzeba, pogarszając jeszcze tylko sytuację. Zapisanie najważniejszej części programu na papierze pozwala spojrzeć na niego z nieco innej perspektywy, niż oglądając go na ekranie, i zachęca do refleksji. Nie stosuj jednak tej techniki rutynowo. Drukowanie kodu programu to marnotrawstwo drzew, a poza tym i tak trudno ogarnąć całą strukturę kodu, jeśli zajmuje on kilka stron. Co więcej, po wprowadzeniu pierwszej zmiany cały wydruk nadaje się do wyrzucenia.

Zrób sobie krótką przerwę. Czasami w kodzie widzisz to, co chciałbyś widzieć, a nie to, co jest w nim rzeczywiście zapisane. Jeśli na chwilę się oderwiesz, to po powrocie może zaczniesz więcej uwagi zwracać na prawdziwe znaczenie kodu.

Przyjij się pokusie poprawiania kodu natychmiast. Warto chwilę się przed tym zastanowić.

Objasnij swój kod komuś innemu. Dobrym sposobem jest objaśnienie napisanego przez siebie kodu innej osobie. Zdarza się, że w ten sposób sami odkrywamy sedno problemu. Czasami wystarczy tylko powiedzieć kilka zdań, aby stwierdzić ze wstydem: „Nieważne, już wiem, co jest nie tak. Przepraszam, że Ci przeszkadzam”. To niezwykle skuteczna metoda. W rolę słuchacza może się wcielić nawet osoba niebędąca programistą. W pewnym uniwersyteckim ośrodku komputerowym przy stanowisku pracy pomocy technicznej umieszczono pluszowego misia. Studenci chcący uzyskać pomoc najpierw musieli swój problem objaśnić misiowi i dopiero potem mogli porozmawiać z człowiekiem.

5.3. Brak pomysłów, trudne błędy

„Nie mam zielonego pojęcia, o co może chodzić”. Jeśli kompletnie nie wiesz, w czym może tkwić problem, to zaczynają się schody.

Wymuś powtarzalność błędu. Pierwszą czynnością, którą należy wykonać, jest sprawienie, aby błąd pojawiał się na żądanie. Tropienie błędu pojawiającego się tylko raz na jakiś czas nie jest przyjemne. Poświęć chwilę na sporządzenie danych wejściowych i opracowanie takich parametrów, które pozwolą Ci niezawodnie spowodować wystąpienie błędu za każdym razem. Następnie zapakuj to wszystko w jeden pakiet, aby móc go przywoływać jednym przyciskiem albo kilkoma klawiszami. Jeśli błąd jest trudny do wytropienia, czynności te trzeba będzie powtórzyć wielokrotnie, a więc lepiej je sobie maksymalnie uprościć.

Jeśli błędu nie da się odtworzyć za każdym razem, spróbuj zrozumieć dlaczego. Czy częstotliwość jego występowania zależy od jakichś specyficznych warunków? Nawet jeżeli nie możesz wymusić pojawienia się błędu za każdym razem, warto spróbować przynajmniej skrócić czas oczekiwania na jego wystąpienie.

Jeśli program może dostarczać danych diagnostycznych, skorzystaj z tej możliwości. Programy symulacyjne, takie jak program generujący łańcuchy Markowa z rozdziału 3., powinny zawierać opcję generowania danych diagnostycznych, np. w celu sprawdzenia wartości początkowej

generatora liczb losowych, dzięki którym można spróbować odtworzyć uzyskane wyniki. Wiele programów zawiera takie opcje i warto je uwzględnić także w swoich programach.

Dziel i rządź. Czy dane wejściowe wywołujące awarię programu można jakoś zmniejszyć albo bardziej skoncentrować? Stwórz minimalny zestaw danych wejściowych, które powodują występowanie błędu, aby zredukować liczbę możliwości. Jakie zmiany powodują, że błąd przestaje się pokazywać? Spróbuj wyodrębnić takie przypadki testowe, które precyzyjnie koncentrują się na szukanym błędzie. Każdy taki przypadek powinien być zaplanowany na uzyskanie określonego wyniku, potwierdzającego lub wykluczającego pewną hipotezę na temat źródła problemów.

Użyj algorytmu przeszukiwania binarnego. Odrzuć połowę danych wejściowych i sprawdź, czy program nadal zwraca niepoprawny wynik. Jeśli nie, wróć do poprzedniego stanu i odrzuć drugą połowę danych wejściowych, a pierwszą tym razem pozostaw. Tę samą metodę można zastosować w odniesieniu do tekstu programu. Usuń jakąś część kodu źródłowego, która Twoim zdaniem nie powinna mieć związku z występującym błędem, i sprawdź, co się stanie. Przy manipulowaniu dużymi przypadkami testowymi i dużymi ilościami kodu źródłowego programu bardzo pomocny jest edytor kodu z opcją cofania zmian, która zapewnia, że nie utracimy błędu.

Przeprowadź numeryczną analizę usterek. Czasami na trop błędu można wpaść, analizując pewne liczbowe cechy usterek. Po napisaniu jednego z podrozdziałów tej książki spostrzeżliśmy, że niektóre litery gdzieś się z niego ulotniły. To było bardzo dziwne. Ponieważ tekst został skopiowany i wklejony do pliku z innego miejsca, doszliśmy do wniosku, że problem tkwi w funkcji kopiowania lub wklejania edytora tekstu. Ale od czego rozpocząć poszukiwanie błędu? Postanowiliśmy dokładniej przyjrzeć się danym i odkryliśmy, że braki znaków występują w równych odstępach w tekście. Obliczyliśmy, że odległość między dwoma kolejnymi brakami zawsze wynosiła 1 023 bajty. Taka regularność jest bardzo podejrzana. Poszukaliśmy w kodzie źródłowym edytora wartości zbliżonych do 1 024 i znaleźliśmy kilka rzeczy wartych uwagi. Jedna z nich znajdowała się w świeżo napisanym kodzie, a więc postanowiliśmy zacząć od niej. Szybko spostrzeżliśmy błąd. Była to klasyczna pomyłka o jeden, która powodowała, że zerowy bajt kasował ostatni znak w buforze o rozmiarze 1 024 bajtów.

Na trop błędu wpadliśmy dzięki przeanalizowaniu liczbowych właściwości związanych z ustereką. Ile czasu nam to zajęło? Kilka minut spędziliśmy w osłupieniu, pięć minut zajęło nam odkrycie prawdziwości w znikaniu znaków i kolejnych pięciu minut potrzebowaliśmy na znalezienie i usunięcie błędu. Rozwiązanie tego problemu przy użyciu programu diagnostycznego byłoby bardzo trudne, gdyż w grę wchodziły dwa wieloprocesowe programy obsługiwane za pomocą myszy i komunikujące się ze sobą poprzez system plików.

Wyświetlaj dodatkowe informacje, aby zorientować się, jak działa program. Jeśli nie rozumiesz, co robi kod, to najłatwiejszym i najmniej kosztownym wydajnościowo sposobem na dowiedzenie się tego jest dodanie instrukcji wyświetlających różne informacje. W ten sposób można upewnić się co do słuszności swoich ocen lub zweryfikować hipotezy na temat tego, co działa źle. Jeśli np. wydaje Ci się, że niemożliwe jest dotarcie do pewnej części kodu, dodaj instrukcję wyświetlającą informację: „Nie można tu wejść”. Jeżeli później komunikat ten zostanie pokazany, przesuń wyświetlającą go instrukcję nieco wyżej, aby dowiedzieć się, w którym miejscu zaczynają się kłopoty. Analogicznie możesz też wyświetlać informację: „Udało się tu wejść” i przesuwać ją stopniowo coraz dalej, by znaleźć ostatnie miejsce, w którym nic złego się nie dzieje. Komunikaty powinny różnić się od siebie, aby za każdym razem było wiadomo, który został wyświetlony.

Komunikaty powinny być zwarte i zawsze mieć jednakowy format, aby dawały się łatwo przeanalizować programiście lub programom pomocniczym, takim jak np. narzędzie grep służące do porównywania wzorców. Programy podobne do grep są nieocenionym wsparciem przy

przeszukiwaniu tekstu — prostą implementację takiego narzędzia przedstawiamy w rozdziale 9. Jeśli wyświetlasz wartości zmiennych, to za każdym razem formatuj komunikat w taki sam sposób. W językach C i C++ wskaźniki prezentuj w postaci liczb szesnastkowych przy użyciu specyfikatorów formatu %x lub %p. Dzięki temu dowiesz się, czy dwa wskaźniki mają tę samą wartość bądź są ze sobą w jakiś sposób powiązane. Naucz się odczytywać wartości wskaźników oraz rozpoznawać prawdopodobne i nieprawdopodobne wartości, np. zero, liczby ujemne, nietypowe wartości i małe liczby. Także znajomość formatów adresów przydaje się podczas używania programu diagnostycznego.

Jeśli jest możliwość, że program zwróci bardzo dużą ilość danych, to może dane te wystarczy wydrukować w postaci pojedynczych liter, np. A, B itd., aby zwięźle pokazać, dokąd program doszedł.

Pisz samosprawdzający się kod. Jeśli potrzebujesz więcej informacji, to możesz napisać własną funkcję sprawdzającą określony warunek, wyświetlającą wartości odpowiednich zmiennych i zamykającą program:

```
/* check: sprawdza warunek, drukuje i kończy działanie */
void check(char *s)
{
    if (var1 > var2) {
        printf("%s: var1 %d var2 %d\n", s, var1, var2);
        fflush(stdout); /* Zapewnia wysłanie wszystkich danych na wyjście */
        abort();       /* Sygnalizuje nienormalne zakończenie działania programu */
    }
}
```

Funkcja check wywołuje standardową funkcję języka C o nazwie abort, która przedwcześnie kończy działanie programu w celu umożliwienia jego analizy w programie diagnostycznym. Oczywiście funkcję check można też zmienić w taki sposób, aby po wydrukowaniu informacji nie zamykała programu.

Następnie wywołaj funkcję check wszędzie tam, gdzie tego potrzebujesz:

```
check("Przed podejrzanym kodem");
/* ... Podejrzany kod ... */
check("Za podejrzanym kodem");
```

Po naprawieniu błędu nie usuwaj funkcji check z kodu źródłowego. Umieść ją w komentarzu albo wyłącz ją za pomocą opcji programu diagnostycznego, aby móc jej użyć ponownie, gdy wystąpi kolejny trudny do rozwiązania problem.

Jeśli pojawią się takie problemy, zakres obowiązków funkcji można rozszerzyć np. o weryfikację i wyświetlanie struktur danych. Można nawet zastosować bardziej ogólne podejście i napisać procedurę na bieżąco sprawdzającą spójność struktur danych i innych informacji. W programach, w których wykorzystywane są skomplikowane struktury danych, warto takie funkcje napisać, **zanim** jeszcze pojawią się problemy, i uczynić je integralną częścią programu. Wówczas w razie kłopotów można je bez przeszkód wyłączyć. Nie ograniczaj się do korzystania z nich tylko podczas usuwania błędów. Możesz ich używać we wszystkich fazach rozwoju programu, a jeśli nie pochłaniają zbyt dużo zasobów, to nawet warto je pozostawić włączone cały czas. W dużych programach, takich jak systemy komutacyjne w komunikacji, często znaczną część kodu stanowią podprogramy monitorujące przepływające informacje i sprzęt i zgłaszające wszelkie ustępki, niekiedy nawet automatycznie je naprawiając.

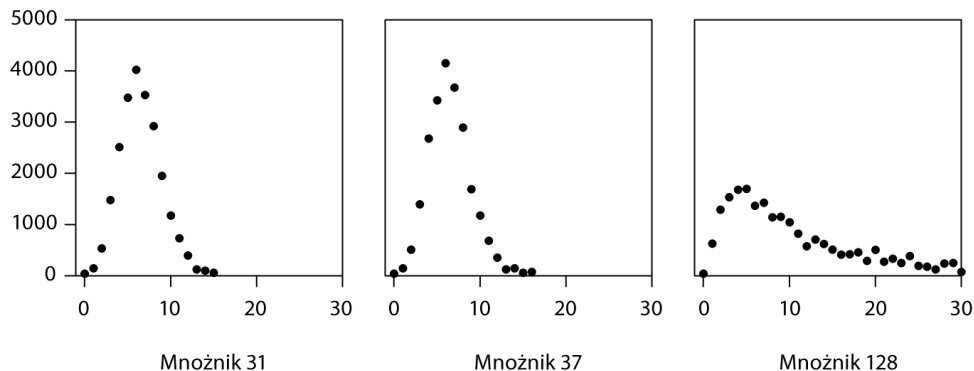
Utwórz dziennik. Kolejnym sposobem jest utworzenie **pliku dziennika**, w którym będą zapisywane dane diagnostyczne w ściśle określonym formacie. W razie wystąpienia awarii w pliku takim powinien znaleźć się zapis tego, co działo się tuż przed tym wydarzeniem. Serwery sieciowe i inne programy działające w sieci utrzymują dzienniki, w których zapisują ogromne ilości informacji o ruchu sieciowym — na ich podstawie kontrolują siebie i swoich klientów. Poniżej przedstawiamy fragment takiego pliku pochodzącego z lokalnego systemu (tekst dopasowany do strony):

```
[Sun Dec 27 16:19:24 1998]
HTTPd: access to /usr/local/httpd/cgi-bin/test.html
failed for ml.cs.bell-labs.com,
reason: client denied by server (CGI non-executable)
from http://m2.cs.bell-labs.com/cgi-bin/test.pl
```

Aby w pliku dziennika pojawiły się rekordy danych, trzeba pamiętać o zapisaniu w nim zawartości buforów wejścia i wyjścia. Funkcje wyjściowe, takie jak `printf`, zwykle buforują swoje wyniki, aby zoptymalizować działanie operacji drukowania. Przy nienormalnym zakończeniu pracy programu informacje te mogą zostać utracone. W języku C zapisanie wszystkich tego typu danych przed zamknięciem programu można wymusić za pomocą funkcji `fflush`. Jej odpowiednikiem w językach C++ i Java jest funkcja `flush` zapisująca dane ze strumieni wyjściowych. Jeżeli nie przeszkadzają Ci dodatkowe koszty wydajnościowe, problem możesz rozwiązać raz na zawsze, wyłączając buforowanie operacji zapisu danych w dzienniku. Służą do tego standardowe funkcje o nazwach `setbuf` i `setvbuf`. Wywołanie funkcji `setbuf(fp, NULL)` spowoduje wyłączenie buforowania w strumieniu `fp`. Standardowe strumienie błędów (`stderr`, `cerr` i `System.err`) mają domyślnie wyłączone buforowanie.

Rysuj obrazy. Czasami w testowaniu i usuwaniu błędów doskonałą pomocą są obrazy. Oczywiście najbardziej pomagają w zrozumieniu struktur danych, o czym przekonaliśmy się w rozdziale 2., i w pisaniu programów graficznych, ale to nie jedyne ich zastosowania. Na wykresie punktowym lepiej widać rozkład wartości niż w kolumnach liczb. Na histogramie można łatwiej wychwycić anomalie w ocenach z egzaminów, losowych liczbach, rozmiarach kubeków alokowanych przez specjalne funkcje i używanych w tablicach mieszania itd.

Jeśli nie rozumiesz, co dzieje się w Twoim programie, spróbuj sobie pomóc, opatrując struktury danych danymi statystycznymi, które dodatkowo przedstawi w postaci wykresu. Poniżej zaprezentowano wykresy sporządzone dla programu Markowa z rozdziału 3. w wersji napisanej w języku C. Na oś x zostały naniesione długości łańcuchów mieszania, a na oś y — liczby elementów w tych łańcuchach. Jako danych wejściowych użyliśmy naszego standardowego tekstu z Księgi Psalmów (42 685 słów, 22 482 przedrostki). Pierwsze dwa wykresy zostały sporządzone dla dobrych mnożników 31 i 37, a trzeci — dla koszarnej wartości 128. W dwóch pierwszych przypadkach długość żadnego łańcucha nie przekracza 15 lub 16 elementów, a większość łańcuchów składa się z 5 i 6 elementów. W trzecim przypadku dane są bardziej rozproszone, najdłuższy łańcuch ma 187 elementów i występuje bardzo dużo łańcuchów zawierających po 20 i więcej elementów.



Korzystaj z narzędzi. Dobrze użyj narzędzi oferowanych przez swoje środowisko pracy. Na przykład program porównujący pliki, taki jak `diff`, zestawia wyniki programu, którego wykonywanie zakończyło się powodzeniem, i takiego, którego wykonywanie zakończyło się niepowodzeniem, dzięki czemu można przeanalizować różnice. Jeśli program diagnostyczny zwraca duże ilości danych, to przeszukuj je za pomocą takiego programu jak `grep` oraz analizuj przy użyciu edytora. Wstrzymaj się od drukowania na papierze danych diagnostycznych: komputery lepiej radzą sobie z analizą dużych ilości danych niż ludzie. Użyj skryptów powłoki, aby zautomatyzować proces przetwarzania danych diagnostycznych.

Pisz proste programy do weryfikacji hipotez i swojego zrozumienia sposobu działania kodu. Czy można np. zwolnić pusty wskaźnik?

```
int main (void)
{
    free(NULL);
    return 0;
}
```

Programy do kontroli kodu źródłowego, takie jak RCS, umożliwiają rejestrację kolejnych wersji programu, dzięki czemu można sprawdzić, jakie zmiany zostały wprowadzone, i w razie potrzeby przywrócić jedną ze starszych wersji. Oprócz funkcji podglądu najnowszych zmian programy te oferują również możliwość znalezienia najczęściej modyfikowanych fragmentów kodu. W tych miejscach często kryją się rozmaite błędy.

Pisz dokumentację. Jeśli poszukiwania źródła problemów będą się przeciągać, po pewnym czasie zapomnisz, co już zostało sprawdzone, a czego jeszcze nie wiesz. Jeżeli zaczniesz zapisywać wykonane testy i ich wyniki, to będziesz mieć pewność, że niczego nie przeoczysz. Notując informacje o problemie, lepiej zapamiętasz, że kiedyś już coś podobnego widziałeś, a przy okazji będziesz mieć pomoc, gdy zechcesz objaśnić problem komuś innemu.

5.4. Ostatnia deska ratunku

Co robić, jeśli żadna z wymienionych technik nie pomaga? Teraz może nadeszła pora na wykonanie programu krok po kroku w programie diagnostycznym. Jeśli masz kompletnie błędne wyobrażenie o tym, jak coś działa, przez co szukasz problemu w niewłaściwym miejscu albo szukasz tam, gdzie trzeba, lecz go nie widzisz, program diagnostyczny może zmusić Cię do

spojrzenia na sprawę z innej perspektywy. Błędy niedające się wykryć z powodu niewłaściwego rozumienia istoty problemu są najgorsze. W takich przypadkach mechaniczna pomoc jest bezcenna.

Czasami błędne przekonanie dotyczy bardzo prostych zagadnień, są to np.: niepoprawna kolejność wykonywania operatorów, użycie niewłaściwego operatora, wcięcia kodu niezgodne z jego strukturą czy błędy zakresu dostępności zmiennych polegające na tym, że zmienna lokalna zasłania zmienną globalną albo zmienna globalna wciąga się w zakres lokalny. Programiści często zapominają przykładowo o tym, że operatory & i | stoją dalej w kolejce do wykonania niż operatory == i !=. Dlatego zdarza im się pisać taki kod:

```
? if (x & 1 == 0)
?     ...
```

i nie mogą zrozumieć, dlaczego ten warunek nigdy nie jest spełniony. Czasami poślizgnie się palec i omyłkowo zamiast jednego znaku równości napiszą się dwa albo odwrotnie:

```
? while ((c == getchar()) != EOF)
?     if (c = '\n')
?         break;
```

Albo podczas pracy nad programem nie zostanie usunięty niepotrzebny kod:

```
? for (i = 0; i < n; i++);
?     a[i++] = 0;
```

Niektóre problemy wynikają z pośpiechu:

```
? switch (c) {
?     case '<':
?         mode = LESS;
?         break;
?     case '>':
?         mode = GREATER;
?         break;
?     default:
?         mode = EQUAL;
?         break;
? }
```

Czasami wpisanie argumentów w niepoprawnej kolejności powoduje błąd, którego nie można wykryć przez mechanizm sprawdzania typów, np.:

```
? memset(p, n, 0); /* Zapisuje n zer w p */
```

zamiast

```
memset(p, 0, n); /* Zapisuje n zer w p */
```

Czasami coś zostaje zmienione bez wiedzy programisty, np. nie wiemy, że jakaś procedura może zmieniać pewne globalne lub współużytkowane zmienne.

Nieraz użyty algorytm lub struktura danych zawierają fatalny błąd, którego po prostu nie dostrzegamy. Przygotowując materiały do omówienia list powiązanych, sporządziliśmy pakiet funkcji służących do tworzenia nowych elementów listy oraz dołączania ich na początku i końcu struktury danych itp. (funkcje te można obejrzeć w rozdziale 2.). Oczywiście sprawdziliśmy, czy wszystko jest w porządku za pomocą specjalnie napisanego w tym celu programu testowego. Kilka pierwszych testów zostało zakończonych pomyślnie, ale w pewnym momencie nastąpiła efektowna awaria. Oto kod źródłowy tamtego programu:

```
? while (scanf("%s %d", name, &value) != EOF) {
?     p = newitem(name, value);
?     list1 = addfront(list1, p);
7     list2 = addend(list2, p);
? }
? for (p = list1; p != NULL; p = p->next)
?     printf("%s %d\n", p->name, p->value);
```

Aż trudno uwierzyć, ile kłopotów sprawiło nam dostrzeżenie, że pierwsza pętla umieszczała ten sam węzeł p w obu listach, przez co gdy przystępowaliśmy do drukowania, wskaźniki były beznadziejnie pomieszane.

Takie błędy są trudne do wykrycia, ponieważ podświadomie widzimy to, co chcielibyśmy wiedzieć. Dlatego w takich przypadkach pomocny jest program diagnostyczny, który zmusza nas do zastanowienia się nad innymi możliwościami i prześledzenia **rzeczywistego** działania programu zamiast myślenia o tym, co on powinien robić. Czasami problem wynika z błędu w ogólnej strukturze programu. Aby wykryć coś takiego, trzeba ponownie przejrzeć swoje wstępne założenia.

Zauważyliśmy przy okazji, że w przykładzie dotyczącym list błąd znajdował się w kodzie testującym, co znacznie utrudniało jego znalezienie. To straszne, jak łatwo można zmarnować czas na poszukiwaniu błędów, których nie ma, bo problem tkwi w programie testującym, albo na testowaniu niewłaściwej wersji programu tudzież ponieważ zaniedbało się aktualizację bądź kompilację programu przed wznowieniem testowania.

Jeśli mimo znacznego wysiłku nie uda Ci się znaleźć błędu, to zrób sobie przerwę. Odpocznij i chwilowo zajmij się czymś innym. Porozmawiaj z kolegą i poproś go o pomoc. Rozwiązanie może pojawić się nagle, nie wiadomo skąd, a nawet jeśli nie, po powrocie do pracy nie będziesz już tkwić w tym samym zaułku.

Zdarza się też, choć niezwykle rzadko, że źródłem problemów jest kompilator, biblioteka, system operacyjny, a nawet sprzęt. Można to podejrzewać zwłaszcza wówczas, gdy błąd wystąpił bezpośrednio po wprowadzeniu zmian w środowisku. Nigdy nie należy rozpoczynać szukania błędów od tych miejsc, ale po wykluczeniu wszystkich innych możliwości to może być ostatnie, co nam zostanie. Kiedyś przenieśliśmy duży program do formatowania tekstu z systemu Unix do komputera PC. Kompilacja zakończyła się bez żadnych problemów, ale program działał bardzo dziwnie: opuszczał mniej więcej co drugi znak w danych wejściowych. Pierwszą naszą myślą było to, że ma to jakiś związek z używaniem 16-bitowych liczb całkowitych zamiast 32-bitowych albo z kolejnością bajtów. Jednak po wydrukowaniu znaków, tak jak były przedstawiane pętli głównej, odkryliśmy, że błąd tkwił w standardowym pliku nagłówka *ctype.h* dostarczonym przez producenta kompilatora. Zawierał on implementację funkcji *isprint* w postaci makra funkcyjnego:

```
? #define isprint(c) ((c) >= 040 && (c) < 0177)
```

a główna pętla pobierania danych była zdefiniowana następująco:

```
? while (isprint(c = getchar()))
?     ...
```

Za każdym razem, gdy na wejściu pojawiała się spacja (o wartości ósemkowej 40, którą stosuje się w złym stylu zamiast zapisu ' ') lub znak o wyższym numerze, funkcja `getchar` była wywoływana po raz drugi, ponieważ makro ewaluowało swój argument dwa razy, przy czym pierwszy znak zniknął bezpowrotnie. Nasz kod źródłowy może nie być szczytem elegancji — warunek pętli mógłby być prostszy — ale plik nagłówkowy od dostawcy kompilatora bez najmniejszych wątpliwości zawierał błąd.

Przykłady tego błędu można spotkać do dziś. Poniższe makro pochodzi z wciąż używanych plików nagłówkowych innego producenta:

```
? #define __iscsym(c) (isalnum(c) || ((c) == '_'))
```

Obfitym źródłem błędów powodujących nienormalne działanie programów są wycieki pamięci, tzn. przypadki nieodzyskania nieużywanych już fragmentów pamięci. Kolejnym jest niezamykanie plików, które prowadzi do wypełnienia tablicy plików otwartych, przez co nie można otwierać następnych. Awaryjne programy zawierające wycieki pamięci często wyglądają bardzo tajemniczo. Ponieważ do usterki dochodzi po wyczerpaniu pewnych zasobów, nie da się odtworzyć specyficznych zdarzeń.

Z rzadka kłopoty sprawia sprzęt. W procesorze Pentium z 1994 roku występował błąd, który powodował, że niektóre obliczenia na liczbach zmiennoprzecinkowych dawały złe wyniki. Ta szeroko nagłośniona usterka w projekcie urządzenia dużo firmę kosztowała, ale gdy już ją zidentyfikowano, błąd dało się powtarzać. Jeden z najdziwniejszych błędów, jakie widzieliśmy w swojej karierze, znajdował się w starym programie kalkulatora działającym w systemie dwuprocessorowym. Czasami dla wyrażenia $0/5$ zwracał wartość 0.5 , a niekiedy drukował jakąś inną wartość, typu 0.7432 , choć trzeba przyznać, że jak już to robił, to konsekwentnie. Nie dało się w żaden sposób przewidzieć, czy w danym przypadku wynik będzie poprawny, czy nie. W końcu odkryto, iż źródłem problemu jest usterka w jednostce odpowiedzialnej za obliczenia zmiennoprzecinkowe w jednym z procesorów. Ponieważ do wykonywania kalkulatora był losowo wybierany albo jeden, albo drugi procesor, raz wyniki były poprawne, a innym razem niedorzeczne.

Wiele lat temu używaliśmy maszyny, której wewnętrzną temperaturę można było oszacować na podstawie liczby niepoprawnych bitów niskich w obliczeniach zmiennoprzecinkowych. Obluzowała się jedna z kart układu elektronicznego i w miarę jak rosła temperatura, karta ta odchyłała się coraz bardziej, co powodowało, że więcej bitów zostawało odciętych od płyty montażowej.

5.5. Błędy niepowtarzalne

Najtrudniejsze do wytropienia są te błędy, które pojawiają się nieregularnie — najczęściej przyczyną ich powstawania nie jest banalne uszkodzenie sprzętu. Jednak cenną wskazówką jest już sam fakt, że tak się zachowują. Można dedukować, że prawdopodobną przyczyną błędu jest nie usterka w algorytmie, lecz raczej to, iż program korzysta z danych, które za każdym razem są inne.

Sprawdź, czy wszystkie zmienne są zainicjalizowane. Możliwe, że któraś z nich otrzymuje losową wartość odpowiadającą temu, co było ostatnio zapisane w przypisywanym jej obszarze

pamięci. W językach C i C++ najczęstszymi sprawcami są zmienne lokalne funkcji i pamięć uzyskiwana za pomocą funkcji alokujących. Wszystkim zmiennym przypisz konkretne wartości. Jeśli w programie używana jest wartość początkowa generatora liczb losowych, której często nadaje się wartość na podstawie aktualnej daty, to przypisz jej jakąś stałą wartość, np. 0.

Jeżeli dodanie kodu diagnostycznego powoduje zmianę zachowania lub wręcz zniknięcie błędu, to można podejrzewać nieprawidłowość przy alokacji pamięci — jakaś instrukcja zapisuje dane poza przydzielonym obszarem i dodanie kodu diagnostycznego wprowadza modyfikację rozmieszczenia elementów w pamięci, której skutkiem jest zmiana efektu wywołanego przez błąd. Większość funkcji wyjściowych, od `printf` po funkcje okien dialogowych, alokuje pamięć samodzielnie, co dodatkowo zaciemnia obraz.

Jeśli miejsce awarii wydaje się odległe od wszystkiego, co mogłoby być zepsute, to najbardziej prawdopodobną przyczyną problemu jest błędne zmieniienie zawartości obszaru pamięci w miejscu, które jest używane dopiero później. Czasami problem dotyczy tzw. wiszącego wskaźnika, czyli omyłkowego zwrócenia przez funkcję wskaźnika na zmienną lokalną i późniejszego jego użycia. Środkiem profilaktycznym przed taką odroczoną katastrofą jest zwrócenie adresu zmiennej lokalnej:

```
? char *msg(int n, char *s)
? {
?     char buf[100];
?
?     sprintf (buf, "Błąd %d: %s\n", n, s);
?     return buf;
? }
```

Zanim wskaźnik zwrócony przez funkcję `msg` zostanie użyty, będzie już wskazywał nic nieznaczące miejsce w pamięci. Musisz przydzielić pamięć za pomocą funkcji `malloc`, użyć statycznej tablicy albo zażądać, aby wywołujący dostarczył pamięć.

Użycie dynamicznie alokowanej wartości już po jej zwolnieniu objawia się w podobny sposób. Wspominaliśmy o tym w rozdziale 2., przy okazji omawiania funkcji `freeall`. Poniższy kod zawiera błąd:

```
? for (p = listp; p != NULL; p = p->next)
?     free (p);
```

Pamięci, która została zwolniona, nie wolno używać, ponieważ jej zawartość mogła się zmienić i nie ma pewności, że instrukcja `p->next` wciąż wskazuje właściwe miejsce w pamięci.

W niektórych implementacjach funkcji `malloc` i `free` dwukrotne zwolnienie elementu powoduje uszkodzenie wewnętrznych struktur danych, ale nie wywołuje to żadnych kłopotów przez dłuższy czas, dopóki kolejne wywołanie nie wyrzuci się na tym bałaganie. Pewne funkcje alokacyjne mają opcje diagnostyczne, za pomocą których można sprawdzić spójność pola działań przed każdym wywołaniem. Włącz je, jeśli próbujesz wytopić nieregularnie zachowujący się błąd. Jeżeli w ten sposób nic nie wskórasz, możesz napisać własną funkcję alokującą, która mogłaby sprawdzać niespójność swoich własnych zachowań albo zapisywać w dzienniku wszystkie wywołania, aby można je było później przeanalizować. Napisanie funkcji alokującej pamięć, gdy nie zależy nam bardzo na szybkości działania, jest łatwe, a więc strategię tę można wykonać, jeżeli problem jest poważny. Istnieją też świetne komercyjne narzędzia służące do sprawdzania zarządzania pamięcią oraz wykrywające błędy i wycieki pamięci. Jeśli nie masz do nich dostępu, możesz wykorzystać niektóre z ich zalet, pisząc własne funkcje `malloc` i `free`.

Jeżeli jedna osoba nie ma problemów z programem, a inna ma, to znaczy, że istnieje jakaś usterka, która ujawnia się tylko w określonych warunkach. Odpowiedzialne za to mogą być jakieś pliki wczytane przez program, prawa dostępu do plików, zmienne środowiskowe, ścieżki dostępu poleceń, ustawienia domyślne lub pliki używane podczas uruchamiania programu. Trudno cokolwiek w takich sytuacjach doradzić, ponieważ aby odtworzyć środowisko, w którym program zawodzi, trzeba być tą drugą osobą.

Ćwiczenie 5.1. Napisz własne wersje funkcji `malloc` i `free`, których będzie można użyć do rozwiązywania problemów z zarządzaniem pamięcią. Jednym z rozwiązań może być sprawdzanie w każdym wywołaniu całej przestrzeni roboczej. Odmiennym podejściem jest zapisywanie danych diagnostycznych w dzienniku, aby mogły zostać przetworzone przez inny program. Bez względu na to, którą metodę wybierzesz, na początku i końcu każdego alokowanego bloku dodaj znaczniki, by ujawnić ewentualne przypadki przekroczenia zakresu z obu stron.

5.6. Narzędzia diagnostyczne

W znajdowaniu błędów pomocne są nie tylko programy diagnostyczne. Istnieje wiele innych narzędzi, które mogą nam pomóc dotrzeć do ważnych informacji w wielkich zbiorach danych, znaleźć anomalie lub tak zmienić układ danych, aby łatwiej można było zobaczyć, co się dzieje. Wiele z nich znajduje się w standardowym wyposażeniu warsztatu. Niektóre zostały napisane w celu znalezienia konkretnego błędu lub przeanalizowania specyficznego problemu.

W tym podrozdziale omówimy prosty program o nazwie `strings`, który jest szczególnie pomocny w przeglądaniu plików składających się głównie ze znaków niedrukowalnych, a więc np. plików wykonywalnych i tajemniczych formatów binarnych używanych przez niektóre edytory tekstu. We wnętrzu często kryją się różne cenne informacje, takie jak tekst dokumentu, komunikaty o błędach i niedokumentowanych opcjach, nazwy plików i katalogów, a także nazwy funkcji, które mogły być wywołane przez program.

Programu `strings` używamy również do znajdowania tekstu w innych plikach binarnych. Wiele plików graficznych zawiera znaki ASCII opisujące program, w którym zostały utworzone, a pliki skompresowane i archiwa (np. ZIP) mogą zawierać nazwy plików. Wszystkie te informacje można odkryć za pomocą programu `strings`.

W systemach uniksowych istnieje już implementacja programu `strings`, chociaż nieco inna od tej, którą przedstawimy tutaj. Rozpoznaje ona programy na wejściu i bada tylko tekst i segmenty danych, ignorując tablicę symboli. Za pomocą opcji `-a` można ją zmusić do zbadania całego pliku.

Program `strings` pobiera tekst ASCII z plików binarnych, tak że można go później wczytać lub przetworzyć przez inne programy. Jeśli znaleziony komunikat o błędzie nie ma żadnego identyfikatora, to może być trudno odgadnąć, jaki program go zgłosił, nie mówiąc już, dlaczego to zrobił. Wówczas może pomóc przeszukanie podejrzanych katalogów przy użyciu polecenia zbliżonego do zapisanego niżej:

```
% strings *.exe *.dll | grep 'Tajemniczy komunikat'
```

Funkcja `strings` wczytuje plik i drukuje wszystkie łańcuchy składające się przynajmniej z `MINLEN = 6` drukowalnych znaków.

```

/* strings: pobiera znaki drukowalne ze strumienia */
void strings(char *name, FILE *fin)
{
    int c, i;
    char buf[BUFSIZ];

    do { /* Jeden raz dla każdego łańcucha */
        for (i = 0; (c = getc(fin)) != EOF; ) {
            if (!isprint(c))
                break;
            buf[i++] = c;
            if (i >= BUFSIZ)
                break;
        }
        if (i >= MINLEN) /* Drukuje, jeśli łańcuch jest wystarczająco długi */
            printf("%s: %.*s\n", name, i, buf);
    } while (c != EOF);
}

```

Łańcuch formatu `%. *s` użyty w wywołaniu funkcji `printf` pobiera długość łańcucha z następnego argumentu (`i`), ponieważ łańcuch (`buf`) nie jest zakończony zerem.

Pętla `do-while` znajduje i drukuje każdy łańcuch, a działanie kończy, gdy napotka znak końca pliku. Dzięki temu, że na końcu funkcji znajduje się sprawdzenie końca pliku, funkcja `getc` oraz pętle łańcuchowe mogą mieć wspólny warunek zakończenia i jedno wywołanie funkcji `printf` może obsługiwać koniec łańcucha, koniec pliku oraz zbyt długie łańcuchy.

W standardowej pętli zewnętrznej ze sprawdzeniem warunku na początku lub pojedynczej pętli z funkcją `getc` i bardziej skomplikowanym kodem źródłowym konieczne by było dwukrotne wywołanie funkcji `printf`. Takie rozwiązanie zastosowaliśmy na początku, ale zrobiliśmy błąd w instrukcji wywołującej funkcję `printf`. Poprawiliśmy go w jednym miejscu, lecz zapomnieliśmy o jeszcze dwóch innych („Czy popełniłem ten sam błąd jeszcze gdzieś indziej?”). Wówczas stało się jasne, że program trzeba napisać ponownie, aby było w nim mniej powtórzeń kodu. Tak doszliśmy do pętli `do-while`.

Funkcja `main` programu `strings` wywołuje funkcję `strings` dla każdego pliku przekazanego jej jako argument:

```

/* main: znajduje znaki drukowalne w plikach */
int main(int argc, char *argv[])
{
    int i;
    FILE *fin;

    setprogname("strings");
    if (argc == 1)
        eprintf("Sposób użycia: nazwy plików");
    else {
        for (i = 1; i < argc; i++) {
            if ((fin = fopen(argv[i], "rb")) == NULL)
                weprintf("Nie można otworzyć pliku %s:", argv[i]);
            else {
                strings(argv[i], fin);
                fclose(fin);
            }
        }
    }
}

```

```

    }
    return 0;
}

```

Może się dziwisz, że funkcja `strings` nie pobiera danych ze swojego standardowego strumienia wejściowego, gdy nie zostaną podane żadne pliki. Początkowo to robiła. Aby wyjaśnić, dlaczego teraz tego nie robi, musimy opowiedzieć historię pewnego błędu.

Oczywistym testem, za pomocą którego można sprawdzić program `strings`, jest uruchomienie go na nim samym. Program działał prawidłowo w systemie Unix, ale w systemie Windows 95 polecenie

```
C:\> strings <strings.exe
```

zwróciło dokładnie pięć wierszy danych:

```

!This program cannot be run in DOS mode
'.rdata
@.data
.idata
.reloc

```

Pierwszy wiersz wygląda jak komunikat o błędzie, przez co zmarnowaliśmy trochę czasu na dowiedzenie się, że jest to łańcuch zapisany w programie, a dane wyjściowe są poprawne, przynajmniej jak na razie. Czasami zdarza się, iż sesja diagnostyczna zostaje przerwana z powodu niezrozumienia źródła pochodzenia komunikatu.

Ale danych wyjściowych powinno być więcej, więc gdzie się podziały? Wreszcie któreś no cy oświeciło mnie („Gdzieś już to widziałem!”). Jest to problem z przenośnością, o którym szerzej piszemy w rozdziale 8. Pierwsza wersja programu wczytywała dane tylko ze standardowego wejścia i używała do tego celu funkcji `getchar`. Ale w systemie Windows funkcja ta zwraca znak końca pliku, jeśli w danych tekstowych napotka konkretny bajt (`0x1A`, czyli znak `Ctrl+Z`). To powodowało przedwczesne kończenie pracy programu.

Jest to całkowicie poprawne zachowanie, ale nie tego oczekiwaliśmy, biorąc pod uwagę nasze doświadczenia z używania programu w systemie Unix. Rozwiązaniem jest otwarcie pliku w trybie binarnym przy użyciu trybu „rb”. Ale strumień `stdin` jest już otwarty i nie da się zmienić jego trybu w żaden standardowy sposób (można by było użyć funkcji takich jak `fdopen` i `setmode`, ale nie należą one do standardu języka C). W efekcie stajemy przed wyborem jednej z kilku nieprzyjemnych możliwości: zmusić użytkownika do podania nazwy pliku, dzięki czemu program będzie dobrze działał w systemie Windows, choć jest to nietypowe rozwiązanie dla systemu Unix; po cichu tworzyć niepoprawne odpowiedzi, gdy użytkownik systemu Windows usiłuje wczytać dane ze standardowego wejścia; albo zastosować kompilację warunkową, by dostosować zachowanie programu do różnych systemów, co zmniejsza jego przenośność. Zdecydowaliśmy się na pierwszą z wymienionych możliwości, ponieważ dzięki temu program wszędzie będzie działał tak samo.

Ćwiczenie 5.2. Program `strings` drukuje łańcuchy zawierające przynajmniej `MINLEN` znaków, co czasami powoduje zwrócenie większej ilości danych, niż potrzeba. Zmodyfikuj program `strings` tak, aby przyjmował opcjonalny argument służący do określania minimalnej długości łańcucha.

Ćwiczenie 5.3. Napisz funkcję `vis` kopiującą dane wejściowe na wyjście i zamieniającą bajty niedrukowalne, takie jak znak *Backspace*, znaki sterujące i znaki nienależące do zestawu ASCII na symbole w formacie `\Xhh`, przy czym `hh` oznacza szesnastkową reprezentację danego znaku. W przeciwieństwie do `strings` funkcja `vis` jest najbardziej przydatna przy analizowaniu danych zawierających niewielką liczbę znaków niedrukowalnych.

Ćwiczenie 5.4. Jaki wynik zwróci funkcja `vis`, jeśli na wejściu otrzyma łańcuch `\X0A`? Co można zrobić, aby funkcja `vis` zwracała niedwuznaczne wyniki?

Ćwiczenie 5.5. Rozszerz zakres działania funkcji, tak aby przetwarzała sekwencje plików, łamała długie wiersze w dowolnym miejscu i usuwała wszystkie niedrukowalne znaki. Jakie jeszcze inne zadania zgodne z przeznaczeniem programu mogłaby spełniać ta funkcja?

5.7. Błędy popełnione przez innych

Niewielu programistów ma przyjemność tworzyć nowy system od podstaw. Znacznie częściej używają, modyfikują, a więc i poprawiają, kod napisany przez innych programistów.

Wszystko, co napisaliśmy do tej pory na temat znajdowania i eliminowania błędów, ma zastosowanie także do błędów popełnionych przez kogoś innego. Przed przystąpieniem do pracy konieczne jest jednak zbadanie organizacji programu oraz zrozumienie sposobu myślenia i pracy poprzednika. W pewnym bardzo dużym projekcie programistycznym użyto określenia „odkrycie”, stanowiącego całkiem dobrą przenośnię. Zadanie polega na odkryciu, o co chodzi w kodzie, którego my nie napisaliśmy.

W takich przypadkach bardzo pomocne są różne narzędzia. Używając programów do przeszukiwania tekstu, takich jak `grep`, można znaleźć wszystkie wystąpienia wybranej nazwy. Generatory odsyłaczy: `g` (ang. *cross-referencer*) pozwalają zapoznać się ze strukturą programu. Wykres przedstawiający wywołania funkcji jest pomocny, jeśli nie jest zbyt duży. Wykonywanie kodu po jednej instrukcji za pomocą programu diagnostycznego pozwala odkryć kolejność zdarzeń. Zglądając do historii wersji programu, można dowiedzieć się, jak program rozwijał się w czasie. Częste zmiany oznaczają, że kod jest słabo zrozumiany albo podlega zmieniającym się wymaganiom, a więc może stanowić potencjalne źródło błędów.

Czasami musisz szukać błędów w oprogramowaniu, za które nie odpowiadasz i którego kod źródłowy nie jest dostępny. W takich przypadkach musisz zidentyfikować i scharakteryzować błąd na tyle dobrze, aby móc go precyzyjnie omówić w raporcie i przy okazji opracować jakieś dobre „obejście” pozwalające go wyeliminować.

Kiedy wyda Ci się, że znalazłeś błąd w nie swoim programie, przede wszystkim upewnij się, iż to na pewno jest błąd, aby nie marnować czasu autora i nie narazić się na utratę reputacji.

Gdy znajdziesz błąd w kompilatorze, również upewnij się, że to rzeczywiście błąd kompilatora, a nie Twojego programu. Przykładowo w językach `Ci C++` nie określono, czy operacja bitowego przesunięcia w prawo powinna wstawiać bity zerowe (przesunięcie logiczne), czy powielać bit znaku (przesunięcie arytmetyczne). Z tego powodu niektórzy początkujący programiści myślą, że konstrukcje typu

```
? i = -1;
? printf ("%d\n", i >> 1);
```


są błędne, jeśli nie zwrócą oczekiwanego wyniku. Jest to jednak kwestia przenośności, gdyż powyższy kod może różnie się zachowywać w rozmaitych systemach i nie będzie to oznaczało błędu. Sprawdź swój test w różnych systemach i upewnij się, że dobrze rozumiesz, co się dzieje. Najlepiej skontroluj też definicję języka.

Sprawdź, czy błąd nie jest znany. Czy masz najnowszą wersję programu? Czy istnieje lista poprawionych błędów? Większość programów jest wydawana w wielu różnych wersjach. Jeśli znajdziesz usterkę w wersji 4.0b1, to wcale nie musi jej być w wersji 4.0b2 albo może w jej miejsce powstać nowa. W każdym razie niewielu programistów pasjonuje się poprawianiem błędów w starszych wersjach programów.

Wreszcie postaw się w roli osoby, która otrzyma Twój raport. Na pewno chcesz dostarczyć właścicielowi programu jak najlepszy przypadek testowy. Nie będziesz zbyt pomocny, jeśli błąd uda Ci się ujawnić tylko przy dużych ilościach danych wejściowych, w wyszukanim środowisku albo przy zastosowaniu wielu plików pomocniczych. Postaraj się ograniczyć test do jak najmniejszego samodzielnego pakietu. Dołącz wszystkie mogące się przydać informacje, takie jak wersja programu, rodzaj użytego kompilatora, system operacyjny czy opis sprzętu. Dla błędnej wersji funkcji `isprint` z podrozdziału 5.4 moglibyśmy dostarczyć poniższy program testowy:

```
/* Program testowy ujawniający błąd w funkcji isprint */
int main(void)
{
    int c;
    while (isprint(c = getchar()) || c != EOF)
        printf ("%c", c);
    return 0;
}
```

Jako przypadek testowy może posłużyć dowolny wiersz tekstu zawierający drukowalne znaki, ponieważ na wyjściu pojawi się tylko połowa danych wejściowych:

```
% echo 1234567890 | isprint_test
24680
%
```

Najlepsze powiadomienia o błędach to takie, które do zademonstrowania błędu wymagają użycia jednego lub najwyżej dwóch wierszy danych wejściowych w świeżym systemie i zawierają rozwiązanie. Wysyłaj takie powiadomienia o błędach, jakie sam chciałbyś otrzymywać.

5.8. Podsumowanie

Przy odrobinie dobrych chęci usuwanie błędów może być dobrą rozrywką, jak rozwiązywanie łamigłówek. Jednak bez względu na to, czy nam się to podoba, czy nie, sztukę tę będziemy uprawiać często i regularnie. Ponieważ fajnie by było, gdyby błędy nie istniały, staramy się pisać jak najlepszy kod od samego początku. W dobrze napisanym kodzie nie tylko jest mniej błędów, lecz także łatwiej je znaleźć, jeśli już się pojawiają.

Po zauważeniu błędu w programie należy najpierw zastanowić się, co można wywnioskować z jego cech szczególnych. Skąd mógł się wziąć? Czy wygląda znajomo? Czy zmieniło się coś w programie? Czy w danych, które spowodowały jego wystąpienie, jest coś szczególnego? Czasami wystarczy kilka dobrze dobranych przypadków testowych i kilka instrukcji drukujących.

Jeśli nie ma żadnych tropów, to i tak najlepiej jest zacząć od dokładnego przemyślenia sprawy i próby zawężenia liczby podejrzanych miejsc. Jedną z możliwości jest stopniowe ograniczanie zbioru danych wejściowych, aby uzyskać niewielki zestaw powodujący awarię. Inną możliwością jest usuwanie po kolei fragmentów kodu źródłowego, które nie powinny mieć z tym nic wspólnego. Można do programu dodać kod sprawdzający, który włącza się dopiero po wykonaniu przez program określonej liczby działań. Wszystkie wymienione techniki to elementy ogólnej strategii „dziel i rządź”, która równie dobrze sprawdza się zarówno w diagnozowaniu programów, jak i w polityce i działaniach wojennych.

Korzystaj także z innych pomocy. Niezwykle przydatne bywa objaśnienie działania kodu komuś innemu (choćby pluszowemu misiowi). Posłuż się programem diagnostycznym do sprawdzenia zawartości stosu wywołań. Użyj któregoś z komercyjnych narzędzi do wykrywania wycieków pamięci, przypadków naruszenia granic tablic, podejrzanego kodu itp. Z możliwości wykonywania kodu po jednej instrukcji skorzystaj wówczas, gdy stanie się jasne, że źle rozumiesz, jak działa kod.

Poznaj siebie i rodzaje błędów, które popełniasz. Kiedy znajdziesz i usuniesz jakiś błąd, sprawdź, czy w innych miejscach programu nie ma jeszcze podobnych usterek. Zastanów się, co się stało, aby móc w przyszłości uniknąć powtórzenia tej sytuacji.

Lektura uzupełniająca

Mnóstwo cennych informacji na temat usuwania błędów można znaleźć w książkach Steve’a Maguire’a *Writing Solid Code* (Microsoft Press, 1993) i Steve’a McConella *Kod doskonały* (Helion, 2010).

Skorowidz

Kobieta: Czy jest tu moja ciotka Minnie?

*Driftwood: Cóż, możesz wejść i poszukać, jeżeli chcesz.
Jeśli jej tu nie ma, to zapewne możesz znaleźć kogoś równie dobrego.*

Bracia Marx, Noc w operze

#define, 31
#ifdef, 207
%f, 128
%lf, 128
&, 17
.length, 32
?, 12, 18
|, 17
++, 19, 22
+ =, 109
< =, 24
= =, 17
0, 31

A

abstrakcja, 112, 208
addfront, 55
aktualizacja komentarza, 35
algorytm, 8, 193
 czas działania, 50
 dane wejściowe, 44
 jasny, 86
 Markowa, 70, 72, 79, 90
 test, 168
 podstawowy, 39, 83
 porównywanie czasu działania, 50
 przeszukiwania,
 binarnego, 41
 sekwencyjnego, 40

przyspieszanie, 181
rola, 39
sortowania, 42, 47
tworzenia,
 łańcuchów elementów, 166
 tekstu, 85
usuwania nieużytków, 116
wybór, 68, 182
wymagania pamięciowe, 50
wyszukiwania binarnego, 42
zakończenie, 77
złożoność,
 oczekiwana, 50
 pesymistyczna, 50
alokacja, 116
 pamięci, 24, 166, 186, 192
analiza
 projektu, 8
 składniowa drzewa, 63
ANSI, 202
 C, 46, 53, 202, 204
 standard, 24
API, 113, 204
application programming interface, 113
argumenty makra, 29
Ariane 5, 165
arytmetyczne
 przesunięcie, 200
assembler, 188
asercja, 150

asocjacyjna
 tablica, 86
 associative array, 86
 atexit, 116
 automatyzacja, 7, 254
 testów, 157
 Awk, 9, 86, 87, 90, 158, 180, 193, 235, 237, 245

B

backwards compatibility, 215
 bajty
 kolejność, 201
 porządek, 211
 balanced tree, 61
 B-drzewo, 63
 Beta wersja, 168
 bezwzględna
 wartość, 49
 biała skrzynka, 167
 biblioteka, 202
 big-endian, 218
 binarne
 przeszukiwanie, 46
 drzewo poszukiwań, 59, 60
 bitowy
 operator, 17
 pole, 201
 błąd
 asercja, 150
 błędne przekonanie, 136
 cechy usterki, 132
 diagnostyka, 127
 dodanie instrukcji wyświetlających informacje, 132
 dziennik, 134
 informacja, 151
 innych programistów, 143
 kompilatora, 143
 komunikat, 118
 minimalny zestaw danych wejściowych, 132
 na żądanie, 131
 nieregularny, 138
 nowy, 129
 obsługa, 99, 101, 117
 po zmianie, 129
 porównywanie plików, 135
 powielenie, 129
 przepełnienia bufora, 164
 roku 2000, 188
 rozmowa z pluszowym misiem, 131
 rzeczywiste działanie programu, 137
 składni, 21
 skutki lekceważenia, 130
 sprzętu, 138
 strumień, 134
 u jednej osoby, 140

usuwanie, 125, 147
 uważne przeczytanie kodu, 131
 wykres, 134
 wykrywanie, 9
 wymuszanie powtarzalności, 131
 zasada obsługi, 119
 znajdowanie, 147
 znany, 144
 boundary condition testing, 148
 Bourne, 166
 break, 27
 Brooks, 69, 95
 bsearch, 46
 bucket, 64
 bufor
 błąd przepełnienia, 164
 danych,
 wejściowych, 187
 wyjściowych, 187
 rozmiar, 76
 bug, 125
 build, 76

C

C, 9, 10, 17, 19, 22, 24, 27, 28, 31, 32, 40, 44, 48, 53,
 54, 64, 66, 73, 81, 83, 89, 96, 103, 107, 114, 116,
 121, 134, 139, 160, 177, 182, 184, 188, 189, 191,
 196, 197, 199, 201, 202, 203, 213, 217, 231, 243
 wada, 79
 zaleta, 79
 C++, 9, 14, 17, 19, 22, 24, 27, 28, 29, 31, 32, 40, 44,
 47, 51, 53, 54, 59, 64, 66, 83, 89, 108, 110, 111,
 113, 115, 116, 134, 139, 160, 163, 177, 182, 184, 188,
 189, 191, 196, 198, 199, 201, 202, 213, 217, 243
 case, 26
 cel testowania, 167
 cerr, 134
 Chain, 80
 char, 200
 char **array, 40
 clock, 177
 Cohen, 218
 comma-separated values, 94
 Comparable, 47
 const, 31
 cost model, 191
 cross-referencer, 143
 CSV, 94, 95, 96, 108, 112
 csvgetline, 96
 ctime, 36, 153
 ctype, 28
 cyclic redundancy check, 67
 cykliczna kontrola nadmiarowa, 67
 czarna skrzynka, 167

czas

- działania algorytmu, 50
- mierzenie, 172, 177
- pracy procesora, 178
- użycia procesora, 177
- wykonywania programu, 177

czytelne formatowanie, 16

D

dane, 64, 155

- globalne, 34
- na wyjściu, 155
- najmniejszy typ, 189
- oznaczanie końca, 77
- statyczne, 54
- struktura, 8, 39, 59
- szkodliwe, 164
- typ źle dobrany, 128
- wejściowe, 187
- wybór struktur, 89
- wyjściowe, 187
- wymiana, 209, 212

Date, 178

debugging, 125

debugowanie, 126, 256

decyzje

- wielokierunkowe, 25

defensive programming, 122

defensywne programowanie, 122, 151

definicja,

- pakietów, 202
- pola, 99

dekrementacja, 19

deque, 84

design patterns, 91

deskryptywna

- nazwa, 13

destruktor, 116

diagnostyka

- błędów, 127
- instrukcji, 127
- kodu, 9

- programu, 126, 140

diff, 135

Dijkstra, 147

długość słów, 85

dobry

- interfejs, 112
- kod, 37
- technika sortowania, 63
- zestaw testów, 161

domyślny rozmiar tablicy, 73

doświadczenia, 253

double, 187

do-while, 23

drukowanie elementów listy, 56

drzewo, 59

- analiza składniowa, 63

korzeń, 59

niezrównoważone, 61

poszukiwań binarne, 59, 60

przeglądanie poprzeczne, 62

zrównoważone, 61

dublowanie elementów, 61

dwuznaczności unikanie, 16

dzieci węzeł, 60

dzielenie, 59

dziennik

- błąd, 134

E

efekty uboczne, 19

efektywność wykorzystania pamięci, 190

elastyczność, 108

element

- dostęp swobodny, 59

dublowanie, 61

grupowanie, 16

liczenie, 57

o zmiennym rozmiarze, 59

połączony, 14, 64

przesuwanie, 53

wstawianie, 59

else, 25, 26

else-if, 25, 26

Ellis, 89

endian, 218

endprintf, 117

enum, 31

EOF, 200

estrdup, 118

ewaluacja, 19, 29

exception, 120

F

fall-through, 26

fclose, 151

fflush, 134

fgets, 24, 148

filtr spamu, 176

final, 31

find, 40

Flandren, 194

float, 187

flush, 134

flushcaches, 249

fopen, 151

for, 22, 23

format

CSV, 94, 95, 96
pętla, 22

formatowanie

czytelne, 16
wyrażenia, 16

fprintf, 151

fragment niejasny, 11

fread, 151

free, 192

funkcja

generująca, 81
komentarz, 34
logiczna, 14
łańcuchowa, 114
mieszająca, 64, 67, 74, 75
nazwa, 13, 14
sortująca, 44

fwrite, 151

G

garbage collection, 116

generate, 242

generator

liczb losowych, 49
odsylaczy, 143
tekstu, 77

getchar, 23, 28

gets, 24, 164

getTime, 178

GIF, 190

globalne

dane, 34
optymalizator, 182
zmiennne, 112

głowa, 54

główny nurt języka, 197

gorący punkt, 178, 184, 186

graficzne operacje, 188

gramatyka, 238

granice tabeli, 166

grep, 143, 172, 229, 234

grupowanie elementów, 16

H

Hashtable, 79, 83

head, 54

hermetyzacja, 112

hierarchiczna struktura danych, 59

Hoare, 42, 47

I

IBM 7094, 248

idealny komentarz, 33

idiom, 22, 24

idiomatyczny kod, 36

if, 20, 21, 25

if...else, 19, 25

implementacja,

niezależne wyniki, 156
programu, 8

indeksowania operator, 85

indexOf, 40

Inferno, 187

informacja

o błędzie, 151
ukrywanie, 99, 100, 112

inicjalizacja, 114

statyczna, 115

tablica, 167

zmienna, 167

inkrementacja, 16, 19

in-order traversal, 62

insert, 62

instrukcja, 16

diagnostyczna, 127
sprawdzająca, 26

Integer, 48

interaktywnego programu test, 168

interfejs, 8, 47, 96, 99, 112, 208, 254, 256

CSV, 112

do tablic rozproszonych, 64

dobry, 112

duży, 113

łatwy w użyciu, 123

poprawny, 150

programistyczny, 113

publiczny, 80, 108

użytkownika, 9, 121

zasady tworzenia, 112

związły, 113

internacjonalizacja, 216

internationalization, 216

internetowy robak, 165

interpreter, 237

poleceń, 234

isspam, 179

isupper, 28

J

jasny algorytm, 86

Java, 9, 14, 17, 22, 24, 27, 28, 31, 40, 47, 48, 51, 54,
58, 64, 79, 83, 89, 113, 115, 116, 121, 134, 178,
188, 199, 213

Java Virtual Machine, 242

jednokierunkowa lista, 54

jednolitość, 123
 język
 mały, 222
 niskiego poziomu, 9, 188
 nurt główny, 197
 programowania, 9
 skryptowy, 236
 standard, 196
 wybór, 221
 wysokiego poziomu, 9
 JIT, 247
 just in time compilation, 247
 JVM, 242

K

klamry, 27
 klarowność, 26, 253
 klasa, 201
 globalna, 13
 kontenerowa, 79
 klucze, 64
 Knuth, 167, 178, 194
 kod
 diagnostyka, 9
 dobry, 37
 generowanie za pomocą makr, 246
 idiomatyczny, 36
 klarowny, 19, 36
 łatwość czytania, 12
 nizany, 240
 optymalizacja, 182
 pokrycie testami, 156
 przejrzysty, 18
 regulacja, 183
 samosprawdzający, 133
 spójny, 12
 sprytny, 18
 struktura, 16
 uwypuklenie, 20
 testowanie w czasie pisania, 151
 wolny od błędów, 147
 wyższej jakości, 151
 zależny od maszyny, 188
 związły, 19, 86
 źródłowy, 12
 Koenig, 245
 kolejka, 14
 dwukierunkowa, 84
 kolejność,
 bajtów, 201
 wykonywania obliczeń, 199
 kolizja, 67
 komentarz, 33, 34
 aktualizacja, 35
 cel stosowania, 37

funkcja, 34
 idealny, 33
 niejasny, 37
 kompilacja
 na czas, 247
 w locie, 247
 warunkowa, 205
 kompilator, 166, 182, 184, 195, 197, 243, 250
 błąd, 143
 optymalizacji kodu, 182
 testowanie, 155, 201
 komputer zasady korzystania, 7
 komunikat, 217
 o błędzie, 118
 konflikt nazw, 113
 konstruktor, 115, 116
 kontener, 83
 konwencje, 12
 konwersji współczynnik, 29
 końcowy warunek, 149
 kopiowanie, 59, 114
 korzeń drzewa, 59
 kosztów model, 191
 krotka, 120
 kubełek, 64

L

last-in-first-out, 59
 liczba, 29, 31
 0, 31
 całkowita, 45, 191
 double, 128
 losowa, 49
 zmiennoprzecinkowa, 191
 liczenie elementów, 57
 licznik odniesień, 116
 LIFO, 59
 liniowe przeszukiwanie, 40
 lista, 54, 83
 drukowanie elementów, 56
 jednokierunkowa, 54
 modyfikacja, 55
 pamięć wolna, 186
 tworzenie, 55
 usuwanie, 57, 58
 liść, 61
 literate programming, 245
 little languages, 222
 locality, 186
 Locanthi, 246
 logiczne przesunięcie, 200
 lokalności zasada, 186
 lookup, 62
 losowa liczba, 49

Ł

łańcuch

- algorytm tworzenia, 166
- nazwa, 13
- Markowa, 70, 79

łańcuchowa funkcja, 114

łatwy w użyciu interfejs, 123

M

mainstream, 197

makro, 28, 31, 32

- argumenty, 29
- generowanie kodu, 246
- problem, 28

malloc, 24, 129, 192

małe języki, 222

map, 79, 84, 85

markov, 70, 91, 163, 170

maszyna,

- stosowa, 240
- wirtualna, 237

Math.abs, 49

mechanizm wyjątków, 120

memcmp, 179, 183

memcpy, 53

memcpy, 113

memmove, 53, 113, 188

memset, 161, 188

metacharacters, 228

metaznaki, 228

Microsoft Visual C++ 5.0, 163

mierzenie czasu, 172, 177

mieszająca funkcja, 74

Mitchell, 89

mocy zmniejszenie, 184

model,

- kosztów, 191
- statystyczny tekstu, 70

Modula-3, 243

modularyzacja, 112

N

nadmiarowa kontrola cykliczna, 67

najmniejsze typy danych, 189

najprostsza struktura danych, 53

najwspanialsze osiągnięcie informatyki, 64

Nameval, 51

NaN, 120

nawiasy, 16

- klamrowe, 20

nazwa, 13

- deskryptywna, 13
- elementy powiązane, 14

funkcja, 14

- logiczna, 14

klasa globalna, 13

konflikt, 113

łańcuch, 13

niespójna, 14

prywatna, 113

stała, 13

struktura globalna, 13

wskaźnik, 13

- zmienna,
 - globalna, 13
 - lokalna, 13
 - pętlowa, 13

negacja, 35

new, 129

niedorzeczne wartości, 128

niejasny

- fragment, 11
- komentarz, 37

niepoprawne dane wejściowe, 122

nieprzezroczysty typ, 112

niesłuchanie duża wartość, 128

niespójna nazwa, 14

nietypowe sytuacje, 120

niezależne implementacje, 156

niezamykanie plików, 138

niezrównoważone drzewo, 61

niskopoziomowy język, 9

nizany kod, 240

not a number, 120

notacja, 221, 254

O, 50

programowanie, 9

nowy węzeł, 61

null, 32

numerycznego programu test, 155

nurt główny języka, 197

nvcmp, 46

O

obciążeniowe testy, 9

obiektu rozmiar, 32

Object, 47, 48

obliczenia

- kolejność wykonywania, 199
- zawczasu, 187

obsługa błędów, 99, 101, 117

- zasada, 119

oczekiwana złożoność algorytmu, 50

odsyłaczy generator, 143

odzyskiwanie zasobów, 116

ogonowa rekurencja, 62

ogólność, 253

- programowania, 7

on the fly compilation, 247
 O-notation, 50
 operacja
 graficzna, 188
 wejścia, 19
 wyjścia, 19
 operator, 16
 bitowy, 17
 indeksowania, 85
 logiczny, 17
 priorytet, 17
 przeciążanie, 189
 przypisania, 17
 relacji, 17, 24
 opóźnienia w dostarczaniu poczty, 172
 optymalizacja, 184, 193
 gospodarowania pamięcią, 189
 kodu kompilatora, 182
 wykorzystania zasobów, 9
 zasada, 171
 optymalizator globalny, 182
 oszczędzanie pamięci, 190
 oznaczanie końca danych, 77

P

pair, 120
 pakietów definicja, 202
 pamięć, 114
 alokowanie, 24, 166, 186, 192
 efektywność wykorzystania, 190
 optymalne gospodarowanie, 189
 oszczędzanie, 190
 podręczna, 186
 wolna, 186
 wyciek, 138
 zwalnianie, 192
 Pathfinder, 130
 Perl, 9, 86, 87, 90, 237
 pesymistyczna złożoność algorytmu, 50
 pętla, 16, 21, 22, 23, 24, 148
 eliminacja, 185
 format, 22
 pierwsze wystąpienie znaku, 40
 pisanie kodu, 151
 piśmienne programowanie, 245
 plik
 dziennika, 134
 nagłówkowy, 202
 niezamykanie, 138
 poczty opóźnienia w dostarczaniu, 172
 podręczna pamięć, 186
 podstawowe algorytmy, 39, 83
 pole
 bitowe, 201
 definicja, 99

pomiary, 193
 wykonywanie, 172
 poprawa wydajności, 175
 poprawność interfejsu, 150
 porównywanie
 czasu działania algorytmów, 50
 liczb całkowitych, 45
 portability, 195
 Portable Operating System Interface, 218
 porządek bajtów, 211
 POSIX, 204, 218
 post-order traversal, 63
 PostScript, 245
 potok, 249
 potomek, 60, 61
 powiązane elementy, 64
 pozycja znaku, 29
 PPM, 190
 praktyka programowania, 7, 12
 Prefix, 81, 82
 pre-order traversal, 63
 printf, 97, 128, 222
 priorytetu operator, 17
 problemu rozmiar, 50
 procedura przeszukująca, 40
 procesor, 195
 prof, 178
 profil, 172, 173, 178
 program
 diagnostyczny, 126, 140
 wady, 127
 zalety, 126
 do powszechnego użytku, 90
 generujący tekst, 69
 graficzny, 155
 implementacja, 8
 interaktywny, 168
 jak napisać, 12
 numeryczny, 155
 odporny na niepoprawne dane wejściowe, 122
 pisze programy, 242
 profilujący, 172, 173, 178
 prototyp, 98
 przekazanie do użytku, 166
 przenośny, 9, 195
 przyspieszanie działania, 183
 rzeczywiste działanie, 137
 spowolnienie, 189
 struktura, 143
 włamanie, 164
 wydajność, 88, 188
 złożoność obliczeniowa, 181
 programowanie
 defensywne, 122, 151
 notacja, 9
 ogólność, 7

programowanie
 piśmienne, 245
 praktyka, 7, 12
 prostota, 7
 przejrzystość, 7
 styl, 8, 38
 zasady, 12

projektu analiza, 8

prostota programowania, 7, 253

prototyp, 197

programu, 98

prywatna nazwa, 113

przechowywanie elementów o zmiennym rozmiarze, 59

przeciążanie operatorów, 189

przeglądanie
 poprzeczne, 62
 wsteczne, 63
 wzdłużne, 63

przejrzystość, 13
 programowanie, 7

przekazanie programu do użytku, 166

przenośność programów, 9, 142, 195, 214, 257

przepelnienie, 46
 bufora, 164

przesunięcie
 arytmetyczne, 200
 elementu, 53
 logiczne, 200

przeszukiwanie
 binarne, 41, 46
 liniowe, 40
 procedura, 40
 sekwencyjne, 40
 tablicy, 154
 tekstu, 143

przydzielanie pamięci, 114

przypisanie, 16
 operator, 17
 wstawianie do warunku pętli, 23

przyspieszanie
 algorytmu, 181
 działania programu, 174, 183
 struktury danych, 181

publiczny interfejs, 80, 108

pułapki językowe, 198

punkt gorący, 184, 186

putchar, 23

Q

qsort, 44, 45, 46
 queue, 14
 quicksort, 42, 49

R

rama testowa, 154, 159, 162, 193

rand, 49

RCS, 135

real, 177

realloc, 129

reduction in strength, 184

reentrant, 116

reference count, 116

referencja, 115

regression testing, 157

regresywne testowanie, 157, 180, 184

regulacja kodu, 183

regular expressions, 228

regularne wyrażenia, 228

rekurencja, 47
 ogonowa, 62

relacji operator, 17, 24

reputacji utrata, 143

rgen, 49

robak internetowy, 165

rozmiar
 bufora, 76
 obiektu, 32
 problemu, 50
 tablicy, 29, 33, 66, 104
 typów danych, 198

rozszerzalna tablica, 51

rozwój, 253

rzutowanie, 53

S

samodzielne testy, 158

samosprawdzający się kod, 133

scalanie, 59

scanf, 96, 128, 151

scmp, 45

sekwencyjne wyszukiwanie, 40

sentinel, 77

shaney, 91

sizeof, 32

składni błędy, 21

skrócenie czasu usuwania usterek, 125

skrypt testowy, 158

skryptowy język, 236

słowa, 73, 85

słownik, 84, 87
 struktura danych, 79

sort, 47, 49

sortowanie
 algorytm, 42, 47
 dobra technika, 63
 funkcja, 44
 szybkie, 42
 tablic łańcuchów, 45

spam, 172
 filtr, 176
 specyfikacja, 101
 zawartość, 101
 split, 103, 105, 109
 sposoby doboru testów, 154
 spójność, 20, 22, 114
 kodu, 12
 zewnętrzna, 114
 sprzętu błąd, 138
 stała, 19, 31
 całkowitoliczbowa, 31
 nazwa, 13
 znakowa, 31
 stanu utrzymywanie, 114
 standard, 197, 202
 ANSI C, 24, 202, 222, 225
 ANSI/ISO języka C, 196
 ISO języka C++, 196
 języka, 196
 Standard Template Library, 83
 State, 73
 statyczna inicjalizacja, 115
 statyczne dane, 54
 stderr, 134
 stdout, 113
 STL, 59, 83, 90, 120, 163
 stos, 59, 186
 wywołań, 126, 130
 strchr, 40, 179
 strcmp, 36, 45
 strcpy, 24
 strdup, 24
 StreamTokenizer, 80
 strerror, 120
 String, 40, 48
 strings, 140
 strlen, 24, 179
 strncmp, 179
 stronicowanie, 189
 strstr, 40, 173, 176, 179
 strtok, 96, 105, 113
 struktura
 danych, 8, 39, 201
 najprostsza, 53
 słownik, 79
 globalna nazwa, 13
 kodu, 16
 programu, 143
 strumienie błędów, 134
 styl, 12, 255
 funkcja, 12
 programowania, 8, 38
 swap, 49
 Swift, 218
 swobodny dostęp do elementów, 59

symbole,
 tablica, 64
 wieloznaczne, 228
 SYS, 177
 system
 zależności, 208
 operacyjny, 195
 System.err, 134
 systematyczne testowanie niewielkich przypadków, 154
 sytuacje nietypowe, 120
 szkodliwe dane wejściowe, 164
 szybko sortowanie, 42

T

tabela
 granice, 166
 tablica, 40, 53
 asocjacyjna, 86
 domyślny rozmiar, 73
 inicjalizacja, 167
 łańcuchów, 45
 mieszająca, 51, 64, 80
 funkcja tworząca, 75
 przeszukiwanie, 154
 rozmiar, 29, 33, 66, 104
 rozproszona, 64
 rozszerzalna, 51
 słowa, 73
 symboli, 64
 znaków, 108
 tail recursion, 62
 Tcl/Tk, 9
 technika usuwania błędów, 125
 tekst
 algorytm tworzenia, 85
 generowanie, 77
 model statystyczny, 70
 przeszukiwanie, 143
 test, 9, 147, 153, 193, 256
 algorytm Markowa, 168
 automatyzacja, 157
 białej skrzynki, 167
 cel, 167
 czarnej skrzynki, 167
 dobry zestaw, 161
 duża ilość danych, 163
 kompilator, 155
 obciążeniowy, 9
 pokrycia kodu, 156
 programu,
 graficznego, 155
 interaktywnego, 168
 numerycznego, 155
 przeciążeniowy, 163
 rama, 154, 159, 162

test

- regresywny, 157, 180, 184
 - samodzielny, 158
 - sposoby doboru, 154
 - stopniowy, 153
 - systematyczny, 153
 - testu, 168
 - ustawienia parametrów wejściowych, 167
 - w czasie pisania kodu, 151
 - wartości brzegowych, 9, 148
 - rozszerzenie metody, 154
 - wzorcowy, 194
 - zautomatyzowanie, 154
 - zestaw, 153, 167, 174
- TEX, 167
- Thompson, 194, 248
- threaded code, 240
- time, 177
- tuple, 120
- tworzenie listy, 55
- typ danych
 - najmniejszych, 189
 - nieprzezroczysty, 112
 - rozmiar, 198
 - źle dobrany, 128

U

- uboczne efekty, 19
- ukrywanie informacji, 99, 100, 112
- unia, 201
- Unicode, 216
- unikanie dwuznaczność, 16
- unquote, 96
- unsigned char, 66
- ustawianie wartości początkowych, 115
- usterki czas usuwania, 125
- usuwanie
 - błędów, 147
 - listy, 57, 58
 - usterek, 125
- UTF-8, 217
- utrata reputacji, 143
- utrzymywanie stanu, 114
- użytkownika interfejs, 9, 121

V

- Vector, 79, 83, 84, 108
- Visual Basic, 9, 243
- void*, 44, 53

W

- wartości
 - bezwzględne, 49
 - brzegowych testowanie, 9
 - niedorzeczne, 128
 - niesłuchanie duże, 128
 - oddzielane przecinkami, 94
 - początkowych ustawianie, 115
 - zmiennej modyfikacja, 19
- wartownik, 77
- warunek, 25
 - brzegowy, 148
 - końcowy, 149
 - pętli, 23
 - wstępny, 149
- warunkowa,
 - kompilacja, 205
 - wyrażenie, 19
- wcięcia, 16, 20, 21
- wczytywanie liczb typu double, 128
- wejście, 19
- wektor, 83
- weprińf, 61, 117
- wersja beta, 168
- węzeł
 - dzieci, 60
 - nowy, 61
- while, 23, 26
- wielokierunkowe decyzje, 25
- wielowejsiowy program, 116
- wildcards, 228
- wiszący wskaźnik, 139
- włamanie do programu, 164
- właściwości danych wejściowych, 155
- wprińf, 203
- wskaźnik
 - brakującego potomka, 60
 - nazwy, 13
 - wiszący, 139
- współczynnik konwersji, 29
- współdzielenie, 114
- wstawianie elementu, 59
- wsteczna zgodność, 215
- wstępny warunek, 149
- wybór
 - algorytmu, 68, 182
 - języka, 9, 221
 - struktur danych, 89
- wycieki pamięci, 138
- wydajność, 171, 257
 - analiza graficzna, 180
 - poprawa, 175
 - program, 88, 188
- wyjątek, 120
- wyjście, 19

wykładnicza złożoność obliczeniowa, 51
wykonywanie pomiarów, 172
wykorzystania zasobów optymalizacja, 9
wykres błędów, 134
wykrywanie błędów, 9
wymagania pamięciowe algorytmu, 50
wymiana danych, 209, 212
wyniki niezależnych implementacji, 156
wyrażenia, 16
 formatowanie, 16
 regularne, 228
 skomplikowane, 17
 warunkowe, 19
wysokiego poziomu język, 9
wyszukiwanie
 binarne, 42
 sekwencyjne, 40
wywołanie
 stos, 126, 130
wzorzec projektowy, 91

Y

Yorktown, 150, 151

Z

zależności systemowe, 208
zarządzanie zasobami, 58, 99, 100, 114
zasada
 korzystania z komputera, 7
 lokalności, 186
 obsługi błędów, 119
 optymalizacji, 171
 programowania, 12

zasoby
 odzyskiwanie, 116
 zarządzanie, 58, 99, 100, 114
zautomatyzowanie testowania, 154
zbiór, 83
 klas kontenerowych, 79
zero, 31
zestaw
 testów, 153, 167, 174
 znaków, 216
zewnętrzna spójność, 114
zgodność wsteczna, 215
złożoność obliczeniowa, 50, 56, 181
 wykładnicza, 51
zmienna, 103
 globalna, 13, 112
 inicjalizacja, 167
 lokalna, 13
 modyfikacja wartości, 19
 nazwa, 13
 pętlowa, 13
 wewnętrzna, 103
zmniejszenie mocy, 184
znak
 pierwsze wystąpienie, 40
 pozycja, 29
 tablica, 108
 zapytania, 12
 zestaw, 216
zrównoważone drzewo, 61
zwalnianie pamięci, 114, 192
zwiększenie wydajności programu, 188
związłość, 13
 kodu, 86

LEKCJA PROGRAMOWANIA

Najlepsze praktyki

KANON INFORMATYKI

Czy zdarzyło Ci się kiedykolwiek...

- pominąć oczywisty błąd w programie i spędzić cały dzień na szukaniu go?
- próbować wprowadzić sensowne zmiany w programie napisanym przez kogoś innego?
- przepisać program od nowa, bo nie dało się go zrozumieć?

Jeśli tak, w przyszłości na pewno chciałbyś tego uniknąć! Takie problemy dla zbyt wielu programistów są niestety chlebem powszednim. Dzieje się tak między innymi dlatego, że testowanie, diagnostyka, przenośność, wydajność czy styl programowania są często traktowane po macoszemu przez osoby tworzące oprogramowanie. A świat rządzony przez olbrzymie interfejsy, wciąż zmieniające się narzędzia, języki czy systemy nie sprzyja podstawowym zasadom tworzenia dobrego kodu – **prostocie, ogólności i przejrzystości**.

Programowanie to coś więcej niż samo pisanie kodu. W książce „Lekcja programowania. Najlepsze praktyki” znajdziesz opis wszystkich zagadnień, z którymi styka się programista – od projektowania, poprzez usuwanie usterek, testowanie kodu czy poprawę jego wydajności, po problemy związane z poprawianiem oprogramowania napisanego przez innych. Wszystko zostało oparte na zacierpiętych z realnych projektów przykładach, napisanych w językach C, C++, Java i innych.

Stwórz swój własny kod w najlepszym stylu!

Brian W. Kernighan i **Rob Pike** pracują w Computing Science Research Center w Bell Laboratories, Lucent Technologies. Brian Kernighan pracuje także dla wydawnictwa Addison-Wesley jako konsultant serii książek „Professional Computing”; napisał też wraz z Dennisem Ritchie książkę „Język ANSI C”.

Rob Pike jest głównym architektem i programistą systemów operacyjnych Plan 9 oraz Inferno. W swojej pracy badawczej interesuje się tworzeniem oprogramowania, które pomaga ludziom pisać ich własne programy.

Tylko tutaj znajdziesz omówienie następujących zagadnień:

- **Styl:** pisanie kodu, który dobrze działa i przyjemnie się czyta
- **Projektowanie:** wybór algorytmów i struktur danych najlepiej nadających się do określonego zadania
- **Interfejsy:** kontrolowanie relacji między składnikami programów
- **Usuwanie błędów:** szybkie i metodyczne wyszukiwanie błędów
- **Testowanie:** zapewnianie niezawodności i poprawności oprogramowania
- **Wydajność:** maksymalizowanie szybkości działania programów
- **Przenośność:** pisanie programów, które działają wszędzie bez żadnych zmian
- **Notacja:** wybór języków i narzędzi, które pozwalają maszynie zrobić więcej



Helion

Nr katalogowy: 6625



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

● <http://helion.pl/promocje>

Książki najchętniej czytane:

● <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

● <http://helion.pl/nowosci>

Helion SA
ul. Kosciuszki 1c, 44-100 Gliwice
tel. 32 230 99 63
e-mail: helion@helion.pl
<http://helion.pl>

helion.pl
księgarnia
internetowa

Cena 47,00 zł

ISBN 978-83-246-3226-8



9 788324 632268

Informatyka w najlepszym wydaniu