helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

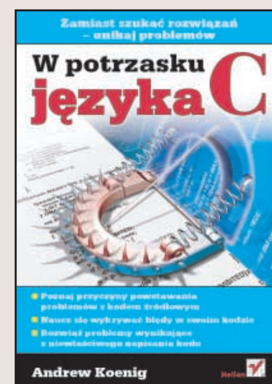
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

W potrzasku języka C

Autor: Andrew Koenig
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-727-2
Tytuł oryginału: C Traps and Pitfalls
Format: B5, stron: 152



Każdy, nawet najbardziej doświadczony programista, popełnia błędy podczas pracy. Niektóre z nich wynikają z pośpiechu, inne — z użycia niewłaściwych konstrukcji, operatorów lub typów. Większość z nich można wykryć i usunąć po kilku minutach uważnej lektury kodu. Zdarzają się jednak i takie błędy, których odnalezienie i skorygowanie zajmuje kilka dni. Błędy te są z reguły łatwe do uniknięcia, jeśli zrozumie się przyczyny ich powstawania.

Książka „W potrzasku C" zawiera omówienie najczęściej spotykanych błędów i przyczyn ich powstawania. Nie zawiera ogólników - jej atutem są konkretne, zaczerpnięte z praktyki, przykłady. Każdy programista prędzej czy później natknie się na jeden z prezentowanych w książce błędów i, dzięki zawartym w niej wiadomościom, będzie w stanie usunąć go i uniknąć w późniejszej pracy.

- Błędy leksykalne i składniowe
- Przepełnienie zakresu
- Problemy z konsolidacją
- Właściwe stosowanie funkcji bibliotecznych
- Makrodefinicje
- Przenośność kodu

Nie trać czasu na usuwanie błędów — dowiedz się co robić, żeby w ogóle nie występowały.

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

Spis treści

Wstęp 7

Wprowadzenie 11

Rozdział 1. Pułapki leksykalne 15
1.1. Porównanie a przypisanie 16
1.2. & i | to nie to samo co && i || 17
1.3. Zachłanność analizatora leksykalnego 18
1.4. Literały stałych całkowitych 19
1.5. Ciągi a znaki 20

Rozdział 2. Pułapki składniowe 23
2.1. Deklaracje funkcji 23
2.2. Priorytety operatorów 26
2.3. Uwaga na średniki! 30
2.4. Instrukcja wyboru switch 32
2.5. Wywołania funkcji 33
2.6. Klauzula else w zagnieżdżonych instrukcjach if 34

Rozdział 3. Pułapki semantyczne 37
3.1. Wskaźniki i tablice 37
3.2. Wskaźniki nie są tablicami 42
3.3. Deklaracje tablic w roli parametrów 43
3.4. Niebezpieczne synekdochy 45
3.5. Wskaźniki puste a ciągi niepuste 46
3.6. Zliczanie a asymetryczne granice zakresów 46
3.7. Kolejność obliczania w wyrażeniu 55
3.8. Operatory &&, || i ! 57
3.9. Przepełnienie zakresu liczby całkowitej 58
3.10. Zwracanie wartości przez funkcję main 59

Rozdział 4. Konsolidacja 63
4.1. Czym jest konsolidator? 63
4.2. Deklaracje a definicje 65
4.3. Kolizje nazw i słowo static 66
4.4. Argumenty, parametry i wartości funkcji 67
4.5. Kontrola typów obiektów zewnętrznych 72
4.6. Pliki nagłówkowe 75

Rozdział 5. Funkcje biblioteczne ... 77
5.1. Funkcja getchar zwraca wartość typu int ... 78
5.2. Aktualizacja pliku sekwencyjnego ... 78
5.3. Buforowanie wyjścia i przydział pamięci ... 80
5.4. Diagnostyka błędów funkcją errno ... 81
5.5. Funkcja signal ... 82

Rozdział 6. Preprocesor ... 85
6.1. Odstępy w makrodefinicjach ... 86
6.2. Makrodefinicje a funkcje ... 86
6.3. Makrodefinicje a instrukcje ... 90
6.4. Makrodefinicje a definicje typów ... 91

Rozdział 7. Kwestie przenośności ... 93
7.1. W obliczu zmian ... 94
7.2. Co z nazwami? ... 95
7.3. Rozmiar liczby całkowitej ... 96
7.4. Czy znaki mają znaki? ... 97
7.5. Operatory przesunięć bitowych ... 98
7.6. Zerowa komórka pamięci ... 99
7.7. Obcinanie przy dzieleniu ... 100
7.8. Rozmiar liczby losowej ... 101
7.9. Zamiana wielkości liter ... 102
7.10. Najpierw zwalniać, potem przydzielać ponownie? ... 103
7.11. Przykładowe problemy nieprzenośności ... 104

Rozdział 8. Porady i odpowiedzi do ćwiczeń ... 109
8.1. Porady ... 110
8.2. Odpowiedzi do ćwiczeń ... 113

Dodatek A Funkcja printf i zmienne listy argumentów ... 129
A.1. Rodzina funkcji printf ... 129
A.1.1. Proste specyfikatory formatu ... 131
A.1.2. Modyfikatory ... 135
A.1.3. Znaczniki ... 138
A.1.4. Zmienna precyzja i szerokość pola ... 140
A.1.5. Neologizmy ... 141
A.1.6. Anachronizmy ... 141
A.2. Zmienne listy argumentów — varargs.h ... 142
A.2.1. Implementacja varargs.h ... 146
A.3. Zmienne listy argumentów w wydaniu ANSI — stdarg.h ... 147

Skorowidz ... 149

Rozdział 1.

Pułapki leksykalne

Czytając zdanie, nie zastanawiamy się nad znaczeniem poszczególnych liter, tworzących kolejne słowa. Litery same w sobie niosą bowiem niewiele treści; grupujemy je więc w słowa i im przypisujemy znaczenie.

Podobnie jest z programami w języku C i innych językach programowania. Pojedyncze znaki programu nie znaczą prawie nic, jeśli rozpatrywać je osobno; znaczenia nabierają dopiero w otoczeniu innych znaków, czyli w pewnym kontekście. Dlatego w wierszu kodu:

```
p->s = "->";
```

oba wystąpienia znaku - znaczą zupełnie co innego. Mówiąc precyzyjniej, oba znaki występują w różnych elementach leksykalnych: pierwszy jest częścią symbolu ->, drugi częścią ciągu znaków. Dalej, znaczenie elementu leksykalnego -> jest całkowicie różne od znaczenia jego poszczególnych znaków.

Pojęcie *symbolu* czy też *elementu leksykalnego* odnosi się do jednostki programu, która odgrywa w nim mniej więcej taką rolę jak słowo w zdaniu — element leksykalny ma podobne znaczenie wszędzie, gdzie występuje. Identyczna sekwencja znaków może w jednym kontekście należeć do jednego elementu leksykalnego, a w innym — do zupełnie innego. Mechanizm kompilatora, który jest odpowiedzialny za podział tekstu programu na elementy leksykalne, nosi często nazwę *analizatora leksykalnego*.

Rozważmy następującą instrukcję:

```
if (x > big) big = x;
```

Pierwszym elementem tej instrukcji jest słowo `if`. Następnym jest znak otwierający nawias, dalej mamy identyfikator `x`, symbol relacji większości, identyfikator `big` itd. W języku C pomiędzy elementy leksykalne możemy wstawiać dowolną liczbę znaków odstępów (spacji, tabulatorów i znaków nowego wiersza), więc instrukcję tę moglibyśmy zapisać również tak:

```
if
(
x
```

```
>
big
)
big
=
x
;
```

W niniejszym rozdziale zajmiemy się szczegółowo niektórymi najczęściej popełnianymi błędami wynikającymi z niezrozumienia znaczenia elementów leksykalnych i zależności pomiędzy nimi a tworzącymi je znakami.

1.1. Porównanie a przypisanie

W większości języków programowania wywodzących się z języka Algol, na przykład w Pascalu czy Adzie, operacja przypisania reprezentowana jest znakami :=, a porównania — znakami ==. W języku C przypisanie reprezentuje znak =, a porównanie — znaki ==. Ponieważ w programie częściej występują przypisania niż porównania, taka reprezentacja jest bardzo wygodna dla programisty, bo dla operacji częstszej stosuje krótszy zapis. Co więcej, w języku C przypisanie jest operatorem, co umożliwia składanie wielu operacji przypisania w jednym wyrażeniu.

Wygoda ta jest jednak źródłem wielu błędów wynikających z omyłkowego zapisania operatora przypisania w miejscu, gdzie powinien znajdować się operator porównania; jak w poniższej instrukcji warunkowej, która powinna wykonywać instrukcję break, kiedy x jest równe y:

```
if (x = y)
    break;
```

W rzeczywistości instrukcja ta przypisuje do x wartość y, a potem sprawdza, czy wynikiem operacji przypisania jest wartość niezerowa. Następnym przykładem może być pętla, która miała pomijać w pliku znaki odstępów (spacji, tabulacji i nowego wiersza):

```
while (c = ' ' || c == '\t' || c == '\n' )
    c = getc(f);
```

W pętli tej w pierwszym porównaniu znaku c zamiast == zapisano =. Ponieważ operator przypisania ma priorytet niższy od operatora ||, „porównanie" w rzeczywistości powoduje przypisanie do c wartości wyrażenia:

```
' ' || c == '\t' || c == '\n'
```

Ponieważ wartość ' ' jest różna od zera, całe wyrażenie ma wartość 1 niezależnie od pierwotnej wartości c. Dlatego pętla spowoduje „przewinięcie" całego pliku. Co stanie się po wyczerpaniu znaków pliku, zależy od tego, czy implementacja języka pozwala na kontynuowanie odczytu po osiągnięciu końca pliku. Jeśli tak, program wejdzie w pętlę nieskończoną.

Niektóre kompilatory języka C próbują wspomagać użytkowników (programistów), prezentując komunikaty ostrzegawcze o warunku w postaci *w1* = *w2*. Kiedy więc w programie zachodzi faktyczna potrzeba przypisania wartości do zmiennej i ustalenia wyniku operacji przypisania, to w celu uniknięcia komunikatów ostrzegawczych należałoby porównanie zapisać jawnie, czyli zamiast:

```
if (x = y)
    foo();
```

napisać:

```
if ((x = y) != 0)
    foo();
```

To upewni kompilator (i innych programistów) co do zamierzeń programisty. Przyczyna konieczności ujęcia przypisania `x = y` w nawiasy została omówiona w podrozdziale 2.2.

Częste są również pomyłki odwrotne, jak poniżej:

```
if ((filedesc == open(argv[i], 0)) < 0)
    error();
```

Funkcja `open` zastosowana w tym przykładzie zwraca `-1`, jeśli wykryje błąd otwarcia pliku, oraz zero albo wartość dodatnią, kiedy operacja zakończy się pomyślnie. Powyższa instrukcja ma więc zapisać wynik operacji otwarcia pliku w zmiennej `filedesc` i równocześnie dokonać sprawdzenia wyniku operacji. Ale zamiast `=` napisano `==`. Z tego względu kod ten faktycznie porównuje wartość funkcji `open` z wartością zmiennej `filedesc` i następnie sprawdza, czy wynik porównania był wartością ujemną. Tymczasem porównanie nigdy nie daje wartości ujemnej, a jedynie albo `1` (dla równych operandów), albo `0` (jeśli operandy są różne). Tak więc funkcja `error` nie zostanie nigdy wywołana, niezależnie od wyniku operacji otwarcia pliku, a wartość zmiennej `filedesc` nie będzie nijak odzwierciedlać wartości funkcji `open`. Niektóre kompilatory ostrzegają programistów o nieskuteczności porównywania z zerem, ale nie polegałbym w takich przypadkach na kompilatorach.

1.2. & i | to nie to samo co && i ||

W przypadku operatorów porównania i przypisania pomyłki wynikają zazwyczaj z nawyków wyniesionych z innych języków programowania, w których porównanie reprezentuje znak `=`. Podobne nawyki dotyczą również operatorów `&` i `&&` oraz `|` i `||`, a to dlatego, ponieważ znaczenie `&` i `|` jest w C odmienne niż w niektórych innych językach. Dokładne znaczenie wszystkich wymienionych operatorów omawiane jest w podrozdziale 3.8.

1.3. Zachłanność analizatora leksykalnego

Niektóre elementy leksykalne języka C, jak /, * czy = to ciągi jednoznakowe. Inne z kolei (/*, == czy wszelkiej maści identyfikatory) składają się z dwóch i większej liczby znaków. Kiedy kompilator napotyka w tekście programu znak /, a za nim znak *, musi zdecydować, czy oba znaki traktować jako niezależne elementy leksykalne czy też połączyć je w jeden element. W języku C wątpliwości rozstrzygane są prostą regułą: „zawsze dopasowuj najdłuższy możliwy ciąg". Konwersja programu w języku C odbywa się od lewej do prawej, a za każdym razem z tekstu programu izolowany jest element o największej możliwej liczbie znaków. Taka strategia dopasowania nosi nazwę *zachłannej*. Twórcy języka C, Kernighan i Ritchie, wyrazili tę zasadę następująco: „jeśli ciąg wejściowy został do danego znaku podzielony na elementy leksykalne, następnym elementem leksykalnym jest najdłuższy ciąg znaków, który można uznać za element leksykalny". Elementy leksykalne, z wyjątkiem literałów znakowych i łańcuchowych (ciągów), nie mogą zawierać żadnych znaków odstępów.

Jeśli uwzględnić zasadę zachłanności analizy leksykalnej, znaki == interpretuje się jako pojedynczy element leksykalny, znaki = = tworzą dwa odrębne elementy, a zapis:

```
a---b
```

znaczy dokładnie tyle, co:

```
a -- - b
```

nigdy zaś:

```
a - -- b
```

Podobnie, jeśli pierwszym znakiem elementu leksykalnego jest /, a następnym *, kompilator rozpoznaje dwuznakowy symbol rozpoczynający komentarz niezależnie od kontekstu.

Poniższe wyrażenie z pozoru ustawia zmienną y wartością x podzieloną przez wartość wskazywaną przez p:

```
y = x/*p       /* p wskazuje na dzielnik */;
```

W rzeczywistości znaki /* rozpoczynają komentarz, więc kompilator zignoruje całość tekstu programu za tymi znakami, aż do znaków symbolu końca komentarza */. Innymi słowy, powyższa instrukcja po prostu przypisuje x do y, ignorując zupełnie wartość p. Aby przed przypisaniem wykonać dzielenie (jak stoi w komentarzu), należałoby zapisać instrukcję tak:

```
y = x / *p       /* p wskazuje na dzielnik */;
```

albo choćby tak:

```
y = x/(*p)       /* p wskazuje na dzielnik */;
```

Tego rodzaju „niejednoznaczności" powodują problemy również w innych kontekstach.

Na przykład niegdyś w języku C operacja przypisania złożonego, reprezentowana dziś operatorem w rodzaju +=, reprezentowana była znakami =+. Niektóre kompilatory do dziś akceptują tę archaiczną składnię, traktując zapis:

```
a=-1;
```

jako:

```
a =- 1;
```

To z kolei znaczy tyle co:

```
a = a - 1;
```

co z pewnością byłoby pewnym zaskoczeniem dla programisty, który najwyraźniej chciał wykonać operację:

```
a = -1;
```

„Archaiczne" kompilatory, o których mowa, potraktowałyby również zapis:

```
a=/*b;
```

jako:

```
a =/ * b ;
```

pomimo że znaki /* z pozoru zapowiadają komentarz.

Starsze kompilatory interpretują jako przypisania złożone również rozłączne elementy leksykalne, przyjmując bez protestów do kompilacji zapis:

```
a >> = 1;
```

który to zapis zostałby przez kompilator ANSI C odrzucony.

1.4. Literały stałych całkowitych

Jeśli pierwszy znak literału stałej całkowitej to cyfra 0, to stała jest interpretowana jako liczba ósemkowa. Stąd znaczenie literałów `10` i `010` jest odmienne. Co gorsza, wiele kompilatorów C przyjmuje bez protestów literały stałych ósemkowych zawierające cyfry „ósemkowe" `8` i `9`. Wartość takich literałów obliczana jest zgodnie z definicją liczby ósemkowej, więc np. 0195 to tyle, co $1{\cdot}8^2+9{\cdot}8^1+5{\cdot}8^0$, czyli dziesiętnie 141, a ósemkowo: 0215. Oczywiście takie stosowanie literałów stałych ósemkowych gorąco odradzam. W języku ANSI C jest ono niedozwolone.

Niewłaściwe wartości stałych ósemkowych dają się we znaki w kontekstach takich jak ten:

```
struct {
    int part_number;
    char* description;
} parttab[] = {
    046, "wichajster lewy",
    047, "wichajster prawy",
    125, "teges",
};
```

1.5. Ciągi a znaki

W języku C znaki pojedynczych i podwójnych cudzysłowów mają zasadniczo różne znaczenie, a ich mylenie w niektórych kontekstach powoduje błędy objawiające się nie tyle komunikatami kompilatora, co właśnie niespodziewanym działaniem programu.

Znak ujęty w znaki pojedynczego cudzysłowu to po prostu alternatywny zapis liczby całkowitej reprezentującej ów znak w zbiorze znaków przyjętym w danej implementacji. Jeśli jest to zbiór ASCII, to `'a'` oznacza tyle co `0141` albo `97`.

Ciąg ujęty w znaki cudzysłowów podwójnych jest z kolei skrótowym zapisem wskaźnika pierwszego znaku nienazwanej tablicy inicjalizowanej znakami składającymi się na ciąg i uzupełnianej dodatkowym znakiem o wartości zerowej.

Zapis:

```
printf("Ahoj, przygodo\n");
```

jest równoznaczny zapisowi:

```
char hello[] = {'A', 'h', 'o', 'j', ',', ' ',
    'p', 'r', 'z', 'y', 'g', 'o', 'd', 'o', '\n', 0};
printf(hello);
```

Ponieważ znak w pojedynczych cudzysłowach reprezentuje wartość całkowitą, a znak w cudzysłowach podwójnych reprezentuje wskaźnik, kompilator, kontrolując typy, wychwytuje zwykle miejsca, w których wartości te są stosowane odwrotnie. Stąd zapis:

```
char *slash = '/';
```

spowoduje błąd kompilacji, ponieważ `'/'` nie jest wskaźnikiem znaku. W niektórych implementacjach języka kontrola typów nie jest jednak dość szczegółowa; dotyczy to zwłaszcza argumentów funkcji `printf`. Zapis:

```
printf('\n');
```

zamiast:

```
printf("\n");
```

może wtedy powodować niespodzianki podczas wykonania programu mimo bezbłędnej kompilacji. Inne tego rodzaju przypadki zostały szczegółowo omówione w podrozdziale 4.4.

Ponieważ liczba całkowita jest w języku C z reguły na tyle pojemna, że mieści kilka wartości znakowych, niektóre kompilatory języka C dopuszczają stosowanie wieloznakowych stałych znakowych; oznacza to, że można zapisać `'tak'` w miejsce `"tak"` i kompilator nie zgłosi błędu. Jednak należy pamiętać, że drugi z tych zapisów oznacza „adres pierwszego z czterech kolejnych komórek pamięci zawierających znaki `t`, `a` i `k`", podczas gdy znaczenie zapisu `'tak'` (choć niezdefiniowane dokładnie) w wielu implementacjach jest równoznaczne „liczbie całkowitej złożonej w jakiś sposób z wartości znaków `t`, `a` i `k`".

Ćwiczenie 1.1

Niektóre z kompilatorów języka C pozwalają na stosowanie komentarzy zagnieżdżonych. Napisz program w języku C, który potrafi wykryć, za pomocą którego z kompilatorów został skompilowany, nie prowokując przy tym *żadnych* błędów kompilacji. Innymi słowy, kod programu powinien być poprawny dla obu reguł interpretacji komentarzy, ale działać różnie dla różnych interpretacji. *Wskazówka*: symbol komentarza wewnątrz ciągu znaków jest interpretowany jako fragment ciągu znaków; znaki `""` wewnątrz komentarza to część komentarza.

Ćwiczenie 1.2

Gdybyś pisał kompilator języka C, czy pozwoliłbyś jego użytkownikom na zagnieżdżanie komentarzy? Gdybyś dysponował kompilatorem dającym możliwość zagnieżdżania komentarzy, czy korzystałbyś z tej możliwości? Czy odpowiedź na drugie z tych pytań wpływa na odpowiedź na pytanie pierwsze?

Ćwiczenie 1.3

Dlaczego `n-->0` oznacza `n-- > 0`, a nie `n- -> 0`?

Ćwiczenie 1.4

Co oznacza zapis `a+++++b`?