

## » Idź do

- Spis treści
- Przykładowy rozdział

## » 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. 032 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
© Helion 1991-2010

## Język ANSI C. Programowanie. Ćwiczenia. Wydanie II

Autorzy: Clovis L. Tondo, Scott E. Gimpel

Tłumaczenie: Paweł Koronkiewicz

ISBN: 978-83-246-2591-8

Tytuł oryginału: The C Answer Book, (2nd Edition)

Format: 158×235, stron: 168



Książka „Język ANSI C. Programowanie. Wydanie II” to jedna z najlepszych dostępnych na rynku pozycji do nauki tego języka, zaliczana do klasyki literatury informatycznej i ciesząca się niemalejącą popularnością. Przejrzyście opisana teoria, liczne przykłady oraz zbiór ćwiczeń to atuty doceniane przez kolejne pokolenia programistów.

Niniejsza książka zawiera rozwiązania wszystkich ćwiczeń zawartych w „Języku ANSI C. Programowanie. Wydanie II”. Oprócz działającego i przetestowanego kodu znajdziesz w niej komentarze do specyficznych konstrukcji i samego sposobu rozwiązywania zadań. Połączenie teorii z praktyką pozwoli Ci błyskawicznie przyswoić wiedzę na temat języka C, a następnie wykorzystać ją w praktyce. Ponadto część rozwiązań z pewnością przyda się w codziennej pracy, dlatego też książka ta sprawdzi się zarówno w rękach adepta języka C, jak i zawodowego programisty.

# Spis treści

<b>Wstęp</b>	<b>5</b>
<b>Rozdział 1. Wprowadzenie</b>	<b>7</b>
<b>Rozdział 2. Typy, operatory i wyrażenia</b>	<b>37</b>
<b>Rozdział 3. Sterowanie wykonywaniem programu</b>	<b>49</b>
<b>Rozdział 4. Funkcje i struktura programu</b>	<b>57</b>
<b>Rozdział 5. Wskaźniki i tablice</b>	<b>77</b>
<b>Rozdział 6. Struktury</b>	<b>121</b>
<b>Rozdział 7. Wejście i wyjście</b>	<b>135</b>
<b>Rozdział 8. Interfejs systemu UNIX</b>	<b>149</b>
<b>Skorowidz</b>	<b>161</b>

# Rozdział 4.

# Funkcje i struktura programu

---

## Ćwiczenie 4.1 (str. 89)

Napisz funkcję `strrindex(s,t)`, która zwraca pozycję *ostatniego* wystąpienia `t` w `s` lub `-1`, jeżeli wyszukiwany ciąg nie został znaleziony.

```
/* strrindex: zwraca index ostatniego wystąpienia t w s lub -1, jeżeli nie występuje */
int strrindex(char s[], char t[])
{
    int i, j, k, pos;

    pos = -1;
    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            pos = i;
    }
    return pos;
}
```

Funkcja `strrindex` jest podobna do `strindex`, przedstawionej w podrozdziale 4.1 podręcznika K&R. Gdy funkcja `strindex` znajduje dopasowany podciąg, zwraca jego pozycję, która jest pozycją pierwszego wystąpienia `t` w `s`. Funkcja `strrindex` nie zwraca pozycji znalezionej podciągu, ale kontynuuje wyszukiwanie, ponieważ jej zadaniem jest określenie położenia ostatniego wystąpienia `t` w `s`:

```
if (k > 0 && t[k] == '\0')
    pos = i;
```

Ten sam problem można rozwiązać także następująco:

```
#include <string.h>

/* strrindex: zwraca index ostatniego wystąpienia t w s lub -1, jeżeli nie występuje */
int strrindex(char s[], char t[])
{
    int i, j, k;

    for (i = strlen(s) - strlen(t); i >= 0; i--) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Jest to rozwiązanie efektywniejsze. Przeglądanie ciągu rozpoczyna się od końca ciągu s minus długość ciągu t. Brak dopasowania powoduje przesunięcie wyszukiwania o jedną pozycję w stronę początku ciągu. Gdy tylko funkcja znajduje t w s, zwraca bieżącą pozycję, i. Jest to ostatnie wystąpienie t w s.

---

## Ćwiczenie 4.2 (str. 91)

Dodaj do funkcji atof możliwość obsługi notacji wykładniczej, postaci:

```
123.45e-6
```

gdzie po liczbie zmiennoprzecinkowej może wystąpić litera e lub E i wykładnik, z opcjonalnym znakiem.

```
#include <ctype.h>

/* atof: konwertuje ciąg znaków s na liczbę double */
double atof(char s[])
{
    double val, power;
    int exp, i, sign;

    for (i = 0; isspace(s[i]); i++) /* pomiń białe znaki */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
```

```

        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    val = sign * val / power;

    if (s[i] == 'e' || s[i] == 'E') {
        sign = (s[++i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')
            i++;
        for (exp = 0; isdigit(s[i]); i++)
            exp = 10 * exp + (s[i] - '0');
        if (sign == 1)
            while (exp-- > 0)                /* wykładnik dodatni */
                val *= 10;
        else
            while (exp-- > 0)                /* wykładnik ujemny */
                val /= 10;
    }
    return val;
}

```

Pierwsza część funkcji to powtórzenie funkcji `atoi` z podrzdziału 4.2 podręcznika K&R. Funkcja pomija białe znaki, zapisuje znak i oblicza liczbę. Pobieranie liczby z kropką dziesiętną wymaga identycznej procedury niezależnie od tego, czy w dalszej części pojawi się wykładnik.

Druga część funkcji odpowiada za konwersję opcjonalnego wykładnika. Jeżeli ta część liczby nie występuje, funkcja zwraca wartość zapisaną w `val`. Jeżeli wykładnik jest obecny, to jego znak zostaje zapisany w zmiennej `sign`, po czym wartość zostaje obliczona i zapisana w zmiennej `exp`.

Końcowa operacja

```

    if (sign == 1)
        while (exp-- > 0)
            val *= 10;
    else
        while (exp-- > 0)
            val /= 10;

```

modyfikuje liczbę odpowiednio do ustalonej wcześniej wartości wykładnika. Jeżeli wykładnik jest dodatni, liczba zostaje pomnożona `exp` razy przez 10. Jeżeli wykładnik jest ujemny, liczba zostaje podzielona `exp` razy przez 10. W zmiennej `val` zostaje zapisany wynik, który jest zwracany do programu wywołującego funkcję.

Zmienna `val` jest dzielona przez 10, a nie mnożona przez 0.1, ponieważ liczba 0,1 nie jest w zapisie binarnym dokładna. Na większości komputerów wartość 0,1 jest reprezentowana jako nieco mniejsza niż 0,1. W efekcie mnożenie 10.0\*0.1 rzadko daje wynik 1.0. Powtarzanie dzielenia przez 10 jest więc lepszym rozwiązaniem niż powtarzanie mnożenia przez 0,1, choć utrata dokładności wciąż występuje.

## Ćwiczenie 4.3 (str. 97)

W oparciu o schemat przedstawiony w przykładach program kalkulatora można łatwo rozbudowywać. Dodaj obsługę operatora modulo (%) i obsługę liczb ujemnych.

```
#include <stdio.h>
#include <math.h>      /* dla atof() */

#define MAXOP 100     /* dopuszczalny rozmiar operandu lub operatora */
#define NUMBER '0'    /* sygnał, że pobrano liczbę */

int getop(char []);
void push(double);
double pop(void);

/* kalkulator z odwrotną notacją polską */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if (op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            case '%':
                op2 = pop();
                if (op2 != 0.0)
                    push(fmod(pop(), op2));
                else
                    printf("error: zero divisor\n");
        }
    }
}
```

```

        break;
    case '\n':
        printf("\t%.8g\n", pop());
        break;
    default:
        printf("error: unknown command %s\n", s);
        break;
    }
}
return 0;
}

```

Zmieniliśmy program główny i funkcję `getop`. Funkcje `push` i `pop` pozostają niezmienione.

Operator modulo (%) jest traktowany podobnie jak operator dzielenia (/). Funkcja biblioteczna `fmod` oblicza resztę z dzielenia dwóch elementów na wierzchołku stosu. `op2` to element pobrany z wierzchołka jako pierwszy.

Oto zmodyfikowana wersja funkcji `getop`:

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMBER '0' /* sygnał, że została znaleziona liczba */

int getch(void);
void ungetch(int);

/* getop: pobiera następny operator lub operand (liczbę) */
int getop(char s[])
{
    int c, i;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    i = 0;
    if (!isdigit(c) && c != '.' && c != '-')
        return c; /* nie jest liczbą */
    if (c == '-')
        if (isdigit(c = getch()) || c == '.')
            s[++i] = c; /* liczba ujemna */
        else {
            if (c != EOF)
                ungetch(c);
            return '-'; /* znak minus */
        }
    if (isdigit(c)) /* pobierz część całkowitą */
        while (isdigit(s[++i] = c = getch()))
            ;
}

```

```
    if (c == '.') /* pobierz część ułamkową */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Funkcja `getop` sprawdza następny znak po znaku `-`, aby określić, czy dane zawierają liczbę ujemną. Przykładowo

`- 1`

to znak minus i liczba. Jednak

`-1.23`

jest liczbą ujemną.

Rozbudowany kalkulator zapewnia obsługę sekwencji

`1 -1 +`  
`-10 3 %`

Pierwsze wyrażenie prowadzi do uzyskania wartości  $0 (1 + (-1))$ . Drugie wyrażenie ma wartość  $-1$ .

---

## Ćwiczenie 4.4 (str. 97)

Utwórz polecenie wypisujące element na wierzchołku stosu bez jego usuwania ze stosu, polecenie duplikujące element na wierzchołku stosu, polecenie zamieniające miejscami dwa górne elementy oraz polecenie usuwające całą zawartość stosu.

```
#include <stdio.h>
#include <math.h> /* dla atof() */

#define MAXOP 100 /* dopuszczalny rozmiar operandu lub operatora */
#define NUMBER '0' /* sygnał, że pobrano liczbę */

int getop(char []);
void push(double);
double pop(void);
void clear(void);

/* kalkulator z odwrotną notacją polską */
main()
{
    int type;
    double op1, op2;
    char s[MAXOP];
```

```

while ((type = getop(s)) != EOF) {
    switch (type) {
        case NUMBER:
            push(atoi(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '?':
            /* wypisz element z wierzchołka stosu */
            op2 = pop();
            printf("\t%.8g\n", op2);
            push(op2);
            break;
        case 'c':
            /* opróżnij stos */
            clear();
            break;
        case 'd':
            /* duplikuj element na wierzchołku stosu */
            op2 = pop();
            push(op2);
            push(op2);
            break;
        case 's':
            /* zamień dwa elementy na wierzchołku */
            op1 = pop();
            op2 = pop();
            push(op1);
            push(op2);
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
    }
}
return 0;
}

```

Znak nowego wiersza powoduje pobranie elementu z wierzchołka stosu i wypisanie go. Dodaliśmy nowy operator, '?', który pobiera element z wierzchołka stosu, wypisuje go, a następnie zwraca na stos. Nie usuwamy elementu z wierzchołka stosu w sposób trwały (tak jak w przypadku użycia znaku nowego wiersza), a stosujemy sekwencję pop-printf-push. Dzięki temu główny program nie musi wiedzieć o stosie ani wykorzystywanych do jego obsługi zmiennych.

Duplikowanie elementu na wierzchołku stosu polega na zdjęciu go ze stosu i dwukrotnym wywołaniu funkcji umieszczającej go na stosie ponownie.

Podobnie zamiana miejscami dwóch elementów na wierzchołku jest realizowana poprzez zdjęcie ich ze stosu i ponowne zapisanie na stosie.

Czyszczenie stosu jest prostą operacją, sprowadzającą się do przypisania zmiennej sp wartości 0. Dodaliśmy nową funkcję, uzupełniającą push i pop, która wykonuje właśnie taką operację. Pozwala to zachować zasadę, że tylko funkcje operujące danymi stosu odwołują się do niego i związanych z nim zmiennych.

```
/* clear: opróżnia stos */
void clear(void)
{
    sp = 0;
}
```

---

## Ćwiczenie 4.5 (str. 97)

Dodaj dostęp do funkcji biblioteki, takich jak sin, exp, i pow. Patrz <math.h> w części 4. dodatku B.

```
#include <stdio.h>
#include <string.h>
#include <math.h> /* dla atof() */

#define MAXOP 100 /* dopuszczalny rozmiar operandu lub operatora */
#define NUMBER '0' /* sygnał, że pobrano liczbę */
#define NAME 'n' /* sygnał, że pobrano nazwę */

int getop(char []);
void push(double);
double pop(void);
void mathfnc(char []);

/* kalkulator z odwrotną notacją polską */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
```

```

switch (type) {
case NUMBER:
    push(atof(s));
    break;
case NAME:
    mathfnc(s);
    break;
case '+':
    push(pop() + pop());
    break;
case '*':
    push(pop() * pop());
    break;
case '-':
    op2 = pop();
    push(pop() - op2);
    break;
case '/':
    op2 = pop();
    if (op2 != 0.0)
        push(pop() / op2);
    else
        printf("error: zero divisor\n");
    break;
case '\n':
    printf("\t%.8g\n", pop());
    break;
default:
    printf("error: unknown command %s\n", s);
    break;
}
}
return 0;
}

```

*/\* mathfnc: sprawdza, czy ciąg s jest nazwą obsługiwanej funkcji matematycznej \*/*

```

void mathfnc(char s[])
{
    double op2;

    if (strcmp(s, "sin") == 0)
        push(sin(pop()));
    else if (strcmp(s, "cos") == 0)
        push(cos(pop()));
    else if (strcmp(s, "exp") == 0)
        push(exp(pop()));
    else if (strcmp(s, "pow") == 0)
        op2 = pop();
        push(pow(pop(), op2));
    } else
        printf("error: %s not supported\n", s);
}

```

Plik źródłowy zmodyfikowanej funkcji `getop`:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMBER '0' /* sygnał, że została znaleziona liczba */
#define NAME 'n' /* sygnał, że pobrano nazwę */

int getch(void);
void ungetch(int);

/* getop: pobiera następny operator, operand lub nazwę funkcji */
int getop(char s[])
{
    int c, i;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    i = 0;
    if (islower(c)) /* polecenie lub nazwa */
        while (islower(s[++i] = c = getch()))
            ;
        s[i] = '\0';
        if (c != EOF)
            ungetch(c); /* pobrany o jeden znak za dużo */
        if (strlen(s) > 1)
            return NAME; /* >1 znak, czyli nazwa */
        else
            return c; /* to może być polecenie */
    }
    if (!isdigit(c) && c != '.')
        return c; /* nie jest liczbą */
    if (isdigit(c)) /* pobierz część całkowitą */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /* pobierz część ułamkową */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Zmodyfikowaliśmy funkcję `getop`, tak aby mogła pobierać ciąg małych liter i zwracać go jako typ `NAME`. Program główny rozpoznaje `NAME` jako jeden z poprawnych typów i wywołuje funkcję `mathfnc`.

Funkcja `mathfnc` jest nowym elementem. Wykonuje ona serię instrukcji `if` aż do znalezienia nazwy funkcji zapisanej w ciągu `s`. Jeżeli nazwa nie zostanie znaleziona, funkcja zgłasza błąd. Jeżeli ciąg `s` jest nazwą jednej z obsługiwanych funkcji matematycznych, `mathfnc` pobiera ze stosu odpowiednią liczbę elementów i wywołuje tę funkcję. Funkcja zwraca wartość, którą `mathfnc` umieszcza na stosie.

Przykładowo funkcja `sin` oczekuje argumentu w radianach, a `sinus PI / 2` ma wartość 1.

```
3.14159265 2 / sin
```

Pierwsza operacja to dzielenie `PI` przez `2`. Wynik zostaje umieszczony na stosie. Funkcja `sin` pobiera tę wartość z wierzchołka stosu, oblicza wartości sinus i zapisuje wynik, 1, ponownie na stosie.

```
3.14159265 2 / sin 0 cos +
```

daje wynik 2, ponieważ `sinus PI / 2` to 1 i `cosinus zera` to 1.

Inny przykład,

```
5 2 pow 4 2 pow +
```

podnosi 5 do potęgi 2, następnie 4 do potęgi 2, po czym dodaje te dwie wartości.

Funkcja `getop` nie zna nazw funkcji matematycznych. Zwraca ona jedynie znalezione ciągi. Zapewnia to możliwość łatwego rozbudowywania funkcji `mathfnc` i wprowadzania obsługi dalszych operacji.

## Ćwiczenie 4.6 (str. 98)

Dodaj polecenia obsługi zmiennych (łatwo jest zapewnić możliwość korzystania z dwudziestu sześciu zmiennych przy użyciu jednoliterowych nazw). Dodaj zmienną przechowującą ostatnią wypisaną wartość.

```
#include <stdio.h>
#include <math.h>      /* dla atof() */

#define MAXOP 100      /* dopuszczalny rozmiar operandu lub operatora */
#define NUMBER '0'     /* sygnał, że pobrano liczbę */

int getop(char []);
void push(double);
double pop(void);

/* kalkulator z odwrotną notacją polską */
main()
{
    int i, type, var = 0;
    double op2, v;
    char s[MAXOP];
```

```
double variable[26];

for (i = 0; i < 26; i++)
    variable[i] = 0.0;
while ((type = getop(s)) != EOF) {
    switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '=':
            pop();
            if (var >= 'A' && var <= 'Z')
                variable[var - 'A'] = pop();
            else
                printf("error: no variable name\n");
            break;
        case '\n':
            v = pop();
            printf("\t%.8g\n", v);
            break;
        default:
            if (type >= 'A' && type <= 'Z')
                push(variable[type - 'A']);
            else if (type == 'v')
                push(v);
            else
                printf("error: unknown command %s\n", s);
            break;
    }
    var = type;
}
return 0;
}
```

Dodane zmienne to wielkie litery, od A do Z. Litery te służą zarazem jako indeksy zmiennej tablicowej. Wprowadziliśmy także zmienną *v*, w której zapisywana jest ostatnia wypisywana wartość.

Gdy program napotyka nazwę zmiennej (od A do Z lub *v*), zapisuje jej wartość na stosie. Dostępny jest także nowy operator, '=', który przypisuje element z wierzchołka stosu zmiennej poprzedzającej operator. Na przykład

```
3 A =
```

przypisuje wartość 3 zmiennej A. Późniejsze

```
2 A +
```

dodaje liczby 2 i 3 (wartość zmiennej A). Po dojściu do znaku nowego wiersza program wypisuje liczbę 5 i przypisuje jednocześnie wartość 5 zmiennej *v*. Jeżeli następną operacją jest

```
v 1 +
```

to wynikiem jest 6: 5+1.

## Ćwiczenie 4.7 (str. 98)

Napisz procedurę `ungets(s)`, która zwraca do danych wejściowych cały ciąg znaków. Czy funkcja ta powinna korzystać ze zmiennych `buf` i `bufp`, czy raczej tylko z funkcji `ungetch`?

```
#include <string.h>

/* ungets: zwraca ciąg znaków do strumienia danych wejściowych */
void ungets(char s[])
{
    int len = strlen(s);
    void ungetch(int);

    while (len > 0)
        ungetch(s[--len]);
}
```

Zmienna `len` zawiera liczbę znaków w ciągu `s` (bez końcowego '\0'), która jest określana przy użyciu funkcji `strlen` (patrz podrozdział 2.3 podręcznika K&R).

Funkcja `ungets` wywołuje procedurę `ungetch` (patrz koniec podrozdziału 4.3 podręcznika K&R) `len` razy, za każdym razem przekazując do strumienia danych wejściowych jeden znak ciągu `s`. Funkcja dba o przekazywanie znaków w odwróconej kolejności.

Funkcja `ungets` nie musi znać zmiennych `buf` i `bufp`. Procedura `ungetch` zapewnia wykrywanie błędów i właściwą pracę z tymi zmiennymi.

## Ćwiczenie 4.8 (str. 98)

Zmodyfikuj funkcje `getch` i `ungetch`, przyjmąwszy założenie, że nigdy nie będzie wycofywany więcej niż jeden znak.

```
#include <stdio.h>

char buf = 0;

/* getch: pobiera znak danych wejściowych (mógł być wcześniej wycofany przez ungetch */
int getch(void)
{
    int c;

    if (buf != 0)
        c = buf;
    else
        c = getchar();
    buf = 0;
    return c;
}

/* ungetch: wycofuje znak do strumienia danych wejściowych */
void ungetch(int c)
{
    if (buf != 0)
        printf("ungetch: too many characters\n");
    else
        buf = c;
}
```

Bufor, `buf`, nie jest już tablicą, ponieważ nigdy nie będzie w nim przechowywany więcej niż jeden znak.

Zmienna `buf` jest inicjalizowana przy ładowaniu programu wartością 0. Funkcja `getch` przywraca tę wartość za każdym razem, gdy pobiera znak. Funkcja `ungetch` przed zapisaniem znaku sprawdza, czy bufor jest pusty. Jeżeli bufor nie jest pusty, wyświetla komunikat błędu.

---

## Ćwiczenie 4.9 (str. 98)

Nasze funkcje `getch` i `ungetch` nie obsługują poprawnie wycofywania znaku EOF. Zastanów się, jakie powinny one mieć cechy w przypadku cofania znaku EOF, po czym zaimplementuj nową koncepcję.

```
#include <stdio.h>

#define BUFSIZE 100
```

```

int buf[BUFSIZE]; /* bufor dla ungetch */
int bufp = 0;     /* następna wolna pozycja w buforze */

/* getch: pobiera znak danych wejściowych (mógł być wcześniej wycofany przez ungetch */
int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

/* ungetch: wycofuje znak do strumienia danych wejściowych */
void ungetch(int c)
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

```

W funkcjach `getch` i `ungetch` przedstawionych w podręczniku K&R bufor, `buf`, jest deklarowany jako tablica znaków:

```
char buf[BUFSIZE];
```

Język C nie wymaga, aby zmienna `char` została określona jako `signed` lub `unsigned` (patrz podrozdział 2.7 podręcznika K&R). Konwersja wartości `char` na `int` nie może prowadzić do uzyskania wartości ujemnej. Na niektórych komputerach, gdy lewy skrajny bit wartości `char` jest równy 1, konwersja na `int` prowadzi do uzyskania wartości ujemnej. Na innych konwersja polega na dodaniu z lewej strony odpowiedniej liczby zer. Takie przekształcenie zawsze prowadzi do uzyskania liczby dodatniej, niezależnie od tego, czy lewy skrajny bit miał wartość 1, czy nie.

W notacji szesnastkowej `-1` to `0xFFFF` (w przypadku 16 bitów). Po zapisaniu wartości `0xFFFF` w zmiennej `char` zmienna zawiera `0xFF`. Konwersja `0xFF` na `int` może prowadzić do uzyskania wartości `0x00FF`, czyli 255, lub `0xFFFF`, czyli `-1`.

liczba ujemna (-1)	-> znak	-> liczba int
0xFFFF	0xFF	0x00FF (255)
0xFFFF	0xFF	0xFFFF (-1)

Jeżeli mamy traktować `E0F (-1)` jak każdy inny znak, zmienna `buf` powinna zostać zadeklarowana jako tablica liczb całkowitych:

```
int buf[BUFSIZE];
```

Nie są wtedy wykonywane żadne operacje konwersji i wartość `E0F (-1)`, podobnie jak każda liczba ujemna, jest obsługiwana w sposób jednolity na wszystkich platformach.

## Ćwiczenie 4.10 (str. 98)

Alternatywna organizacja pracy z danymi wejściowymi opiera się na użyciu `getline` w celu pobrania całego wiersza. Dzięki temu funkcje `getch` i `ungetch` nie są potrzebne. Przekształć kalkulator, tak aby jego praca opierała się na takim podejściu do danych wejściowych.

```
#include <stdio.h>
#include <ctype.h>

#define MAXLINE 100
#define NUMBER '0' /* sygnał, że została znaleziona liczba */

int getline(char line[], int limit);

int li = 0; /* indeks wiersza wejściowego */
char line [MAXLINE]; /* jeden wiersz wejściowy */

/* getop: pobiera następny operator lub operand (liczbę) */
int getop(char s[])
{
    int c, i;

    if (line[li] == '\0')
        if (getline(line, MAXLINE) == 0)
            return EOF;
        else
            li = 0;
    while ((s[0] = c = line[li++]) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* nie jest liczbą */
    i = 0;
    if (isdigit(c)) /* pobierz część całkowitą */
        while (isdigit(s[++i] = c = line[li++]))
            ;
    if (c == '.') /* pobierz część ułamkową */
        while (isdigit(s[++i] = c = line[li++]))
            ;
    s[i] = '\0';
    li--;
    return NUMBER;
}
```

Zamiast `getch` i `ungetch` używamy w `getop` funkcji `getline`. `line` to tablica zawierająca jeden pełny wiersz danych wejściowych. `li` to indeks kolejnego znaku w `line`. Deklarujemy `line` i `li` jako zmienne wewnętrzne, aby zachowywały wartości między wywołaniami.

Gdy `getop` dochodzi do końca wiersza (lub żaden wiersz nie został jeszcze pobrany),

```
if (line[li] == '\0')
```

następuje wywołanie `getline` w celu pobrania nowego wiersza danych.

W oryginalnej wersji (podrozdział 4.3 podręcznika K&R) funkcja `getop` wywołuje `getch` za każdym razem, gdy potrzebny jest nowy znak. W tej wersji pobierany jest znak na pozycji `li` w tablicy `line`, po czym wartość `li` jest zwiększana. Na końcu funkcji, zamiast wywoływać `ungetch` w celu zwrócenia znaku do strumienia danych wejściowych, zmniejszamy `li`, aby wycofać się o jeden znak.

Warto pamiętać, że każda funkcja ma możliwość wykorzystywania i modyfikowania zmiennych zewnętrznych stosowanych w innych funkcjach, więc zmienne `li` i `line` mogą zostać zmienione przez funkcję inną niż `getop`. Czasem wskazane jest zabezpieczenie programu przed takimi sytuacjami. Umożliwia to zadeklarowanie zmiennych jako `static`. Nie zrobiliśmy tego, bo zmienne `static` zostaną omówione dopiero w podrozdziale 4.6 podręcznika K&R.

## Ćwiczenie 4.11 (str. 102)

Zmodyfikuj funkcję `getop` w taki sposób, aby nie korzystała z funkcji `ungetch`. Wskazówka: użyj wewnętrznej zmiennej statycznej.

```
#include <stdio.h>
#include <ctype.h>

#define NUMBER '0' /* sygnał, że została znaleziona liczba */

int getch(void);

/* getop: pobiera następny operator lub operand (liczbę) */
int getop(char s[])
{
    int c, i;
    static int lastc = 0;

    if (lastc == 0)
        c = getch();
    else {
        c = lastc;
        lastc = 0;
    }
    while ((s[0] = c) == ' ' || c == '\t')
        c = getch();
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* nie jest liczbą */
    i = 0;
```

```
    if (isdigit(c)) /*pobierz część całkowitą*/
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /*pobierz część ułamkową*/
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        lastc = c;
    return NUMBER;
}
```

Zmodyfikowaliśmy funkcję `getop`, tak aby korzystała ze zmiennej `static`, która pamięta ostatni znak, który powinien zostać wycofany do strumienia danych wejściowych. Ponieważ nie korzystamy z funkcji `ungetch`, zapisujemy ten znak w zmiennej `lastc`.

Wywołanie funkcji `getop` prowadzi do sprawdzenia, czy `lastc` zawiera wycofany znak. Jeżeli nie, następuje wywołanie `getch` w celu pobrania nowego znaku. Jeżeli `lastc` zawiera wycofany znak, to funkcja `getop` kopiuje go do zmiennej `c` i zeruje wartość `lastc`. Pierwsza instrukcja `while` uległa pewnym zmianom. Wynikają one z tego, że `getop` musi pobierać nowy znak tylko po zakończeniu przetwarzania bieżącego znaku w `c`.

---

## Ćwiczenie 4.12 (str. 107)

Zaadaptuj koncepcję funkcji `printfd` do napisania rekurencyjnej wersji funkcji `itoa`. Innymi słowy, przekształć liczbę całkowitą na ciąg znaków, wywołując procedurę rekurencyjną.

```
#include <math.h>

/* itoa: konwertuje liczbę n na ciąg znaków s; wersja rekurencyjna */
void itoa(int n, char s[])
{
    static int i;

    if (n / 10)
        itoa(n / 10, s);
    else {
        i = 0;
        if (n < 0)
            s[i++] = '-';
    }
    s[i++] = abs(n) % 10 + '0';
    s[i] = '\0';
}
```

Funkcja `itoa` pobiera dwa argumenty: liczbę całkowitą `n` i tablicę znaków `s`. Jeżeli wynik dzielenia całkowitego `n/10` jest różny od zera, funkcja wywołuje samą siebie, przekazując jako argument `n/10`:

```
    if (n / 10)
        itoa(n / 10, s);
```

Gdy w jednym z kolejnych wywołań rekurencyjnych `n/10` ma wartość 0, mamy do czynienia z najbardziej znaczącą cyfrą `n`. Statyczna zmienna `i` jest indeksem tablicy `s`. Jeżeli liczba `n` jest ujemna, umieszczamy znak minus na pierwszej pozycji tablicy i zwiększamy `i`. Gdy `itoa` powraca z kolejnych wywołań rekurencyjnych, obliczane są kolejne cyfry, od lewej do prawej strony. Zwróćmy uwagę, że na każdym poziomie zostaje dodane `'\0'` kończące ciąg, które na kolejnym poziomie zostaje zastąpione następną cyfrą liczby. Wyjątkiem jest jedynie zakończenie ostatniego wywołania `itoa`, po którym zapisany znacznik końca ciągu pozostaje w tablicy znaków.

## Ćwiczenie 4.13 (str. 107)

Napisz rekurencyjną wersję funkcji `reverse(s)`, odwracającej „w miejscu” ciąg znaków `s`.

```
#include <string.h>

/* reverse: odwraca w miejscu ciąg s */
void reverse(char s[])
{
    void reverser(char s[], int i, int len);

    reverser(s, 0, strlen(s));
}

/* reverser: odwraca w miejscu ciąg s; algorytm rekurencyjny */
void reverser(char s[], int i, int len)
{
    int c, j;

    j = len - (i + 1);
    if (i < j) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
        reverser(s, ++i, len);
    }
}
```

Musimy zachować ten sam interfejs procedury `reverse` niezależnie od implementacji. Oznacza to, że możemy przekazać do niej tylko ciąg znaków.

Funkcja `reverse` określa długość ciągu i wywołuje funkcję `reverser`, która wykonuje właściwą operację odwrócenia ciągu `s` w miejscu.

Funkcja `reverser` pobiera trzy argumenty: `s` jest odwracanym ciągiem, `i` to indeks ciągu (od lewej), a `len` to długość ciągu (`strlen(s)`); patrz podrozdział 2.3 podręcznika K&R).

Początkowo parametr `i` ma wartość 0. `j` to indeks ciągu, wskazujący pozycję względem prawego końca tego ciągu. Wartość `j` jest obliczana jako

```
j = len - (i + 1);
```

Znaki ciągu są zamieniane miejscami, począwszy od skrajnych, aż do środka ciągu — najpierw zamieniane są `s[0]` i `s[len-1]`, potem `s[1]` i `s[len-2]` itd. Wartość indeksu `i` jest zwiększana o jeden przed każdym kolejnym wywołaniem funkcji `reverser`:

```
reverser(s, ++i, len);
```

Zamienianie znaków jest kontynuowane do momentu, gdy dwa indeksy wskazują ten sam znak (`i == j`) albo indeks liczony od lewej strony wskazuje znak na prawo od znaku wskazywanego przez indeks liczony od lewej strony (`i > j`).

Nie jest to korzystne zastosowanie rekurencji. Pewne problemy dobrze poddają się rozwiązaniom rekurencyjnym — przykładem może być funkcja `treeprint` przedstawiona w podrozdziale 6.5 podręcznika K&R. Inne lepiej rozwiązywać innymi sposobami. Do tej ostatniej kategorii należy problem odwracania ciągu.

---

## Ćwiczenie 4.14 (str. 110)

Zdefiniuj makro `swap(t,x,y)` wymieniające wartości dwóch argumentów, których typ to `t` (pomocna będzie struktura blokowa).

```
#define swap(t, x, y) { t _z; \
                        _z = y; \
                        y = x; \
                        x = _z; }
```

Używając nawiasów klamrowych, definiujemy blok. Na początku bloku możemy zadeklarować zmienne lokalne. `_z` to zmienna lokalna typu `t`, która pomaga zamienić dwa argumenty.

Makro `swap` działa poprawnie, o ile żaden z argumentów nie ma nazwy `_z`. Jeżeli tak jest,

```
swap(int, _z, x);
```

to po rozwinięciu makra uzyskujemy

```
{ int _z; _z = _z; _z = x; x = _z; }
```

i wymiana nie następuje. Przyjmujemy więc założenie, że `_z` nie będzie wykorzystywane jako nazwa zmiennej.