WYDANIE II

JĘZYK

PROGRAMOWANIE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

NAJLEPSZA KSIĄŻKA
O NAJPOPULARNIEJSZYM JĘZYKU!

Helion

Tytuł oryginału: The C Programming Language, Second Edition

Tłumaczenie: Paweł Koronkiewicz

ISBN: 978-83-8322-556-2

Authorized translation from the English language edition, entitled: The C Programming Language, Second Edition, ISBN 0131103628, by Brian W. Kernighan, Dennis M. Ritchie; published by Pearson Education, Inc, publishing as Prentice Hall; Copyright © 2002 by Prentice Hall PTR.

Polish language edition published by Helion S.A.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jansvv
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 4.
Funkcje i struktura programu

Funkcje dzielą duże zadania obliczeniowe na mniejsze oraz umożliwiają wielokrotne wykorzystywanie tego samego kodu. Właściwie napisane funkcje ukrywają szczegóły swoich mechanizmów przed innymi częściami programu, dla których są one nieistotne. Zapewnia to przejrzystość i znacznie ułatwia wprowadzanie zmian.

Język C został zaprojektowany w taki sposób, aby korzystanie z funkcji było efektywne i łatwe. Program składa się z reguły z dużej liczby małych funkcji. Duże funkcję są stosowane rzadko. Program może być zapisany w jednym lub wielu plikach. Pliki źródłowe programu mogą być kompilowane niezależnie i później jednocześnie ładowanedo pamięci razem z wcześniej skompilowanymi funkcjami bibliotek. Nie będziemy omawiać tu dokładnie tego rodzaju procedur, ponieważ różnią się one w zależności od stosowanego systemu.

Deklaracja i definicja funkcji to obszar, w którym norma ANSI wprowadziła najbardziej rzucające się w oczy zmiany w języku C. Jak widzieliśmy już w rozdziale 1., można teraz określać w deklaracji funkcji typy jej argumentów. Składnia definicji funkcji również jest zmieniona, dzięki czemu deklaracja i definicja mają taką samą postać. Umożliwia to kompilatorowi wykrycie znacznie większej liczby błędów niż wcześniej. Co więcej, właściwy sposób deklarowania argumentów zapewnia automatyczne konwersje typów.

Standard uściśla reguły dotyczące zakresu nazw. W szczególności wymaga on, aby każdy obiekt zewnętrzny miał tylko jedną definicję. Mechanizm inicjalizacji został uogólniony — w ANSI C można inicjalizować tablice i struktury automatyczne.

Preprocesor języka również został usprawniony. Jego nowe mechanizmy obejmują pełniejszy zbiór dyrektyw kompilacji warunkowej, możliwość budowania ciągów znakowych z argumentów makr oraz większą kontrolę nad procesem rozwijania makra.

4.1. Funkcje — podstawy

Na początek zaprojektujemy i napiszemy program, który wypisuje każdy wiersz danych wejściowych zawierający określony wzorzec — ciąg znaków (będzie to uproszczona wersja programu `grep` systemu UNIX). Przykładowo wyszukiwanie wzorca „ould" w zbiorze wierszy

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

spowoduje wypisanie

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

Tak postawione zadanie można podzielić w naturalny sposób na trzy części:

```
while (jest kolejny wiersz)
    if (wiersz zawiera wzorzec)
        wypisz wiersz
```

Choć jest oczywiście możliwe umieszczenie całego kodu w funkcji `main`, lepszym podejściem okazuje się wykorzystanie możliwości strukturalizowania kodu i zapisanie każdej części w odrębnej funkcji. Z trzema małymi elementami łatwiej pracować niż z jednym dużym — nieistotne szczegóły pozostają ukryte w funkcjach, a prawdopodobieństwo wystąpienia niepożądanych interakcji jest ograniczone do minimum. Co więcej, gotowe elementy mogą znaleźć zastosowanie w innych programach.

„while *jest kolejny wiersz*" to funkcja `getline`, którą napisaliśmy już w rozdziale 1. „*wypisz wiersz*" to funkcja `printf`, dostępna w standardowej bibliotece. Oznacza to, że musimy jedynie napisać procedurę określającą, czy wiersz zawiera wzorzec.

Możemy rozwiązać ten problem, pisząc funkcję `strindex(s,t)`, która zwraca pozycję (indeks) w ciągu `s`, od którego zaczyna się ciąg `t`, lub `-1`, jeżeli `s` nie zawiera `t`. Ponieważ pierwsza pozycja w tablicach języka C ma indeks 0, indeksy będą miały wartości dodatnie lub 0, a wartość ujemna, taka jak `-1`, może zostać wykorzystana do sygnalizowania nieudanego wyszukiwania. Jeżeli w przyszłości będzie potrzebny bardziej wyszukany mechanizm wyszukiwania wzorców, będzie można wymienić funkcję `strindex` na inną. Reszta kodu pozostanie bez zmian (standardowa biblioteka zawiera funkcję `strstr`, która jest podobna do `strindex`, ale zwraca wskaźnik zamiast indeksu).

Po takim przygotowaniu projektu napisanie właściwego programu jest już czynnością stosunkowo prostą. Poniżej przedstawiono całość, zarówno główny program, jak i stosowane funkcje, aby Czytelnik mógł wygodnie przeanalizować ich współdziałanie. W tej wersji wyszukiwany ciąg jest literałem (stałą), więc trudno mówić o ogólności rozwiązania. Do inicjalizowania tablic znakowych powrócimy już wkrótce, natomiast w rozdziale 5.

pokażemy, jak przekształcić wzorzec w parametr przekazywany przy uruchamianiu programu. Kod zawiera także nieco zmodyfikowaną funkcję `getline`. Porównanie jej z wersją z rozdziału 1. może dostarczyć wartościowych spostrzeżeń.

```
#include <stdio.h>
#define MAXLINE 1000      /* dopuszczalna długość wiersza */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould"; /* wzorzec do wyszukania */

/* wyszukuje wszystkie wiersze zawierające wzorzec */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: pobiera wiersz do s, zwraca długość */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

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

    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')
            return i;
    }
    return -1;
}
```

Definicja funkcji ma zawsze następującą postać:

```
typ_zwracany nazwa_funkcji(deklaracje_argumentów)
{
    deklaracje i instrukcje
}
```

Różne elementy można pomijać. Absolutne minimum to

```
dummy () {}
```

czyli funkcja, która nic nie robi i nic nie zwraca. Funkcja tego rodzaju okazuje się czasem przydatna jako tymczasowa „atrapa" w trakcie pracy nad programem. Jeżeli zwracany typ danych nie został określony, kompilator przyjmuje, że jest to `int`.

Program to po prostu zbiór definicji zmiennych i funkcji. Komunikacja między funkcjami odbywa się za pośrednictwem argumentów funkcji, wartości zwracanych przez funkcje i zmiennych zewnętrznych. Funkcje mogą być umieszczone w pliku źródłowym w dowolnej kolejności, a program może być podzielony na wiele plików źródłowych, o ile tylko każda funkcja znajduje się w całości w jednym pliku.

Instrukcja `return` reprezentuje mechanizm zwracania wartości z funkcji wywoływanej do funkcji lub środowiska wywołującego. Po słowie `return` może znajdować się dowolne wyrażenie:

```
return wyrażenie;
```

Jeżeli to konieczne, wartość wyrażenia jest przekształcana na typ zadeklarowany jako zwracany przez funkcję. Wyrażenie następujące po słowie `return` ujmuje się często w nawiasy, ale nie jest to wymagane.

Funkcja wywołująca może w każdym przypadku zignorować zwracaną wartość. Co więcej, wyrażenie po słowie `return` nie jest elementem wymaganym. Gdy zostanie pominięte, funkcja nie będzie zwracała żadnej wartości. Sterowanie zostaje przekazane do funkcji wywołującej bez zwracania wartości także po dojściu do końcowego nawiasu klamrowego. Jest to dopuszczalne, ale — gdy funkcja z jednego miejsca zwraca wartość, a z innego nie — sygnalizuje występowanie nieprawidłowości w pracy programu. W każdym przypadku, w którym funkcja nie zwraca wartości, próba jej odczytania prowadzi do uzyskania przypadkowych danych (śmieci).

Program wyszukujący ciąg zwraca z funkcji `main` informację o przebiegu jego wykonywania, którą w tym przypadku jest liczba znalezionych wierszy. Wartość ta może być wykorzystywana przez środowisko, które wywołało program.

Mechanika kompilowania i ładowania programu C, który został zapisany w wielu plikach źródłowych, różni się w zależności od systemu. Przykładowo w systemie UNIX zadanie to realizuje wspomniane w rozdziale 1. polecenie `cc`. Załóżmy, że trzy funkcje przykładowego programu są zapisane w trzech plikach, o nazwach *main.c*, *getline.c* i *strindex.c*. W takiej sytuacji polecenie

```
cc main.c getline.c strindex.c
```

kompiluje trzy wymienione pliki, umieszcza kod obiektów w plikach *main.o*, *getline.o* i *strindex.o*, a następnie ładuje je wszystkie do pliku wykonywalnego o nazwie *a.out*. W przypadku wystąpienia błędu, na przykład w *main.c*, plik może zostać skompilowany ponownie niezależnie od innych i załadowany razem z przygotowanymi wcześniej. Umożliwia to polecenie

```
cc main.c getline.o strindex.o
```

Polecenie `cc` wykorzystuje rozszerzenia *.c* i *.o* do odróżniania plików źródłowych od plików wynikowych.

Ćwiczenie 4.1. Napisz funkcję `strrindex(s,t)`, która zwraca pozycję *ostatniego* wystąpienia `t` w `s` lub `-1`, jeżeli wyszukiwany ciąg nie został znaleziony.

4.2. Zwracanie wartości innych niż int

Dotychczas przykładowe funkcje albo nie zwracały żadnej wartości (`void`), albo zwracały liczbę `int`. Co z funkcjami zwracającymi wartości innych typów? Wiele funkcji liczbowych, takich jak `sqrt`, `sin` czy `cos`, zwraca typ `double`. Inne wyspecjalizowane funkcje zwracają jeszcze inne typy. Aby zilustrować pracę z takimi funkcjami, napiszemy i wywołamy funkcję `atof(s)`, która konwertuje ciąg `s` na jego odpowiednik typu zmiennoprzecinkowego, podwójnej precyzji. Funkcja `atof` jest rozwinięciem funkcji `atoi`, której wersje zostały przedstawione w rozdziałach 2. i 3. `atof` zapewnia obsługę opcjonalnego znaku liczby oraz kropki dziesiętnej, a także sytuacji, w których nie występuje część całkowita lub część ułamkowa wartości. Przedstawiona tu wersja *nie jest* wysokiej jakości procedurą konwersji danych wejściowych. Taka funkcja zajęłaby stanowczo zbyt wiele miejsca. Dopracowaną wersję `atof` zawiera standardowa biblioteka języka — jest ona zdefiniowana w nagłówku `<stdlib.h>`.

Przede wszystkim typ zwracanej wartości, jeżeli nie jest to `int`, musi zostać określony w samej funkcji. Nazwę typu umieszcza się przed nazwą funkcji:

```
#include <ctype.h>

/* atof: konwertuje ciąg s na liczbę double */
double atof(char s[])
{
    double val, power;
    int 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;
    }
    return sign * val / power;
}
```

Drugą, równie ważną rzeczą jest to, że procedura wywołująca musi wiedzieć, że funkcja `atof` zwraca wartość inną niż `int`. Jedną z możliwości zapewnienia tego jest jawne zadeklarowanie `atof` w tej procedurze. Deklarację taką widać w programie minimalistycznego kalkulatora (nadającego się chyba tylko do podliczania wypłat z bankomatu), który sumuje pobieraną z wejścia pojedynczą kolumnę liczb. Liczby mogą zawierać znak, a po każdej jest drukowana suma pobranych już wartości:

```
#include <stdio.h>

#define MAXLINE 100

/* prymitywny kalkulator */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

Deklaracja

```
double sum, atof(char []);
```

mówi, że `sum` to zmienna typu `double`, a `atof` to funkcja, która pobiera jeden argument `char[]` i zwraca wartość `double`.

Deklaracja i definicja funkcji muszą być zgodne. Jeżeli definicja funkcji i jej wywołanie w `main` mają niespójnie określone typy, a są w tym samym pliku źródłowym, kompilator zgłosi błąd. Jednak gdy (co jest bardziej prawdopodobne) funkcja `atof` będzie kompilowana niezależnie, brak zgodności nie zostanie wykryty, a funkcja zwróci liczbę `double`, która w `main` będzie traktowana jako `int` — uzyskiwane wtedy wartości będą niemal zupełnie przypadkowe.

W świetle tego, co powiedzieliśmy o dopasowaniu deklaracji do definicji, może się to wydawać zaskakujące. Przyczyną takiej niezgodności jest zasada, że gdy brak prototypu funkcji, jej deklaracja następuje automatycznie w chwili pierwszego użycia w wyrażeniu, na przykład

```
sum += atof(line)
```

Jeżeli nazwa, która nie została wcześniej zadeklarowana, występuje w wyrażeniu, a bezpośrednio po niej jest umieszczony otwierający znak nawiasu, następuje deklaracja na podstawie kontekstu — dana nazwa jest uznawana za nazwę funkcji, która zwraca wartość `int`. Nie są natomiast przyjmowane żadne założenia dotyczące jej argumentów. Co więcej, jeżeli deklaracja funkcji nie zawiera argumentów, jak w instrukcji

```
double atof();
```

to również wstrzymuje kompilator od przyjmowania założeń dotyczących argumentów. Sprawdzanie poprawności parametrów zostaje całkowicie wyłączone. Ta szczególna interpretacja pustej listy argumentów ma umożliwić kompilowanie starszych programów w języku C przez nowsze kompilatory. Nie należy jednak stosować takiej składni w nowych programach. Jeżeli funkcja pobiera argumenty, deklarujemy je. Jeżeli nie pobiera żadnych, używamy typu `void`.

Jeśli dysponujemy (właściwie zadeklarowaną) funkcją `atof`, możemy wykorzystać ją do utworzenia prostej funkcji `atoi` (konwertującej ciąg znaków na liczbę `int`):

```
/* atoi: konwertuje ciąg s na liczbę całkowitą przy użyciu atof */
int atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```

Zwróćmy uwagę na strukturę deklaracji i instrukcję `return`. Wartość wyrażenia w wierszu

```
return wyrażenie;
```

zostaje przekształcona na typ wartości zwracanej przez funkcję przed wyjściem z tej funkcji. Wartość `atof`, typu `double`, jest konwertowana automatycznie na `int` po dojściu do tego wiersza — funkcja `atoi` ma zwracać liczbę całkowitą. Operacja taka może prowadzić do utraty części danych (części ułamkowej liczby), więc niektóre kompilatory generują po jej napotkaniu ostrzeżenie. Operacja `(int)` jest jawną informacją o tym, że konwersja typu jest zamierzona, dzięki czemu ostrzeżenie nie jest wyświetlane.

Ćwiczenie 4.2. 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.

4.3. Zmienne zewnętrzne

Program w języku C składa się ze zbioru obiektów zewnętrznych — zmiennych i funkcji. Wewnątrz funkcji definiowane są obiekty wewnętrzne, czyli jej argumenty i zmienne lokalne. Zmienne zewnętrzne są definiowane poza funkcjami, dzięki czemu mogą być dostępne nie w jednej, ale w wielu funkcjach. Same funkcje są zawsze obiektami zewnętrznymi, ponieważ język C nie dopuszcza definiowania funkcji wewnątrz funkcji. Standardowo zewnętrzne zmienne i funkcje mają tę właściwość, że wszystkie odwołania do nich, czyli takie, które używają tej samej nazwy, nawet w funkcjach kompilowanych niezależnie, pozostają odwołaniami do tego samego obiektu (norma określa tę właściwość terminem „dowiązywanie obiektów zewnętrznych", ang. *external linkage*). Pod tym względem zmienne zewnętrzne zachowują się tak jak bloki `COMMON` języka Fortran lub zmienne w najbardziej zewnętrznym bloku w języku Pascal. Wkrótce pokażemy, jak definiować zmienne i funkcje zewnętrzne, które są widoczne jedynie w obrębie pojedynczego pliku źródłowego.

Ponieważ zmienne zewnętrzne są dostępne globalnie, stanowią alternatywę dla argumentów i wartości zwracanych przez funkcje — również umożliwiają wymianę danych między funkcjami. Każda funkcja może uzyskać dostęp do zmiennej zewnętrznej przy użyciu jej nazwy, o ile tylko nazwa ta została wcześniej w pewien sposób zadeklarowana.

Jeżeli funkcje mają korzystać wspólnie z wielu różnych zmiennych, zmienne zewnętrzne są wygodniejsze i efektywniejsze niż długie listy argumentów. Jak jednak pisaliśmy w rozdziale 1., korzystanie z tej możliwości powinno wiązać się z pewną ostrożnością, może mieć bowiem zły wpływ na strukturę programu i prowadzić do kodu z nadmiernie złożoną siecią powiązań między funkcjami.

Zmienne zewnętrzne znajdują także zastosowania wynikające bezpośrednio z ich większego zakresu i dłuższego „czasu życia". Zmienne automatyczne to wewnętrzne obiekty funkcji. Powstają w chwili wejścia do funkcji i zostają zlikwidowane w chwili wyjścia z niej. Zmienne zewnętrzne są trwałe, zachowują swoją wartość pomiędzy wywołaniami różnych funkcji. Jeżeli więc dwie funkcje muszą korzystać z tych samych danych, a nie występuje sytuacja, w której jedna z nich wywołuje drugą, zapisanie wspólnych danych w zmiennych zewnętrznych jest często najwygodniejszym rozwiązaniem, pozwalającym uniknąć wprowadzania dodatkowego mechanizmu przekazywania wartości do i z każdej ze współdziałających funkcji.

Przeanalizujmy to zagadnienie na konkretnym przykładzie. Naszym zadaniem jest napisanie programu kalkulatora, który umożliwia korzystanie z operatorów +, -, * i /. Ponieważ jest to prostsze w implementacji, kalkulator będzie korzystał z odwrotnej notacji polskiej, a nie notacji infiksowej (odwrotna notacja polska jest używana przez niektóre kalkulatory kieszonkowe oraz języki programowania, na przykład Forth i PostScript).

W odwrotnej notacji polskiej wszystkie operandy poprzedzają operator. Wyrażenie infiksowe, na przykład

```
(1 – 2) * (4 + 5)
```

jest wprowadzane jako

```
1 2 - 4 5 + *
```

Nawiasy nie są potrzebne. Notacja jest jednoznaczna, o ile tylko liczba operandów każdego operatora jest stała.

Implementacja jest prosta. Każdy operand zostaje umieszczony na stosie. Po pobraniu operatora program zdejmuje ze stosu właściwą liczbę operandów (dwa w przypadku operatorów binarnych), wykonuje operację i zapisuje wynik ponownie na stosie. W powyższym przykładzie oznacza to umieszczenie na stosie liczb 1 i 2, następnie zastąpienie ich różnicą, –1. W kolejnym kroku na stos trafiają liczby 4 i 5, które zostają następnie zastąpione sumą, 9. Kolejna operacja to mnożenie, więc ze stosu zostają pobrane wartości –1 i 9, które zastępuje następnie ich iloczyn, –9. Na zakończenie, po dojściu do końca wiersza, wartość ze szczytu stosu zostaje wypisana na ekranie.

Struktura programu jest więc pętlą, która wykonuje odpowiednie operacje na pobieranych kolejno operatorach i operandach:

```
while (następny operator lub operand nie jest znakiem końca pliku)
    if (liczba)
        zapisz na stosie
    else if (operator)
        zdejmij operandy ze stosu
        wykonaj operację
        zapisz wynik na stosie
    else if (znak nowego wiersza)
        zdejmij wartość ze szczytu stosu i wypisz
    else
        błąd
```

Operacje umieszczania danych na stosie i zdejmowania z niego są banalne, ale do czasu uzupełnienia programu o wykrywanie i obsługę błędów pozostają wystarczająco złożone, aby uzasadniało to umieszczenie ich w osobnych funkcjach. Pozwoli to przede wszystkim uniknąć powtarzania kodu. Również odrębna funkcja powinna odpowiadać za pobieranie kolejnego operatora lub operandu.

Głównym założeniem projektowym jest to, gdzie konkretnie jest stos, a właściwie — które procedury mają do niego bezpośredni dostęp. Jedną z możliwości jest pozostawienie jego obsługi w `main`. Można przekazywać stos i bieżącą pozycję stosu do procedur, które pobierają i zapisują wartości. Jednak w funkcji `main` nie są potrzebne zmienne sterujące stosem. Wykonuje ona tylko operacje zapisania danych i odczytania ich. Zdecydowaliśmy więc o przechowywaniu stosu i związanych z nim informacji w zmiennych zewnętrznych, dostępnych funkcjom `push` i `pop`, ale nie `main`.

Zapisanie takiego projektu w postaci kodu nie jest trudne. Jeżeli mamy zapisać program w jednym pliku źródłowym, będzie on wyglądał tak:

```
#include ... /* wiersze include */
#define  ... /* wiersze define */
```

```
deklaracje funkcji dla main
main() { ... }

zewnętrzne zmienne dla push i pop
void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

procedury wywoływane przez getop
```

Zagadnieniem dzielenia programu na dwa pliki źródłowe lub więcej zajmiemy się już niedługo.

Funkcja `main` to pętla zawierająca rozbudowaną instrukcję `switch`, która rozgałęzia sterowanie w zależności od typu operatora lub operandu. Jest to bardziej typowy przykład jej użycia niż ten przedstawiony w podrozdziale 3.4.

```
#include <stdio.h>
#include <stdlib.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 '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %s\n", s);
                break;
            }
        }
        return 0;
    }
```

Ponieważ + i * to operatory działań przemiennych, kolejność zdejmowania operandów ze stosu nie ma znaczenia. Jednak w przypadku operatorów – i / musi istnieć rozróżnienie między wartością po lewej stronie znaku i wartością po prawej stronie znaku. W instrukcji

```
push(pop() – pop());    /* BŁĄD */
```

kolejność obliczania wartości wywołań `pop` nie jest określona. Aby zagwarantować właściwą, konieczne jest wcześniejsze pobranie pierwszej wartości do zmiennej tymczasowej. Widać to w kodzie funkcji `main`.

```
#define MAXVAL 100  /* dopuszczalna głębokość stosu wartości */

int sp = 0;         /* następna wolna pozycja stosu */
double val[MAXVAL]; /* stos */

/* push: zapisuje f na stosie */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop: zdejmuje i zwraca wartość z wierzchołka stosu */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

Zmienna jest zmienną zewnętrzną, jeżeli jest zdefiniowana poza funkcją, tak więc współużytkowane przez funkcje `pop` i `push` zmienne stosu i indeksu stosu zostają zdefiniowane poza tymi funkcjami. Jednak funkcja `main` nie odwołuje się do stosu ani jego indeksu — reprezentacja może pozostać ukryta.

Przejdźmy teraz do implementacji `getop`, funkcji, która pobiera kolejny operator lub operand. Zadanie jest proste. Pomijamy spacje i tabulatory. Jeżeli następny znak nie jest cyfrą lub kropką dziesiętną, zwracamy go. W pozostałych przypadkach pobieramy ciąg cyfr (który może zawierać kropkę dziesiętną) i zwracamy `NUMBER`, czyli wartość sygnalizującą, że pobrana została liczba.

```
#include <ctype.h>

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

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

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

Co to za funkcje `getch` i `ungetch`? Często zdarza się, że program nie może określić, czy odczytał wystarczającą ilość danych wejściowych aż do momentu, gdy odczyta ich zbyt dużo. Takim przypadkiem jest właśnie odczytywanie znaków tworzących liczbę: do czasu odczytania pierwszego znaku, który nie jest cyfrą, pobierana liczba pozostaje niekompletna. Jednak jest to moment, gdy program odczytał już o jeden znak za dużo, znak, na który nie jest przygotowany.

Problem byłby rozwiązany, gdyby istniała możliwość cofnięcia operacji odczytu ostatniego znaku danych wejściowych. Wówczas program, który odczyta o jeden znak za dużo, mógłby „oddać” ten znak do strumienia, a inne elementy programu działałyby tak,

jak gdyby znak ten nigdy nie był odczytywany. Okazuje się, że skonstruowanie takiego mechanizmu nie jest trudne, wystarczy para współpracujących ze sobą funkcji. `getch` zwraca kolejny znak danych wejściowych. `ungetch` zapamiętuje znaki zwrócone na wejście w taki sposób, aby dalsze wywołania `getch` zwracały je przed odczytaniem nowych z rzeczywistego strumienia.

Ich współpraca jest prosta. `ungetch` zapisuje wycofane znaki we wspólnym buforze — tablicy znaków. `getch` odczytuje zawartość bufora, jeżeli nie jest on pusty. W pozostałych przypadkach wywołuje po prostu funkcję `getchar`. Niezbędna jest również zmienna indeksująca, która rejestruje pozycję bieżącego znaku w buforze.

Ponieważ bufor i indeks wykorzystują dwie funkcje, `getch` i `ungetch`, a wartości tych zmiennych muszą zostać zachowane między wywołaniami, konieczne jest użycie zmiennych zewnętrznych. Obie funkcje i deklaracje zmiennych można zapisać tak:

```
#define BUFSIZE 100

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

int getch(void) /* pobiera znak (może być znakiem wcześniej wycofanym) */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

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

Standardowa biblioteka zawiera funkcję `ungetc`, która umożliwia wycofanie jednego znaku. Omówimy ją w rozdziale 7. W powyższym przykładzie użyliśmy tablicy, a nie pojedynczego znaku, aby zaprezentować bardziej ogólne podejście.

Ćwiczenie 4.3. W oparciu o schemat przedstawiony w przykładach program kalkulatora można łatwo rozbudowywać. Dodaj obsługę operatora modulo (%) i obsługę liczb ujemnych.

Ćwiczenie 4.4. 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.

Ćwiczenie 4.5. Dodaj dostęp do funkcji biblioteki, takich jak `sin`, `exp`, i `pow`. Patrz `<math.h>` w części 4. dodatku B.

Ćwiczenie 4.6. Dodaj polecenia obsługi zmiennych (łatwo jest zapewnić możliwość korzystania z dwudziestu sześciu zmiennych przy użyciu jednoliterowych nazw). Dodaj zmienną przechowującą ostatnią wypisaną wartość.

Ćwiczenie 4.7. 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`?

Ćwiczenie 4.8. Zmodyfikuj funkcje `getch` i `ungetch`, przyjąwszy założenie, że nigdy nie będzie wycofywany więcej niż jeden znak.

Ćwiczenie 4.9. 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.

Ćwiczenie 4.10. 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.

4.4. Zakres

Funkcje i zmienne zewnętrzne tworzące program w języku C nie muszą być kompilowane jednocześnie. Źródłowy tekst programu można przechowywać w wielu plikach, a wcześniej skompilowane procedury mogą być ładowane z bibliotek. Wiąże się to z kilkoma istotnymi pytaniami:

- Jak zapisywać deklaracje, aby deklarowanie zmiennych właściwie przebiegało w czasie kompilacji?
- Jaki powinien być układ deklaracji, aby wszystkie elementy zostały właściwie połączone w chwili ładowania programu?
- Jaki układ deklaracji zapewnia, że nie są one powtarzane?
- Jak inicjuje się zmienne zewnętrzne?

Omówimy te zagadnienia na przykładzie programu kalkulatora, który teraz podzielony zostanie na kilka plików. Z praktycznego punktu widzenia jest to zbyt mały program, aby faktycznie warto było go dzielić, jednak wystarczy on do zilustrowania problemów, które pojawiają się w większych projektach.

Zakres (ang. *scope*) nazwy to część programu, w której nazwę tę można stosować. Dla zmiennej automatycznej, deklarowanej na początku funkcji, zakresem jest funkcja, w której zmienna została zadeklarowana. Zmienne lokalne o tej samej nazwie, ale w różnych funkcjach nie mają ze sobą żadnego związku. To samo można powiedzieć o parametrach funkcji — są one w praktyce zmiennymi lokalnymi.

Zakres zmiennej zewnętrznej lub funkcji sięga od punktu jej zadeklarowania do końca kompilowanego pliku. Jeżeli na przykład `main`, `sp`, `val`, `push` i `pop` są zdefiniowane w jednym pliku, w kolejności przedstawionej wcześniej, czyli

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

to zmienne `sp` i `val` można stosować w funkcjach `push` i `pop`, po prostu wymieniając ich nazwę. Nie są wymagane dodatkowe deklaracje. Jednak nazwy te nie są widoczne w `main`, podobnie jak funkcje `push` i `pop`.

Z drugiej strony, jeżeli odwołania do zmiennej zewnętrznej mają wystąpić przed jej zdefiniowaniem lub zmienna ta jest definiowana w innym pliku źródłowym niż ten, w którym jest wykorzystywana, konieczne staje się użycie deklaracji `extern`.

Ważne jest, aby rozróżniać **deklarację** zmiennej zewnętrznej od jej **definicji**. Deklaracja informuje o właściwościach zmiennej (przede wszystkim jej typie). Definicja powoduje dodatkowo przydzielenie pamięci. Jeżeli wiersze

```
int sp;
double val[MAXVAL];
```

pojawiają się poza funkcjami, są to *definicje* zmiennych zewnętrznych `sp` i `val`. Powodują one przydzielenie pamięci, pełnią także funkcje deklaracji dla kodu w pozostałej części pliku źródłowego. Z drugiej strony wiersze

```
extern int sp;
extern double val[];
```

deklarują na potrzeby kodu w dalszej części pliku, że `sp` ma typ `int`, a `val` to tablica liczb `double` (której rozmiar jest określony gdzie indziej). Nie tworzą one jednak zmiennych i nie rezerwują pamięci.

We wszystkich plikach tworzących program źródłowy może wystąpić tylko jedna *definicja* zmiennej zewnętrznej. Inne pliki mogą zawierać deklaracje `extern` umożliwiające dostęp do tej zmiennej (deklaracje `extern` mogą znaleźć się także w pliku zawierającym definicję). Rozmiar tablicy musi zostać określony w definicji, a w deklaracji `extern` jest opcjonalny.

Inicjalizacja zmiennej zewnętrznej może zostać połączona tylko z jej definicją.

Choć w tym przypadku układ taki nie ma raczej uzasadnienia, funkcje `push` i `pop` mogą być zdefiniowane w jednym pliku, a zmienne `val` i `sp` w innym. Wówczas ich powiązanie zostanie zapewnione przez następujący układ definicji i deklaracji:

W pliku file1:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

W pliku file2:

```
int sp = 0;
double val[MAXVAL];
```

Ponieważ deklaracje `extern` w pliku *file1* poprzedzają definicje funkcji, zmienne można w tych funkcjach stosować. Jedna para deklaracji wystarczy dla zapewnienia dostępności zmiennych w całym pliku *file1*. Taki sam układ należałoby zastosować, gdyby definicje `sp` i `val` znajdowały się w tym samym pliku, ale po definicjach funkcji, w których są stosowane.

4.5. Pliki nagłówkowe

Rozważmy podzielenie programu kalkulatora na kilka plików źródłowych. Mogłoby to być potrzebne, gdyby poszczególne jego komponenty zostały znacznie rozbudowane. Przyjmijmy, że funkcja `main` trafia do pliku *main.c*, `push`, `pop` i ich zmienne do pliku *stack.c*, funkcja `getop` do pliku *getop.c*, a `getch` i `ungetch` — do *getch.c*. Oddzielamy te ostatnie od pozostałych, ponieważ w rzeczywistym programie byłyby częścią odrębnie kompilowanej biblioteki.

Pozostaje jeden problem do rozwiązania — definicje i deklaracje elementów wykorzystywanych w więcej niż jednym pliku. Dążymy do maksymalnej centralizacji budowanego systemu, aby każda z jego części miała tylko jedno właściwe miejsce, nieulegające zmianie w toku dalszej ewolucji kodu. Aby osiągnąć ten cel, umieszczamy wspólne elementy w **pliku nagłówkowym** (ang. *header file*, najczęściej nazywany krótko nagłówkiem), *calc.h*. Plik ten będzie włączony do kodu plików, które korzystają z jego zawartości, dyrektywą `#include`. Dyrektywę tę opiszemy dokładnie w podrozdziale 4.11. Program wygląda tak:

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

Mamy tu do czynienia z problemem wyważenia między dążeniem do tego, aby każdy plik miał dostęp wyłącznie do tych informacji, które są mu niezbędne, a prozaiczną potrzebą codziennej praktyki — praca ze zbyt dużą liczbą plików nagłówka jest uciążliwa. Do pewnych granic dobrym rozwiązaniem jest stosowanie jednego nagłówka dla całego programu zawierającego wszystko, co jest używane przez więcej niż jedną jego część. Takie rozwiązanie zastosowaliśmy w przykładzie. Większe programy wymagają bardziej rozbudowanej struktury i większej liczby nagłówków.

4.6. Zmienne statyczne

Zmienne `sp` i `val` w pliku *stack.c* oraz `buf` i `bufp` w pliku *getch.c* służą do prywatnego użytku przez funkcje znajdujące się w tym samym pliku źródłowym. Żadne inne nie powinny mieć do nich dostępu. Deklaracja `static` zastosowana w odniesieniu do zmiennej zewnętrznej lub funkcji ogranicza zakres obiektu do pozostałej części kompilowanego pliku źródłowego. Zewnętrzna deklaracja `static` jest więc metodą ukrywania nazw takich jak `buf` i `bufp` — nazw, które muszą być zewnętrzne, bo są współużytkowane przez różne funkcje, ale nie powinny być widoczne dla kodu wywołującego te funkcje.

Statyczne przechowywanie zmiennych określamy, wstawiając na początku zwykłej deklaracji słowo `static`. Jeżeli dwie procedury i dwie zmienne są kompilowane w tym samym pliku, jak w przykładzie

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

int getch(void) { ... }

void ungetch(int c) { ... }
```

to żadna inna procedura nie ma dostępu do zmiennych `buf` i `bufp`, a ich nazwy nie wchodzą w konflikt z takimi samymi nazwami w innych plikach tego samego programu. W taki sam sposób można ukryć zmienne wykorzystywane przez funkcje `push` i `pop` do obsługi stosu — deklarując `sp` i `val` jako `static`.

Zewnętrzna deklaracja `static` jest najczęściej stosowana w odniesieniu do zmiennych, ale może być użyta także w odniesieniu do funkcji. Normalnie nazwy funkcji mają charakter globalny — są widoczne w całym programie. Jeżeli jednak funkcja jest zadeklarowana jako `static`, jej nazwa nie jest widoczna poza plikiem, w którym została zadeklarowana.

Deklaracji `static` można także użyć w odniesieniu do zmiennych wewnętrznych. Wewnętrzne zmienne `static` pozostają zmiennymi lokalnymi funkcji, podobnie jak zmienne automatyczne, jednak w przeciwieństwie do zmiennych automatycznych nie przestają istnieć w chwili wyjścia z funkcji. W efekcie wewnętrzne zmienne statyczne to prywatna pamięć trwała pojedynczej funkcji.

Ćwiczenie 4.11. Zmodyfikuj funkcję `getop` w taki sposób, aby nie korzystała z funkcji `ungetch`. Wskazówka: użyj wewnętrznej zmiennej statycznej.

4.7. Zmienne rejestrowe

Deklaracja `register` zwraca uwagę kompilatora na to, że dana zmienna będzie wyjątkowo intensywnie wykorzystywana. Ideą tej deklaracji jest wskazanie, że pewne zmienne powinny zostać umieszczone w rejestrach komputera. Z zasady prowadzi to do szybszych i mniejszych programów. Kompilator może, ale nie musi, dostosować się do takiego zalecenia.

Oto przykłady deklaracji `register`:

```
register int x;
register char c;
```

Deklaracje takie można stosować wyłącznie w odniesieniu do zmiennych automatycznych i parametrów formalnych funkcji. W przypadku parametrów formalnych wygląda to tak:

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

W praktyce zmienne rejestrowe podlegają pewnym ograniczeniom wynikającym z możliwości wykorzystywanej platformy sprzętowej. Tylko kilka zmiennych w każdej funkcji można przechowywać w rejestrach i tylko wybrane typy są dopuszczalne. Nadmiar deklaracji `register` jest jednak nieszkodliwy, ponieważ w przypadku zbyt dużej liczby

tak opisanych zmiennych lub niezgodności typów słowo `register` jest ignorowane. Dodatkowo nie można pobrać adresu zmiennej rejestrowej (ten temat omówimy w rozdziale 5.), niezależnie od tego, czy została ona faktycznie umieszczona w rejestrze. Zakres ograniczeń co do typów i liczby zmiennych rejestrowych jest zależny od komputera.

4.8. Struktura blokowa

Język C nie jest językiem, w którym struktura programu opiera się na blokach, jak jest na przykład w Pascalu — nie można definiować funkcji wewnątrz funkcji. Mimo to struktura blokowa obowiązuje przy definiowaniu zmiennych. Deklaracje zmiennych (i ich inicjalizacja) mogą zostać umieszczone po nawiasie klamrowym otwierającym *dowolną* instrukcję blokową, a nie tylko po nawiasie klamrowym otwierającym blok instrukcji funkcji. Zmienne deklarowane w ten sposób przesłaniają zmienne o takich samych nazwach występujące poza blokiem, a ich „czas życia" kończy się wraz z wyjściem z bloku. Na przykład w kodzie

```
if (n > 0) {
    int i;  /* deklaracja nowej zmiennej i */

    for (i = 0; i < n; i++)
        ...
}
```

zakres zmiennej `i` to blok wykonywany przy wartości warunku „prawda". Zmienna ta nie ma żadnych powiązań ze zmiennymi o nazwie `i` poza blokiem, w którym jest zadeklarowana. Zmienna automatyczna deklarowana i inicjalizowana w bloku jest deklarowana i inicjalizowana przy każdym wejściu do tego bloku. Analogiczna zmienna `static` jest inicjalizowana przy pierwszym wejściu do bloku.

Zmienne automatyczne, w tym parametry formalne, również przesłaniają zmienne zewnętrzne i funkcje o tej samej nazwie. W układzie deklaracji

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

wewnątrz funkcji `f` wszystkie wystąpienia `x` odnoszą się do parametru (typu `double`). Poza funkcją `f` nazwa zmiennej `x` odnosi się do liczby `int`, zmiennej zewnętrznej. To samo można powiedzieć o zmiennej `y`.

Do dobrej praktyki programowania należy unikanie stosowania nazw zmiennych, które przesłaniają nazwy używane w szerszym zakresie. Jest to bowiem najkrótsza droga do pomyłek i błędów.

4.9. Inicjalizacja

O inicjalizacji wspominaliśmy już kilkukrotnie, ale zawsze pozostawała ona na marginesie innych tematów. W tym podrozdziale, po omówieniu różnych klas pamięci danych, możemy przejść do usystematyzowania reguł tego procesu.

Gdy brak jawnej inicjalizacji, zmienne zewnętrzne i statyczne mają wartość 0, a zmienne automatyczne i rejestrowe pozostają niezdefiniowane — nie zawierają użytecznej wartości.

Zmienne skalarne można inicjalizować przy ich definiowaniu — wystarczy wprowadzić po ich nazwie znak równości i wyrażenie:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milisekund/dzień */
```

Wartość inicjalizująca zmienne zewnętrzne i statyczne musi być wyrażeniem o stałej wartości. Inicjalizacja jest wykonywana jednokrotnie, jeszcze przed rozpoczęciem właściwego procesu wykonywania programu. Inicjalizacja zmiennych automatycznych i rejestrowych następuje przy każdym wejściu wykonywanego programu do funkcji lub bloku.

Wartość inicjalizująca zmienne automatyczne i rejestrowe nie musi być stała — może to być dowolne wyrażenie oparte na wartościach wcześniej zdefiniowanych, a nawet wywołaniach funkcji. Przykładowo inicjalizacja programu wyszukiwania binarnego z podrozdziału 3.3 może być zapisana następująco:

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

Nie jest wymagane pisanie:

```
int low, high, mid;
low = 0;
high = n - 1;
```

W efekcie inicjalizacja zmiennych automatycznych i rejestrowych to po prostu skrócona forma łącząca instrukcje deklaracji i przypisania. Wybór jest kwestią stylu. W książce z zasady nie łączymy deklaracji i przypisania, ponieważ wartość początkowa określona w bloku deklaracji jest łatwa do przeoczenia, a odrębne przypisanie może nastąpić w miejscu, w którym zmienna jest wykorzystywana.

Tablicę można zainicjalizować, umieszczając po deklaracji listę wartości elementów — ujętą w nawiasy klamrowe i rozdzielaną przecinkami. Aby na przykład zainicjalizować tablicę `days` długościami miesięcy, piszemy:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

Gdy rozmiar tablicy nie jest określony, kompilator określa ją, zliczając wartości początkowe elementów. W tym przypadku jest ich 12.

Jeżeli lista początkowych wartości elementów tablicy zawiera mniej elementów niż tablica, pozostałym przypisywana jest wartość 0. Dotyczy to zmiennych zewnętrznych, statycznych i automatycznych. Podanie zbyt długiej listy wartości jest błędem. Nie ma składni umożliwiającej powtarzanie wartości na liście albo inicjalizowanie elementów wewnętrznych bez podania wartości wszystkich elementów poprzedzających.

Tablice znaków są traktowane w sposób szczególny. W miejsce nawiasów klamrowych i rozdzielonej przecinkami listy można użyć ciągu:

```
char pattern = "ould";
```

Jest to skrót dłuższej, choć równoważnej konstrukcji:

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

W tym przypadku rozmiar tablicy to 5 (cztery znaki plus końcowa stała `'\0'`).

4.10. Rekurencja

Funkcje języka C mogą być wywoływane rekurencyjnie. Oznacza to, że funkcja może, bezpośrednio lub pośrednio, wywołać siebie samą. Rozważmy przykład wypisywania liczby jako ciągu znaków. Jak pisaliśmy wcześniej, cyfry są wypisywane w niewłaściwej kolejności — mniej znaczące są dostępne przed bardziej znaczącymi. Kolejność ich wypisywania musi być odwrotna.

Są dwa rozwiązania tego problemu. Pierwszym jest zapisanie cyfr w tablicy i odwrócenie kolejności zapisanych elementów. Tak zrobiliśmy w przykładowej funkcji `itoa` w podrozdziale 3.6. Alternatywę stanowi rozwiązanie rekurencyjne, w którym funkcja `printd` rozpoczyna pracę od wywołania samej siebie w celu wyświetlenia cyfr bardziej znaczących niż cyfra aktualnie przetwarzana. Dopiero po powrocie z wywołanej funkcji wypisywana jest cyfra bieżąca. Poniżej przedstawiamy taką funkcję, ponownie w wersji niezapewniającej poprawnego przetwarzania największej liczby ujemnej.

```
#include <stdio.h>

/* printd: wypisuje n jako liczbę dziesiętną */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
```

```
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Gdy funkcja wywołuje rekurencyjnie samą siebie, każde wywołanie otrzymuje nowy zestaw wszystkich zmiennych automatycznych, całkowicie niezależny od wcześniejszego. W efekcie po wywołaniu `printd(123)` pierwsza funkcja `printd` otrzymuje argument `n = 123`. Przekazuje ona `12` do drugiej funkcji `printd`, która z kolei przekazuje `1` trzeciej. Ta ostatnia wypisuje znak `1` i kończy pracę. Wówczas funkcja na drugim poziomie wypisuje znak `2` i również kończy pracę. Funkcja najwyższego poziomu wypisuje `3` i przetwarzanie początkowego wywołania `printd(123)` zostaje zakończone.

Innym ciekawym przykładem rekurencji jest algorytm sortowania quicksort, opracowany przez C.A.R. Hoare'a w 1962 roku. Z tablicy wybierany jest jeden element, a pozostałe zostają podzielone na dwa podzbiory — elementów mniejszych oraz elementów większych lub równych. Ten sam proces jest następnie powtarzany rekurencyjnie dla każdego z podzbiorów. Gdy podzbiór ma mniej niż dwa elementy, dalsze sortowanie nie jest potrzebne i proces rekurencji zostaje zakończony.

Nasza wersja programu sortującego metodą quicksort nie jest najszybsza, ale za to jest jedną z najprostszych. Podział bazuje na środkowym elemencie każdej podtablicy.

```
/* qsort: sortuje v[left]...v[right] rosnąco */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) /* nic nie rób, jeżeli tablica zawiera */
        return;        /* mniej niż dwa elementy */
    swap(v, left, (left + right)/2); /* przenieś element partycji */
    last = left;                     /* do v[0] */
    for (i = left + 1; i <= right; i++) /* partycja */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);          /* przywróć element partycji */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Przenieśliśmy operację zamieniania elementów miejscami do osobnej funkcji `swap` — jest przecież wywoływana w trzech miejscach.

```
/* swap: zamienia miejscami v[i] i v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
```

```
        v[i] = v[j];
        v[j] = temp;
    }
```

Standardowa biblioteka zawiera wersję funkcji `qsort`, która potrafi sortować obiekty dowolnego typu.

Rekurencja nie przyczynia się do oszczędzania pamięci — stos wykorzystywanych przez kolejne poziomy wywołań wartości musi być gdzieś przechowywany. Nie jest też rozwiązaniem szybszym. Jednak kod rekurencyjny jest bardziej zwarty i często łatwiejszy do napisania i intuicyjnego zrozumienia niż jego nierekurencyjny odpowiednik. Rekurencja jest szczególnie wygodna przy przetwarzaniu rekurencyjnie zdefiniowanych struktur danych, takich jak drzewa. Ciekawy przykład znajdziemy w podrozdziale 6.5.

Ćwiczenie 4.12. Zaadaptuj koncepcję funkcji `printd` do napisania rekurencyjnej wersji funkcji `itoa`. Innymi słowy, przekształć liczbę całkowitą na ciąg znaków, wywołując procedurę rekurencyjną.

Ćwiczenie 4.13. Napisz rekurencyjną wersję funkcji `reverse(s)`, odwracającej „w miejscu" ciąg znaków `s`.

4.11. Preprocesor języka C

Język C realizuje pewne mechanizmy językowe za pośrednictwem preprocesora. Jest to pierwszy krok wykonywany przed właściwym procesem kompilacji. Dwie najczęściej stosowane dyrektywy preprocesora to `#include`, włączająca do procesu kompilacji zawartość innego pliku, i `#define`, zastępująca nazwę wskazanym ciągiem znaków. W tym podrozdziale opiszemy też inne możliwości preprocesora: kompilację warunkową i makra z argumentami.

4.11.1. Wstawianie plików

Mechanizm wstawiania plików ułatwia przede wszystkim obsługę zbiorów dyrektyw `#define` i deklaracji. Każdy wiersz postaci

```
#include "nazwa_pliku"
```

lub

```
#include <nazwa_pliku>
```

zostaje zastąpiony zawartością pliku *nazwa_pliku*. Jeżeli nazwa pliku jest ujęta w cudzysłów, wyszukiwanie pliku rozpoczyna się najczęściej w katalogu programu źródłowego. Jeżeli plik nie zostanie w nim znaleziony albo gdy zamiast cudzysłowu użyto znaków < i >, wyszukiwanie przebiega zgodnie z zasadami określonymi przez implementację. Włączane dyrektywą `#include` pliki mogą także zawierać wiersze `#include`.

Na początku pliku źródłowego znajduje się najczęściej cała grupa wierszy `#include`, które włączają do programu podstawowe instrukcje `#define` i deklaracje `extern`. Mogą również zapewniać dostęp do deklaracji prototypów funkcji bibliotecznych, zapisanych w nagłówkach takich jak `<stdio.h>` (ściślej: nagłówki nie muszą być plikami; zasady dostępu do nagłówków wyznacza implementacja).

Włączanie wierszem `#include` to podstawowa metoda łączenia deklaracji w dużych programach. Gwarantuje ona, że wszystkie pliki źródłowe będą miały dostęp do tych samych definicji i deklaracji zmiennych. Eliminuje to jeden z najbardziej uciążliwych rodzajów błędów w kodzie. Naturalnie gdy włączany plik ulega zmianie, wszystkie zależne od niego pliki programu muszą być kompilowane ponownie.

4.11.2. Makra

Definicja ma postać:

```
#define nazwa tekst_zastępujący
```

Mamy tu do czynienia z najprostszą postacią makra, opartą na substytucji — wszystkie dalsze wystąpienia *nazwa* zostaną zastąpione przez *tekst_zastępujący*. Nazwa w `#define` ma taką samą postać jak nazwa zmiennej. Tekst zastępujący może być dowolny. Normalnie są to wszystkie znaki do końca wiersza, ale długa definicja może zostać podzielona na kilka kolejnych wierszy przez wstawienie znaku `\` na końcu każdego wiersza, który ma być kontynuowany. Zakres nazwy wskazanej w `#define` sięga od wiersza `#define` do końca kompilowanego pliku źródłowego. Definicja może korzystać z wcześniejszych definicji. Substytucja nie obejmuje miejsc, w których nazwa jest częścią dłuższej nazwy i fragmentów ujętych w cudzysłów. Po zdefiniowaniu na przykład nazwy `YES` substytucja nie nastąpi w `printf("YES")` ani w `YESMAN`.

Zastępujący nazwę tekst może być dowolny. Na przykład

```
#define forever for (;;) /* pętla nieskończona */
```

definiuje nowe słowo, `forever`, które będzie zastępowane pętlą nieskończoną.

Można także definiować makra z argumentami, dzięki którym tekst zastępujący jest różny w poszczególnych wywołaniach makra. Przykładem może być makro `max`:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Choć wygląda jak wywołanie funkcji, użycie `max` sprowadza się do rozwinięcia nazwy w kod wstawiany wewnątrz wiersza. Każde wystąpienie parametru formalnego (tutaj `A` i `B`) zostanie zastąpione podanym argumentem. Tak więc wiersz

```
x = max(p+q, r+s);
```

przyjmie postać

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Dopóki argumenty są spójne, makro max może pracować z dowolnym typem danych. Nie ma potrzeby definiowania różnych nazw max dla różnych typów danych, tak jakby to było w przypadku zastosowania funkcji.

Gdy przyjrzymy się sposobowi rozwijania makra max, zwrócimy uwagę, że wiąże się on z pewnymi pułapkami. Wartości wyrażeń są obliczane dwukrotnie. Staje się to istotnym problemem, gdy pojawiają się efekty uboczne wynikające ze stosowania operatorów zwiększania i zmniejszania albo operacji wejścia-wyjścia. Przykładowo

```
max(i++, j++) /* BŁĄD */
```

prowadzi do dwukrotnego zwiększenia większej wartości. Często warto zadbać o ujęcie wyrażenia w nawiasy, aby zapewnić właściwą kolejność wykonywania obliczeń. Pomyślmy, co się stanie, gdy makro

```
#define square(x) x * x /* BŁĄD */
```

zostanie wywołane w wyrażeniu square(z+1).

Makra są bardzo wartościowym narzędziem. Jednym z praktycznych przykładów ich zastosowania jest włączanie do kompilacji pliku <stdio.h>, w którym operacje getchar i putchar są często zdefiniowane jako makra. Pozwala to uniknąć obciążenia programu mechanizmem wywoływania funkcji przy odczycie pojedynczych znaków. Również funkcje w <ctype.h> są zazwyczaj implementowane jako makra.

Definicje nazw można wycofywać dyrektywą #undef. Możliwość tę wykorzystuje się często w celu uzyskania gwarancji, że dana procedura będzie funkcją, a nie makrem:

```
#undef getchar

int getchar(void) { ... }
```

Parametry formalne nie są zastępowane w ciągach znakowych otoczonych znakami cudzysłowu. Jeżeli jednak nazwę parametru poprzedza w tekście zastępującym znak #, to zostanie on zamieniony na ujęty w cudzysłów ciąg znaków, w którym parametr jest zastąpiony podanym argumentem faktycznym. W połączeniu z konkatenacją ciągów pozwala to na przykład utworzyć następujące makro wyświetlające wartości potrzebne w procesie debugowania:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Po jego wywołaniu, na przykład w instrukcji

```
dprint(x/y);
```

makro zostaje rozwinięte do postaci

```
printf("x/y" " = &g\n", x/y);
```

ciągi znakowe są automatycznie łączone, więc w efekcie uzyskujemy

```
printf("x/y = &g\n", x/y);
```

W argumencie faktycznym każdy znak " jest zastępowany przez \", a każdy znak \ przez \\, dzięki czemu wynik to poprawna stała tekstowa.

Operator preprocesora ## umożliwia konkatenowanie argumentów faktycznych w trakcie rozwijania makr. Jeżeli parametr w tekście zastępującym sąsiaduje ze znakami ##, parametr ten jest zastępowany argumentem faktycznym, znaki ## i białe znaki zostają usunięte, a wynik jest analizowany ponownie. Przykładowo makro `paste` łączy dwa argumenty:

```
#define paste(front, back) front ## back
```

więc `paste(name, 1)` tworzy nazwę `name1`.

Reguły zagnieżdżania operatora ## są dość złożone. Szczegóły można znaleźć w dodatku A.

Ćwiczenie 4.14. Zdefiniuj makro `swap(t,x,y)` wymieniające wartości dwóch argumentów, których typ to `t` (pomocna będzie struktura blokowa).

4.11.3. Warunkowe wstawianie kodu

Istnieje możliwość sterowania pracą samego preprocesora przy użyciu instrukcji warunkowych, wykonywanych w trakcie jego działania. Zapewnia to możliwość wybiórczego wstawiania kodu, w zależności od warunków, których wartości są obliczane w czasie kompilowania.

Wiersz `#if` oblicza wartość stałego wyrażenia całkowitego (które nie może zawierać operatora `sizeof`, konwersji typów i stałych `enum`). Jeżeli wyrażenie ma wartość różną od zera, wstawione zostają dalsze wiersze, aż do wiersza `#endif`, `#elif` lub `#else` (instrukcja preprocesora `#elif` odpowiada `else if`). Wyrażenie `defined(`*nazwa*`)` w wierszu `#if` ma wartość 1, jeżeli *nazwa* została wcześniej zdefiniowana, a 0 w pozostałych przypadkach.

Aby na przykład zapewnić, że zawartość pliku `hdr.h` będzie włączana do kodu tylko raz, można otoczyć ją wierszami dyrektyw warunkowych:

```
#if !defined(HDR)
#define HDR

/* tu znajduje się właściwa treść nagłówka hdr.h */

#endif
```

Pierwsza operacja włączania pliku *hdr.h* powoduje zdefiniowanie nazwy `HDR`. Przy kolejnych próbach włączenia preprocesor stwierdza, że nazwa została już zdefiniowana, i przechodzi do wiersza `#endif`. Podejście takie można stosować bardzo szeroko. Zachowanie pełnej konsekwencji pozwala w każdym nagłówku włączać do kompilacji dowolne inne wymagane nagłówki bez ciągłego śledzenia ich wzajemnych zależności.

Następująca sekwencja sprawdza tekst powiązany z nazwą SYSTEM, aby określić, która wersja nagłówka ma zostać włączona do kodu:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Wiersze #ifdef i #ifndef to wyspecjalizowane formy sprawdzenia, czy nazwa została zdefiniowana. Wcześniejszy przykład z #if można zapisać jako

```
#ifndef HDR
#define HDR

/* tu znajduje się właściwa treść nagłówka hdr.h */

#endif
```

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Drogi Czytelniku, właśnie trzymasz w rękach nowe wydanie książki zaliczanej do klasyki literatury informatycznej. Napisana przez autorów języka ANSI C w najlepszy możliwy sposób przedstawia arkana tego języka. A co można powiedzieć o samym języku? To też klasyka. To język wymagający systematyczności i skupienia, ale dający w zamian wiele możliwości i świetne wyniki. To najczęściej nauczany język programowania — jego znajomość stanowi znakomity fundament do poznania kolejnych, bardziej złożonych języków. Mimo swojego zaawansowanego wieku jest on ceniony i w wielu dziedzinach wciąż niezastąpiony.

Dzięki tej książce zdobędziesz kompletną wiedzę na temat języka C. Poznasz wszystkie dostępne typy, operatory i wyrażenia. Nauczysz się sterować wykonywaniem programu oraz wykorzystywać funkcje. Ponadto dogłębnie poznasz coś, co sprawia początkującym programistom najwięcej problemów — wskaźniki. Następnie zapoznasz się także z funkcjami wejścia i wyjścia. Dowiesz się, jak uzyskać dostęp do plików, formatować dane wyjściowe oraz obsługiwać błędy. Książka ta jest bogata w przykłady, a każdy z nich został przetestowany przez autorów. „Język ANSI C. Programowanie. Wydanie II" to niezastąpiona pozycja na półce każdego studenta informatyki, pasjonata programowania i zawodowca. Wraz z książką został wydany zeszyt zawierający rozwiązania do wszystkich zawartych w niej ćwiczeń.

- Zmienne i wyrażenia arytmetyczne w języku C
- Kompilowanie kodu
- Wykorzystanie preprocesora języka C
- Typy i operatory
- Metody sterowania wykonywaniem programu
- Wykorzystanie funkcji
- Struktura programu
- Zasada działania wskaźników
- Struktury danych
- Operacje wejścia i wyjścia
- Zastosowanie rekurencji

POZNAJ TAJNIKI JĘZYKA C!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-556-2
9 788383 225562
Cena: 77,00 zł