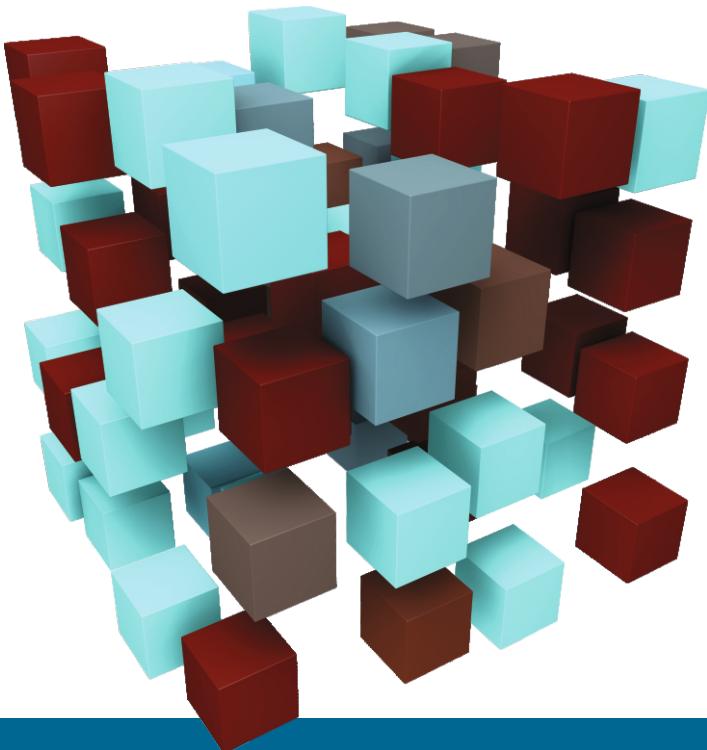


# PODSTAWY PROGRAMOWANIA W JĘZYKU C

Zadania z rozwiązaniami

ANNA ŁUPIŃSKA-DUBICKA  
MAREK TABĘDZKI



Anna Łupińska-Dubicka • Marek Tabędzki

# **PODSTAWY PROGRAMOWANIA W JĘZYKU C**

Zadania z rozwiązaniami



OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ  
BIAŁYSTOK 2022

**Recenzent:**  
dr inż. Mariusz Rybnik

**Redaktor naukowy dyscypliny matematyka:**  
prof. dr hab. Jarosław Stepaniuk

**Korekta językowa:**  
mgr Natalia Popławska

**Skład, opracowanie graficzne:**  
dr inż. Anna Łupińska-Dubicka  
dr inż. Marek Tabędzki

**Okładka:**  
Marcin Dominów  
**Zdjęcie na okładce:**  
Peggy\_Marco

<https://pixabay.com/pl/illustrations/matryca-sie%C4%87-wymiana-danych-wirus-1013612/>

© Copyright by Politechnika Białostocka, Białystok 2022

ISBN 978-83-67185-48-6 (eBook)  
DOI: 10.24427/978-83-67185-48-6



Publikacja jest udostępniona na licencji  
Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 4.0  
(CC BY-NC-ND 4.0).

Pełną treść licencji udostępniono na stronie  
[creativecommons.org/licenses/by-nc-nd/4.0/legalcode.pl](http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode.pl).  
Publikacja jest dostępna w Internecie na stronie Oficyny Wydawniczej PB.

## Spis treści

<b>1 Instrukcje warunkowe i wyboru .....</b>	<b>7</b>
1.1 Zadanie 1 (rok przestępny) .....	7
1.2 Zadanie 2 (układ współrzędnych) .....	8
1.3 Zadanie 3 (dni tygodnia) .....	10
<b>2 Instrukcje iteracyjne .....</b>	<b>13</b>
2.1 Zadanie 1 (suma) .....	13
2.2 Zadanie 2 (ile parzystych) .....	14
2.3 Zadanie 3 (zgadnij liczbę) .....	16
<b>3 Funkcje .....</b>	<b>20</b>
3.1 Zadanie 1 (liczba pierwiastków równania kwadratowego) .....	20
3.2 Zadanie 2 (równanie kwadratowe) .....	23
3.3 Zadanie 3 (pierwiastki równania kwadratowego) .....	24
<b>4 Tablice jednowymiarowe .....</b>	<b>28</b>
4.1 Zadanie 1 (średnia elementów) .....	29
4.2 Zadanie 2 (element minimalny) .....	30
4.3 Zadanie 3 (unikalne elementy) .....	33
<b>5 Łańcuchy znaków .....</b>	<b>39</b>
5.1 Zadanie 1 (wystąpienie znaku) .....	39
5.2 Zadanie 2 (palindrom) .....	41
5.3 Zadanie 3 (ostatnie wystąpienie znaku) .....	44
<b>6 Tablice dwuwymiarowe .....</b>	<b>46</b>
6.1 Zadanie 1 (podzielne przez 3 lub 4) .....	46
6.2 Zadanie 2 (wiersz o najwyższej średniej elementów) .....	48
6.3 Zadanie 3 (usuwanie pustych kolumn) .....	50

<b>7 Wskaźniki, dynamiczny przydział pamięci</b>	56
7.1 Zadanie 1 (zdublowane elementy)	56
7.2 Zadanie 2 (tablica bez powtórzeń)	64
7.3 Zadanie 3 (splice)	72
<b>8 Typ strukturalny</b>	78
8.1 Zadanie 1 (odległość punktów)	78
8.2 Zadanie 2 (tablica punktów)	80
8.3 Zadanie 3 (radar)	83
<b>9 Pliki tekstowe</b>	89
9.1 Zadanie 1 (rozdzielanie danych)	89
9.2 Zadanie 2 (najwyższa punktacja)	94
9.3 Zadanie 3 (wyniki testów)	97
<b>10 Listy jednokierunkowe</b>	101
10.1 Zadanie 1 (dodawanie)	101
10.2 Zadanie 2 (usuwanie)	109
10.3 Zadanie 3 (usuwanie powtórzeń)	112

## Wstęp

Język C jest proceduralnym językiem ogólnego przeznaczenia średniego poziomu (ponieważ obsługuje zarówno funkcje niskiego, jak i wysokiego poziomu). Został stworzony w latach 1969-1973 przez Dennis'a Ritchiego oraz Briana Kernighana. Główną ideą jego powstania było uniezależnienie systemu operacyjnego od używanych ówcześnie 16-bitowych mikrokomputerów PDP-11. Umożliwiło to przeniesienie systemu UNIX na inne platformy sprzętowe. W 1989 roku zatwierdzono jego standard jako ANSI X3.159-1989 Programming Language C (obecny standard języka opublikowany w 2018 roku to ISO/IEC 9899:2018, a kolejna wersja testowa czeka na publikację).

Język C posłużył także jako podstawa i wzór do budowy innych języków: przede wszystkim C++, ale także Javy, C#, Objective-C, czy Pythona. Ponieważ mają one więcej bibliotek, sprawdzają się znacznie lepiej w popularnych zadaniach programistycznych. Jednakże, pomimo ich powszechności i możliwości, C wciąż jest popularny. Jego główne cechy, jakimi są m.in. niskopoziomowy dostęp do pamięci, prosty zestaw słów kluczowych, sprawiają, że język C nadaje się do programowania systemowego, takiego jak system operacyjny czy rozwój kompilatorów. Kod tworzony w C jest prawie, tak szybki jak kod pisany w języku asemblerowym. W C zostały napisane i są nadal rozwijane nie tylko jądra systemów operacyjnych Windows, Linux czy MacOS oraz ich mobilne wersje, ale również oprogramowanie o wysokiej złożoności manipulacji danymi, jak animacje 3D czy bazy danych (w C zostały napisane m.in. Oracle czy też MS SQL Server). Język C należy do nielicznej grupy języków, które sprawdzają się w środowiskach produkcyjnych. Z racji swojej szybkości i wydajności spełnia zasadę im prościej, tym lepiej obowiązującą w przypadku systemów wbudowanych, stąd też jest często wykorzystywany w ich tworzeniu. Deweloperzy sięgają po język C przy projektowaniu programów obsługujących standardowe czujniki IoT z uwagi na jego arbitralny dostęp do adresów pamięci i arytmetykę wskaźnikową. Należy przyznać, że C nie jest doskonały i nie w każdej sytuacji będzie najlepszym wyborem. Nie należy również do łatwych języków. Wymaga od programisty dużo czujności i wiedzy o działaniu kodu. To, co jest jego zaletą, czyli m.in. niskopoziomowy dostęp do pamięci, stanowi również jego wadę. Jednakże, cytując twórców, „umiejętność posługiwania się językiem C wzrasta wraz z doświadczeniem jego użytkownika”.

Niniejsza publikacja nie stanowi samodzielniego podręcznika do nauki języka C. Pomyślana jest raczej jako pomoc dla osób stawiających pierwsze kroki w nauce

programowania. Przeznaczona jest dla studentów pierwszego roku *Informatyki* oraz *Informatyki i ekonometrii*. Początki nauki programowania bywają trudne. Nie wystarczy bowiem poznać instrukcje i funkcje języka programowania ani nauczyć się algorytmów. Programowanie nie polega na powtarzaniu znanych rzeczy, ale przede wszystkim na szukaniu rozwiązań. To wymaga wyrobienia w sobie umiejętności odpowiedniego myślenia o problemie. Gdy stawiamy pierwsze kroki, może przydać się pomocna dłoń, która nas przez nie poprowadzi. Tym właśnie ma być ta publikacja. Znajdziecie w niej szereg ćwiczeń wraz z rozwiązaniami. Zakres tematyczny treści pokrywa się z sylabusem przedmiotu *Podstawy programowania* wykładanego na Wydziale Informatyki Politechniki Białostockiej w pierwszym semestrze studiów na wspomnianych wyżej kierunkach. Zagadnienia prezentowane w skrypcie ułożone zostały w takiej samej kolejności, jak tematy realizowane w ramach prowadzonych przez nas zajęć. Jednak poza kodem i opisem rozwiązań przedstawiliśmy tu też krok po kroku proces prowadzącego do nich myślenia. Publikacja przeznaczona jest szczególnie dla tych, którzy mają problem z przestawieniem się na myślenie jak programista. Liczymy jednak na to, że nawet osoby sprawnie programujące, będą w stanie czegoś się z niej nauczyć i dzięki lekturze nabrać większej biegłości. Jeśli w tekście publikacji znajdziecie błąd lub zechcecie podzielić się jakąś sugestią czy pomysłem odnośnie zadań bądź rozwiązań – zachęcamy do kontaktu. Nasze adresy e-mail znajdziecie na stronie Wydziału Informatyki Politechniki Białostockiej: <https://wi.pb.edu.pl/pracownicy/lista-pracownikow/>. Życzymy miłej lektury i sukcesów w programowaniu.

Białystok, listopad 2022 r.

Anna Lopińska-Dubicka  
Marek Tabędzki

# Rozdział 1

## Instrukcje warunkowe i wyboru

### 1.1 Zadanie 1 (rok przestępny)

Napisz program wczytujący z klawiatury liczbę całkowitą reprezentującą rok, a następnie wypisujący informację o tym, czy jest to rok przestępny.

Po zadeklarowaniu zmiennej typu `int` wczytujemy jej wartość od użytkownika:

```
int rok;  
printf("Podaj rok: ");  
scanf("%d", &rok);
```

Rok jest rokiem przestępny, jeżeli dzieli się przez 4 i nie dzieli się przez 100 lub dzieli się przez 400. Skorzystamy z instrukcji warunkowej, a do sprawdzenia podzielności użyjemy operatora modulo, czyli reszty z dzielenia, który oznaczany jest symbolem `%`.

```
if(rok%4==0 && rok%100!=0 || rok%400==0)  
{  
    printf("Rok jest przestępny.\n");  
}
```

W języku C *instrukcja warunkowa* składa się ze słowa kluczowego `if`, po którym w nawiasach okrągłych znajduje się warunek. Jeżeli warunek jest prawdziwy, to wykonywane są instrukcje zawarte w nawiasach klamrowych.

W naszym przypadku użyliśmy warunku złożonego, korzystając z operatorów koniunkcji (`&&`) i alternatywy (`||`). Tak zapisany warunek odczytujemy następująco: reszta z dzielenia przez 4 równa 0 i reszta z dzielenia przez 100 różna od 0 lub reszta z dzielenia przez 400 równa zero. W przypadku koniunkcji wszystkie jej elementy muszą być prawdziwe, w przypadku alternatywy wystarczy jeden prawdziwy. Zatem zmienna `rok` jednocześnie nie może dzielić się przez 4 i przez 100. Należy pamiętać, że operator koniunkcji ma wyższy priorytet niż operator alternatywy (analogicznie jak operatory mnożenia i sumowania).

W przypadku, gdy chcemy również zareagować na niespełnienie warunku, możemy rozbudować instrukcję warunkową o blok `else` (w przeciwnym przypadku).

```
if(rok%4==0 && rok%100!=0 || rok%400==0)  
{
```

```

        printf("Rok jest przestępny.\n");
    }
else
{
    printf("Rok nie jest przestępny.\n");
}

```

Instrukcje w bloku `else` wykonują się w sytuacji, gdy warunek zawarty w `if` nie został spełniony.

Cały program wygląda zatem następująco:

```

#include <stdio.h>
int main()
{
    int rok;
    printf("Podaj rok: ");
    scanf("%d", &rok);
    if(rok%4==0 && rok%100!=0 || rok%400==0)
        printf("Rok jest przestępny.\n");
    else
        printf("Rok nie jest przestępny.\n");
    return 0;
}

```

## 1.2 Zadanie 2 (układ współrzędnych)

*Napisz program, który po wczytaniu od użytkownika a i b współczynników prostej wyświetli na ekranie numery ćwiartek układu współrzędnych, przez które ona przechodzi.*

Zaczynamy od deklaracji zmiennych oraz wczytania ich wartości:

```

float a, b;
printf("Podaj parametry prostej: ");
scanf("%f%f", &a, &b);

```

Następnie przystępujemy do rozpatrzenia możliwych wartości zmiennej `a`. Musimy uwzględnić trzy przypadki:  $a > 0$ ,  $a < 0$  oraz  $a = 0$ .

```

if(a>0)
{
}
else if (a<0)

```

```
{
}
else //a==0
{
}
```

Powyższy kod możemy odczytać w następujący sposób: *jeżeli  $a > 0 \dots$  w przeciwnym wypadku, jeżeli  $a < 0 \dots$  w przeciwnym wypadku.*... Nowy element **else if** rozbudowuje instrukcję warunkową o możliwość uwzględnienia więcej niż dwóch możliwych warunków. Liczba zastosowanych instrukcji **else if** zależy tylko od rozpatrywanego problemu – może być jedna, dwie albo więcej. W analogiczny sposób, jako zagnieżdżone instrukcje warunkowego, przeanalizujemy wartości zmiennej **b**:

```
if(a>0)
{
    if(b>0)
        printf("przechodzi przez ćwiartki: I, II, III.\n");
    else if (b<0)
        printf("przechodzi przez ćwiartki: I, III, IV.\n");
    else
        printf("przechodzi przez ćwiartki: I, III.\n");
}
```

Pełny kod programu wyglądać będzie następująco:

```
#include <stdio.h>
int main()
{
    float a, b;
    printf("Podaj parametry prostej: ");
    scanf("%f%f", &a, &b);

    printf("Prosta o równaniu %.2fx+%.2f=0 ", a, b);

    if(a>0)
    {
        if(b>0)
            printf("przechodzi przez ćwiartki: I, II, III.\n");
        else if (b<0)
            printf("przechodzi przez ćwiartki: I, III, IV.\n");
        else
            printf("przechodzi przez ćwiartki: I, III.\n");
    }
```

```

else if (a<0)
{
    if(b>0)
        printf("przechodzi przez ćwiartki: I, II, IV.\n");
    else if (b<0)
        printf("przechodzi przez ćwiartki: II, III, IV.\n");
    else
        printf("przechodzi przez ćwiartki: II, IV.\n");
}
else //a==0
{
    if(b>0)
        printf("przechodzi przez ćwiartki: I, II.\n");
    else if (b<0)
        printf("przechodzi przez ćwiartki: III, IV.\n");
    else
        printf("pokrywa się z osią OX.\n");
}
return 0;
}

```

### 1.3 Zadanie 3 (dni tygodnia)

*Napisz program, który na podstawie wartości liczbowej będącej numerem dnia tygodnia poda nam jego nazwę.*

Możemy go zrealizować, używając złożonej instrukcji warunkowej w poniższy sposób:

```

#include <stdio.h>
int main()
{
    int dzien;
    printf("Podaj numer dnia ");
    printf("(poniedziałek traktujemy jako dzień pierwszy)\n");
    scanf("%d", &dzien);
    if (dzien == 1) printf("Poniedziałek\n");
    else if (dzien == 2) printf("Wtorek\n");
    else if (dzien == 3) printf("Środa\n");
    else if (dzien == 4) printf("Czwartek\n");
    else if (dzien == 5) printf("Piątek\n");
}

```

```

    else if (dzien == 6) printf("Sobota\n");
    else if (dzien == 7) printf("Niedziela\n");
    else printf("Wartość spoza zakresu!!!\n");
    return 0;
}

```

W niektórych sytuacjach, aby ograniczyć wielokrotne stosowanie instrukcji `if–else if–else`, możemy zamiast niej użyć instrukcji `switch`. Służy ona do podejmowania decyzji wyłącznie na podstawie wartości jednej zmiennej.

W instrukcji `switch` inaczej niż w instrukcji `if` nie określamy warunku, który musi być prawdziwy, aby wykonały się pewne instrukcje. Zamiast tego określamy wyrażenie wyboru dla jednej zmiennej i określamy warianty, które mają się wykonać w zależności od tego, jaka jest wartość tego wyrażenia. Warianty oznaczamy słowem kluczowym `case`, za którym umieszczaćmy wartość, dla której instrukcje tego wariantu mają być wykonane. Dodatkowo, instrukcja `switch` umożliwia określenie wariantu domyślnego: `default`. Wariant ten wykonywany jest, jeśli dla danej wartości wyrażenia wyboru nie określono wariantu typu `case`. Jego odpowiednikiem jest `else` w instrukcji warunkowej.

```

#include <stdio.h>
int main()
{
    int dzien;
    printf("Podaj numer dnia ");
    printf("(poniedzialek traktujemy jako dzień pierwszy)\n");
    scanf("%d", &dzien);
    switch(dzien)
    {
        case 1: printf("Poniedzialek\n");
                  break;
        case 2: printf("Wtorek\n");
                  break;
        case 3: printf("Środa\n");
                  break;
        case 4: printf("Czwartek\n");
                  break;
        case 5: printf("Piątek\n");
                  break;
        case 6: printf("Sobota\n");
                  break;
        case 7: printf("Niedziela\n");
                  break;
        default:printf("Podano wartość spoza zakresu!\n");
    }
}

```

```
        break;
    }
    return 0;
}
```

## Rozdział 2

### Instrukcje iteracyjne

#### 2.1 Zadanie 1 (suma)

Napisz program, który będzie wczytywał liczby, dopóki ich suma będzie mniejsza od 100. Po zakończeniu wczytywania wyświetli, ile liczb zostało wprowadzonych przed osiągnięciem granicy 100.

Zaczynamy od zadeklarowania trzech zmiennych typu int: **liczba** (do przechowywania wprowadzonych wartości), **suma** (do obliczania sumy wprowadzonych wartości) oraz **licznik** (do zliczania liczb). Dodatkowo zerujemy zmienne **suma** i **licznik** w momencie deklaracji.

```
int liczba, suma=0, licznik=0;
```

Następnie przechodzimy do tworzenia instrukcji iteracyjnej. W tym przykładzie użyjemy pętli **while**.

```
while(suma<=100)
{}
```

Dopóki (**while**) warunek jest spełniony, to instrukcje wewnętrz pętli będą wykonywane. Gdy tylko warunek przestanie być prawdziwy – program opuści pętlę i przejdzie do wykonywania dalszych instrukcji.

W naszym przypadku warunkiem jest nieosiągnięcie wartości granicznej, stąd **suma<=100**. W momencie, gdy dodanie kolejnej liczby spowoduje przekroczenie 100, program przerwie powtarzanie iteracji. Wewnątrz pętli wczytujemy kolejną wartość i aktualizujemy zmienne suma oraz licznik.

```
while(suma<=100)
{
    printf("Podaj liczbę.\n");
    scanf("%d", &liczba);
    suma += liczba;
    licznik++;
}
```

Zostało jeszcze napisanie instrukcji poza pętlą.

```
printf("Podano %d wartości, a ich suma wynosi %d\n",
       licznik, suma);
```

Cały program będzie wyglądał zatem następująco:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int liczba, suma=0, licznik=0;
    while(suma<=100)
    {
        printf("Podaj liczbę.\n");
        scanf("%d", &liczba);
        suma += liczba;
        licznik++;
    }
    printf("Podano %d wartości, a ich suma wynosi %d\n",
           licznik, suma);
    return 0;
}
```

Po jego uruchomieniu okaże się jednak, że wypisana suma przekracza wartości 100, a licznik jest o jeden za duży. Dzieje się tak dlatego, że ostatnia wprowadzona wartość (ta, która powoduje przekroczenie 100) jest dodawana do sumy, wewnętrz pętli. Analogicznie zwiększały jest licznik. Możemy to naprawić zmniejszając wartości zmiennych suma i licznik po wyjściu z pętli.

```
suma -=liczba;
licznik--;
printf("Podano %d wartości, a ich suma wynosi %d\n",
       licznik, suma);
```

Tak zaktualizowany kod będzie wyświetlał wartości zgodnie z treścią polecenia.

## 2.2 Zadanie 2 (ile parzystych)

*Napisz program, który wczyta od użytkownika n liczb i zliczy, ile z nich jest parzystych. Wartość n podawana jest przez użytkownika na początku działania programu.*

Zaczynamy od deklaracji zmiennych i wczytania żądanych wartości. Dodatkowo zadeklarowaliśmy zmienną *i*, która będzie pełnić rolę licznika powtórzeń pętli.

```
int i, n, liczba, ile_parzystych;  
printf("Ile wartości chcesz wczytać?\n");  
scanf("%d", &n);
```

W tym zadaniu mamy do czynienia z iteracją o znanej liczbie powtórzeń. Moglibyśmy użyć pętli `while`, ale wygodniejsza w użyciu może okazać się pętla `for`. Pozwala ona na zwiększenie czytelności kodu – ustawianie zmiennej, jej inkrementacja oraz sprawdzanie warunku jest zapisywane w jednej linii.

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

Konstrukcja pętli `for` składa się z trzech elementów (wyrażeń) oddzielonych średnikiem. Każde z nich ma swoje określone miejsce i znaczenie:

1. wyrażenie pierwsze (`i=0`) to instrukcja wykonana *przed* pierwszym przebiegiem pętli – zazwyczaj jest to inicjalizacja zmiennej pracującej jako licznik przebiegów pętli,
2. wyrażenie drugie (`i<n`) jest warunkiem zakończenia pętli – podobnie jak w przypadku `while`, pętla wykonuje się dopóki jest on spełniony,
3. wyrażenie trzecie (`i++`) to instrukcja wykonywana po każdym przebiegu pętli (również po ostatnim) – zawiera instrukcje zwiększające licznik przebiegów pętli o odpowiednią wartość.

Następnie wewnętrz nawiasów klamrowych umieszczamy instrukcje wykonywane przy każdym przebiegu pętli.

```
for(i=0; i<n; i++)  
{  
    printf("Podaj liczbę #%d.\n", i+1);  
    scanf("%d", &liczba);  
    if(liczba%2==0)  
        ile_parzystych++;  
}
```

Przeanalizujmy jej działanie. Przed wejściem do pętli wykonuje się wyrażenie `i=0`. Wykonywane jest zawsze, nawet jeśli warunek przebiegu pętli w wyrażeniu drugim jest od początku fałszywy.

Kolejnym krokiem jest sprawdzenie warunku. Jeżeli jest prawdziwy, to wykonywane są instrukcje wewnętrz pętli.

Po zakończeniu wszystkich instrukcji wewnętrz pętli jako ostatnie wykonywane jest `i++` (równoważny zapisowi `i=i+1`). Zostanie ono wykonane nawet wtedy, gdy był to ostatni przebieg pętli.

Musimy jeszcze zadbać o początkową wartość zmiennej `ile_parzystych`. Powinna być ustawiona na zero. Możemy to zrobić w momencie jej deklaracji

lub w dowolnym miejscu przed wejściem do pętli. Możemy również nadać jej wartość w samej pętli **for**, w pierwszym wyrażeniu:

```
for(i=0, ile_parzystych = 0; i<n; i++)
```

W obrębie każdego wyrażenia możemy umieścić dowolną liczbę instrukcji. Istotne jest tylko to, że są one od siebie oddzielone przecinkami. Średniki są zarezerwowane do oddzielania poszczególnych trzech wyrażeń i nie może być ich więcej niż dwa.

Pelnny kod naszego programu może zatem wyglądać następująco:

```
#include <stdio.h>
int main()
{
    int i, n, liczba, ile_parzystych;
    printf("Ile wartości chcesz wczytać?\n");
    scanf("%d", &n);
    for(i=0, ile_parzystych = 0; i<n; i++)
    {
        printf("Podaj liczbę #%d.\n", i+1);
        scanf("%d", &liczba);
        if(liczba%2==0)
            ile_parzystych++;
    }
    printf("Spośród %d liczb, parzystych jest %d.\n",
           n, ile_parzystych);
    return 0;
}
```

## 2.3 Zadanie 3 (zgadnij liczbę)

*Napisz program, który pyta o pewną wylosowaną przez program liczbę tak długo, aż zostanie odgadnięta. Po każdej nieudanej próbie informuje, czy szukana liczba jest większa, czy mniejsza od podanej.*

Zaczynamy od zadeklarowania zmiennych typu **int**: **liczba** (będzie przechowywać wartość podawaną przez użytkownika) oraz **x** (wartość, którą użytkownik ma odgadnąć).

```
int liczba, x;
```

Do losowania wartości służy funkcja **rand()** znajdująca się w bibliotece **<stdlib.h>**. Zwraca ona kolejną liczbę pseudolosową z przedziału domkniętego **<0, RAND\_MAX>**.

```
liczba = rand();
```

Aby otrzymać liczbę z mniejszego przedziału, należy posłużyć się operatorem modulo (%) lub operacjami na liczbach rzeczywistych, np.

```
{liczba = rand() % 11;}
```

wylosuje wartości z przedziału <0, 10>.

Uprzednio omówione pętle while i for w pewnych przypadkach mogą nie wykonać się ani razu – czyli ani razu nie dojdzie do przebiegu całej pętli. Aby mieć pewność, że nasza pętla będzie miała co najmniej jeden przebieg, możemy zastosować pętlę do-while.

```
do
{
    printf("Podaj wartość ");
    scanf("%d", &x);
    if(x<liczba)
        printf("Szukana liczba jest większa.\n");
    else if(x>liczba)
        printf("Szukana liczba jest mniejsza.\n");
}while(x!=liczba);
```

Ponieważ warunek (`x!=liczba`) sprawdzany jest na końcu (a nie jak w przypadku pętli while czy for) przed pierwszą iteracją, mamy gwarancję, że użytkownik przynajmniej raz wprowadzi wartość. Jeśli uda mu się odgadnąć za pierwszym razem, pętla nie wykona się ponownie. Jeśli wprowadzona wartość nie będzie równa zgadywanej, program wyświetli komunikat i ponownie wykona instrukcje zawarte wewnętrz pętli.

Cały program może wyglądać następująco (w celach testowych znajduje się w nim instrukcja wypisująca wylosowaną wartość na ekran):

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int liczba, x;
    liczba = rand();
    printf("Liczba = %d\n", liczba);
    do
    {
        printf("Podaj wartość ");
        scanf("%d", &x);
        if(x<liczba)
            printf("Szukana liczba jest większa.\n");
```

```

    else if(x>liczba)
        printf("Szukana liczba jest mniejsza.\n");
    }while(x!=liczba);
    printf("Brawo!");
    return 0;
}

```

Jeśli uruchomimy go kilka razy, zauważymy, że za każdym razem losowana jest taka sama wartość. Oczywiście takie zachowanie nie jest pożądane. Aby rozwiązać ten problem, musimy skonfigurować generator liczb losowych. W tym celu trzeba posłużyć się funkcją `srand()` z biblioteki `<stdlib.h>`. Ustawia punkt startowy dla mechanizmu generowania kolejnych liczb całkowitych podanym w argumencie zarodkiem, przykładowo możemy wywołać są następujaco: `srand(1234)`. W dalszym ciągu jednakże będziemy losować takie same wartości przy kolejnych uruchomieniach programu. Aby sprawić, by za każdym uruchomieniem programu zarodek liczb pseudolosowych był inny, musimy użyć zmieniającej się wartości. Często stosowaną w tym celu techniką jest uzależnienie zarodka od momentu uruchomienia programu. Instrukcja

```
srand(time());
```

najpierw wywołuje funkcję `time`, podającą liczbę sekund, które upłyнуły od dnia 1 stycznia 1970 roku godziny 0:00:00 czasu uniwersalnego i używa jej jako zarodka generatora. Użycie funkcji `time` wymaga włączenia do kodu źródłowego pliku nagłówkowego `<time.h>`. Funkcji `srand()` nie trzeba wywoływać przed każdym losowaniem – wystarczy raz przed użyciem funkcji `rand()`.

Po tych zmianach kod naszego programu wygląda następująco:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    int liczba, x;
    srand(time());
    liczba = rand();
    printf("Liczba = %d\n", liczba);
    do
    {
        printf("Podaj wartość ");
        scanf("%d", &x);
        if(x<liczba)
            printf("Szukana liczba jest większa.\n");
        else if(x>liczba)

```

```
    printf("Szukana liczba jest mniejsza.\n");
}while(x!=liczba);
printf("Brawo!");
return 0;
}
```

# Rozdział 3

## Funkcje

### 3.1 Zadanie 1 (liczba pierwiastków równania kwadratowego)

Napisz funkcję obliczającą i zwracającą liczbę pierwiastków równania kwadratowego. Parametrami funkcji są trzy współczynniki równania. Uwzględnij przypadek równania liniowego, sprzecznego i tożsamościowego. Funkcja zwraca liczbę rozwiązań (2, 1, 0 lub -1 w przypadku, gdy równanie jest tożsamościowe).

Pisanie funkcji zaczynamy od określenia jej nagłówka:

```
int liczbaPierwiastkow(float a, float b, float c)
```

Nagłówek funkcji opisuje, jakie argumenty przyjmuje funkcja i jaką wartość zwraca (funkcja może przyjmować wiele argumentów, lecz może zwracać tylko jedną wartość). Na początku podajemy typ zwracanej wartości, w naszym wypadku `int`. Następnie określamy nazwę funkcji i w nawiasach listę parametrów (typ każdego parametru podajemy oddzielnie). Nazwa funkcji, podobnie jak nazwa zmiennej, może składać się z liter, cyfr oraz znaku podkreślenia `_`, przy czym cyfra nie może znajdować się na początku nazwy.

Po zdefiniowaniu nagłówka możemy przystąpić do pisania ciała funkcji, czyli wszystkich jej instrukcji. Umieszczać je w nawiasach klamrowych.

```
int liczbaPierwiastkow(float a, float b, float c)
{
}
```

Ciało tej funkcji może wyglądać następująco:

```
int liczbaPierwiastkow(float a, float b, float c)
{
    int ile = 0;
    if (a==0)
    {
        if (b!=0)
            ile = 1;
        else if (c!=0)
            ile = 0;
        else ile = -1;
    }
}
```

```

    else
    {
        float delta = b*b-4*a*c;
        if (delta > 0)
            ile = 2;
        else if (delta == 0)
            ile = 1;
    }
    return ile;
}

```

Pierwszą instrukcją jest deklaracja zmiennej całkowitej `ile`, do której będziemy przypisywać liczbę rozwiązań równania. Jest to zmienna lokalna, czyli niewidoczna poza funkcją. Dalej przeprowadzamy odpowiednie działania i zwracamy rezultat za pomocą instrukcji `return`.

Powyzszą funkcję można napisać również w inny sposób – korzystając z faktu, że instrukcja `return` powoduje natychmiastowe opuszczenie funkcji:

```

int liczbaPierwiastkow(float a, float b, float c)
{
    if (a==0)
    {
        if (b!=0)
            return 1;
        if (c!=0)
            return 0;
        return -1;
    }
    float delta = b*b-4*a*c;
    if (delta > 0)
        return 2;
    if (delta ==0)
        return 1;
    return 0;
}

```

Tym razem nie deklarujemy dodatkowej zmiennej. Za każdym razem, gdy możemy określić liczbę rozwiązań, wymuszamy przerwanie funkcji i zwracamy wartość. Wszystkie późniejsze instrukcje są ignorowane, podobnie jak w przypadku instrukcji `break` w instrukcjach iteracyjnych.

Program korzystający z tej funkcji może mieć postać:

```

#include <stdio.h>
int main()
{
    float a, b, c;
    printf("Podaj parametry równania kwadratowego:\n");
    scanf("%f%f%f", &a, &b, &c);
    int liczba = liczbaPierwiastkow(a, b, c);
    if (liczba>=0)
        printf("Liczba rozwiązań równania: %d\n", liczba);
    else
        printf("Równanie ma nieskończenie wiele rozwiązań.\n");
    return 0;
}

```

Funkcję wywołujemy, podając jej nazwę i przekazując wartości do funkcji `liczbaPierwiastkow(a, b, c)`. W naszym przykładzie przypisaliśmy wynik działania funkcji do dodatkowej zmiennej, żeby zróżnicować wyświetlany na ekranie komunikat.

W którym miejscu programu powinniśmy umieścić naszą funkcję? Funkcja, podobnie jak zmienna, musi zostać zadeklarowana, zanim zostanie użyta. Naturalnym zatem wydaje się umieszczenie jej przed funkcją `main`.

Czasem jednak, w przypadku większych programów z dużą liczbą funkcji, trudno jest zadbać nam o zachowanie tego warunku. Poza tym pilnowanie, która funkcja powinna być pierwsza, która druga itd., może sprawiać trudności, szczególnie jeżeli wywołują się one nawzajem. W takiej sytuacji, możemy rozdzielić deklarację funkcji od jej definicji. Deklaracja funkcji to nic innego jak jej nagłówek, zakończony średnikiem:

```
int liczbaPierwiastkow(float a, float b, float c);
```

Wystarczy zatem, że umieścimy go przed funkcją `main`, a jej pełną postać (nagłówek i ciało) ulokujemy w dowolnym miejscu programu, np. na dole. Pisząc deklarację funkcji, możemy zrezygnować z nadawania nazw poszczególnym parametrom:

```
int liczbaPierwiastkow(float, float, float);
```

Na etapie komplikacji, kiedy kompilator sprawdza poprawność składniową kodu, wystarczy, że zna on typ i nazwę funkcji oraz liczbę i typy jej parametrów. Nazwy parametrów potrzebne są nam dopiero w trakcie pisania jej ciała, czyli tam, gdzie się do nich bezpośrednio odwołujemy.

Pełny kod naszego programu może zatem wyglądać następująco:

```

#include <stdio.h>
int liczbaPierwiastkow(float, float, float);
int main()
{
    float a, b, c;
    printf("Podaj parametry równania kwadratowego:\n");
    scanf("%f%f%f", &a, &b, &c);
    int liczba = liczbaPierwiastkow(a, b, c);
    if (liczba>=0)
        printf("Liczba rozwiązań równania: %d\n", liczba);
    else
        printf("Równanie ma nieskończenie wiele rozwiązań.\n");
    return 0;
}
int liczbaPierwiastkow(float a, float b, float c)
{
//kod funkcji liczbaPierwiastkow
}

```

### 3.2 Zadanie 2 (równanie kwadratowe)

Napisz funkcję, wyświetlającą wzór równania kwadratowego (np.  $x^2 + 2x - 3 = 0$ ) dla zadanych wartości współczynników  $a$ ,  $b$  oraz  $c$ . Uwzględnij różne wartości oraz znaki współczynników (aby np. nie wyświetlać  $0x^2 + -1x + 0 = 0$ ).

Ta funkcja nic nie zwraca, jej zadaniem jest wyświetlenie wyniku na ekranie. W takim wypadku jako typ zwracany należy wpisać słowo kluczowe `void` oznaczające właśnie *nic* po angielsku. Słowo kluczowe `void` informuje kompilator (jak również i programistę), że funkcja nie zwraca żadnej wartości.

```

void wyswietlRownanie(float a, float b, float c)
{
    if(a!=0)
        printf("%.2fx^2", a);
    if(b>0)
        printf("+%.2fx ", b);
    else if (b<0)
        printf("%.2fx", b);
    if (c>0)
        printf("+%.2f", c);
    else if (c<0)

```

```
    printf("%.2f", c);
    printf(" = 0\n");
}
```

Wywołanie tej funkcji w programie może wyglądać następująco:

```
int main()
{
    float a, b, c;
    printf("Podaj parametry równania kwadratowego:\n");
    scanf("%f%f%f", &a, &b, &c);

    wyswietlRownanie(a, b, c);

    return 0;
}
```

Zwrót uwagę, w jaki sposób wywołujemy funkcje typu `void`. Nie mogą być one użyte jako element wyrażenia, ponieważ nie zwracają żadnej wartości.

### 3.3 Zadanie 3 (pierwiastki równania kwadratowego)

*Napisz funkcję, która będzie obliczała i zwracała pierwiastki równania kwadratowego, zwracając jednocześnie liczbę możliwych do obliczenia rozwiązań (2, 1, 0 lub -1 w przypadku, gdy równanie jest tożsamościowe).*

Analizując opis funkcji, dochodzimy do wniosku, że nasza funkcja musi zwrócić trzy informacje: liczbę rozwiązań i wartości pierwiastków. Jednocześnie wiemy, że funkcja może zwracać tylko jedną wartość. W takim wypadku samo skorzystanie z instrukcji `return` nie wystarczy. Musimy skorzystać z innej możliwości wynoszenia wyniku poza funkcję – przez parametry.

We wcześniejszych funkcjach parametry były przekazywane przez wartość – co oznacza, że po wywołaniu funkcji tworzone były lokalne kopie zmiennych skojarzonych z jej argumentami. W funkcji widoczne są one pod postacią parametrów funkcji. Parametry mogą być traktowane jak zmienne lokalne, którym przypisano początkową wartość. Po zakończeniu działania funkcji wszystkie zmienne powiązane z parametrami przekazywanymi do funkcji przestają istnieć. Po wyjściu z funkcji znów odwołujemy się do oryginalnej zmiennej, która nie została zmodyfikowana.

Przekazywanie argumentów do funkcji przez wskaźnik polega na tym, że

do funkcji przesyłane są *adresy* zmiennych będących jej argumentami. Wszystkie operacje wykonywane w funkcji na takich argumentach będą odnosiły się do zmiennych z funkcji wywołującej.

Używaliśmy już tego sposobu, korzystając z funkcji `scanf`:

```
float a, b, c;  
printf("Podaj parametry równania kwadratowego:\n");  
scanf("%f%f%f", &a, &b, &c);
```

Argumentami funkcji `scanf` są właśnie adresy zmiennych, do których wpisywane są wczytane z klawiatury wartości.

Nagłówek naszej funkcji szukającej pierwiastków może wyglądać następująco:

```
int pierwiastkiRownania(float a, float b, float c,  
                           float *x1, float *x2)
```

Pojawiły się dwa dodatkowe parametry: `int *x1` oraz `int *x2`. Symbol gwiazdki `*` umieszczony przed nazwą parametru oznacza, że ten parametr specjalnym rodzajem zmiennej (wskaźnikiem). *Wskaźnik*, jak sama nazwa sugeruje służy do wskazywania (czyli do pokazywania) na zmienne dowolnego typu. Każdy wskaźnik posiada określony typ. Na tym etapie przyjmiemy, że wskaźnik może pokazywać na zmienne tylko takiego typu, jakiego został zadeklarowany. W naszej funkcji obydwa wskaźniki przechowują adresy zmiennych typu `int`. Dzięki temu do funkcji trafi wartość adresu pamięci, do którego będziemy mogli się odwołać i zmodyfikować zapisaną pod nim wartość.

Jak poprzez wskaźnik do zmiennej można „dostać” się do samej zmiennej? Robi się to poprzez operator dereferencji, zwany też operatorem wyłuskania wartości, oznaczany przez gwiazdkę `*`.

```
(*x1) = (*x2) = 0;
```

Powyższa linia ustawia wartość 0 pod adresami pamięci, na które wskazują wskaźniki `x1` oraz `x2`.

Zauważ, że mimo iż gwiazdka pojawia się zarówno w momencie deklaracji wskaźnika (w parametrach funkcji), jak i miejscu wydobycia wartości zmiennej, na którą wskaźnik wskazuje, to z operatorem wyłuskania mamy do czynienia tylko w drugim przypadku. Po prostu niektóre symbole wykorzystano do oznaczania różnych operacji.

Nasza funkcja może wyglądać następująco. Znaczna część instrukcji jest taka sama jak w zadaniu wcześniejszym, dodaliśmy tylko wyznaczanie pierwiastków równania w poszczególnych miejscach kodu i wpisywanie ich wartości do

odpowiednich zmiennych. W przypadku, gdy równanie ma jedno rozwiązańe, obydwa parametry otrzymują tę samą wartość, ale przy wywołaniu funkcji będziemy brać pod uwagę tylko jeden z nich.

Aby wyliczyć wartość rozwiązań równania, musimy wyznaczyć pierwiastek z delty. Funkcja pierwiastkująca w C nazywa się `sqrt` i znajduje się w bibliotece `<math.h>`. Jej parametrem jest liczba rzeczywista. Należy pamiętać, że funkcja nie sprawdza poprawności danych – przed wywołaniem funkcji powinniśmy być pewni, że w przekazywanym parametrze znajduje się liczba nieujemna (bo dla takich jest określone działanie pierwiastkowania). Funkcja zwraca wyznaczony pierwiastek ze swojego parametru.

```
int pierwiastkiRownania(float a, float b, float c,
                         float *x1, float *x2)
{
    (*x1) = (*x2) = 0;
    if (a==0)
    {
        if (b!=0)
        {
            (*x1) = (*x2) = -c/b;
            return 1;
        }
        if (c!=0)
            return 0;
        return -1;
    }

    float delta = b*b-4*a*c;
    if (delta > 0)
    {
        (*x1) = (-b-sqrt(delta))/(2*a);
        (*x2) = (-b+sqrt(delta))/(2*a);
        return 2;
    }

    if (delta == 0)
    {
        (*x1) = (*x2) = (-b+sqrt(delta))/(2*a);
        return 1;
    }
}
```

```
        return 0;  
}
```

Wywołanie powyższej funkcji w programie głównym ma następującą postać:

```
#include <stdio.h>  
#include <math.h>  
int main()  
{  
    float a, b, c;  
    printf("Podaj parametry równania kwadratowego:\n");  
    scanf("%f%f%f", &a, &b, &c);  
    float x1, x2;  
    int liczba = pierwiastkiRownania(a, b, c, &x1, &x2);  
    if (liczba==2)  
        printf("Równanie ma dwa rozwiązania: %.2f i %.2f.\n",  
               x1, x2);  
    else if (liczba==1)  
        printf("Równanie ma jedno rozwiązanie: %.2f.\n", x1);  
    else if (liczba==0)  
        printf("Równanie nie ma rozwiązania.\n");  
    else  
        printf("Równanie ma nieskończenie wiele rozwiązań.\n");  
  
    return 0;  
}
```

## Rozdział 4

### Tablice jednowymiarowe

W dotychczasowych programach używaliśmy niewielkiej liczby zmiennych. Wyobraźmy sobie jednak, co by było, gdybyśmy chcieli, by użytkownik z jakiegoś powodu musiał podać 100 liczb, a nasz program miałby je wypisać na ekranie i zsumować. Czy program sam w sobie byłby skomplikowany? Oczywiście, że nie. Ale niewygodnie by się go pisało. Po pierwsze, żeby przechować wartości potrzebowalibyśmy 100 zmiennych (przykładowo  $x_1$ ,  $x_2$ , ...,  $x_{100}$ ). Następnie każdą zmienną trzeba wczytać – daje nam to 100 wywołań funkcji `scanf`, wypisać – kolejne 100 wywołań funkcji `printf`. Na koniec musimy jeszcze policzyć sumę, czyli wymienić wszystkie zmienne po kolei.

Tablice służą do organizacji danych tego samego typu. Jest to ciąg wartości tego samego typu, np. dziesięć znaków lub pięćdziesiąt liczb rzeczywistych przechowywanych w pamięci jedna obok drugiej. Tablica ma swoją nazwę oraz typ, a dostęp do poszczególnych jej elementów jest możliwy za pomocą indeksu, czyli liczby naturalnej wskazującej pozycję elementu w tablicy.

Deklaracja tablicy ma następującą postać: `nazwa_typu nazwa_tablicy[rozmiar];` gdzie:

- nazwa\_typu – przechowywany typ danych, np. `double`, `int`, `char`,
- nazwa\_tablicy – zasady określające nazwę tablicy są takie same, jak przy nazwach zmiennych,
- rozmiar – liczba elementów, które chcemy przechowywać w tablicy (ta wartość musi być liczbą naturalną).

Tablicę możemy wypełnić wartościami początkowymi w momencie deklaracji, podobnie jak w przypadku zmiennych:

```
int liczby[5] = {1, 2, 3, 4, 5};  
int liczby2[5] = {1, 2, 3};  
int liczby3[5] = {};
```

Tablica `liczby` zostanie wypełniona wartościami od 1 do 5, tablica `liczby2` wartościami 1, 2, 3, 0, 0 – kompilator automatycznie uzupełni brakujące wartości liczbą 0. W analogiczny sposób zostanie wypełniona tablica `liczby3` – samymi zerami. Takie wpisywanie wartości do tablicy możliwe jest tylko w momencie deklaracji. Jeżeli będziemy chcieli zmienić zawartość tablicy w dalszej części kodu, to musimy odwoływać się do każdego elementu oddzielnie.

Do elementów tablicy odwołujemy się, podając indeks elementu. Ważne jest, by pamiętać, że pierwszy element w tablicy ma indeks 0, a ostatni **rozmiar-1**.

```
liczby[0] = 2;  
liczby[2] = 4;
```

Powyższe instrukcje zmienią zawartość tablicy liczby na wartości: 2, 2, 4, 4, 5.

## 4.1 Zadanie 1 (średnia elementów)

Napisz program tworzący 100-elementową tablicę liczb typu int i wypełnij ją liczbami losowymi z przedziału [0, 1000] oraz wypisz zawartość tablicy na ekranie. Następnie oblicz i wypisz średnią arytmetyczną elementów tablicy.

Program zaczynamy od deklaracji tablicy.

```
int tab[100];
```

Zgodnie z wcześniejszymi informacjami nasza tablica ma nazwę (**tab**), typ (**int**) oraz określony rozmiar (100).

Ponieważ w naszym programie wykonujemy te same instrukcje (wczytanie, wyświetlenie, sumowanie), dla każdego elementu tablicy możemy użyć pętli.

```
for (i=0; i<100; i++) tab[i] = rand()%1000+1;
```

W pętli **for** wartość licznika **i** na początek ustawiana jest na 0, a zakończy się w momencie, gdy osiągnie wartość 100. Gwarantuje to nam nieprzekroczenie rozmiaru tablicy (indeks ostatniego elementu jest równy 99). W pierwszej iteracji wylosowana wartość zostanie przypisana do pierwszego elementu tablicy, czyli **tab[0]**. Po każdej iteracji wartość licznika zwiększy się o 1, czyli druga iteracja wypełni **tab[1]**, trzecia – **tab[2]**, itd.

W analogiczny sposób wyświetlimy zawartość tablicy na ekranie:

```
for(i=0; i<100; i++) printf("%5d", tab [i]);
```

i wyznaczymy sumę elementów:

```
for(i=0; i<100; i++) srednia += tab [i];
```

Zmienna **srednia** jest typu **float**, ponieważ średnia elementów całkowitych może być wartością rzeczywistą.

Cały program może wyglądać następująco:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>
```

```

int main()
{
    int tablica[100], i;

    srand(time(0));
    for (i=0; i<100; i++)
        tab[i] = rand()%1000+1;

    for(i=0; i<100; i++)
        printf("%4d", tab [i]);

    float srednia = 0;
    for(i=0; i<100; i++)
        srednia += tab [i];
    srednia /= 100;

    printf("\nŚrednia arytmetyczna elementów wynosi %.2f\n", srednia);

    return 0;
}

```

## 4.2 Zadanie 2 (element minimalny)

*Napisz program, który po pobraniu od użytkownika rozmiaru tablicy wypełni ją wartościami wczytanymi z klawiatury. Następnie wyświetli na ekranie indeks najmniejszego elementu tablicy. W przypadku, gdy wartość minimalna występuje w tablicy więcej niż jeden raz, na ekranie powinien pojawić się indeks jej pierwszego wystąpienia.*

Nasz program zaczynamy od wczytania rozmiaru od użytkownika:

```

int rozmiar;
printf("Podaj rozmiar tablicy:");
scanf("%d", &rozmiar);

```

Zmienna przechowująca rozmiar musi być typu całkowitego. Następnie musimy upewnić się, że podana wartość na sens – liczba elementów w tablicy nie może być mniejsza od jedynki. Moglibyśmy użyć instrukcji warunkowej, ale w ten sposób sprawdzilibyśmy podaną wartość tylko raz. Poniższa pętla gwarantuje nam pobieranie rozmiaru do momentu, aż będzie on prawidłowy:

```

while(rozmiar <= 0)
{
    printf("Podaj poprawną wartość!");
    scanf("%d", &rozmiar);
}

```

Mając wczytany rozmiar, możemy zadeklarować tablicę.

```
int tablica[rozmiar];
```

Dlaczego dopiero teraz? W momencie wykonywania tej instrukcji w pamięci rezerwowane jest miejsce na tyle elementów, ile podamy w nawiasach kwadratowych, czyli odczytywana jest wartość zmiennej `rozmiar`. Pamiętasz, że zmienne w momencie deklaracji mają już pewną wartość – to może być 0, -10 albo 100. Jeżeli sami nie zadbane o prawidłową wartość zmiennej `rozmiar`, to program będzie próbował stworzyć tablicę w oparciu o tę początkową, losową wartość. Może to się zakończyć błędem wykonywania programu – zmienna `rozmiar` będzie miała ujemną lub zerową wartość. Druga opcja jest bardziej trudniejsza do wykrycia – zmienna `rozmiar` będzie miała „sensowną” wartość, np. 15. W takim wypadku istnieją dwie możliwości – albo nasz program będzie działał poprawnie (bo podany przez nas rozmiar jest mniejszy niż początkowa losowa wartość i po prostu stworzymy większą tablicę, niż nam potrzeba), albo będzie się zachowywał „dziwnie” podczas wykonywania późniejszych instrukcji. Taki błąd może być trudny do wykrycia.

Do tego tematu jeszcze wróćmy w późniejszych rozdziałach. Zapamiętaj, że tablicę deklarujemy dopiero po podaniu rozmiaru.

Wypełnienie tablicy wartościami z klawiatury możemy zrealizować np. za pomocą poniższej pętli:

```

for (i=0; i<rozmiar; i++)
{
    printf("Podaj wartość: ");
    scanf("%d", &tablica[i]);
}

```

Po wczytaniu elementów przystępujemy do szukania wartości minimalnej oraz jej indeksu.

Potrzebujemy w tym celu dwóch dodatkowych zmiennych:

```
int minimum = tablica[0], indeks=0;
```

Ustawiamy wartość początkową zmiennej `minimum` na pierwszy element tablicy, a zmiennej `indeks` na wartość 0. Dlaczego? Przyjrzyjmy się następującemu fragmentowi kodu:

```

for(i=1; i<rozmiar; i++)
{
    if (tablica[i]<minimum)
    {
        minimum = tablica[i];
        indeks = i;
    }
}

```

Algorytm poszukiwania wartości minimalnej polega na przeszukaniu całego zbioru element po elemencie. W każdej iteracji porównujemy dotychczasowe tymczasowe minimum z kolejną wartością ze zbioru (`tablica[i]`). Jeżeli bieżąca wartość jest mniejsza niż dotychczasowe minimum, to za nowy element minimalny przyjmujemy porównywany element zbioru:

```
minimum = tablica[i];
```

Gdy przejdziemy cały zbiór, w tymczasowym minimum znajdzie się element minimalny z całego zbioru. Dlatego ważne jest, by początkowa wartość minimum należała do zbioru elementów, wśród których szukamy wartości minimalnej. Wyobraźmy sobie sytuację, w której wartością początkową minimum jest 0, a wszystkie elementy tablicy są liczbami większymi od 0. W takim wypadku, żadna z porównywanych wartości nie będzie mniejsza od początkowej wartości elementu minimalnego. Oczywiście możemy ustawić dowolną wartość ze zbioru, ale pierwszy element tablicy jest najprościej pobrać. Dodatkowo możemy zmniejszyć liczbę przebiegów pętli i zacząć ją od indeksu nr 1. Początkowa wartość zmiennej `indeks` jest konsekwencją początkowej wartości zmiennej `minimum`.

Cały program wygląda zatem następująco:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int rozmiar;
    printf("Podaj rozmiar tablicy:");
    scanf("%d", &rozmiar);
    while(rozmiar <= 0)
    {
        printf("Podaj poprawną wartość!");
        scanf("%d", &rozmiar);
    }
}

```

```

int tablica[rozmiar], i;
for (i=0; i<rozmiar; i++)
{
    printf("Podaj wartość: ");
    scanf("%d", &tablica[i]);
}

int minimum = tablica[0], indeks=0;
for(i=1; i<rozmiar; i++)
{
    if (tablica[i]<minimum)
    {
        minimum = tablica[i];
        indeks = i;
    }
}

printf("\nElement minimalny %d", minimum);
printf("znajduje się pod indeksem %d\n", indeks);

return 0;
}

```

### 4.3 Zadanie 3 (unikalne elementy)

*Napisz program, który wypełni 25-elementową tablicę całkowitymi wartościami losowymi z przedziału  $-25 \dots 25$ . Elementy tablicy nie mogą się powtarzać, czyli program sprawdza, czy nowa wylosowana wartość nie wystąpiła już wcześniej. Jeśli tak, ponawia losowanie elementu. Wypełnioną tablicę należy wypisać na ekranie. Następnie program sortuje tablicę w porządku niemalejącym i ponownie wyświetla na ekranie.*

Program zaczynamy od zadeklarowania tablicy i wylosowania jej elementów:

```

int tablica[25], n=25, i;
srand(time(0));
for (i=0; i<n; i++)
    tablica[i] = rand()%51-25;

```

Wprowadźmy dla wygody dodatkową zmienną przechowującą rozmiar tablicy ( $n=25$ ). Oczywiście powyższy kod wypełni tablicę nie zwracając jeszcze

uwagi na powtórzenia. W którym miejscu należy to sprawdzać? Po wypełnieniu całej tablicy jest już za późno. Musimy to robić na bieżąco, zanim umieścimy wylosowaną wartość w tablicy.

Dodajmy do naszego kodu trzy zmienne:

```
int liczba, j;
bool wystepuje;
```

Zatrzymajmy się na chwilę przy deklaracji zmiennej `wystepuje`. Będzie nam ona służyła do oznaczenia, czy wylosowana wartość jest powtórzeniem, czy nie. Moglibyśmy do tego celu użyć typu całkowitego i ustawiać jej wartość na 0 lub 1. Możemy również użyć kolejnego typu danych, jakim jest typ boolowski. Zmienna tego typu przyjmuje dwie wartości: `true` lub `false`. Korzystanie z tego typu możliwe jest po dołączeniu biblioteki `<stdbool.h>`.

Zastanówmy się, w jaki sposób sprawdzić, czy nowo wylosowana liczba nie pojawiła się już wcześniej. Założymy, że losujemy pierwszą wartość. W takim wypadku nie mamy nic do sprawdzenia. Przy losowaniu drugiej wartości musimy zweryfikować, czy nie została już ona wstawiona pod indeksem 0 tablicy, przy trzeciej – sprawdzamy indeksy 0 i 1 itd. Uogólniając, wstawienie  $i$ -tej liczby wymaga od nas sprawdzenia indeksów od 0 do  $i - 1$ . Przed każdym losowaniem zakładamy, że nowa wartość nie wystąpiła do tej pory i ustawiamy zmienną `wystepuje` na `false`. Pętla `for` sprawdza wcześniejsze elementy tablicy i w przypadku, gdy pod  $j$ -tym indeksem znajduje się wylosowana liczba ustawia zmienną `wystepuje` na `true` i przerwuje działanie pętli.

```
wystepuje = false;
liczba = rand()%51-25;
for(j=0; j<i; j++)
{
    if (liczba == tablica[j])
    {
        wystepuje = true;
        break;
    }
}
```

Pętla wypełniająca całą tablicę może wyglądać zatem następująco:

```
for (i=0; i<n; i++)
{
    do
    {
        wystepuje = false;
        liczba = rand()%51-25;
        for(j=0; j<i; j++)
            if (liczba == tablica[j])
```

```

        wystepuje = true;
}while(wystepuje);

tablica[i] = liczba;
}

```

Zewnętrzna pętla `for` wykona się tyle razy, ile elementów liczy tablica. W każdej iteracji, losowanie i sprawdzanie zostały dodatkowo umieszczone w pętli `do-while`. Zapewni to losowanie wartości  $i$ -tego elementu tak długo, aż będzie on unikalny. Nową wartość wpisujemy do tablicy dopiero po opuszczeniu pętli `do-while`. Poniższy kod wyświetli zawartość tablicy na ekranie:

```

for(i=0; i<n; i++)
    printf("%4d", tablica[i]);

```

Kolejną częścią naszego zadania jest posortowanie tablicy. Użyjemy do tego celu algorytmu sortowania bąbelkowego. Zasada jego działania opiera się na porównywaniu dwóch sąsiadujących elementów tablicy oraz zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operacje te wykonujemy tak długo, aż cała tablica zostanie posortowana.

```

for(j = 0; j < n - 1; j++)
    for(i = 0; i < n - j; i++)
        if(tablica[i] > tablica[i + 1])
        {
            int tmp = tablica[i];
            tablica[i] = tablica[i+1];
            tablica[i+1] = tmp;
        }

```

Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna kontrolowana przez zmienną  $j$  wykona się  $n - 1$  razy (bo tyle mamy par elementów). Wewnętrznej jej umieszczona jest druga pętla, sterowana przez zmienną  $i$ . Liczba jej powtórzeń zależy od numeru iteracji pętli zewnętrznej – wykonuje się ona  $n - i$  razy. Dlaczego? Po pierwszej iteracji pętli zewnętrznej największy element tablicy zostanie umieszczony na końcu tablicy. W drugiej iteracji nie musimy już zatem sprawdzać ostatniego indeksu, ponieważ znajduje się tam właściwa wartość. W trzeciej iteracji możemy pominąć ostatni i przedostatni indeks, itd.

Pętla wewnętrzna sprawdza, czy sąsiadujące elementy zachowują porządek rosnący. Jeżeli nie, to są one zamieniane wartościami.

Cały kod tego zadania może wyglądać następująco:

```

#include <stdio.h>
#include <stdlib.h>

```

```

#include <time.h>
#include <stdbool.h>
int main()
{
    int tablica[25], n=25, i;
    int liczba, j;
    bool wystepuje;
    srand(time(0));
    for (i=0; i<n; i++)
    {
        do
        {
            wystepuje = false;
            liczba = rand()%51-25;
            for(j=0; j<i; j++)
                if (liczba == tablica[j])
                {
                    wystepuje = true;
                    break;
                }
        }while(wystepuje);
        tablica[i] = liczba;
    }
    for(i=0; i<n; i++)
        printf("%4d", tablica[i]);
    for(j = 0; j < n-1; j++)
        for(i = 0; i < n-j; i++)
            if(tablica[i] > tablica[i + 1])
            {
                int tmp = tablica[i];
                tablica[i] = tablica[i+1];
                tablica[i+1] = tmp;
            }
    printf("\n\n");
    for(i=0; i<n; i++)
        printf("%4d", tablica[i]);
    return 0;
}

```

Zwróćmy jednak uwagę, że pewne jego fragmenty powtarzają się – wyświetlanie tablicy na ekranie. Moglibyśmy tego uniknąć, stosując funkcje.

```

void wyswietl_tablice(int n, int tab[])
{
    int i;
    for(i=0; i<n; i++)
        printf("%4d", tab[i]);
    printf("\n");
}

```

Funkcja `wyswietl_tablice` otrzymuje dwa parametry: rozmiar (`int n`) oraz tablicę (`int tab[]`). Nawiąsy kwadratowe umieszczone przy `tab` informują, że przekazujemy tablicę, a dokładniej adres jej początku w pamięci. Ciało funkcji to pętla przechodząca po całej tablicy (do tego właśnie potrzebny jest rozmiar przekazany jako parametr) i wyświetlająca wartość na ekranie.

Powyzsza funkcja mogłaby mieć nagłówek napisany również następująco:

```
void wyswietl_tablice(int n, int *tab);
```

W tym wypadku dokładniej widać, czym jest drugi parametr – wskaźnikiem. Tablice, jako złożone typy danych, nie są przekazywane do funkcji przez wartość, czyli nie jest tworzona ich kopia lokalna. Do funkcji trafia adres, pod którym w pamięci znajduje się pierwszy element tablicy. Konsekwencją tego jest to, że jakakolwiek zmiana wykonana na elementach tablicy wewnętrz funkcji jest trwała i będzie widoczna również po zakończeniu funkcji. Wywołanie funkcji będzie wyglądać następująco:

```
wyswietl_tablice(n, tablica);
```

Przekazując tablicę, wpisujemy tylko jej nazwę bez nawiasu kwadratowego.

Kod po powyższych modyfikacjach wygląda tak:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
void wyswietl_tablice(int n, int *tab)
int main()
{
    int tablica[25], n=25, i;
    int liczba, j;
    bool wystepuje;
    srand(time(0));
    for (i=0; i<n; i++)
    {
        do

```

```

{
    wystepuje = false;
    liczba = rand()%51-25;
    for(j=0; j<i; j++)
        if (liczba == tablica[j])
    {
        wystepuje = true;
        break;
    }
}while(wystepuje);
tablica[i] = liczba;
}
wyswietl_tablice(n, tab);
for(j = 0; j < n-1; j++)
    for(i = 0; i < n-j; i++)
        if(tablica[i] > tablica[i + 1])
    {
        int tmp = tablica[i];
        tablica[i] = tablica[i+1];
        tablica[i+1] = tmp;
    }
printf("\n\n");
wyswietl_tablice(n, tab);
return 0;
}
void wyswietl_tablice(int n, int *tab)
{
    int i;
    for(i=0; i<n; i++)
        printf("%4d", tab[i]);
    printf("\n");
}

```

# Rozdział 5

## Łańcuchy znaków

Napis (inne nazwy to tekst, łańcuch, ang. *string*) jest to grupa znaków traktowanych jako całość. Napis może zawierać litery, cyfry, znaki specjalne. W języku C nie ma wbudowanego typu napisowego, a rolę zmiennych napisowych pełnią tablice znaków, przy czym koniec napisu oznaczany jest znakiem '\0'. Rozmiar napisu powinien być większy o 1 niż długość przechowywanego tekstu.

Aby móc przechowywać napisy musimy stworzyć tablicę typu `char`, np.:

```
char napis[15];
```

Jak każdą tablicę, napis możemy zainicjować w momencie jego deklaracji.

```
char napis[15] = "Programowanie";
```

Możemy także pominąć rozmiar napisu. W takim wypadku kompilator sam wyliczy potrzebny rozmiar tablicy.

```
char napis[] = "Programowanie";
```

Kompilator automatycznie uzupełnia stałą tekstową (ciąg znaków w cudzysłowach " ") ogranicznikiem tekstu '\0'.

### 5.1 Zadanie 1 (wystąpienie znaku)

*Napisz program, który wczytuje z klawiatury słowo, a następnie sprawdza, ile razy wystąpiła w nim litera 'a'. Przyjmij, że czytane słowo może mieć maksymalnie 40 znaków.*

Zacznijmy od deklaracji zmiennej napisowej:

```
char słowo[41];
```

Ustawiamy rozmiar tablicy znaków na 41 elementów, ponieważ musimy uwzględnić dodawany automatycznie znak końca łańcucha ('\0'). Do wczytania słowa z klawiatury użyjemy znanej już funkcji `scanf`:

```
scanf("%s", słowo);
```

W porównaniu z wcześniejszymi wywołaniami funkcji `scanf` możemy zauważać dwie rzeczy. Po pierwsze pojawił się nowy format – %s. Jest to format

poświęcony napisom. Po drugie, zwróć uwagę na brak znaku & w wywołaniu funkcji `scanf`. Pamiętasz, że służy on do pobierania adresu zmiennej. W przypadku tablic nazwa tablicy określa adres jej początku w pamięci.

Aby policzyć wystąpienia litery 'a' w słowie, musimy przejść przez wszystkie jego wczytane znaki i dla każdego sprawdzić, czy jest literą 'a'. Ponieważ napisy są tablicami, używamy pętli:

```
for(i=0, ile=0; i<strlen(slowo); i++)
    if (slowo[i]=='a')
        ile++;
```

Uwagę powinien zwrócić warunek stopu pętli. Nie podajemy tutaj, że ma się wykonać 41 razy (mimo, że taki jest rozmiar tablicy). Dlaczego? Podane przez użytkownika słowo może być krótsze niż 40 znaków, a my mamy sprawdzić tylko te wczytane znaki. Skąd zatem wiedzieć, ile wpisał użytkownik? W momencie zakończenia wczytywania funkcja `scanf` umieszcza w napisie wspomniany wcześniej znak końca łańcucha '\0'. I rzeczywista długość napisu to liczba elementów w tablicy przed tym znakiem. Możemy oczywiście samodzielnie policzyć, ile ich jest. Możemy również skorzystać z funkcji bibliotecznej – `strlen`, która zwraca nam długość napisu (bez znaku kończącego). Znajduje się ona w bibliotece `<string.h>`.

W każdej iteracji sprawdzamy, czy bieżący znak jest równy znakowi 'a'. Jeżeli tak, to zwiększamy licznik.

```
printf("W napisie \"%s\" litera 'a' występuje %d razy", slowo, ile);
```

Wypisując na ekran napis funkcją `printf`, używamy również formatu `%s`.

Pełny kod naszego programu może wyglądać następująco:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char slowo[41];
    int i, ile;
    printf("Podaj wyraz: ");
    scanf("%s", slowo);
    for(i=0, ile=0; i<strlen(slowo); i++)
        if (slowo[i]=='a')
            ile++;
    printf("W napisie \"%s\" litera 'a' występuje %d razy", slowo, ile);
```

```
        return 0;  
    }
```

## 5.2 Zadanie 2 (palindrom)

Napisz program, który wczytuje ze standardowego wejścia napis do tablicy i sprawdza, czy jest on palindromem. Wczytany napis ma mieć maksymalnie 40 znaków. Wielkość liter nie powinna mieć znaczenia w trakcie porównywania.

Sprawdzanie, czy napis jest palindromem, napiszemy w postaci funkcji. Jej parametrem będzie sprawdzane słowo `s`. Zauważ, że nie ma tu potrzeby podawania rozmiaru tablicy (tym bardziej, że długość napisu może być mniejsza niż zarezerwowany rozmiar). Liczbę interesujących nas znaków sprawdzimy funkcją `strlen`.

```
bool czy_palindrom(char s[])  
{  
    int i=0, j = strlen(s)-1;  
    while(i<j)  
    {  
        if(tolower(s[i])!=tolower(s[j]))  
            return false;  
        i++;  
        j--;  
    }  
  
    return true; //wyraz jest palindromem  
}
```

Do przejścia napisu użyjemy dwóch liczników: `i` oraz `j`. Pierwszy będzie przehodził przez znaki napisu od początku i będzie zwiększany w każdej iteracji, a drugi od końca – jego początkowa wartość to długość napisu i po każdym przebiegu pętli jego wartość zostanie zmniejszona o 1. W momencie, gdy liczniki się spotkają, zakończymy sprawdzanie. Warunkiem stopu pętli `while` jest zatem  $i < j$ .

Instrukcja:

```
if(tolower(s[i])!=tolower(s[j]))
```

sprawdza, czy dwa znaki napisu są takie same. Użycie funkcji `tolower` z biblioteki `<ctype.h>` powoduje, że mała i wielka litery będą traktowane jako ten sam znak. Jako parametr funkcja przyjmuje pojedynczy znak i jeżeli jest on

wielką literą, to zwraca jej mały odpowiednik. W przeciwnym wypadku zwraca ten sam znak, który otrzymała w parametrze.

Gdy sprawdzane znaki nie będą się zgadzać, to wyraz nie jest palindromem i przerwamy działanie funkcji, zwracając jednocześnie wartość `false`.

Kod całego programu może wyglądać następująco:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

bool czy_palindrom(char s[])
{
    int i=0, j = strlen(s)-1;
    while(i<j)
    {
        if(s[i]!=s[j])
            return false;
        i++;
        j++;
    }

    return true;
}

int main()
{
    char slowo[40];
    printf("Podaj wyraz: ");
    scanf("%s", slowo);
    if(czy_palindrom(slowo))
        printf("\'%s\' jest palindromem.", slowo);
    else
        printf("\'%s\' nie jest palindromem.", slowo);
    return 0;
}
```

Powyższy program będzie działał pod warunkiem, że z klawiatury wprowadzimy pojedyncze słowo. W przypadku, gdybyśmy chcieli sprawdzić całe zdanie, okaże się, że do zmiennej `slowo` trafił tylko pierwszy wyraz wprowadzonego zdania. Dzieje się ta dlatego, że funkcja `scanf` traktuje białe znaki (spacja, tabulator, enter) jako sygnał końca wprowadzania danych. W przypadku, gdy zależy

nam na wczytaniu do napisu również białych znaków, musimy skorzystać z funkcji `fgets` umieszczonej w bibliotece `<stdio.h>`.

```
fgets(zdanie, 41, stdin);
```

Funkcja `fgets` przyjmuje trzy argumenty: tablicę, do której zapisze wczytane znaki, rozmiar tej tablicy oraz informację o tym, skąd ma czytać – w tym wypadku jest to klawiatura, czyli `stdin`. O ile pierwszy i trzeci parametr funkcji nie budzą wątpliwości, to pojawić się może pytanie, po co rozmiar tablicy. W odróżnieniu od funkcji `scanf`, funkcja `fgets` kontroluje liczbę wczytywanych znaków. Możliwe jest wczytanie tylko *rozmiar*-1 znaków do tablicy. Ostatni znak zarezerwowany jest dla znaku końca łańcucha ('`\0`'). Zapobiega to wyjściu poza zakres tablicy i utracie innych danych w pamięci.

Należy również wspomnieć, że funkcja `fgets` wczytuje do napisu również kończący znak *enter*. Gdybyśmy chcieli się go pozbyć, to wystarczy, że przesuniemy znak końca łańcucha o jedną pozycję w lewo. Zamażemy w ten sposób ostatni znak, którym jest przejście do nowej linii. Warto wcześniej jednak sprawdzić, czy niepotrzebny nam znak *enter* zmieścił się w tablicy:

```
if (slowo[strlen(slowo)-1]=='\n')  
    slowo[strlen(slowo)-1] = '\0';
```

Po zmianie funkcji wczytującej musimy również zmienić funkcję sprawdzającą, czy napis jest palindromem – powinna ona teraz pomijać te znaki w napisie, które nie są literami.

Zrealizujemy to poprzez usunięcie niechcianych znaków. Nie będziemy jednak modyfikować napisu przekazanego przez parametr – ta zmiana byłaby widoczna również po zakończeniu funkcji. Stworzymy tymczasowy napis, do którego przekopiujemy tylko litery z napisu oryginalnego. Deklarujemy łańcuch o długości takiej samej, jak napis wejściowy.

```
char slowo[strlen(s)+1];
```

Następnie, przechodząc napis oryginalny znak po znaku sprawdzamy, czy aktualny znak jest literą. Korzystamy tutaj z funkcji `isalpha`, która znajduje się w bibliotece `<ctype.h>`. Funkcja `isalpha` zwraca wartość niezerową, gdy przekazany jej znak jest literą i zero w przeciwnym wypadku.

```
int i, j;  
for(i=0, j=0; i<strlen(s); i++)  
    if (isalpha(s[i])) slowo[j++]=s[i];
```

Zwróć uwagę, że zmienna `j` zwiększana jest dopiero po dodaniu nowego znaku do tymczasowego napisu. Na koniec musimy jeszcze uzupełnić stworzony napis o znak końca łańcucha:

```
slowo[j] = '\0';
```

Po opuszczeniu pętli `for` zmienna `j` będzie miała wartość o jeden większą niż liczba przepisanych znaków, zatem nie ma potrzeby jej zwiększać.

Po tych modyfikacjach funkcja `czy_palindrom` będzie miała następującą postać:

```
bool czy_palindrom(char s[])
{
    char slowo[strlen(s)];
    int i, j;
    for(i=0, j=0; i<strlen(s); i++)
        if (isalpha(s[i])) slowo[j++]=s[i];
    slowo[j] = '\0';
    i=0;
    j=strlen(slowo)-1;
    while(i<j)
    {
        if(tolower(slowo[i])!=tolower(slowo[j]))
            return false;
        ++i;
        --j;
    }
    return true;
}
```

### 5.3 Zadanie 3 (ostatnie wystąpienie znaku)

Napisz program, który wczytuje od użytkownika łańcuch znaków, a następnie pojedynczy znak i podaje indeks ostatniego wystąpienia tego znaku w zadanym łańcuchu. Wyświetlane indeksy mają być liczone od jedynki. Wczytany napis ma mieć maksymalnie 20 znaków.

Przykład: łańcuch: "Ola ma kota" znak: 'o', wynik: 9

Po deklaracji zmiennych i wczytaniu ich wartości:

```
char zdanie[41], znak;
int i, pozycja;
printf("Podaj zdanie: ");
fgets(zdanie, 41, stdin);
printf("Podaj znak: ");
scanf("%c", &znak);
```

możemy przeszukać nasz napis. Poniższa pętla `for` iteruje po napisie, sprawdzając, czy dany jego znak jest taki sam, jak podany przez użytkownika. W tym przypadku uwzględniamy wielkość liter:

```
for(i=0, pozycja=0; i<strlen(zdanie); i++)
    if (zdanie[i] == znak)
        pozycja = i+1;
```

W przypadku zgodności przypisujemy zmiennej pozycja wartość indeksu bieżącego znaku z tablicy powiększoną o jeden, by zachować zgodność z treścią zadania. Jeżeli okaże się, że poszukiwany znak nie występuje w napisie, to zmienna pozycja zachowią swoją początkową wartość, czyli zero. Skorzystamy z tego przy wypisywaniu wyniku:

```
if (pozycja>0)
    printf("Ostatnie wystąpienie '%c' na pozycji %d.",
           znak, pozycja);
else
    printf("Znak '%c' nie występuje w zdaniu.", znak);
```

Kod całego programu wygląda następująco:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char zdanie[41], znak;
    int i, pozycja;

    printf("Podaj zdanie: ");
    fgets(zdanie, 41, stdin);
    printf("Podaj znak: ");
    scanf("%c", &znak);
    for(i=0, pozycja=0; i<strlen(zdanie); i++)
        if (zdanie[i] == znak)
            pozycja = i+1;
    if (pozycja>0)
        printf("Ostatnie wystąpienie '%c' na pozycji %d.",
               znak, pozycja);
    else
        printf("Znak '%c' nie występuje w zdaniu.", znak);
    return 0;
}
```

# Rozdział 6

## Tablice dwuwymiarowe

### 6.1 Zadanie 1 (podzielne przez 3 lub 4)

Napisz program, który wypełni dwuwymiarową tablicę liczb całkowitych  $t[3][4]$  wartościami wczytanymi od użytkownika, a następnie obliczy i wyświetli liczbę elementów tej macierzy, które są podzielne przez 3 lub 4.

Kod programu zaczynamy od deklaracji tablicy i pozostałych zmiennych:

```
int tab[N][M], i, j, ile;
```

Tablicę dwuwymiarową deklarujemy w sposób analogiczny do dwuwymiarowej – z tym, że podajemy dwa wymiary: pierwszy to liczba wierszy (N), drugi liczba kolumn (M). Zostały one wcześniej zdefiniowane w programie przy użyciu dyrektywy `#define`:

```
#define N 3  
#define M 4
```

Aby wypełnić wartościami tablicę dwuwymiarową, potrzebowaliśmy jednej pętli. W przypadku tablic dwuwymiarowych niezbędne są dwie pętle (pierwsza przechodzi po wierszach, druga po kolumnach):

```
for(i=0; i<N; i++)  
    for(j=0; j<M; j++)  
        scanf("%4d", &tab[i][j]);
```

Powyższy kod możemy odczytać następująco: dla każdego wiersza  $i$  dla każdej kolumny  $j$  w wierszu  $i$  wczytaj wartość elementu `tab[i][i]`. Widzimy również, w jaki sposób odwołać się do konkretnego elementu w tablicy – podając jego „współrzędne”, czyli numer wiersza i numer kolumny tabeli. W analogiczny sposób wypiszemy zawartość całej tablicy na ekranie:

```
for(i=0; i<N; i++)  
{  
    for(j=0; j<M; j++)  
        printf("%4d", tab[i][j]);  
    printf("\n");  
}
```

A także zliczymy, ile elementów tej tablicy jest podzielnych przez 3 lub przez 4:

```

for(i=0, ile=0; i<N; i++)
    for(j=0; j<M; j++)
        if(tab[i][j]%3==0 || tab[i][j]%4==0)
            ile++;

```

Kod całego programu będzie wyglądał zatem tak:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

#define N 3
#define M 4

int main()
{
    int tab[N][M], i, j, ile;

    printf("Podaj %d liczb: ", N*M);
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            scanf("%d", &tab[i][j]);

    for(i=0; i<N; i++)
    {
        for(j=0; j<M; j++)
            printf("%4d", tab[i][j]);
        printf("\n");
    }

    for(i=0, ile=0; i<N; i++)
        for(j=0; j<M; j++)
            if(tab[i][j]%3==0 || tab[i][j]%4==0)
                ile++;

    printf("Wartości podzielnych przez 3 lub 4 jest %d\n", ile);
    return 0;
}

```

## 6.2 Zadanie 2 (wiersz o najwyższej średniej elementów)

Napisz program, który wypełni losowymi wartościami z przedziału  $<-10; 10>$  dwuwymiarową tablicę liczb całkowitych  $t[8][6]$ , a następnie znajdzie numer wiersza, dla którego średnia elementów jest największa. W przypadku, gdy kilka wierszy w tablicy ma taką samą średnią, program powinien wyświetlić numer pierwszego z nich.

Po deklaracji niezbędnych zmiennych ( $N$  i  $M$  ponownie zostały definiowane przez `#define`):

```
int tab[N][M], i, j, ktory;  
float srednia, maks_srednia;
```

losujemy zawartość tablicy:

```
srand(time(0));  
for(i=0; i<N; i++)  
    for(j=0; j<M; j++)  
        tab[i][j] = rand()%21-10;
```

Przypomnijmy, że funkcja `srand` gwarantuje losowanie innych wartości przy każdym uruchomieniu programu.

Mając wypełnioną tablicę, możemy przejść do poszukiwania wiersza o maksymalnej średniej. Algorytm poszukiwania wartości maksymalnej jest analogiczny do omówionego przy okazji tablic jednowymiarowych algorytmu znalezienia minimum. Przed przystąpieniem do przeszukiwania tablicy musimy nadać wartości początkowe zmiennej `ktory` oraz `maks_srednia` z tą różnicą, że w `maks_srednia` umieścimy średnią elementów pierwszego wiersza, a w zmiennej `ktory` numer pierwszego wiersza, czyli 0:

```
ktory = 0;  
for(j=0, srednia=0; j<M; j++)  
    maks_srednia += tab[0][j];
```

Moglibyśmy w tym momencie podzielić `maks_srednia` przez długość wiersza, czyli  $M$ . Ponieważ jednak nasza tablica jest prostokątna, to każdy wiersz jest tej samej długości. Wiersz o najwyższej średniej będzie zatem również wierszem o największej sumie – mianownik średniej każdego wiersza wynosi tyle samo:  $M$ . Czyli w tym wypadku możemy zrezygnować z dzielenia i poprzestać na wyznaczaniu sumy elementów.

Po ustaleniu wartości początkowych możemy zacząć szukanie w pozostałą części tablicy:

```
for(i=1; i<N; i++)  
{
```

```

srednia = 0;
for(j=0; j<M; j++)
    srednia += tab[i][j];
if(srednia>maks_srednia)
{
    maks_srednia = srednia;
    ktory = i;
}
}

```

Zewnętrzna pętla for kontrolowana przez i przechodzi po każdym wierszu. Wartość i zaczyna się od wartości 1, ponieważ już wyżej wyliczyliśmy średnią dla wiersza pierwszego (o indeksie 0).

W każdym przebiegu pętli musimy pamiętać o wyzerowaniu zmiennej **srednia**, ponieważ każdy wiersz traktujemy osobno. Następnie w pętli wewnętrznej obliczamy sumę elementów *i*-tego wiersza. Po zakończeniu pętli sprawdzamy, czy suma *i*-tego wiersza nie jest większa od dotychczasowego maksimum. Jeżeli jest, to aktualizujemy zmienne **maks\_srednia** oraz **ktory** na bieżące wartości. Kod całego programu (uzupełnionego o wypisywanie tablicy na ekranie) prezentuje się następująco:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 8
#define M 6

int main()
{
    int tab[N][M], i, j, ktory;
    float srednia, maks_srednia;

    srand(time(0));
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            tab[i][j] = rand()%21-10;

    for(i=0; i<N; i++)
    {
        for(j=0; j<M; j++)
            printf("%4d", tab[i][j]);
        printf("\n");
    }
}

```

```

}

ktory = 0;
for(j=0, srednia=0; j<M; j++)
    maks_srednia += tab[0][j];

for(i=1; i<N; i++)
{
    srednia = 0;
    for(j=0; j<M; j++)
        srednia += tab[i][j];
    if(srednia>maks_srednia)
    {
        maks_srednia = srednia;
        ktory = i;
    }
}

printf("Wiersz o największej średniej elementów to: %d.", ktory);
return 0;
}

```

### 6.3 Zadanie 3 (usuwanie pustych kolumn)

Napisz funkcję, która w tablicy dwuwymiarowej  $N$  „usunie” puste (wypełnione samymi zerami) kolumny (przesuwając „niepuste” w lewo).

Aby lepiej wyobrazić sobie cel zadania, założymy, że na wejściu dostajemy następującą macierz:

```

2 0 1 0 0
3 0 2 0 2
5 0 1 0 2

```

Widzimy tu dwie kolumny, będące kandydatkami do usunięcia – zawierają one same zera. Usuwanie nie ma polegać na zmniejszaniu rozmiaru macierzy, a jedynie na przesunięciu w lewo kolumn, które nie są puste. Jako rezultat powinniśmy zatem otrzymać:

```

2 1 0 0 0
3 2 2 0 0
5 1 2 0 0

```

Przyjmiemy, że rozmiar macierzy jest stały, np.:

```
#define N 3  
#define M 5
```

Nie zmieni to naszego algorytmu, ale uprości testy. Nagłówek funkcji może wyglądać zatem jak poniżej:

```
void zeroshift(int mat[N][M])
```

Na wejściu otrzymujemy tablicę 2D, wszystkie operacje wykonujemy na niej, zatem nie musimy zwracać żadnego wyniku.

Pierwszym krokiem powinno być znalezienie pustej kolumny (o ile jakaś istnieje). Potrzebujemy zatem jednej pętli, która przejrzyc po kolejne wszystkie kolumny:

```
for (int col = 0; col < M; col++)  
{  
}
```

Zmienna iteracyjna została nazwana `col`, aby dla czytającego kod nie było wątpliwości, że chodzi o kolumnę. Wewnątrz tej pętli umieścimy kolejną – w każdej kolumnie musimy przejrzeć wszystkie elementy i sprawdzić, czy wszystkie są zerami:

```
int count = 0;  
for (int row = 0; row < N; row++)  
{  
    if (mat[row][col] == 0)  
        count++;  
}
```

Nie różni się to niczym od kodu, który sprawdzałby czy wszystkie elementy wektora (tablicy jednowymiarowej) są zerami, z tym, że tym razem dotyczy to elementów kolumny `col`. Iteracja przebiega przez wiersze macierzy, zatem zmienna iteracyjna została nazwana `row`.

Samo sprawdzanie pewnie moglibyśmy załatwić nieco inaczej (np. gdy znajdziemy pierwszy element różny od zera, nie ma potrzeby sprawdzać dalej), ale te modyfikacje niech będą ćwiczeniem dla czytelnika. Póki co, zostawmy powyższą, prostą w odczycie postać – biegnimy z góry do dołu i liczymy ile zer występuje w kolumnie.

Gdy to określmy, możemy sprawdzić, czy jest ich tyle, ile wynosi wysokość kolumny (liczba wierszy):

```
if (count == N)
```

Jeśli tak – oznacza to, że odnaleźliśmy pustą kolumnę. Pora zatem na przesuwanie. Jeśli pustą kolumną jest kolumna 2, to na jej miejsce należy wpisać

wszystkie elementy z kolumny 3, z kolei do kolumny 3 trafi zawartość kolumny 4 itd. Potrzebujemy zatem podwójnej pętli, która dokona odpowiedniego podstawienia:

```
for (int i = 0; i < N; i++)
{
    for (int j = col; j < M - 1; j++)
    {
        mat[i][j] = mat[i][j + 1];
    }
}
```

Pierwsza pętla przebiega przez kolejne wiersze, zaś druga – przez kolumny. Ich kolejność nie ma znaczenia – gdyby zamienić je miejscami, operacja wykonywałaby się tak samo. Różnica polega na tym, czy najpierw przesuniemy wszystkie elementy pierwszego wiersza, a potem przejdziemy do następnego, by zrobić to samo, czy też raczej najpierw przepiszemy jedną kolumnę, potem następną, potem kolejną itd. Efekt jest identyczny. Znaczenie ma natomiast, skąd zaczniemy to robić. W drugiej pętli zmienią się od wartości `col` – bowiem `col` jest tą odnalezioną, pustą kolumną, której zawartość chcemy wymazać. Zatem na miejsce kolumny `col` trafią wartości z kolumny `col+1`, potem na miejsce `col+1` trafią wartości z `col+2` itd. Tak jak jest zapisane wewnątrz podwójnej pętli:

```
mat[i][j] = mat[i][j + 1];
```

Jeszcze jedna ważna rzecz to warunek końca pętli – pętla po kolumnach (zmieniona iteracyjna `j`) iteruje się do `M-1`, a nie `M`. Na miejsce przedostatniej kolumny trafią wartości z kolumny ostatniej (`j+1`), natomiast dla ostatniej kolumny nie możemy wykonać już tej operacji (ponieważ `j+1` wyszłoby poza dozwolony zakres – nie mielibyśmy skąd wziąć tych wartości). Ostatnia kolumna powinna zostać wyzerowana. Zrobimy to w osobnej pętli:

```
for (int i = 0; i < N; i++)
{
    mat[i][m - 1] = 0;
}
```

Tu iterujemy tylko po wierszach i ostatnią wartość w każdym wierszu ustawiamy na 0.

Jeszcze jedna ważna zmiana. Jeśli w ten sposób „usuniemy” kolumnę numer 2, co stanie się dalej? Pętla po kolumnach (zewnętrzny `for`) dalej robi swoje – wykonuje operację `col++` i sprawdza kolejną kolumnę. Ale to oznacza, że jedną kolumnę pominieliśmy. Do kolumny 2 przepisaliśmy zawartość kolumny 3, potem przesuwamy się do 3 (gdzie znajduje się zawartość kolumny 4), ale nie

sprawdziliśmy co było w kolumnie 3. Czasem możemy nawet nie dostrzec tego błędu, ale gdyby np. dwie kolumny z rzędu były wypełnione zerami, to jednej z nich nie zdolamy w ten sposób wykryć. Po „usunięciu” kolumny powinniśmy zatem sprawdzić ją jeszcze raz (tym razem dla nowych wartości). Aby to zapewnić wystarczy zmniejszyć `col` po dokonaniu przesunięcia i uzupełnienia zerami:

```
col--;
```

To zniweluje operację `col++`, która za chwilę zostanie wykonana jako element pętli `for`.

To jednak nie koniec. W tej chwili nasza pętla po kolumnach wykonuje się zawsze do `M`, czyli liczby kolumn macierzy. Jeśli jednak wykonaliśmy przesuwanie i uzupełnianie, to ostatnia kolumna na pewno jest wypełniona zerami, a zatem nie ma potrzeby jej sprawdzać. I nie jest to tylko kwestia optymalizacji. Sprawdzenie jej może spowodować błąd – bo skoro jest wypełniona zerami, to powinniśmy wykonać przesunięcie i uzupełnienie (które nie przyniosą żadnego efektu), następnie `col-`, aby wykonać sprawdzenie jeszcze raz – które znów dostarczy nam informacji, że jest to kolumna zer, potem jeszcze raz, jeszcze raz, jeszcze raz. Taki program zapętli się w nieskończoność. Należy zatem upewnić się, że sprawdzanie zakończy się w odpowiednim momencie. Dodajmy na początku funkcji zmienną, która przechowuje nam liczbę kolumn.

```
int m = M;
```

Teraz w kodzie pętli zastąpmy stałą `M` zmienną `m`. I dodajmy jedną linijkę po usunięciu wypełnionej zerami kolumny:

```
m--;
```

Skoro usunęliśmy kolumnę, to mamy o jedną mniej do sprawdzenia. Teraz funkcja powinna prawidłowo zadziałać. Cały jej kod przedstawiono poniżej:

```
void zeroshift(int mat[N][M])
{
    // szukamy pustej kolumny:
    int m = M;
    for (int col = 0; col < m; col++)
    {
        int count = 0;
        for (int row = 0; row < N; row++)
        {
            if (mat[row][col] == 0)
                count++;
        }
        // jest pusta kolumna!
```

```

if (count == N)
{
    // przesuwamy wszystko w lewo
    for (int i = 0; i < N; i++)
    {
        for (int j = col; j < m - 1; j++)
        {
            mat[i][j] = mat[i][j + 1];
        }
    }
    // ostatnią zerujemy
    for (int i = 0; i < N; i++)
    {
        mat[i][m - 1] = 0;
    }
    // tą, którą przesunęliśmy sobie z prawej,
    // wciąż musimy sprawdzić
    col--;
    // mamy o jedną mniej do sprawdzania!
    m--;
}
}
}

```

Jak mógłby wyglądać kod testujący jej działanie? Zawartość tablicy moglibyśmy wczytać od użytkownika. To jednak uczyni testowanie żmudnym zajęciem i za każdym razem będzie wymagać wpisania, powiedzmy, 15 wartości. Alternatywą jest losowanie, ale szansa na to, że wylosuje się nam jakąś pusta kolumna, może być zbyt mała, by test miał sens. Dlatego najwygodniej będzie przyjąć jakąś stałą zawartość:

```

int mat[N][M] = {
    2, 0, 1, 0, 0,
    3, 0, 2, 0, 2,
    5, 0, 1, 0, 2};

```

Przyda się fragment kodu wypisujący ją na ekranie, aby użytkownik testujący program miał wgląd w jej zawartość:

```

for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        printf("%d ", mat[i][j]);
    }
}

```

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

Teraz możemy wywołać naszą funkcję, przekazując tablicę jako parametr:

```
zeroshift(mat);
```

A następnie znów wypisać ją na ekranie – licząc, że puste kolumny zostaną poprawnie zastąpione tymi, przesuniętymi z prawej.

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        printf("%d ", mat[i][j]);
    }
    printf("\n");
}
```

# Rozdział 7

## Wskaźniki, dynamiczny przydział pamięci

### 7.1 Zadanie 1 (zdublowane elementy)

Napisz program, który sprawdza, ile elementów jednej tablicy jednowymiarowej występuje w drugiej. Pierwsza tablica wypełniana jest losowymi wartościami z przedziału  $<1; 10>$ , druga liczbami pobranymi z klawiatury.

Zacznijmy od wyjaśnienia czym jest tablica dynamiczna i czym się różni od dotychczas stosowanych, tzw. automatycznych. Zazwyczaj zmienne programu przechowywane w pamięci typu stos (ang. *stack*) – powstają, gdy program wchodzi do bloku, w którym zmienne są zadeklarowane oraz usuwane w momencie zakończenia tego bloku. Dla tablic przechowywanych na stosie rozmiar musi być znany w momencie komplikacji, żeby kompilator wygenerował kod rezerwujący odpowiednią ilość pamięci. Stąd też deklaracja takich tablic wyglądała następująco:

```
int tablica[10];
```

lub:

```
int n = 10;  
int tablica[n];
```

Dostępny jest jeszcze inny rodzaj pamięci. Jest to tzw. sterta (ang. *heap*). Sterta to obszar pamięci wspólny dla całego programu, przechowywane są w nim zmienne, których czas życia nie jest związany z poszczególnymi blokami. Aby uzyskać do niej dostęp, sami musimy je rezerwować (a także je później zwalniać), ale dzięki temu możemy to zrobić w dowolnym momencie działania programu.

Tablice dynamiczne są właśnie tworzone na stercie. Aby z nich skorzystać, potrzebujemy specjalnych zmiennych wskaźnikowych do przechowywania adresu w pamięci. Zmienną wskaźnikową deklarujemy następująco:

```
int *t;
```

Spotkaliśmy się już z nimi przy okazji funkcji i przekazywania parametrów przez adres. Warto w tym miejscu zaznaczyć, że deklaracja:

```
int *t1, t2;
```

Tworzy tylko jedną zmienną wskaźnikową **t1** i zmienną typu całkowitego **t2**. Nie ma znaczenia, czy symbol **\*** wpiszemy przy nazwie zmiennej, czy nazwie

typu. Zawsze odnosi się ona do najbliższej zmiennej. A jednoczesna deklaracja dwóch wskaźników ma postać:

```
int *t1, *t2;
```

Mając zmienną wskaźnikową, możemy poprosić o dostęp do pamięci na stercie. Służy do tego funkcja `malloc` znajdująca się w bibliotece `<stdlib.h>`:

```
t = malloc(n*sizeof(int));
```

Funkcja `malloc` oczekuje, że w parametrze otrzyma rozmiar pamięci w bajtach. W naszym przypadku możemy wyliczyć tę wartość, mnożąc liczbę elementów tablicy przez rozmiar pojedynczego elementu. Aby poznać rozmiar elementu, korzystamy z operatora `sizeof`. Wywołanie `sizeof(int)` zwraca nam liczbę bajtów, jaką zajmuje w pamięci zmienna typu `int` zgodnie z architekturą maszyny, na której uruchamiamy nasz program.

Funkcja `malloc` zwraca adres pierwszego bajtu przydzielonego bloku pamięci, który przypisujemy naszemu wskaźnikowi `t`. W przypadku, gdy nie uda znaleźć się wystarczająco dużo wolnego obszaru, to zwróci wskaźnik zerowy (`NULL`). Dobrą praktyką jest sprawdzanie, czy alokacja pamięci przebiegła pomyślnie.

```
if (t==NULL)
{
    printf("Alokacja zakończona niepowodzeniem\n.");
    //dalej kod
}
```

Powyższą instrukcję przydziału pamięci możemy również zapisać trochę inaczej:

```
t = malloc(n*sizeof(*t));
```

Jaka jest różnica? Zamiast `sizeof(int)` zapisaliśmy `sizeof(*t)`. Tak naprawdę te dwa zapisy są ze sobą równoważne. Zmienna `t` jest wskaźnikiem na typ `int`, czyli instrukcja `*t` (operator wyłuskania) daje nam zmienną typu `int`. Zatem `sizeof(*t)` to po prostu `sizeof(int)`. Jeżeli chodzi o działanie programu, to, który zapis wybierzymy, nie ma znaczenia. Różnica pojawia się w chwili, gdy w postanowimy zmienić typ przechowywanych elementów w tablicy, np. na `float`. Jeżeli alokowaliśmy pamięć stosując zapis `sizeof(int)`, będziemy musieli zmienić `float` na `int` w każdym wywołaniu funkcji `malloc` dla tego wskaźnika. Z kolei użycie `sizeof(*t)` zwalnia nas z tego obowiązku, ponieważ rozmiar zostanie wyliczony automatycznie na podstawie typu podanego przy deklaracji wskaźnika. Mając stworzoną tablicę, możemy przystąpić do wypełniania jej wartościami:

```
int i;
srand(time(0));
```

```
for(i=0; i<n; i++)
    t[i] = rand()%10+1;
```

Odwoływanie się do elementów tablic dynamicznych wygląda tak samo, jak w przypadku wcześniejszych tablic automatycznych. Podajemy indeks, pod który wstawiamy wartość.

Wiemy już, w jaki sposób przydzielić pamięć. Stwórzmy teraz funkcję, która będzie tworzyć i wypełniać tablicę oraz ją zwracać. Nagłówek tej funkcji będzie wyglądał następująco:

```
int* tworzLosowo(int n)
```

Parametrem funkcji jest rozmiar tworzonej tablicy. Zwracana wartość to zmieniona wskaźnikowa. Będzie ona zawierała adres zwrócony przez `malloc` wewnątrz funkcji:

```
return t;
```

Cała funkcja zatem ma postać:

```
int* tworzLosowo(int n)
{
    int *t = (int*)malloc(n*sizeof(*t));
    if (t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem\n.");
        return NULL;
    }
    int i;
    srand(time(0));
    for(i=0; i<n; i++)
        t[i] = rand()%10+1;

    return t;
}
```

Zauważmy, że w przypadku, gdy przydział pamięci nie powiodł się, nasza funkcja zwraca `NULL`. Wykorzystamy to w późniejszym kodzie.

W analogiczny sposób tworzymy drugą funkcję, tę, która wypełnia elementy tablicy wartościami pobranymi od użytkownika:

```
int* tworzKlawiatura(int n)
{
    int *t = (int*)malloc(n*sizeof(*t));
    if (t==NULL)
```

```

{
    printf("Alokacja zakończona niepowodzeniem\n.");
    return NULL;
}
int i;
srand(time(0));
printf("Wprowadź elementy tablicy:\n");
for(i=0; i<n; i++)
    scanf("%d", &t[i]);

return t;
}

```

Następną częścią naszego zadania jest sprawdzenie, ile elementów pierwszej tablicy występuje w drugiej. Zaczniemy od nagłówka funkcji:

```
int ileWystepuje(int *t1, int n1, int *t2, int n2)
```

Funkcja `ileWystepuje` przyjmuje cztery parametry: dwie tablice, a dokładniej adresy ich pierwszych elementów (`t1` i `t2`) oraz ich rozmiary (odpowiednio `n1` i `n2`). Zwieracana wartość to liczba wystąpień, czyli typem funkcji jest typ `int`. Zastanówmy się nad algorytmem poszukiwania wystąpień. Musimy взять każdy element z pierwszej tablicy i porównać go z każdym elementem drugiej tablicy. W przypadku, gdy są równe, zwiększamy licznik wystąpień i przerywamy sprawdzanie:

```

for(i=0, ile=0; i<n1; i++)
{
    for(j=0; j<n2; j++)
        if(t1[i] == t2[j])
    {
        ile++;
        break;
    }
}

```

Dzięki instrukcji `break` w pętli wewnętrznej unikniemy nadmiarowego podbijania licznika. Na przykład dla tablicy `t1` [1, 2, 3] i tablicy `t2` [2, 5, 1, 5, 1] wynik naszej funkcji wyniesie 2 – wartości 1 oraz 2 pojawiają się w obydwu tablicach. Gdybyśmy jednak przeszukiwali za każdym razem tablicę `t2` do końca, to wynik wyniósłby 3, ponieważ wartość 1 występuje w `t2` dwukrotnie.

Zostało nam jeszcze zwrócić wyznaczoną wartość:

```
return ile;
```

Pełna funkcja `ileWystepuje` ma zatem postać:

```

int ileWystepuje(int *t1, int n1, int *t2, int n2)
{
    int i, j, ile = 0;

    for(i=0, ile=0; i<n1; i++)
    {
        for(j=0; j<n2; j++)
            if(t1[i] == t2[j])
            {
                ile++;
                break;
            }
    }
    return ile;
}

```

Możemy teraz przejść do funkcji głównej programu. Zaczynamy od deklaracji potrzebnych zmiennych:

```
int n1, *t1, n2, *t2;
```

Po wczytaniu rozmiaru pierwszej tablicy, tworzymy ją korzystając z napisanej wcześniej funkcji:

```
t1 = tworzLosowo(n1);
if(t1==NULL)
    return -1;
```

Gdyby wewnątrz funkcji nie powiodła się alokacja pamięci (czyli `tworzLosowo` zwróciła `NULL`) kończymy działanie całego programu.

W analogiczny sposób stworzymy drugą tablicę:

```
t2 = tworzKlawiatura(n2);
if(t2==NULL)
    return -1;
```

Mając tablice, możemy wywołać funkcję zliczającą wspólne wystąpienia liczb w tablicach:

```
printf("Odpowiedź: %d\n", ileWystepuje(t1, n1, t2, n2));
```

Wynik otrzymany z funkcji wyświetlamy od razu na ekranie.

Nie jest to jednak koniec naszego programu. Korzystając z tablic dynamicznych, musimy pamiętać o jeszcze jednym aspekcie. Skoro to my przyjęliśmy

na siebie zarządzanie pamięcią, to poza jej przydzieleniem i używaniem powinniśmy ją również oddać, gdy nie jest nam już potrzebna. Ten proces nazywamy „zwolnieniem pamięci”. Używamy to tego celu funkcji `free` z biblioteki `<stdlib.h>`.

```
free(t1);
```

Funkcja `free` dealokuje (zwalnia) obszar pamięci wskazany przez wskaźnik dany jako parametr, o ile jest on różny od `NULL`. W przeciwnym wypadku nie robi nic. Argument ten musi być wskaźnikiem do obszaru uprzednio przydzielonego przez funkcję `malloc`. Ale co to dokładnie znaczy? Czyści ją? Usuwa ze sterty?

Najprościej mówiąc, `free` zaznacza zwalniany blok pamięci jako dostępny. Możemy to sobie wyobrazić następująco: gdzieś w pamięci, która jest niedostępna bezpośrednio dla naszego programu, istnieje pewna lista. Każdy element tej listy to para adresów: początek i koniec pewnego bloku w pamięci. W momencie wywołania funkcji `malloc` w tej tablicy powstaje zapis mówiący, odkąd i dokąd sięga zarezerwowany blok. Od tego momentu ten obszar jest „nietykalny” dla innych wywołań funkcji `malloc`.

Wywołanie funkcji `free` usuwa z tej listy wpis o zarezerwowanym bloku. I od tej pory, może on być ponownie przydzielony. Co ważne, funkcja `free` nie czyści zwalnianej pamięci. Są w niej cały czas te wartości, które tam umieściliśmy. Funkcja `free` nie zmienia również wartości wskaźnika! Ma on nadal tę samą wartość, którą uzyskał po wywołaniu funkcji `malloc`. Może to prowadzić do błędów. Aby temu częściowo zapobiec, możemy po wywołaniu funkcji `free` ustawić wartość na `NULL`:

```
t1 = NULL;
```

Pamięć zwalniamy dopiero wtedy, kiedy nie będzie ona nam więcej potrzebna. Ostatnie dwie omówione instrukcje powodują, że nie mamy już dostępu do tablicy `t1` i każda próba odwołania do jej elementów zakończy się błędem.

Wróćmy jeszcze na moment do tego fragmentu programu:

```
t2 = tworzKlawiatura(n2);
if(t2==NULL)
    return -1;
```

Powinniśmy go uzupełnić o zwolnienie pamięci tablicy `t1`. W momencie, gdy nie udało się nam stworzyć tablicy `t2`, tablica `t1` już istnieje w pamięci. A ponieważ każdemu zakończonemu sukcesem wywołaniu funkcji `malloc` powinno towarzyszyć wywołanie funkcji `free`, to nasz kod powinien wyglądać tak:

```
t2 = tworzKlawiatura(n2);
if(t2==NULL)
```

```
{
    free(t1); t1 = NULL;
    return -1;
}
```

Lącząc wszystkie powyższe fragmenty, otrzymujemy kod całego programu (uzupełnionego o funkcję wypisującą tablicę na ekranie):

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int* tworzLosowo(int n)
{
    int *t = (int*)malloc(n*sizeof(*t));
    if (t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem\n.");
        return NULL;
    }
    int i;
    srand(time(0));
    for(i=0; i<n; i++)
        t[i] = rand()%10+1;

    return t;
}

int* tworzKlawiatura(int n)
{
    int *t = (int*)malloc(n*sizeof(*t));
    if (t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem\n.");
        return NULL;
    }
    int i;
    srand(time(0));
    printf("Wprowadź elementy tablicy:\n");
    for(i=0; i<n; i++)
        scanf("%d", &t[i]);

    return t;
}
```

```

}

void wypiszTablice(int *t, int n)
{
    int i;

    for(i=0; i<n; i++)
        printf("%4d", t[i]);
    printf("\n");
}

int ileWystepuje(int *t1, int n1, int *t2, int n2)
{
    int i, j, ile = 0;

    for(i=0, ile=0; i<n1; i++)
    {
        for(j=0; j<n2; j++)
            if(t1[i] == t2[j])
            {
                ile++;
                break;
            }
    }
    return ile;
}

int main()
{
    int n1, *t1, n2, *t2;
    do
    {
        printf("Podaj rozmiar pierwszej tablicy: ");
        scanf("%d", &n1);
    }while(n1<=0);
    t1 = tworzLosowo(n1);
    if(t1==NULL)
        return -1;

    do
    {
        printf("Podaj rozmiar drugiej tablicy: ");
        scanf("%d", &n2);

```

```

}while(n2<=0);
t2 = tworzKlawiatura(n2);
if(t2==NULL)
{
    free(t1); t1 = NULL;
    return -1;
}

wypiszTablice(t1, n1);
wypiszTablice(t2, n2);

printf("Odpowiedź: %d\n", ileWystepuje(t1, n1, t2, n2));

free(t1); t1 = NULL;
free(t2); t2 = NULL;
return 0;
}

```

## 7.2 Zadanie 2 (tablica bez powtórzeń)

*Napisz funkcję, która na podstawie tablicy liczb całkowitych stworzy nową w taki sposób, by nie zawierała powtarzających się wartości. Parametrem funkcji jest oryginalna tablica oraz jej rozmiar. Pamiętaj również o rozmiarze nowej tablicy – on również musi „wyjść”, poza funkcję. Następnie wywołaj tę funkcję w programie głównym i wypisz zawartość zwróconej tablicy na ekran.*

Zaczynamy od zdefiniowania nagłówka funkcji:

```
int* bezPowtorzen(int *t, int n, int *m)
```

Funkcja zwraca adres pierwszego elementu utworzonej tablicy (zapisany w zmiennej wskaźnikowej typu `int*`). Jej parametry to istniejąca tablica wraz z rozmiarem. Trzeci parametr określać będzie rozmiar nowej tablicy. Ponieważ język C nie oferuje możliwości sprawdzenia rozmiaru tablicy, musimy go wprowadzić z funkcji. Instrukcja `return` będzie zajęta przez adres tablicy, zatem jedynym sposobem, jaki nam pozostał, jest zwrócenie w parametrze, czyli przez adres.

Zaczynamy od zadeklarowania niezbędnych zmiennych:

```
int *nowa = NULL;
int i, j, k;
bool wystepuje = false;
```

Zmienna `nowa` będzie przechowywała adres tablicy, a zmienna `wystepuje` pomoże nam wyszukać powtarzające się wartości.

Zanim stworzymy nową tablicę, musimy poznać jej rozmiar, czyli liczbę unikalnych wartości w tablicy źródłowej. Algorytm ich zliczenia może wyglądać następująco: *dla każdego elementu w tablicy źródłowej sprawdź, czy nie wystąpił on w lewej części tablicy. Jeśli nie, zwiększ licznik.*

```
for(i=0, (*m)=0; i<n; i++)
{
```

```
    wystepuje = false;
```

Wewnątrz pętli dla każdego  $i$ -tego elementu tablicy źródłowej ustawiamy zmienną pomocniczą `wystepuje` na `false` (zakładamy, że bieżąca wartość jest unikalna):

```
for(j=0; j<i; j++)
    if (t[i] == t[j])
    {
        wystepuje = true;
        break;
    }
```

Następnie sprawdzamy w lewej części tablicy, czy rzeczywiście do tej pory się nie pojawiła w tablicy. Jeżeli znaleźliśmy taką wartość, przedstawiamy zmienną `wystepuje` na `true` i kończymy sprawdzanie. Nie ma potrzeby sprawdzać wszystkich elementów leżących z przodu. Informacja, ile razy dana wartość już wystąpiła, nie jest nam potrzebna.

```
if(!wystepuje)
    (*m)++;
```

Jeżeli zmienna `wystepuje` ma nadal wartość `false` po wyjściu z pętli wewnętrznej, to znaczy, że bieżąca wartość tablicy jest unikalna. W takim wypadku zwiększamy licznik.

Po zakończeniu pętli zewnętrznej mamy ustalony rozmiar nowej tablicy i możemy przydzielić jej pamięć, sprawdzając jednocześnie, czy alokacja się powiodła:

```

nowa = malloc((*m)*sizeof(*nowa));
if(nowa==NULL)
{
    printf("Alokacja zakończona niepowodzeniem.\n");
    return NULL;
}

```

Proces wypełnienia tablicy wartościami przebiega analogicznie do zliczania wartości unikalnych. Tym razem tylko wstawiamy wartość do nowej tablicy. Zmienna `k` przechowuje indeksy elementów nowej tablicy:

```

for(i=0, k=0; i<n; i++)
{
    wystepuje = false;
    for(j=0; j<i; j++)
        if (t[i] == t[j])
    {
        wystepuje = true;
        break;
    }
    if(!wystepuje)
        nowa[k++] = t[i];
}

```

Pelny kod funkcji, uzupełniony o zwarcanie adresu nowej tablicy, przedstawia się zatem następująco:

```

int* bezPowtorzen(int *t, int n, int *m)
{
    int *nowa = NULL;
    int i, j, k;
    bool wystepuje = false;

    for(i=0, (*m)=0; i<n; i++)
    {
        wystepuje = false;
        for(j=0; j<i; j++)
            if (t[i] == t[j])
        {
            wystepuje = true;
            break;
        }
        if(!wystepuje)
            (*m)++;
    }
}

```

```

}

nowa = malloc((*m)*sizeof(*nowa));
if(nowa==NULL)
{
    printf("Alokacja zakończona niepowodzeniem.\n");
    return NULL;
}
for(i=0, k=0; i<n; i++)
{
    wystepuje = false;
    for(j=0; j<i; j++)
        if (t[i] == t[j])
    {
        wystepuje = true;
        break;
    }
    if(!wystepuje)
        nowa[k++] = t[i];
}

return nowa;
}

```

Jeżeli dokładnie mu się przyjrzymy, zauważymy, że pewien jego fragment występuje dwukrotnie. Jest to sprawdzanie unikalności elementu. Zastanówmy się, czy można tego uniknąć. Nie możemy przydzielić pamięci na całą tablicę, nie znając jej rozmiaru. Nie możemy wpisać wszystkich elementów tablicy, nie przydzieliwszy wcześniej na nią pamięci. Możemy jednak przydzielać pamięć w razie potrzeby, czyli w momencie, gdy znajdziemy unikalną wartość i będziemy ją chcieli wstawić do tablicy. Wykorzystamy do tego nową funkcję `realloc` z biblioteki `<stdlib.h>`.

Zaczynamy od wpisania pierwszego elementu tablicy źródłowej do nowej tablicy. Będzie on na pewno unikalny. Najpierw jednak musimy przydzielić na niego pamięć, ustawiając jednocześnie rozmiar nowej tablicy na 1.:

```

(*m) = 1;
nowa = malloc(sizeof(*nowa));
nowa[0] = t[0];

```

Następnie w taki sam sposób, jak wcześniej iterujemy po tablicy źródłowej, sprawdzając unikalność bieżącego elementu. Tym razem jednak zaczynamy

od drugiego elementu tablicy (o indeksie 1, ponieważ pierwszy już wykorzystaliśmy):

```
for(i=1, k=1; i<n; i++)
{
    wystepuje = false;
    for(j=0; j<i; j++)
        if (t[i] == t[j])
    {
        wystepuje = true;
        break;
    }
    if(!wystepuje)
    {
        //...
    }
}
```

W przypadku, gdy element ma być wstawiony do tablicy, wykonujemy następujące instrukcje. Zwiększamy dotychczasowy rozmiar tablicy:

```
(*m)++;
```

Dokonujemy tzw. realokacji pamięci:

```
nowa = realloc(nowa, (*m)*sizeof(*nowa));
```

Wstawiamy nowy element:

```
nowa[k++] = t[i];
```

Działanie funkcji `realloc` jest proste. Zwraca ona wskaźnik do bloku pamięci o pożądanej wielkości (lub `NULL`, gdy zabrakło pamięci). Istnieje możliwość, że będzie miał on inną wartość niż dotychczasowy adres, jeżeli bowiem będziemy zwiększać obszar pamięci, a za zaalokowanym aktualnie obszarem nie będzie wystarczająco dużo wolnego miejsca, to funkcja znajdzie nowe miejsce i przekoniuje tam starą zawartość. Nie jest to efektywne rozwiązanie z punktu widzenia optymalizacji (jest kosztowne czasowo) i jego nadużywanie nie jest dobrą praktyką.

Druga wersja naszej funkcji będzie zatem wyglądać następująco:

```
int* bezPowtorzen2(int *t, int n, int *m)
{
    int *nowa = NULL;
    int i, j, k;
    bool wystepuje = false;
```

```

(*m) = 1;
nowa = malloc(sizeof(int));
nowa[0] = t[0];
for(i=1, k=1; i<n; i++)
{
    wystepuje = false;
    for(j=0; j<i; j++)
        if (t[i] == t[j])
    {
        wystepuje = true;
        break;
    }
    if(!wystepuje)
    {
        (*m)++;
        nowa = realloc(nowa, (*m)*sizeof(*nowa));
        nowa[k++] = t[i];
    }
}

return nowa;
}

```

Po uzupełnieniu programu o funkcje `tworzLosowo` oraz `wypiszTablice` z po-przedniego zadania otrzymamy następujący kod:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
int* tworzLosowo(int n)
{
    int *t = (int*)malloc(n*sizeof(*t));
    if (t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem.\n");
        return NULL;
    }
    int i;
    srand(time(0));
    for(i=0; i<n; i++)
        t[i] = rand()%5+1;
}

```

```

        return t;
    }
void wypiszTablice(int *t, int n)
{
    int i;

    for(i=0; i<n; i++)
        printf("%4d", t[i]);
    printf("\n");
}

int* bezPowtorzen1(int *t, int n, int *m)
{
    int *nowa = NULL;
    int i, j, k;
    bool wystepuje = false;

    for(i=0, (*m)=0; i<n; i++)
    {
        wystepuje = false;
        for(j=0; j<i; j++)
            if (t[i] == t[j])
            {
                wystepuje = true;
                break;
            }
        if (!wystepuje)
            (*m)++;
    }

    nowa = malloc((*m)*sizeof(*nowa));
    if (nowa==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem.\n");
        return NULL;
    }
    for(i=0, k=0; i<n; i++)
    {
        wystepuje = false;
        for(j=0; j<i; j++)
            if (t[i] == t[j])

```

```

    {
        wystepuje = true;
        break;
    }
    if(!wystepuje)
        nowa[k++] = t[i];
}

return nowa;
}

int* bezPowtorzen2(int *t, int n, int *m)
{
    int *nowa = NULL;
    int i, j, k;
    bool wystepuje = false;

    (*m) = 1;
    nowa = malloc(sizeof(int));
    nowa[0] = t[0];
    for(i=1, k=1; i<n; i++)
    {
        wystepuje = false;
        for(j=0; j<i; j++)
            if (t[i] == t[j])
            {
                wystepuje = true;
                break;
            }
        if(!wystepuje)
        {
            (*m)++;
            nowa = realloc(nowa, (*m)*sizeof(*nowa));
            nowa[k++] = t[i];
        }
    }

    return nowa;
}

int main()
{

```

```

int n, *t, n2, *t2, n3, *t3;
do
{
    printf("Podaj rozmiar tablicy: ");
    scanf("%d", &n);
}while(n<=0);
t = tworzLosowo(n);
if(t==NULL)
    return -1;
wypiszTablice(t, n);

t2 = bezPowtorzen1(t, n, &n2);
wypiszTablice(t2, n2);

t3 = bezPowtorzen2(t, n, &n3);
wypiszTablice(t3, n3);

return 0;
}

```

### 7.3 Zadanie 3 (splice)

*Napisz funkcję, która na wejściu otrzymuje tablicę, zaś na wyjściu zwraca jej kopię, z której wskazany fragment został usunięty i zastąpiony zawartością innej tablicy.*

Ponieważ zadanie nie wskazało nagłówka funkcji, w ramach pierwszego kroku zastanówmy się, jak powinien on wyglądać. To też uświadomi nam, jakie operacje mają być wykonane na przekazanej tablicy. Funkcję nazwiemy `splice`, gdyż tak właśnie nazywany jest przeważnie ten rodzaj operacji:

`splice(`

Po pierwsze – określenie wejściowej tablicy (potrzebny nam jest wskaźnik na pierwszy element oraz jej rozmiar):

`splice(int *ar1, int len1,`

Następnie – wskazanie, który fragment ma zostać usunięty. Niech będzie to indeks pierwszej wartości do usunięcia oraz liczba usuwanych wartości:

`splice(int *ar1, int len1, int idx, int count,`

Na koniec – druga tablica. Ta, która ma zostać wstawiona w miejsce usuwanych elementów:

```
splice(int *ar1, int len1, int idx, int count,
       int *ar2, int len2)
```

Ponieważ funkcja ma zwracać tablicę – jako typ zwracany podajemy wskaźnik na `int`:

```
int *splice(int *ar1, int len1, int idx, int count,
            int *ar2, int len2)
```

Zwróćmy uwagę, jak wszechstronna będzie to funkcja. Mając np. wejściową tablicę jak poniżej:

[2, 1, 3, 0, 5, 4],

możemy wskazać, że na miejsce 2 wartości, zaczynając od indeksu 2 (wartości 3, 0) ma być wstawiona zawartość tablicy [6, 7]. Rezultat będzie wtedy taki:

[2, 1, 6, 7, 5, 4].

Równie dobrze jednak w miejsce dwóch elementów możemy wstawić trzy (wtedy rozmiar tablicy się powiększy)

[2, 1, 6, 7, 8, 5, 4].

lub jeden (wówczas się zmniejszy)

[2, 1, 6, 5, 4]

Możemy też po prostu usunąć fragment i nic nie wstawić na jego miejsce (długość drugiej tablicy równa 0)

[2, 1, 5, 4]

lub wstawić coś bez usuwania (liczba usuwanych elementów równa 0)

[2, 1, 6, 7, 3, 0, 5, 4].

Jak jednak tego dokonać? Ponieważ zadanie wymaga od nas zwrócenia nowej tablicy, powinniśmy stworzyć ją dynamicznie (przydzielić na nią pamięć przy pomocy funkcji typu `malloc`):

```
int *ar3 = (int *)malloc(len3 * sizeof(int));
```

Zanim to zrobimy, musimy policzyć jej rozmiar. Ile elementów potrzebujemy? Tablica wejściowa ma długość `len1`, usuwamy z niej `count` elementów i dodajemy `len2` elementów z drugiej tablicy. A zatem:

```
int len3 = len1 - count + len2;
```

Teraz musimy wypełnić tablicę `ar3` wartościami pobranymi odpowiednio z tablic `ar1` oraz `ar2`. Ponieważ dopiero wartości od `idx` mają być usuwane, to początek możemy bezpiecznie przepisać z `ar1` bez żadnych zmian:

```
for (int i = 0; i < idx; i++)
    ar3[i] = ar1[i];
```

Jeśli tak wyglądała tablica `ar1`:

```
[2, 1, 3, 0, 5, 4],
```

natomiast jako `idx` podano 2, to w `ar3` powinny się teraz znaleźć 2 pierwsze wartości z `ar1`:

```
[2, 1, , , , ...]
```

Kolejny fragment ma być usunięty, ale w tej chwili ważniejsze jest, że w jego miejsce przepiszemy zawartość tablicy `ar2`:

```
for (int i = 0; i < len2; i++)
    ar3[idx + i] = ar2[i];
```

Ważne jest tutaj, aby zwrócić szczególną uwagę na indeksy. W tablicy `ar3` dotarliśmy już do elementu `idx`, natomiast od niego zaczynamy wypełnianie. Z tablicy `ar2` wartości pobieramy od indeksu 0. Wykonujemy tę operację aż do przepisania wszystkich elementów z `ar2` (czyli warunek to `i < len2`).

Jeśli w przykładzie tablica `ar2` wyglądała tak:

```
[6, 7, 8],
```

to w `ar3` powinno się teraz znaleźć:

```
[2, 1, 6, 7, 8, ...]
```

Ostatni krok to przepisanie pozostałych wartości z tablicy `ar1`. Te „pozostałe” wartości to te od indeksu `idx`, ale... z pominięciem tych, które chcemy „usunąć”. Usuwanie jest w cudzysłowie, bo oczywiście z tablicy wejściowej niczego nie usuwamy, ani jej nie modyfikujemy. To po prostu fragment tablicy, który chcemy opuścić.

```
for (int i = idx + count; i < len1; i++)
    ar3[i - count + len2] = ar1[i];
```

Pętla `for` idzie po indeksach tablicy `ar1` – dotarliśmy już do `idx`, opuszczamy `count`, zatem możemy zacząć przepisywanie od elementu `idx+count`. W naszym przykładzie, jeśli `idx=2` i `count=2`, to zaczynamy od indeksu 4, czyli wartości 5 z tablicy `ar1`:

```
[2, 1, 3, 0, 5, 4].
```

Zwróciły też uwagę, gdzie trafią te wartości do tablicy `ar3`:

```
ar3[i - count + len2] = ar1[i];
```

Z pewnością nie na tę samą pozycję, na której znajdowały się w tablicy `ar1`. Tak stałoby się tylko w przypadku, gdybyśmy dodawali tyle samo elementów, ile usunęliśmy. Jeśli usuwaliśmy elementy, to długość tablicy ulega skróceniu i wartości z końcówki znajdą się bardziej po lewej (odejmujemy `count`). Z kolei dodanie wartości powiększy tablicę i ostatnie wartości powędrują w prawo, by zrobić miejsce na zawartość `ar2` (dlatego dodajemy `len2`):

```
ar3[i - count + len2] = ar1[i];
```

Tak utworzona tablica zostanie zwrócona jako wynik funkcji:

```
return ar3;
```

Jeszcze jeden drobiazg. Któżkolwiek użyje naszej funkcji, otrzyma tablicę. Czy jednak będzie znał jej rozmiar, by móc się nią prawidłowo posłużyć (np. wypisać na ekranie)? Pewnie mógłby sam to sobie policzyć (skoro znałby rozmiary przekazywanych nam tablic i wartości wszystkich parametrów), jednak należy do dobrego zwyczaju, aby funkcja robiła wszystko, co do niej należy, włącznie z takimi obliczeniami. Jak jednak mamy zwrócić tę wartość? Nie możemy tego zrobić przy pomocy `return`, jako wynik funkcji zwracamy już bowiem tablicę `ar3`. Użyjmy zatem przekazywania przez adres. Dodamy jeszcze jeden parametr do nagłówka naszej funkcji:

```
int *splice(int *ar1, int len1, int idx, int count,
            int *ar2, int len2, int *length)
```

Parametr `length` posłuży nam do zwrócenia obliczonej długości wynikowej tablicy. Nie jest on typu `int` (wówczas nic byśmy nie działały), ale typu `int*` – wskaźnik na `int`. Jest to adres, pod który możemy wpisać wyliczoną wartość:

```
if (length != NULL)
    *length = len3;
```

Warunek jest jedynie dla bezpieczeństwa – gdyby użytkownik funkcji nie przekazał nam adresu. Prawdopodobnie podobne warunki warto byłoby dodać i do innych zmiennych, aby upewnić się że nie próbujemy dokonać niemożliwego (np. przepisywać wartości z tablicy o ujemnej długości).

Cała funkcja będzie zatem wyglądać tak:

```
int *splice(int *ar1, int len1, int idx, int count,
            int *ar2, int len2, int *length)
{
    // nowa tablica - wynikowa
    int len3 = len1 - count + len2;
    int *ar3 = (int *)malloc(len3 * sizeof(int));
    // przepisujemy pierwszy fragment:
```

```

for (int i = 0; i < idx; i++)
    ar3[i] = ar1[i];
// przepisujemy wstawiany fragment:
for (int i = 0; i < len2; i++)
    ar3[idx + i] = ar2[i];
// przepisujemy ostatnie wartości:
// w tablicy ar1 dotarliśmy do idx,
// ale opuszczamy count elementów
// w tablicy ar3 ustawianie zaczynamy od idx+len2,
// bo tam skończyliśmy wpisywać elementy z ar2
for (int i = idx + count; i < len1; i++)
    ar3[i - count + len2] = ar1[i];
// przekażmy jeszcze rozmiar wynikowej tablicy
if (length != NULL)
    *length = len3;
// zwracamy wynik (przydzielony dynamicznie,
// zatem do usunięcia w main!)
return ar3;
}

```

I jeszcze przykład kodu testującego. Założmy, że stworzyliśmy sobie pomocnicze funkcje – do tworzenia tablicy i wypełniania jej losowymi wartościami:

```

int *randomarray(int length)
{
    int *result = (int *)malloc(length * sizeof(int));
    for (int i = 0; i < length; i++)
    {
        result[i] = rand() % 20;
    }
    return result;
}

```

Oraz do wypisywania zawartości tablicy na ekranie:

```

void printar(int *ar, int len)
{
    printf("[%d", ar[0]);
    for (int i = 1; i < len; i++)
    {
        printf(", %d", ar[i]);
    }
    printf("]\n");
}

```

Kod testujący może teraz wyglądać tak:

```
int *ar1 = randomarray(10);
int *ar2 = randomarray(4);
printar(ar1, 10);
printar(ar2, 4);
int len;
int *ar3 = splice(ar1, 10, 3, 2, ar2, 4, &len);
printar(ar3, len);
free(ar1);
free(ar2);
free(ar3);
```

Skoro wszystkie tablice zostały stworzone dynamicznie, nie powinniśmy zapominać o ich prawidłowym zwolnieniu (`free`). Podając inne parametry funkcji `splice`, możemy sprawdzić, czy prawidłowo zachowa się w sytuacji, gdy długość ulega zmniejszeniu (więcej usuwamy, niż wstawiamy):

```
int *ar3 = splice(ar1, 10, 3, 6, ar2, 4, &len);
```

lub gdy w ogóle nic nie usuwamy:

```
int *ar3 = splice(ar1, 10, 3, 0, ar2, 4, &len);
```

albo nic nie wstawiamy:

```
int *ar3 = splice(ar1, 10, 3, 2, NULL, 0, &len);
```

# Rozdział 8

## Typ strukturalny

### 8.1 Zadanie 1 (odległość punktów)

Dana jest następująca struktura:

```
struct punkt {  
    float x, y;  
};
```

Napisz program, który wczyta od użytkownika współrzędne dwóch punktów i wyświetli na ekranie odległość między nimi.

Język C umożliwia tworzenie własnych typów danych. Jednym z takich typów złożonych są struktury. Struktura może składać się z różnych typów zmiennych, również tablicowych. Nie tworzy rzeczywistego obiektu w pamięci, jedynie określa, z czego składa się taki obiekt. Deklaracja struktury wygląda następująco:

```
struct punkt {  
    float x, y;  
};
```

Słowo kluczowe **struct** mówi nam, że będziemy mieli do czynienia ze strukturą. Dalej następuje nazwa, w naszym przypadku jest to **punkt**. Następnie, w nawiasach klamrowych, znajduje się lista pól struktury. Każde pole musi zostać prawidłowo zadeklarowane i kończyć się średnikiem. Pole może być dowolnego typu – nawet inną strukturą (mamy wtedy do czynienia ze strukturą zagieźdzoną). Na końcu deklaracji struktury musi pojawić się średnik!

Zmienna strukturalna deklarowana jest podobnie do „zwykłej” zmiennej typu np. **int**. W naszym zadaniu wygląda tak:

```
struct punkt p1, p2;
```

Przetwarzając taką linię, kompilator tworzy zmienne **p1** i **p2**, rezerwując łącznie w pamięci miejsce dla czterech zmiennych zmiennoprzecinkowych typu **float** (po dwie na każdą zmienną strukturalną). Zmienne strukturalne, podobnie jak każde inne (również tablice), możemy zainicjować w momencie deklaracji:

```
struct punkt p = {12.5, 0.0};
```

W naszym zadaniu będziemy jednakże wczytywali współrzędne punktów z klawiatury. W tym celu musimy uzyskać dostęp do poszczególnych elementów zmiennej strukturalnej. Jej elementy są wskazywane za pomocą indeksu – czyli nazwy pola, do którego chcemy się odwołać, a na pole wskazuje symbol '.'.

```
scanf("%f%f", &p1.x, &p1.y);
scanf("%f%f", &p2.x, &p2.y);
```

Powyższy kod wczytuje od użytkownika kolejne współrzędne punktów. Zwróć uwagę na zapis `&p1.x` i analogiczne. Wskazuje on, że chcemy dostać się do składowej `x` zmiennej strukturalnej `p1` i pod jej adres wstawić pobraną wartość.

Odległość pomiędzy dwoma punktami obliczymy, korzystając z następującego wzoru:

```
sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y))
```

Funkcja `sqrt` z biblioteki `<math.h>` oblicza pierwiastek kwadratowy z wartości przekazanej w parametrze. Tym parametrem jest natomiast wyrażenie równe sumie kwadratów różnic pomiędzy współrzędnymi obydwu punktów. Zmienne strukturalne, jak każde inne, możemy przekazywać pomiędzy funkcjami. Stwórzmy zatem funkcję, która jako parametry przyjmie dwa punkty i zwróci odległość między nimi:

```
float odleglosc(struct punkt A, struct punkt B)
{
    return sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y));
}
```

Wywołanie tej funkcji będzie wyglądać następująco:

```
odleglosc(p1, p2)
```

Lącząc powyższe fragmenty instrukcji, otrzymamy następujący kod:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct punkt
{
    float x, y;
};

float odleglosc(struct punkt A, struct punkt B)
{
    return sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y));
```

```

}

int main()
{
    struct punkt p1, p2;
    printf("Wprowadź dane pierwszego punktu: \n");
    scanf("%f%f", &p1.x, &p1.y);
    printf("Wprowadź dane drugiego punktu: \n");
    scanf("%f%f", &p2.x, &p2.y);

    printf("Odległość między punktami wynosi %.2f.", 
           odleglosc(p1, p2));
    return 0;
}

```

## 8.2 Zadanie 2 (tablica punktów)

*Dana jest następująca struktura:*

```

struct punkt {
    float x, y;
};

```

*Napisz funkcję, która utworzy i wypełni danymi podawanymi od użytkownika tablicę n punktów. Wartość n przekaż jako parametr funkcji. Następnie stwórz funkcję, która wypisze stworzoną tablicę na ekranie. Napisz również kod programu pokazującego działanie tych funkcji.*

Parametrem funkcji tworzącej i wypełniającej tablicę będzie liczba punktów, zaś zwracaną wartością adres tablicy:

```
struct punkt* tworzTablice(int n);
```

Deklarujemy zmienną wskaźnikową, przydzielając jednocześnie pamięć:

```
struct punkt *t = (struct punkt*)malloc(n*sizeof(struct punkt));
```

W przypadku niepowodzenia kończymy funkcję zwracając NULL:

```

if(t==NULL)
{
    printf("Alokacja zakończona niepowodzeniem.\n");
    return NULL;
}

```

Następnie pobieramy współrzędne punktów, wpisując je do tablicy:

```

for(i=0; i<n; i++)
{
    printf("Wprowadź punkt nr %d: \n", i);
    scanf("%f%f", &t[i].x, &t[i].y);
}

```

I zwracamy adres pierwszego elementu tablicy:

```
return t;
```

Pełny kod funkcji wygląda następująco:

```

struct punkt* tworzTablice(int n)
{
    struct punkt *t = (struct punkt*)malloc(n*sizeof(struct punkt));
    if(t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem.\n");
        return NULL;
    }
    int i;
    for(i=0; i<n; i++)
    {
        printf("Wprowadź punkt nr %d: \n", i);
        scanf("%f%f", &t[i].x, &t[i].y);
    }
    return t;
};

```

Kolejną funkcją jest funkcja wypisująca tablicę na ekranie. Przyjmuje dwa parametry: adres tablicy oraz jej rozmiar. Nie zwraca żadnej wartości, więc jej typ ustawiamy na void.

```
void wypiszTablice(struct punkt *t, int n)
```

Przechodzimy przez wszystkie elementy tablicy, wyświetlając je pojedynczo na ekranie:

```

for(i=0; i<n; i++)
    printf("(%.4.1f, %.4.1f)\n", t[i].x, t[i].y);

```

Następnie w funkcji głównej tworzymy potrzebne zmienne i ustawiamy rozmiar tablicy:

```

struct punkt *t;
int n;
do
{

```

```
printf("Podaj liczbę punktów: ");
scanf("%d", &n);
}while(n<=0);
```

Mając pobrany rozmiar, tworzymy tablicę punktów:

```
t = tworzTablice(n);
if(t==NULL)
    return -1;
```

I wypisujemy ją na ekranie:

```
wypiszTablice(t, n);
```

Na koniec zostało jeszcze zwolnienie pamięci:

```
free(t); t = NULL;
```

Składając powyższe instrukcje w całość, otrzymujemy pełny kod programu:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct punkt
{
    float x, y;
};

struct punkt* tworzTablice(int n)
{
    struct punkt *t = (struct punkt*)malloc(n*sizeof(struct punkt));
    if(t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem.\n");
        return NULL;
    }
    int i;
    for(i=0; i<n; i++)
    {
        printf("Wprowadź punkt nr %d: \n", i);
        scanf("%f%f", &t[i].x, &t[i].y);
    }
    return t;
};
```

```

void wypiszTablice(struct punkt *t, int n)
{
    int i;
    for(i=0; i<n; i++)
        printf("(%.4f, %.4f)\n", t[i].x, t[i].y);
}

int main()
{
    struct punkt *t;
    int n;

    do
    {
        printf("Podaj liczbę punktów: ");
        scanf("%d", &n);
    }while(n<=0);

    t = tworzTablice(n);
    if(t==NULL)
        return -1;
    wypiszTablice(t, n);

    free(t); t = NULL;
    return 0;
}

```

### 8.3 Zadanie 3 (radar)

*Dana jest następująca struktura:*

```

struct punkt {
    float x, y;
};

```

*Korzystając z tablicy utworzonej w poprzednim zadaniu, napisz funkcję, która przepisze do nowej tablicy te punkty, które znajdują się w zasięgu radaru o zadanym środku i promieniu. Nową tablicę wyświetl na ekranie.*

Zacznijmy od stworzenia nagłówka funkcji `znajdzPunkty`:

```
struct punkt* znajdzPunkty(struct punkt s, float r,
                           struct punkt *t, int n, int *m)
```

Funkcja zwraca wskaźnik na strukturę `punkt`. Będzie to adres pierwszego elementu stworzonej tablicy. Przyjmuje pięć parametrów. Dwa pierwsze tyczą się radaru – współrzędnych jego środka oraz długości promienia. Kolejna para parametrów to informacje o istniejącej tablicy punktów – wskaźnik na jej początek oraz rozmiar. Ostatni parametr, wskaźnik na zmienną typu `int`, posłuży nam do wyniesienia z funkcji rozmiaru nowej tablicy.

Deklarujemy zmienne:

```
struct punkt *wewnatrz = NULL;
int i, j;
```

Zanim przydzielimy pamięć na nową tablicę, musimy wyznaczyć jej rozmiar. W tym celu przeszukamy tablicę wejściową w poszukiwaniu punktów leżących w zasięgu radaru. Korzystamy tu z funkcji `odleglosc` z wcześniejszego zadania. W przypadku, gdy odległość między  $i$ -tym elementem tablicy a środkiem radaru nie przekracza długości jego promienia, zwiększamy wartość zapisaną pod adresem ze zmiennej `m`. (Dla przypomnienia zmienność `m` jest wskaźnikiem, czyli przechowuje pewien adres innej zmiennej, w tym wypadku przekazanej z zewnątrz funkcji. Aby dostać się do tej zmiennej, musimy dokonać operacji wyłuskania, czyli używamy zapisu `*m`. Nawiąsy wskazują kolejność operatorów: najpierw wyłuskanie, a potem inkrementacja).

```
for(i=0, (*m)=0; i<n; i++)
    if(odleglosc(t[i], s)<=r)
        (*m)++;
```

Może się zdarzyć, że żaden z punktów w tablicy nie znajdzie się w zasięgu radaru. W takim przypadku (gdy `(*m)==0`) wychodzimy z funkcji, zwracając wartość `NULL`. Oznaczać to będzie, że nowa tablica jest pusta.

```
if ((*m)==0)
    return NULL;
```

W przeciwnym wypadku alokujemy pamięć na nową tablicę:

```
wewnatrz = (struct punkt*)malloc((*m)*sizeof(struct punkt));
```

Sprawdzamy, czy przydział pamięci się powiodł:

```
if(wewnatrz==NULL)
{
    printf("Alokacja zakończona niepowodzeniem.\n");
    return NULL;
}
```

Jeśli nie, to opuszczamy funkcję. W przeciwnym wypadku wpisujemy elementy do tablicy:

```
for(i=0, j=0; i<n; i++)
    if(odleglosc(t[i], s)<=r)
        wewnatrz[j++] = t[i];
```

i zwracamy adres stworzonej tablicy:

```
return wewnatrz;
```

Zakładając, że w funkcji `main` zadeklarowaliśmy:

```
// liczba punktów
int n;
// adres tablicy punktów pobranych od użytkownika
struct punkt *t=NULL;
// koordynaty środka radaru
struct punkt srodek;
// zasięg radaru
float r;
// rozmiar tablicy z punktami wewnątrz radaru
int m;
// adres tablicy z punktami wewnątrz radaru
struct punkt *wzasiegu=NULL;
```

i nadaliśmy im odpowiednie wartości, wywołanie naszej funkcji wyglądać będzie następująco:

```
wzasiegu = znajdzPunkty(srodek, r, t, n, &m);
```

Przed wyświetleniem nowej tablicy na ekranie warto sprawdzić, czy na pewno została ona utworzona. W tym celu skorzystamy z faktu, że funkcja zwraca wartość `NULL` w przypadku, gdy żaden punkt nie znalazł się w zasięgu radaru. Do wypisania tablicy na ekranie użyjemy funkcji `wypiszTablice` z poprzedniego zadania:

```
if(wzasiegu!=NULL)
    wypiszTablice(wzasiegu, m);
else
    printf("Brak punktów.");
```

Zbierając powyższe fragmenty kodu w całość, otrzymamy poniższy program. Omówienie funkcji `tworzTablice`, `wypiszTablice` oraz `odleglosc` znajduje się we wcześniejszych zadaniach.

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

```

#include <math.h>

struct punkt
{
    float x, y;
};

struct punkt* tworzTablice(int n)
{
    struct punkt *t = (struct punkt*)malloc(n*sizeof(struct punkt));
    int i;
    if(t==NULL)
    {
        printf("Alokacja zakończona niepowodzeniem.\n");
        return NULL;
    }

    for(i=0; i<n; i++)
    {
        printf("Wprowadź punkt nr %d: \n", i);
        scanf("%f%f", &t[i].x, &t[i].y);
    }
    return t;
};

void wypiszTablice(struct punkt *t, int n)
{
    int i;
    for(i=0; i<n; i++)
        printf("(%.4.1f, %.4.1f)\n", t[i].x, t[i].y);
}

float odleglosc(struct punkt p1, struct punkt p2)
{
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}

struct punkt* znajdzPunkty(struct punkt s, float r, struct punkt *t,
                           int n, int *m)
{
    struct punkt *wewnatrz = NULL;
    int i, j;

```

```

for(i=0, (*m)=0; i<n; i++)
    if(odleglosc(t[i], s)<=r)
        (*m)++;
if ((*m)==0)
    return NULL;

wewnatrz = (struct punkt*)malloc((*m)*sizeof(struct punkt));
if(wewnatrz==NULL)
{
    printf("Alokacja zakończona niepowodzeniem.\n");
    return NULL;
}
for(i=0, j=0; i<n; i++)
    if(odleglosc(t[i], s)<=r)
        wewnatrz[j++] = t[i];

return wewnatrz;
};

int main()
{
    int n;
    do
    {
        printf("Podaj liczbę punktów: ");
        scanf("%d", &n);
    }while(n<=0);
    struct punkt *t = NULL;

    t = tworzTablice(n);
    if(t==NULL)
        return -1;
    wypiszTablice(t, n);

    struct punkt srodek, *wzasiegu=NULL;
    float r;
    int m;
    printf("Podaj koordynaty radaru: ");
    scanf("%f%f", &srodek.x, &srodek.y);
    printf("Podaj promień radaru: ");
    scanf("%f", &r);
}

```

```
wzasiegu = znajdzPunkty(srodek, r, t, n, &m);
if(wzasiegu!=NULL)
    wypiszTablice(wzasiegu, m);
else
    printf("Brak punktów.");
free(t); t = NULL;
free(wzasiegu); wzasiegu = NULL;
return 0;
}
```

# Rozdział 9

## Pliki tekstowe

### 9.1 Zadanie 1 (rozdzielanie danych)

Napisz funkcję, która mając dany plik tekstowy zawierający liczby całkowite, stworzy dwa pliki i do pierwszego z nich przepisze wszystkie wartości mniejsze od średniej, a do drugiego – większe bądź równe. Ścieżki do plików są przekazane jako parametry funkcji.

Zacznijmy od napisania nagłówka funkcji. Parametrami wejściowymi są nazwy trzech plików – jednego wejściowego, z którego odczytamy liczby, oraz dwóch wyjściowych, do których przepiszemy je zgodnie z treścią zadania.

```
void rozdziel(const char *nazwaWejscie, const char *nazwaMniejsze,  
              const char *nazwaWieksze)
```

Nazwa pliku przekazywana jest jako ciąg znaków, czyli wskaźnik na `char`. Atrybut `const` oznacza, że wartość ta nie ulegnie zmianie wewnątrz funkcji – jest traktowana jako wartość stała. Funkcję tę będziemy mogli wywołać przykładowo w ten sposób:

```
rozdziel("liczby.txt", "mniejsze.txt", "wieksze.txt");
```

Oczywiście zamiast samych nazw plików możemy podać ścieżki (pełne lub względne). Funkcja powinna odczytać zawartość pliku `liczby.txt`, a następnie przepisać je do plików o nazwach `mniejsze.txt` i `wieksze.txt`. W pliku wejściowym spodziewamy się ciągu liczb oddzielanych spacjami, enterami bądź innymi białymi znakami – nie powinno mieć to wpływu na działanie funkcji. Jego zawartość może być zatem taka:

```
7 17 3 12 5 6 9 13 15 18.
```

Zacznijmy od otwarcia pliku:

```
FILE *wejscie = fopen(nazwaWejscie, "r");
```

Funkcja `fopen` jako parametry przyjmuje nazwę pliku oraz kod mówiący o sposobie jego otwarcia i obsługiwanego. Litera "`r`" wskazuje, że jest to plik tekstowy otwarty w trybie „do odczytu”. Tego pliku nie będziemy modyfikować. Jako wynik funkcja zwracany jest uchwyt do pliku. Jest to wskaźnik na typ `FILE`. Nie musimy, a nawet nie powinniśmy, wniknąć w jego zawartość – ta zmienna służy nam jedynie jako klucz, przy pomocy którego odwołujemy się

do otwartego pliku, bowiem każda kolejna funkcja chcącą wykonać operację na pliku będzie posługiwać się właśnie tym kluczem (a nie jego nazwą).

Każdy otwarty plik, gdy już skończymy na nim pracować, należy zamknąć. W przeciwnym razie system operacyjny może blokować do niego dostęp innym programom, myśląc że wciąż go używamy, ewentualnie zmiany, które w nim wprowadziliśmy, mogą przepaść. Aby o tym nie zapomnieć, najlepiej jest napisać kod zamykający plik od razu po jego otwarciu (i umieścić na końcu funkcji):

```
fclose(wejscie);
```

Zawartość pliku będziemy wczytywać w pętli – liczba po liczbie. Może to wyglądać np. tak:

```
while(!feof(wejscie)) {  
    int liczba;  
    fscanf(wejscie, "%d", &liczba);  
    printf("%d, ", liczba);  
}
```

Pętla `while` sprawdza, czy nie dotarliśmy do końca pliku. Funkcja `feof` zwraca prawdę, gdy osiągnięty został koniec pliku (*end of file*). Wewnątrz pętli do odczytania liczby z pliku tekstowego używamy wariantu funkcji `scanf` – tym razem z przedrostkiem `f` oznaczającym plik. Różni się ona od `scanf` tym, że jako pierwszy parametr przyjmuje uchwyty do otwartego pliku tekstowego. Ponadto jej działanie jest identyczne – skanuje zawartość w poszukiwaniu znaków, które interpretuje zgodnie z podanym ciągiem formatującym (w tym przypadku `"%d"`). Odczytane informacje są wpisywane pod podany adres zmiennej (w tym przypadku jest to zmienna `liczba`). Podany kod powinien wypisać odczytane liczby na ekranie.

Warto wiedzieć, że funkcja `fscanf` ma jeszcze dodatkowe działanie – zwraca informację o tym, ile wartości udało się odczytać. W tym przypadku próbujemy odczytać jedną liczbę typu całkowitego – więc oczekujemy, że zwrócona zostanie wartość 1. Jeśli chcemy uczynić nasz kod bezpieczniejszym i mniej podatnym na błędy, warto sprawdzać, co zostanie zwrócone, zwłaszcza gdy odczytujemy zawartość plików. Możemy zatem dwie ostatnie linijki pętli zamienić na:

```
if (1 == fscanf(wejscie, "%d", &liczba) )  
    printf("%d, ", liczba);
```

Kiedy funkcja `fscanf` może zwrócić wartość mniejszą od 1? Na przykład w sytuacji, gdy w pliku dotrzymy do miejsca, gdzie nie ma liczb, ale jest jakiś tekst, którego nie będziemy mogli zinterpretować jako wartości całkowitej. Lub – bardzo typowy problem – gdy w pliku tekstowym po ostatniej liczbie postawiono znak enter lub spację. Po odczytaniu ostatniej wartości pętla `while`

sprawdza wówczas, czy dotarło do końca pliku. Odpowiedź brzmi nie – w pliku są jeszcze znaki, zatem można czytać dalej. Kolejne wywołanie `fscanf` nie zauważuje jednak żadnych liczb. Kontrolę wczytywania wartości możemy wpleść też do warunku samej pętli – skoro nie jesteśmy już w stanie odczytywać liczb z pliku, nie ma sensu próbować dalej i można zakończyć pętlę. Może to wyglądać np. tak:

```
int liczba;
while (EOF != fscanf(wejscie, "%d", &liczba))
{
    printf("%d ", liczba);
}
```

Funkcja `fscanf` zwróci nam `EOF` (*end of file*), jeśli przy próbie odczytania danych dotrzymy do końca pliku.

Poza plikiem wejściowym, z którego odczytujemy liczby, potrzebujemy jeszcze dwóch plików wyjściowych. Te otworzymy w trybie do zapisu:

```
FILE *mniejsze = fopen(nazwaMniejsze, "w");
FILE *wieksze = fopen(nazwaWieksze, "w");
```

Tryb `"w"` oznacza, że do pliku będziemy zapisywać dane „od czysta” – jeśli plik już istniał, to zostanie wyczyszczony, jeśli zaś nie istniał – zostanie stworzony. Oczywiście pamiętamy też o zamknięciu:

```
fclose(mniejsze);
fclose(wieksze);
```

Pomyślmy nad algorytmem działania. Powinniśmy wczytywać wartości jedna po drugiej i decydować, do którego z dwóch plików ją przepisać. Szkielet procedury będzie zatem wyglądał jak poniżej:

```
int liczba;
while (EOF != fscanf(wejscie, "%d", &liczba))
{
    if (...)

    {
        fprintf(mniejsze, "%d ", liczba);
    }
    else
    {
        fprintf(wieksze, "%d ", liczba);
    }
}
```

W pętli wczytujemy liczby i zależnie od tego, czy wartość jest mniejsza od średniej, czy też nie, zapisujemy ją do właściwego pliku. Ponieważ pliki docelowe są plikami tekstowymi, wartości zapisujemy przy pomocy funkcji `fprintf`, w parametrach przekazując uchwyty do pliku, string formatujący oraz wartość.

Pozostało określenie warunku instrukcji `if`. Aby sprawdzić, czy liczba jest mniejsza od średniej, musimy najpierw obliczyć, ile ta średnia wynosi. Nie zrobimy tego bez wczytania wszystkich liczb, zatem konieczne będzie dwukrotne przejście całego pliku – raz w celu obliczenia średniej i drugi raz w celu przepisania liczb do dwóch plików. Przed pętlą dodajemy zatem kod liczący średnią:

```
int liczba, ile = 0;
float srednia = 0;
while (EOF != fscanf(wejscie, "%d", &liczba))
{
    srednia += liczba;
    ile++;
}
srednia /= ile;
```

W celu obliczenia średniej musimy odczytać wszystkie liczby z pliku, aby określić, ile wynosi ich suma oraz ile tych liczb jest. Następnie musimy rozpocząć czytanie tego samego pliku od początku – moglibyśmy w tym momencie po prostu go zamknąć i otworzyć jeszcze raz albo użyć funkcji, która przesunie nas z powrotem na początek i umożliwi odczytanie wszystkich wartości jeszcze raz:

```
fseek(wejscie, 0, 0);
```

Teraz jesteśmy gotowi, aby dopisać warunek do instrukcji `if`:

```
while (EOF != fscanf(wejscie, "%d", &liczba))
{
    if (liczba < srednia)
    {
        fprintf(mniejsze, "%d ", liczba);
    }
    else
    {
        fprintf(wieksze, "%d ", liczba);
    }
}
```

Poniżej znajduje się pełen kod funkcji:

```

void rozdziel(const char *nazwaWejscie, const char *nazwaMniejsze,
              const char *nazwaWieksze)
{
    // uchwyty do plików:
    FILE *wejscie = fopen(nazwaWejscie, "r");
    FILE *mniejsze = fopen(nazwaMniejsze, "w");
    FILE *wieksze = fopen(nazwaWieksze, "w");

    // liczenie średniej
    int liczba, ile = 0;
    float srednia = 0;
    while (EOF != fscanf(wejscie, "%d", &liczba))
    {
        srednia += liczba;
        ile++;
    }
    srednia /= ile;

    // powrót na początek pliku
    fseek(wejscie, 0, 0);

    // rozdzielenie:
    while (EOF != fscanf(wejscie, "%d", &liczba))
    {
        if (liczba < srednia)
        {
            fprintf(mniejsze, "%d ", liczba);
        }
        else
        {
            fprintf(wieksze, "%d ", liczba);
        }
    }

    // zamykanie plików
    fclose(wejscie);
    fclose(mniejsze);
    fclose(wieksze);
}

```

## 9.2 Zadanie 2 (najwyższa punktacja)

W pliku tekstowym są zapisane dane o osobach i ich ocenach. Do pliku wynikowego należy wpisać nazwiska osób z najwyższą oceną.

Zaczniemy od nagłówka funkcji. Ponieważ mamy dwa pliki – wejściowy i wyjściowy – jako parametry przyjmiemy ścieżki do tych dwóch plików:

```
void oceny(const char *nazwaWoj, const char *nazwaWyj)
```

Krok pierwszy to otwarcie obu plików – jednego tylko do odczytu, drugiego do zapisu:

```
FILE *wejscie = fopen(nazwaWoj, "r");
FILE *wyjscie = fopen(nazwaWyj, "w");
```

Od razu też – podobnie jak w poprzednim zadaniu – dopiszmy kod zamykania plików:

```
fclose(wejscie);
fclose(wyjscie);
```

Pętla wczytywania danych, która będzie stanowić osią naszej funkcji, może wyglądać jak poniżej:

```
while (!feof(wejscie))
{
    char nazwisko[100];
    int ocena;
    if (2 != fscanf(wejscie, "%s %d", nazwisko, &ocena))
        break;
}
```

Każda linijka pliku zawiera nazwisko i liczbę punktów – zatem w każdym kroku pętli musimy odczytać obie te wartości. Warunek przerwania pętli jest zabezpieczeniem, na wypadek nieudanej próby ich odczytania przed końcem pliku.

Treść zadania mówi o zapisaniu do pliku wynikowego nazwisk osób (liczba mnoga) o najwyższej ocenie – może się bowiem zdarzyć, że więcej niż jedna osoba ma tę samą ocenę. Jak będzie wyglądać procedura naszego postępowania? Moglibyśmy najpierw, w pierwszym przebiegu przez plik, określić, jaka jest wartość maksymalna oceny, a następnie (zaczytując wczytywanie od początku) przepisać pasujące wartości. To podejście zostawmy jako ćwiczenie dla czytelnika, a tu spróbujmy postąpić w inny sposób – odczytując dane jednokrotnie. Zadeklarujmy (przed pętlą) zmienną, w której przechowamy wartość maksymalną. Za chwilę zastanowimy się, na jaką wartość powinniśmy ją zainicjować, na razie zostawmy to jako:

```
int max;
```

W samej pętli natomiast będziemy oceniać, czy wartość oceny właśnie odczytanej osoby jest większa, mniejsza czy też równa wartości maksymalnej.

Jeśli ocena ma tę samą wartość, sprawa jest jasna – znaleźliśmy kolejną osobę o największej wartości, zatem zapiszmy jej nazwisko do pliku wynikowego:

```
if (ocena == max)
{
    fprintf("%s\n", nazwisko);
}
```

Gdyby wartość oceny była mniejsza, sprawa również jest prosta – nie robimy nic. Osoby o niższej ocenie w tej chwili nas nie interesują, możemy przejść dalej. Co jednak, jeśli ocena jest większa?

```
if (ocena > max)
{
}
```

Przede wszystkim – oznacza to, że znaleźliśmy nową „wartość maksymalną”. Możemy ją zatem zapamiętać w zmiennej `max`, tak jak w standardowym algorytmie poszukiwania wartości maksymalnej:

```
max = ocena;
```

Oczywiście powinniśmy też wpisać do pliku wynikowego nazwisko osoby, którą właśnie odczytaliśmy – w końcu ma (jak dotąd) największą wartość. Ale uwaga – być może w trakcie przetwarzania wcześniejszej porcji pliku, już kogoś tam wpisaliśmy. Te osoby, które wydawały nam się największe, póki nie dotarliśmy do obecnej, większej od nich. Co powinniśmy z nimi zrobić? Oczywiście usunąć. W jaki sposób możemy opróżnić zawartość pliku? Najprostsze rozwiązanie to zamknąć i otworzyć go raz jeszcze:

```
fclose(wyjscie);
wyjscie = fopen(nazwaWyj, "w");
```

Ponownie otwierając plik w trybie `"w"`, jego poprzednia zawartość zostanie wymazana. Można do tego również użyć funkcji `freopen` (możesz o niej poczytać w dokumentacji). Teraz możemy dopisać tu naszą nową osobę:

```
fprintf("%s\n", nazwisko);
```

Ostatnia rzecz, o której należy zadbać, to początkowa wartość zmiennej `max`. Mamy tu kilka możliwych rozwiązań. Najbardziej uniwersalne, choć nieco komplikujące kod, to zrobienie pierwszego kroku poza pętlą – pierwsza wczytana osoba na pewno będzie miała ocenę „jak dotąd największą”, zatem możemy ją zanotować w zmiennej `max`, a nazwisko osoby wpisać do pliku

wyjściowego. Alternatywnie, jeśli znamy możliwy zakres ocen, możemy wstawić do zmiennej `max` najmniejszą z możliwych wartości. Jeśli zaś nie znamy zakresu – najmniejszą możliwą wartość danego typu, np.:

```
int max = INT_MIN;
```

Podsumowując, kompletny kod funkcji będzie wyglądał jak poniżej:

```
void oceny(const char *nazwaWej, const char *nazwaWyj)
{
    FILE *wejscie = fopen(nazwaWej, "r");
    FILE *wyjscie = fopen(nazwaWyj, "w");

    int max = INT_MIN;
    while (!feof(wejscie))
    {
        char nazwisko[100];
        int ocena;
        if (2 != fscanf(wejscie, "%s %d", nazwisko, &ocena))
            break;
        if (ocena > max)
        {
            // mamy nowego największego! otwieramy jeszcze raz
            fclose(wyjscie);
            wyjscie = fopen(nazwaWyj, "w");
            fprintf("%s\n", nazwisko);
            max = ocena;
        }
        else if (ocena == max)
        {
            // drugi tak samo duży
            fprintf("%s\n", nazwisko);
        }
    }

    fclose(wejscie);
    fclose(wyjscie);
}
```

### 9.3 Zadanie 3 (wyniki testów)

Jako parametr funkcji przekazana jest nazwa pliku z wynikami testu. Każda linijka pliku zawiera: imię i nazwisko studenta oraz następujące po nim N oddzielonych spacjami liter – odpowiedzi na poszczególne pytania testowe.

Jeden wiersz może zatem zawierać np:

```
Jan Kowalski a c d a b d c a a d
```

W drugim pliku zapisano prawidłowe odpowiedzi (również jako N liter).

Do trzeciego pliku należy wypisać nazwiska tych studentów, którzy udzielili przy najmniej 50% poprawnych odpowiedzi. W każdym wierszu powinny się znaleźć: imię, nazwisko oraz procent poprawnych odpowiedzi.

Zacznijmy standardowo – od nagłówka funkcji. Jako parametry przekażemy nazwy trzech plików.

```
void testy(const char *nazwaWejscie, const char *nazwaPrawidlowe,
           const char *nazwaWyjscie)
```

Następnie otwieramy każdy z plików: do odczytu z rozwiązaniami studentów oraz poprawnymi odpowiedziami i do zapisu z rezultatami testu.

```
FILE *wejscie = fopen(nazwaWejscie, "r");
FILE *wyjscie = fopen(nazwaWyjscie, "w");
FILE *pravidlowe = fopen(nazwaPrawidlowe, "r");
```

Pamiętamy też o zamknięciu przed wyjściem z funkcji:

```
fclose(wejscie);
fclose(wyjscie);
fclose(pravidlowe);
```

Podstawowa pętla wczytująca będzie podobna do tej z poprzednich zadań:

```
while (!feof(wejscie))
{
    char imie[50], nazwisko[100];
    if (EOF == fscanf(wejscie, "%s", imie))
        return;
    if (EOF == fscanf(wejscie, "%s", nazwisko))
        return;
```

Najpierw odczytujemy imię i nazwisko. Po nich następują odpowiedzi. Zadanie mówi o N odpowiedziach. Przyjmijmy, że jest to wartość stała, znana w momencie komplikacji programu:

```
#define N 10
```

Odpowiedzi danego studenta możemy zatem wpisać do N-elementowej tablicy znaków:

```
char odpowiedzi[N];
```

Uwaga – to nie napis, tylko tablica znaków. Nie wczytujemy tam zatem wartości, korzystając ze specyfikatora formatu "%s" tylko "%c" (kolejno, znak po znaku):

```
for (int i = 0; i < N; i++)
{
    if (EOF == fscanf(wejscie, " %c", odpowiedzi + i))
        break;
}
```

W powyższym kodzie warto zwrócić uwagę na dwie rzeczy. Pierwsza to string formatujący: " %c", który zaczyna się spacją. Pozwoli ona zignorować znaki białe znajdujące się przed znakiem, który chcemy wczytać. Jeśli bowiem odczytano najpierw imię i nazwisko, co może odczytać kolejny fscanf, gdybyśmy podali string formatujący "%c"? Oczywiście kolejny znak, czyli spację. Tego jednak chcielibyśmy uniknąć. Spacja w stringu formatującym skonsumuje tę początkową spację z pliku wejściowego i pozwoli przejść dalej – do faktycznych znaków z udzielonymi odpowiedziami.

Druga rzecz to sposób podania adresu zmiennej, do której chcemy wczytać dane. Oczywiście *i*-ta wartość powinna być wczytana pod *i*-ty indeks tablicy, czyli do odpowiedzi[i]. Moglibyśmy zatem podać &odpowiedzi[i]. Ponieważ samo odpowiedzi jest adresem (zmienna tablicowa to wskaźnik na pierwszy element tablicy), a dodanie wartości *i* przesuwa nas w pamięci na kolejne elementy tablicy – te zapisy są równoważne.

Kolejna rzecz do zrobienia to sprawdzenie, ile z udzielonych odpowiedzi jest poprawnych. Najlepiej byłoby przedtem wczytać klucz – zestaw prawidłowych odpowiedzi, który zapisany jest w osobnym pliku. Dzięki temu nie będziemy musieli robić tego co krok. Przed pętlą czytającą plik wejściowy umieszczaamy zatem:

```
char klucz[N];
for (int i = 0; i < N; i++)
{
    if (EOF == fscanf(prawidlowe, " %c", klucz + i))
        break;
}
```

Dane z pliku z odpowiedziami czytamy dokładnie tak samo, jak odpowiedzi studenta. Teraz jesteśmy w stanie porównać udzielone odpowiedzi z kluczem. Możemy to zrobić od razu w pętli czytającej te odpowiedzi:

```

int score = 0;
for (int i = 0; i < N; i++)
{
    if (odpowiedzi[i] == klucz[i])
        score++;
}

```

Jeśli znak jest taki sam jak w kluczu – uznajemy to za prawidłową odpowiedź. Ostatni krok to sprawdzenie, jaki uzyskaliśmy procent poprawnych odpowiedzi, i zapisanie wyniku do pliku wynikowego (bądź nie – gdyż zapisujemy tylko tych studentów, których wynik przekroczył limit 50%):

```

if (score >= N * 0.5)
{
    fprintf(wyjscie, "%s %s %.2f%%\n",
            imie, nazwisko, score * 100.0 / N);
}

```

Wynik (procent poprawnych odpowiedzi) wypisujemy z dokładnością do dwóch miejsc po przecinku (%.2f) wraz z symbolem procentu (%). W obliczeniu procentów użyliśmy rzeczywistej wartości zamiast całkowitej (100.0 a nie 100), aby wskutek konwersji wynik był wyliczany również jako wartość rzeczywista. Pełen kod funkcji przedstawiono poniżej:

```

void testy(const char *nazwaWejscie, const char *nazwaPrawidlowe,
           const char *nazwaWyjscie)
{
    FILE *wejscie = fopen(nazwaWejscie, "r");
    FILE *wyjscie = fopen(nazwaWyjscie, "w");
    FILE *pravidlowe = fopen(nazwaPrawidlowe, "r");

    // wczytajmy "klucz"
    char klucz[N];
    for (int i = 0; i < N; i++)
    {
        if (EOF == fscanf(pravidlowe, " %c", klucz + i))
            break;
    }

    // czytamy testy:
    while (!feof(wejscie))
    {
        char imie[50], nazwisko[100];
        if (EOF == fscanf(wejscie, "%s", imie))

```

```
        break;
if (EOF == fscanf(wejscie, "%s", nazwisko))
    break;
char odpowiedzi[N];
int score = 0;
for (int i = 0; i < N; i++)
{
    if (EOF == fscanf(wejscie, " %c", odpowiedzi + i))
        break;
// oceniamy:
if (odpowiedzi[i] == klucz[i])
    score++;
}
// czy jest zaliczenie?
if (score >= N * 0.5)
{
    fprintf(wyjscie, "%s %s %.2f%%\n",
            imie, nazwisko, score * 100.0 / N);
}
}

fclose(wejscie);
fclose(wyjscie);
fclose(prawidlowe);
}
```

# Rozdział 10

## Listy jednokierunkowe

### 10.1 Zadanie 1 (dodawanie)

Napisz funkcję, która doda pewną wartość jako  $i$ -ty element listy (gdzie  $i$  jest wskazaną pozycją, licząc od 0), np. dla listy:

2->4->1->3->5

jeżeli  $i=2$ , nowa wartość zostanie dodana pomiędzy 4 (pozycja 1) a 1 (pozycja 2):

2->4->**nowa**->1->3->5.

Pojedynczy element listy będzie reprezentowany przez następującą strukturę:

```
typedef struct element
{
    int value;
    struct element *next;
} element;
```

Jak widać, składa się ona z wartości, którą chcemy przechować (zmienna `value`), oraz wskaźnika na kolejny element listy (zmienna `next`). Lista zbudowana będzie zatem z takich właśnie elementów. Każdy z nich wskazuje na kolejny (czyli przechowuje jego adres) aż do elementu ostatniego, gdzie wartość `next` wynosi `NULL` (adres pusty).

Nagłówek funkcji może wyglądać następująco:

```
element *insertList(element *first, int idx, int value);
```

Na wejściu potrzebujemy wskaźnika na początek listy (parametr `first`), miejsca, w którym ma się znaleźć nasza wartość (parametr `idx`), oraz oczywiście tejże wartości (parametr `value`). Po co nam wartość zwracana – o tym za moment.

Pierwszym krokiem będzie utworzenie nowego elementu łańcucha. Posłużymy się funkcją `malloc`, która przydzieli pamięć na nowy element. Jako parametr podajemy rozmiar naszej struktury w bajtach. W rezultacie otrzymamy wskaźnik na nasz nowy element (czyli jego adres).

```
element *nowy = malloc(sizeof(element));
```

To w nim przechowamy naszą wartość, a zatem włożymy ją tam od razu:

```
nowy->value = value;
```

Teraz czas na najważniejsze zadanie: znalezienie miejsca, w które ma trafić nasza nowa wartość. Założymy, że, tak jak w przykładzie, chcemy umieścić element pod indeksem 2 we wskazanej liście:

2->4->1->3->5->NULL.

Ponieważ liczymy od 0, nasz nowy element ma trafić pomiędzy wartości 4 a 1. Jak ją odnaleźć? Nie jest to tablica, zatem nie możemy po prostu posłużyć się indeksem, nie mamy też żadnej wiedzy o tym, gdzie w pamięci rozmieszczone są poszczególne elementy łańcucha. Jedyny sposób to po prostu je policzyć. Musimy zatem przeiterować przez elementy listy, dopóki nie dotrzemy do szukanej pozycji. Może do tego posłużyć fragment kodu podobny do tego:

```
int i = 0;
while(first != NULL) {
    if(i == idx) {
        // to jest tutaj!
        printf("szukana wartość: %d\n", first->value);
        break;
    }
    i++;
    first = first->next;
}
```

To dość standardowy sposób postępowania z listami. Pierwszym krokiem jest wyzerowanie pomocniczej zmiennej, która posłuży nam do liczenia elementów:

```
int i = 0;
```

Następnie tworzymy pętlę odwiedzającą każdy element listy – dopóki wskaźnik nie trafi na jej koniec (pusty adres), wykonujemy „krok do przodu” – czyli przechodzimy na element下一个 względem obecnego.

```
while(first != NULL) {
    first = first->next;
}
```

Wewnątrz pętli podbijamy nasz licznik:

```
i++;
```

Gdy dotrzemy do właściwego numeru, oznacza to że znaleźliśmy element na którym nam zależało i możemy z nim zrobić to, co zaplanowaliśmy. Możemy też przerwać iterację, gdyż nie ma potrzeby przeszukiwać listy dalej.

```
if(i == idx) {
    // to jest tutaj!
    break;
}
```

W naszym wypadku być może lepiej będzie jednak tę pętlę zapisać w ten sposób:

```
for (int i = 0; i < idx; i++)
{
    first = first->next;
}
```

Skoro wiemy, którego elementu szukamy, wiemy też, ile kroków mamy wykonać – pętla typu `for` pasuje do tej sytuacji idealnie. Aby znaleźć drugi element listy, należy dwukrotnie wykonać operację przejścia do kolejnego elementu:

```
first = first->next;
```

Na początku `first` wskazuje na początek listy (czyli element o numerze 0), pierwsze wykonanie tej operacji przesunie nas na element o numerze 1, a kolejne – o numerze 2. Czy to jest to, czego szukaliśmy? Nie do końca. Założymy, że nasza lista wyglądała tak:

2->4->1->3->5->NULL.

Na początku `first` wskazywał na zerowy element (wartość 2), po wykonaniu dwóch kroków wskazuje na element o numerze 2 (czyli wartość 1). Właśnie w jego miejsce musimy wstawić nasz nowy element. Wstawienie nie może powodować jednak przerwania łańcucha. Elementy nadal muszą tworzyć całość – łącząc się zaznaczonymi na naszym diagramie strzałkami. Każdy element musi wskazywać na swojego następnika, ale i być wskazywanym przez swego poprzednika. Nie wystarczy, że strzałka prowadząca z naszego nowego elementu będzie wskazywać wartość 1. Strzałka prowadząca z 4 ma teraz wskazywać na naszą nową wartość. Tylko wtedy będziemy mogli mówić o faktycznym „wstawieniu” elementu do listy. Jednak teraz, mając do dyspozycji jedynie wskaźnik na 1, nie jesteśmy w stanie wrócić do 4 i zmodyfikować go w jakikolwiek sposób. Rozwiązanie jest proste – wystarczy zrobić o jeden krok mniej.

```
for (int i = 0; i < idx - 1; i++)
{
    first = first->next;
}
```

Teraz nasza iteracja zatrzymuje się o jedną pozycję wcześniej (`idx - 1`), czyli na elemencie o wartości 4. Mając do niego dostęp, możemy bez problemu dołączyć za nim kolejny element. Wymaga to od nas dwóch operacji. Najpierw ustawimy następnika naszego nowo utworzonego elementu:

```
nowy->next = first->next;
```

Stanie się nim element za naszym bieżącym elementem. Dokładnie tak, jak powinno być – skoro `first` wskazuje w tym momencie na wartość 4, to znaczy,

że za nim (`first->next`) znajduje się 1 – to właśnie będzie następnik naszej nowej wartości.

Gdyby okazało się, że `first` jest ostatnim elementem listy (czyli za nim jest już jedynie `NULL`), to nic nie szkodzi – nasz kod działa nadal. Nie interesuje nas, jaka wartość kryje się pod `first->next`. Jeśli jest to `NULL`, to ten `NULL` znajdzie się w `nowy->next`, czyli nasz nowy element będzie teraz ostatnim elementem listy.

Druga ze wspomnianych dwóch operacji to:

```
first->next = nowy;
```

Nasz nowy element ma się stać następnikiem elementu bieżącego, czyli w naszym przykładzie wartości 4. To faktycznie połączy wszystkie elementy w spójny łańcuch:

```
2->4->value->1->3->5->NULL.
```

Tu uwaga: tych operacji nie możemy wykonać w dowolnej kolejności. Gdybyśmy zamienili kolejność działań, wówczas operacja łącząca element 4 z nowym elementem sprawiłaby, że stracilibyśmy dalszą część listy – skoro za 4 znalazła się nowa wartość, to już nie wiemy, co było tam wcześniej. Nie mamy możliwości dotrzeć teraz do 1, a co za tym idzie, i kolejnych elementów łańcucha. To byłby błąd.

Czy pomyśleliśmy o wszystkim i czy ten kod nie zawiedzie nas w żadnej sytuacji? Warto rozważyć sytuacje graniczne. Na przykład, co będzie, jeśli ktoś poda wartość `idx = 0`. Gdzie mamy umieścić nasz nowy element? Nie znajdzie się on za zerowym elementem ani za pierwszym, ani za żadnym innym – ale przed elementem o indeksie 0. Stanie się zatem nowym początkiem listy. Napisany powyżej kawałek kodu nie zapewni nam takiego działania, ponieważ w nim umieszczałyśmy nowy element za pewnym innym elementem. Tę sytuację najlepiej będzie zatem rozstrzygnąć osobno.

```
if (idx == 0)
{
    // dodawanie na początek listy
}
else
{
    // dodawanie w środku listy
}
```

Jeśli `idx` wynosi 0, oznacza to, że chcemy dodać element na początku listy. W przeciwnym wypadku (`else`) wykonamy szukanie elementu `idx-1` i wstawianie elementu za nim – czyli dokładnie ten kod, który zapisaliśmy powyżej.

Dodawanie elementu na początku jest znacznie prostsze. Ponieważ to początek łańcucha, wystarczy nam ustawić jednej „strzałki” – tej prowadzącej z nowego elementu:

```
nowy->next = first;
```

Jeśli zatem lista wyglądała tak:

```
2->4->1->3->5->NULL,
```

teraz będzie wyglądać tak:

```
value->2->4->1->3->5->NULL.
```

Jest jeszcze jednak ważna rzecz. Opisana powyżej zmiana będzie niestety widoczna tylko wewnętrz funkcji. Ktakolwiek wywołał naszą funkcję, przekazał tam wskaźnik na początek listy. Na przykład tak:

```
element *first = NULL;  
// tu tworzymy listę  
  
insertList(first, 2, 10);  
  
// a tu chcemy nadal korzystać z listy
```

I pewnie po wykonaniu naszego kodu nadal będzie się nim posługiwał. Jeśli udało nam się wstawić cokolwiek do środka listy (`idx` był większy od 0), to w porządku. Poczynając od elementu pierwszego, po nitce do kłębka, dotrze do elementu, który wstawiliśmy, a potem do wszystkich kolejnych, aż do końca listy. A co, jeśli `idx` był równy 0? Niestety, `first` nadal będzie wskazywał na poprzedni początek listy, nie sposób się dowiedzieć, czy coś przed nim wstawiliśmy, czy też nie. Najprostszy sposób to zwrócić z funkcji nasz „nowy początek”. Teraz będziemy posługiwać się nią tak:

```
first = insertList(first, 2, 10);
```

Jakikolwiek byłby wskaźnik na pierwszy element, po wywołaniu funkcji dostaniemy jego nową wartość i nią będziemy się teraz posługiwać. Po to właśnie był nam wskaźnik na element jako wartość zwracana w nagłówku funkcji:

```
element *insertList(element *first, int idx, int value);
```

Dodajmy zatem kawałki kodu, które zwrócią poprawny wskaźnik na ten „nowy początek”. Jeśli dodaliśmy coś pod indeksem 0, nasz nowy element jest właśnie tym nowym początkiem:

```
return nowy;
```

Jeśli jednak dodaliśmy coś w środku (`else` naszego `if`), początek się nie zmienia – czyli możemy po prostu zwrócić `first`. Moglibyśmy, gdyby nie pewne przeoczenie. Nasza pętla zmieniła `first` i początek gdzieś nam przepadł. Oto odpowiedzialna operacja:

```
first = first->next;
```

Musimy zatem albo przechować gdzieś wskaźnik na ten „prawdziwy początek” listy, albo do iteracji nie używać `first`, ale jakiejś innej zmiennej pomocniczej, np. w ten sposób:

```
element *tmp = first;
for (int i = 0; i < idx - 1; i++)
{
    tmp = tmp->next;
}
nowy->next = tmp->next;
tmp->next = nowy;
```

Zmienna `tmp` będzie użyta w liczącej elementy pętli, a potem posłużymy się nią, by połączyć nowy element z jego poprzednikiem i następnikiem. Zmienna `first` pozostała natomiast niezmieniona. Możemy zatem ją zwrócić:

```
return first;
```

Cała funkcja będzie zatem wyglądać następująco:

```
element *insertList(element *first, int idx, int value)
{
    // tworzymy nowy
    element *nowy = malloc(sizeof(element));
    nowy->value = value;

    if (idx == 0)
    {
        // dodawanie na początku listy
        nowy->next = first;
        first = nowy;
        return nowy;
    }
    else
    {
        // dodawanie w środku listy
        element *tmp = first;
        for (int i = 0; i < idx - 1; i++)
        {
```

```

        tmp = tmp->next;
    }
    nowy->next = tmp->next;
    tmp->next = nowy;
    return first;
}
}

```

Czy na pewno zadbaliśmy o wszystko? Na pewno nie. Pozostało kilka wyjątkowych sytuacji do rozstrzygnięcia. Co będzie, jeśli ktoś przekaże NULL jako wskaźnik na pierwszy element? A co, jeśli indeks będzie większy (lub równy – gdyż liczymy od 0) niż liczba elementów listy? Można to uznać za „błędы użytkownika” (a raczej osoby korzystającej z naszej funkcji), ale być może dobrze byłoby zadbać, aby nie powodowały zawieszenia programu. Można sygnalizować to błędne działanie komunikatem albo może po prostu przewidzieć jakieś standardowe zachowanie w takich sytuacjach (np. podanie zbyt dużego indeksu spowoduje dodanie elementu na końcu listy, a przekazanie pustego wskaźnika może oznaczać pustą listę, a zatem nasza nowa wartość stanie się jej pierwszym i jedynym elementem). To ćwiczenie pozostawiam czytelnikowi.

Aby móc przetestować funkcję z zadania, przydadzą nam się trzy funkcje pomocnicze. Pierwsza to tworzenie listy:

```

element *randList(int n)
{
    element *first = NULL;
    for (int i = 0; i < n; i++)
    {
        element *nowy = malloc(sizeof(element));
        nowy->value = rand() % 20;
        nowy->next = first;
        first = nowy;
    }
    return first;
}

```

Jako parametr podajemy długość listy, a jako wynik otrzymujemy wskaźnik na jej pierwszy element. Elementy inicjowane są losowymi wartościami – to wygodne do testów, ponieważ sprawdzanie poprawności działania programu nie wymaga interakcji z użytkownikiem (i każdorazowo wprowadzania danych z klawiatury).

Druga funkcja: wypisująca zawartość listy na ekranie:

```

void printList(element *first)
{

```

```

while (first != NULL)
{
    printf("%d -> ", first->value);
    first = first->next;
}
printf("NULL\n");
}

```

To dość standardowy kod iterujący i wypisujący elementy. Strzałki dodane są jedynie jako dekoracja (i wskazanie że mamy do czynienia z listą).

Wreszcie funkcja usuwająca listę – skoro przydzieliśmy pamięć, należy zadbać również o jej zwolnienie:

```

void destroyList(element *first)
{
    while (first != NULL)
    {
        element *next = first->next;
        free(first);
        first = next;
    }
}

```

Poprawność działania możemy przetestować np. w poniższy sposób:

```

void main()
{
    srand(time(NULL));

    element *first = randList(6);

    printList(first);
    first = insertList(first, 2, 10);
    printList(first);
    first = insertList(first, 0, 20);
    printList(first);
    first = insertList(first, 8, 30);
    printList(first);

    destroyList(first);
}

```

Powyższy kod najpierw tworzy listę sześciu losowych wartości, następnie doda je do niej kolejne trzy wartości (w różnych miejscach), za każdym razem wypisując ją od nowa na ekran. Na końcu lista jest usuwana.

## 10.2 Zadanie 2 (usuwanie)

Napisz funkcję, która usunie z listy wszystkie wartości spoza zakresu  $[a, b]$ .

Zadanie opiera się na identycznej strukturze listy:

```
typedef struct element
{
    int value;
    struct element *next;
} element;
```

Jako argumenty wejściowe funkcji powinniśmy otrzymać wskaźnik na pierwszy element listy oraz wartości  $a$  i  $b$  – początek i koniec zakresu. Jako wynik funkcji zwrócimy wskaźnik na początek przefiltrowanej listy (czyli złożonej z elementów, które zdecydowaliśmy się zachować). Zapiszmy to zatem jako:

```
element *filter(element *first, int a, int b);
```

Zadanie wymaga sprawdzenia wartości każdego elementu listy, zatem jego trzonem będzie pętla iterująca przez wszystkie elementy:

```
while (first != NULL)
{
}
```

Gdzieś w środku znajdzie się pewnie:

```
    first = first->next;
```

Zaczekajmy jednak z tym jeszcze, bo być może nie zawsze będzie to takie proste. Najpierw zidentyfikujmy, czy mamy do czynienia z elementem, który chcielibyśmy zachować, czy też usunąć:

```
if (first->value >= a && first->value < b)
{
    // ten zachowamy
}
else
{
    // ten usuńemy
}
```

Zachowujemy wartości należące do zakresu od  $a$  do  $b$  (zwróć uwagę, że  $b$  nie należy do zachowywanych wartości – zgodnie z treścią zadania). Samo usuwanie/zachowywanie moglibyśmy teraz zrealizować na kilka sposobów – spróbujmy zrobić to tak – na początku programu stworzymy dodatkową zmienną:

```
element *keep = NULL;
```

Będzie to lista elementów, które chcemy zachować. Jeśli zatem zdecydujemy się jakiś element zatrzymać – przeniesiemy go na tę listę. Nasza bieżąca lista będzie natomiast stopniowo opróżniana. W zależności od decyzji element będzie albo usuwany, albo przenoszony na listę `keep`.

Usuwanie jest zatem proste:

```
element *tmp = first;
first = first->next;
free(tmp);
```

Usunięcie to zwolnienie pamięci (`free`), ale musimy się upewnić, że w kolejnym kroku odwiedzimy następny element.

Jeśli zachowujemy element – dodajemy go na drugą listę. Ta operacja może zależeć od tego, czy na liście już coś jest, czy też jest (tak jak deklarujemy ją na samym początku) jeszcze pusta. Jeśli jest pusta, dodanie tam pierwszego elementu jest proste:

```
if (keep == NULL)
{
    // to pierwszy, jaki zachowujemy
    keep = first;
    first = first->next;
    tail->next = NULL;
}
else
{
    // już mamy jakieś zachowane
}
```

Zachowujemy `first`, przesuwamy się na `first->next` i oczywiście ustawiamy, by następnikiem naszego zachowanego elementu (pierwszego a zarazem ostatniego elementu listy `keep`) był `NULL`.

A co, jeśli na liście już coś jest? Jeśli nie chcemy, by nasza funkcja odwróciła kolejność elementów, następny zachowywany element powinien powędrować na koniec listy `keep`. Jeśli mamy początek, koniec zawsze możemy znaleźć, jednak prościej (i wydajniej) będzie przechowywać gdzieś wskaźnik na ostatni element listy `keep`. Zadeklarujmy tę zmienną na początku funkcji:

```
element *tail = NULL;
```

Oczywiście na początku jest ona równa `NULL` – jeśli nie mamy początku listy `keep`, nie mamy też końca. Pierwsza wartość pojawi się tam, gdy dodamy do listy pierwszy element:

```
keep = tail = first;
```

Gdy na liście `keep` jest jeden element, jest on jednocześnie jej początkiem i końcem. Teraz możemy przystąpić do pisania `else` naszego `if`-a. Jeśli lista zawiera jakieś elementy, kolejny dodajemy do końca (czyli za ostatnim elementem):

```
tail->next = first;
```

To będzie teraz nasz nowy koniec listy:

```
tail = first;
```

Oczywiście musimy przesunąć się do kolejnego elementu do sprawdzenia:

```
first = first->next;
```

Ponieważ jest to teraz koniec listy, to wskaźnik `next` musi prowadzić nas do adresu pustego:

```
tail->next = NULL;
```

Tu uwaga: tych dwóch operacji nie moglibyśmy wykonać w odwrotnej kolejności. Choć pozornie może się wydawać, że nie mają na siebie wpływu, zwróćmy uwagę, że `tail` i `first` mają tę samą wartość. Wyzerowanie `tail->next` uniemożliwiłoby zatem prawidłowe przejście do następnika elementu `first`. Kompletna funkcja będzie wyglądać następująco:

```
element *filter(element *first, int a, int b)
{
    element *keep = NULL;
    element *tail = NULL;
    while (first != NULL)
    {
        if (first->value >= a && first->value < b)
        {
            // ten zachowamy
            if (keep == NULL)
            {
                // to pierwszy, jaki zachowujemy
                keep = tail = first;
                first = first->next;
                tail->next = NULL;
            }
            else
            {
                // już mamy jakieś zachowane
                tail->next = first;
                tail = first;
            }
        }
    }
}
```

```

        first = first->next;
        tail->next = NULL;
    }
}
else
{
    // ten usuniemy
    element *tmp = first;
    first = first->next;
    free(tmp);
}
// zwracamy wskaźnik na nowy początek listy:
return keep;

```

Napisanie kodu testującego może być dobrym ćwiczeniem dla czytelnika.

### 10.3 Zadanie 3 (usuwanie powtórzeń)

*Dana jest lista ocen filmów zdefiniowana przy pomocy poniższych struktur.*

```

struct movie
{
    char title[50]; // tytuł filmu
    float score;    // ocena w skali 0-10
};

typedef struct item
{
    struct movie mv;
    struct item *next;
} item;

```

*Należy zwrócić listę po usunięciu z niej powtórzeń. Jeśli jakiś film został oceniony więcej niż raz, wszystkie jego oceny zostaną zastąpione jedną – o wartości równej ich średniej.*

Zaczniemy zwyczajowo od napisania nagłówka funkcji:

```
void highscore(item *first)
```

To powinno nam wystarczyć – na wejściu przyjmujemy wskaźnik na początek listy. Nie musimy nic zwracać, ponieważ wszystkich zmian dokonamy w danych,

które zostały dostarczone (nie będziemy tworzyć nowych elementów ani zmieniać ich kolejności).

Podobnie jak w poprzednim zadaniu chcemy przejrzeć wszystkie elementy listy, zatem osią funkcji będzie pętla iterująca:

```
while (first != NULL)
{
    ...
    first = first->next;
}
```

Wyobraźmy sobie zatem, że zaczynamy przeglądanie naszej listy filmów i przechujemy właśnie na jej pierwszy element. Co powinniśmy z nim zrobić? Zachować? Usunąć? Zmodyfikować? Zostawić go, tak jak jest? Aby o tym zdecydować, musimy dowiedzieć się, co jest dalej. Czy to jest jedyny wpis na temat tego filmu? A może gdzieś w dalszej części listy mamy kolejne? A zatem, poczynając od elementu następnego, musimy przejrzeć całą listę w poszukiwaniu wpisów dotyczących tego samego filmu. To oznacza konieczność utworzenia pętli w pętli:

```
while (first != NULL)
{
    item *p = first->next;
    while (p != NULL)
    {
        p = p->next;
    }
    first = first->next;
}
```

Element `p` to jest nasz „kolejny element do sprawdzenia”. Zaczynamy od elementu następującego bezpośrednio po elemencie `first` i kontynuujemy sprawdzanie aż do końca listy. Będziemy szukać wpisu o filmie pod tym samym tytułem co `first`. Jeśli go znajdziemy – oznacza to, że taki wpis należy usunąć (a oceny uśrednić). Wszystkie pozostałe filmy pozostaną niezmienione.

Zwrócmy uwagę, że wewnętrzna pętla nie rozpoczyna się od początku listy, ale od naszej „aktualnej pozycji” iteracji zewnętrznej pętli. Jeśli pętla zewnętrzna dotarła już, powiedzmy, do połowy listy (`first` zmienia się w każdym kroku, a więc w tym momencie przestaje już wskazywać na rzeczywisty początek listy), to nie musimy sprawdzać, czy dany film występuje gdzieś w części, którą już raz sprawdziliśmy – skoro dotarliśmy aż tutaj, za każdym razem usuwając wszystkie powtórzenia, to znaczy, że nie możemy teraz trafić na tytuł, który znajduje się gdzieś za nami.

Co zatem mamy robić z elementem `p`? Powinniśmy sprawdzić, czy nie ma przypadkiem tego samego tytułu, co element `first`:

```

if (strcmp(p->mv.title, first->mv.title) == 0)
{
    // to ten sam!
}
else
{
    // to jakiś inny film
}

```

Uwaga: porównanie wykonujemy przy pomocy funkcji `strcmp` – nie możemy po prostu użyć operatora `==`, ponieważ chcemy porównać zawartość napisów (zaś napis jest po prostu tablicą znaków), a nie sam adres (wartość tablicy to adres jej pierwszego elementu).

Jeśli znaleźliśmy wpis o tym samym filmie, powinniśmy go usunąć. To oczywiście rodzi problem z ciągłością listy. Usunięcie elementu nie polega jedynie na wywołaniu `free` i zwolnieniu pamięci. Zachowana ma być kolejność elementów. Nasz poprzedni element powinien zacząć wskazywać na naszego następnika.

Jeśli kolejność elementów w liście była: A, B, C:

A->B->C->...,

to po usunięciu elementu B element A powinien wskazywać na element C:

A->C->...

Aby wprowadzić modyfikację na poprzednim elemencie, musimy mieć do niego wskaźnik. Najprościej będzie po prostu przechowywać dodatkową zmienną. Nasza pętla szukająca kolejnego wystąpienia danego filmu przybierze zatem postać:

```

item *p = first->next; // kolejny element do sprawdzenia
item *prev = first;    // poprzednik kolejnego
while (p != NULL)
{
    ...
}

```

Teraz chcąc usunąć odnaleziony element, mamy dość danych, aby to zrobić:

```

if (strcmp(p->mv.title, first->mv.title) == 0)
{
    // to ten sam, zatem usuwamy
    prev->next = p->next;
    free(p);
    p = prev->next;
}

```

Pierwsza linijka to przepięcie wskaźnika – element poprzedni (`prev`) wskazywał oczywiście na nasz element aktualny (`p`), teraz ma wskazywać na naszego następnika (`p->next`). Potem możemy bezpiecznie usunąć element `p`. Ostatni krok to aktualizacja zmiennej `p`. Nie możemy zrobić po prostu:

```
p = p->next;
```

ponieważ `p` zostało usunięte – nie byłoby to zatem bezpieczne. Wartość ta znajduje się jednak w `prev->next`, zatem to właśnie będzie nasz kolejny element do sprawdzenia. Wartość `prev` się nie zmieni:

```
A->B->C->...
```

Jeśli `B` to był nasz element do usunięcia (`p`), `A` to jego poprzednik (`prev`), to w kolejnym kroku:

```
A->C->...
```

elementem do sprawdzenia (i ewentualnego usunięcia) będzie `C`, natomiast `A` będzie nadal „poprzednikiem” (czyli `prev`).

Oczywiście zanim to zrobimy, powinniśmy zadbać o jeszcze jedną rzecz. Mieliśmy nie tylko usuwać powtarzające się filmy, ale też wyliczać średnią ocenę. To wymaga z naszej strony drobnej modyfikacji. Przed pętlą `while` po `p` dodajmy dwie zmienne:

```
float score = first->mv.score;
int scores = 1;
```

Użyjemy ich, aby policzyć średnią – potrzebujemy sumy ocen oraz liczby wystąpień danego filmu. Inicjujemy je na wartość oceny filmu, od którego zaczęliśmy, oraz 1 (bo jak dotąd mamy jeden film). Teraz w samej pętli, zanim usuniemy znaleziony film, zaktualizujmy wartość obu tych zmiennych:

```
score += p->mv.score;
scores++;
```

zaś po wyjściu z pętli obliczymy średnią i wstawmy do jedynego wystąpienia tego filmu, które zachowaliśmy:

```
first->mv.score = score / scores;
```

Ostatnią rzeczą jest dodanie `else` do warunku sprawdzającego tytuł filmu:

```
prev = p;
p = p->next;
```

Skoro dany film nie jest tym, którego szukaliśmy, przechodzimy do kolejnego. Zaktualizowane musi być nie tylko `p` (film aktualny), ale również `prev` (film poprzedni) – aby w razie potrzeby móc go poprawnie usunąć, tak jak opisano wcześniej.

Ostatnią rzeczą, na którą powinniśmy zwrócić uwagę, jest wartość zwracana. W tym wypadku ustawiliśmy ją na `void` – i tyle wystarczy. Gdyby zmieniał się początek naszej listy, byłaby konieczność jakiegoś przekazania tej informacji osobie wywołującej funkcję. W tym jednak wypadku taka sytuacja nie zaistnieje. Pierwszy film na liście zawsze zostanie zachowany – niezależnie od tego, czy jego tytuł powtarza się, czy też nie, pierwsze wystąpienie zachowujemy, usuwamy jedynie kolejne. Struktura listy może się zmieniać, jeśli jednak pierwszy element pozostaje ten sam.

Kompletny kod funkcji może wyglądać, tak jak poniżej:

```
void highscore(item *first)
{
    while (first != NULL)
    {
        float score = first->mv.score;
        int scores = 1;
        // najpierw sprawdzimy, czy za elementem first są
        // jakieś oceny tego samego filmu:
        item *p = first->next; // kolejny element do sprawdzenia
        item *prev = first;     // poprzednik kolejnego
        while (p != NULL)
        {
            if (strcmp(p->mv.title, first->mv.title) == 0)
            {
                // to ten sam!
                score += p->mv.score;
                scores++;
                // zatem usuwamy
                prev->next = p->next;
                free(p);
                p = prev->next;
            }
            else
            {
                prev = p;
                p = p->next;
            }
        }
        // aktualizacja oceny:
        first->mv.score = score / scores;
        first = first->next;
    }
}
```

