Politechnika Wrocławska

Bardzo szybkie podsumowanie: wykład 4

wer. 10 z drobnymi modyfikacjami!

Wojciech Myszka

2023-05-15 06:20:43 +0200

HR EXCELLENCE IN RESEARCH

Uwagi

1. Obowiązuje cały materiał!
2. Tu tylko podsumowanie.

Politechnika Wrocławska

Struktury danych, unie

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2023-04-24 10:25:18 +0200

Struktury danych

1. W Pascalu — rekordy.
2. Struktura „podobna” do tablicy, ale pozwalająca na przechowywanie danych różnych typów.
3. Przykłady:
   - Lista płac:
     - Imię
     - Nazwisko
     - PESEL
     - Numer konta bankowego
     - Kwota do wypłaty
     - ...
   - Współrzędne punktu:
     - współrzędna X
     - współrzędna Y
     - (ewentualnie) współrzędna Z

Zacznijmy od przykładu

Ułamki

Zadanie

- Chcemy (musimy?) napisać program–kalkulator wykonujący obliczenia na ułamkach „zwykłych”.
- Ułamki takie to liczby postaci:

$$\frac{p}{q}$$

  gdzie $p, q \in N$ oraz $q \neq 0$.
- Kalkulator będzie wykonywał operacje dodawania, odejmowania, mnożenia i dzielenia.
- Do każdej operacji musimy napisać funkcję.
- Dodatkowo potrzebne będą funkcje upraszczania ułamków i ich drukowania. Chcemy żeby wydruk ułamków wyglądał jak najbardziej naturalnie:

```
     17
-35---
    123
```

Sposób przechowywania danych

1. Pierwszą sprawą wymagającą naszej uwagi jest sposób przechowywania danych.
2. Najwygodniej (czemu?) przechowywać wszystkie dane, na których prowadzimy operacje jako ułamki (niewłaściwe)
3. Wydaje się, że najwygodniej będzie dane przechowywać jako dwie liczby:
   - licznik przechowujący również znak liczby
   - mianownik
4. Po każdej operacji ułamek będzie normalizowany (upraszczany).

Politechnika Wrocławska

Operacje

- dodawanie/odejmowanie

$$\frac{a}{b} \pm \frac{c}{d} = \frac{a * d \pm c * b}{b * d}$$

- mnożenie

$$\frac{a}{b} * \frac{c}{d} = \frac{a * c}{b * d}$$

- dzielenie

$$\frac{a}{b} / \frac{c}{d} = \frac{a * d}{b * c}$$

Uwagi

Ponieważ licznik przechowuje znak — w wyniku dzielenia może okazać się, że mianownik stanie się ujemny. Trzeba to sprawdzić i skorygować.
Do rozstrzygnięcia pozostaje kwestia działań podejmowanych gdy mianownik będzie równy zeru.

Naiwna próba realizacji

- Napiszmy funkcję suma.
- argumenty: wartości a, b, c, d (zmienne typu **int**)
- wynik: nie będzie mógł być przekazany poleceniem **return** — można tak przesłać tylko jedną zmienną
  czyli:
  - albo piszemy dwie funkcje do wyliczania wartości licznika i mianownika będziemy potrzebowali cztery funkcje do wyliczania liczników dla sumy, różnicy, iloczynu i ilorazu oraz dwie funkcje do wyliczania mianownika: jedną dla sumy, różnicy i iloczynu, oraz drugą dla ilorazu;
  - albo wyniki przekazujemy „przez adres” jako wskaźniki

  Drugie rozwiązanie wydaje się bardziej racjonalne

Politechnika Wrocławska

Suma

```
void suma ( int a, int b, int c, int d, int *p, int *q )
{
 *p = a * d + c * b;
 *q = b * d;
}
```

Użycie:

```
int a, b, c, d, e, f;
...
suma (a, b, c, d, &e, &f);
```

Politechnika Wrocławska

Czy można inaczej?

1. Możliwość posiadania zmiennej, która będzie mogła przechowywać więcej niż jedną wartość ułatwiłaby nasz problem.
2. Taki typ zmiennych istnieje w języku C i nazywa się **strukturą**.
3. Deklarujemy najpierw strukturę danych:

```
struct Ulamek
{
 int l;
 int m;
};
```

4. Używamy polecenia **typedef** aby łatwiej używać zmiennych

```
typedef struct Ulamek ulamek;
```

i możemy teraz deklarować zmienne w sposób następujący:

```
ulamek A, B, C;
```

Politechnika Wrocławska

Zalety i wady takiego postępowania

+ Mamy zmienną mogącą przechowywać więcej niż jedną wartość.
+ Przechowywane wartości mogą być różnego typu.
+ Ponieważ korzystamy z nowego typu — funkcje mogą używać tego typu we wszelkich obliczeniach.
– Nie można wykonywać operacji na takich zmiennych złożonych.
– Dostęp do składowych struktury jest nieco bardziej skomplikowany;
– Nie można (łatwo) nadawać wartości zmiennym tego typu (nie istnieją stałe typu strukturalnego).

Nowe sformułowanie funkcji suma

```
struct Ulamek
{
 int l;
 int m;
};
typedef struct Ulamek ulamek;
ulamek suma ( ulamek A, ulamek B )
{
 ulamek Wynik;
 Wynik.l = A.l * B.m + B.l * A.m;
 Wynik.m = A.m * B.m;
 return Wynik;
}
...
ulamek aa, bb, cc;
...
cc = suma ( aa, bb );
```

Politechnika Wrocławska

Co dalej?

1. Możemy wykonywać operacje złożone (**ulamek** aa, bb, cc;):

```
cc = iloczyn(suma(aa, bb), roznica(aa, bb))
```

2. Deklarując zmienną, można nadać jej wartość (ale tylko tu!):

```
ulamek aa = {1, 2}; bb = {3, 4}, cc;
```

3. Ma sens coś takiego:

```
int z = suma(aa, bb).l;
```

Politechnika Wrocławska

Struktury danych

Deklaracja

1. Deklaracja struktury, tak na prawdę, to deklaracja nowego typu danych!

```
struct point {
    int x;
    int y;
};
```

2. x i y to **składowe** struktury.
3. Deklaracja zmiennej:

```
struct point punkt1, punkt2, punkt3, pt;
```

4. Można też tak:

```
struct point3D {
    int x;
    int y;
    int z;
} p1, p2, p3;
```

   Można też w dwu etapach (jak w ułamku):

   4.1 najpierw definiujemy strukturę,
   4.2 później definiujemy nowy typ używając **typedef**.

Struktury

Dane

1. Inicjalizacja:

```
struct point maxpt = { 320, 200 };
```

2. W wyrażeniach dostęp uzyskuje się *nazwa_struktury.składowa*
3. Kropka to **operator** składowej struktury.
4. Przykłady:

4.1
```
printf("%d,%d", punkt1.x, punkt1.y);
```

4.2 Odległość między punktami

```
double dist, sqrt(double); /* sqrt: pierwiastek */
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Struktury

Przykład cd

- Zagnieżdzona struktura

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Gdy zadeklarujemy screen:

```
struct rect screen;
screen.pt1.x = 100;
```

nadaje wartośc współrzędnej x pierwszego punktu (pt1) struktury ekran

- Inicjacja struktury złożonej

```
struc rect ekran = { { 0, 0 }, { 1024, 768 }};
```

Struktury i funkcje I

1. Funkcja może zwracać daną typu strukturalnego.

```
/* makepoint: utworz punkt ze
              wspolrzednych x i y */
struct point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

   Użycie punkt2 = makepoint(100, 59);

2. Parametrami funkcji mogą być dane typu strukturalnego

Struktury i funkcje II

```
/* addpoint: dodaj dwa punkty */
struct point addpoint(struct point p1, \
                      struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

użycie:

```
int main (void)
{
        struct rect ekran = { { 0, 0 }, { 1024, 768 } };
        struct point srodek;
        srodek = makepoint(addpoint(ekran.pt1,
                                    ekran.pt2).x/2,\
                                    addpoint(ekran.pt1,
                                    ekran.pt2).y/2);
        printf("(%d,%d)", srodek.x, srodek.y);
```

Struktury i funkcje III

```
        return 0;
}
```

Politechnika Wrocławska

Struktury i wskaźniki I

1. W sytuacji gdy do funkcji jako daną mamy przekazać bardzo rozbudowaną strukturę (będzie ona **kopiowana** do zmiennej tymczasowej) lepiej jest użyć wskaźnika.
2. Deklaracja jak zwykle:

```
struct point origin, *pp;
```

   `origin` to struktura (o składowych x i y), `pp` to wskaźnik do struktury typu point; `(*pp).x` oraz `(*pp).y` to jej składowe.

3. Użycie

```
pp = &origin;
printf("punkt poczatkowy (%d,%d)\n", \
                        (*pp).x, (*pp).y);
```

Struktury i wskaźniki II

4. Ponieważ struktur używa się bardzo często podobnie jak wskaźników do nich, zeby uprościć życie wymyślono specjalną notację. Jeżli `p` jest wskaźnikiem do struktury to aby się dobrać do składowej, używamy takiej notacji: `p->skladowa_struktury`. Zatem poniższy zapis jest równoważny poprzedniemu:

```
printf("punkt poczatkowy (%d,%d)\n", \
                          pp->x, pp->y);
```

5. W przypadku struktur zagnieżdżonych sprawa nieco się komplikuje (to samo można zapisać na kilka różnych sposobów!):

```
struct rect r, *rp = &r;
```

Następujące wyrażenia są równoważne:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Struktury i wskaźniki III

6. Operatory strukturowe ., −> oraz nawiasy okrągłe () wywołania funkcji oraz nawiasy kwadratowe [] indeksowania tablicy mają najwyższy priorytet (najsilniej wiążą swoje argmenty).

Tablice struktur

1. Podobnie jak z danych innych typów, również ze struktur można tworzyć tablice (bazy danych???)

2. ```
   struct point dane[100];
   ```

   deklaruje tablicę o 100 elementach, z których każdy to dana typu point — struktura złożona z dwu składowych x i y.
   ```

Politechnika Wrocławska

Tablice struktur

Przykład

1. Piszemy program zliczający częstość występowania słów kluczowych języka C.
2. Dane: tablica, każdym jej elementem jest para (słowo kluczowe, liczba wystąpień) — struktura.
3. Schemat blokowy:
   - 3.1 Jeżeli koniec pliku — **skończ**.
   - 3.2 Wprowadź słowo (z pliku).
   - 3.3 Znajdź słowo w danych.
   - 3.4 Jezeli wystąpiło — zwiększ odpowiedni licznik.
   - 3.5 (Jeżeli nie wystąpiło — przechodzimy dalej.)
   - 3.6 Przejdź na początek.

Sam program znajduje się w [1]:


B. W. Kernighan, D. M. Ritchie.
Język ANSI C.
WNT, Warszawa, 2007.

Tablice struktur

Przykład — dane

1. Zakładamy, że słowa kluczowe uszeregowane są w kolejności rosnącej — ułatwi to wyszukiwanie.
2. Do wyszukiwania wykorzystamy metody szukania binarnego (podział na pół).
3. Ile miejsca w pamięci (bajtów) zajmuje struktura key?

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    /* ... */
    "volatile", 0,
    "while", 0
};
```

Pola bitowe

1. Pamięć jest tania i zazwyczaj nie ma na potrzeby jej oszczędzać.
2. Czasami chcemy jednak operować an bitach.
3. Definiujemy sobie coś takiego:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

   ta 1 (po dwukropku) do długość pola w bitach.
4. Dostajemy zmienną o nazwie flags zawierającą trzy jednobitowe pola.
5. Bardzo łatwo możemy manipulować bitami (zerować je, ustawiać i sprawdzać ich wartość), na przykład:

```
flags.is_extern = flag.is_static = 0;
```

Politechnika Wrocławska

Unie

1. Na pierwszy rzut oka deklaracja unii jest bardzo podobna do deklaracji struktury:

```
union uni {
    int val;
    float fval;
    char sval[4];
} zmienna;
```

2. Podobieństwo jest jednak złudne.
3. Unia to obszar pamięci, w którym zapisać można wartości różnych typów; kompilator dba o przydział odpowiednio dużego obszaru pamięci (mieszczącego „największy” z podnych typów).
4. Użycie: `zmienna.val = 5` albo `x = zmienna.fval`
5. Wykorzystanie...

Politechnika Wrocławska

Deklaracja nowego typu I

1. Standardowy zestaw typów w języku C jest niezbyt bogaty (**int**, **float**, **double**, **char**, **struct**).
2. Niektóre typy mogą być modyfikowane za pomocą słów **short**, **long**, **unsigned**.
3. Język C pozwala na definiowanie „aliasów" dla struktur istniejących (ułatwiają one konstruowanie programów). Służy do tego instrkcja **typedef**:

   ```
   typedef int Lenght;
   ```

   deklarująca **nowy** typ danych o nazwie **Lenght** (służący do deklarowania wszystkich zmiennych związanych z długością):

   ```
   Lenght len, maxlen, minlen;
   ```

4. Możemy w ten sposób stworzyć sobie typ zespolony:

   ```
   typedef struct { double r, theta; } Complex;
   ```

Deklaracja nowego typu II

albo

```
typedef struct { double real, imaginary; }
                Complex;
```

5. Albo cośą takiego:

```
typedef char* string;
```

(Do czego może się to przydać? Ale korzystać ostrożnie!)

Politechnika Wrocławska

Deklaracje typów

```
typedef struct {
     char forename[20];
     char surname[20];
     float age;
     int childcount;
     } person;
```

1. Definiujemy nowy typ o nazwie **person**.
2. Jest to struktura danych (rekord?) o zawartości: imię, nazwisko, wiek, liczba potomstwa.
3. Deklaracja poszczegolnych zmiennych staje się bardzo prosta:

```
person Osoba1, Osoba11, Osoba111;
int a, b, c;
double x, y, z;
```

Wejście/Wyjście

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2023-05-08 07:31:07 +0200

Strumienie

1. W czasach przed-uniksowych program wykonujący operacje wejścia/wyjścia musiał „podłączyć" wszystkie urządzenia, z których chciał (musiał) korzystać.
2. Unix wprowadził abstrakcyjne urządzenia wejścia/wyjścia zwane strumieniami.
3. Strumień to uporządkowany ciąg bajtów, które mogą być odczytywane jeden po drugim aż do napotkania znacznika końca.
4. Unix wprowadził również ideę automatycznego uaktywniania tych strumieni. (Wcześniejsze systemy operacyjne wymagały wykonywania pewnych, czasami skomplikowanych, czynności.)
5. Standardowo program ma do dyspozycji trzy strumienie:
   5.1 Standardowe wejście (*standard input*) — **stdin**,
   5.2 Standardowe wyjście (*standard output*) — **stdout**,
   5.3 Standardowy strumień błędów (standard error) — **stderr**.

Standardowe wejście

- Pewno powinno się mówić *standardowy strumień wejścia*...
- Standardowo dane pobierane są z terminala, z którego został uruchomiony program.
- Strumień może być „przekierowany" (*redirection*).
- Nie wszystkie programy z niego korzystają.
- Używany do wprowadzania informacji do programu.
- Deskryptor 0.

Standardowe wyjście

- Pewno powinno się mówić *standardowy strumień wyjścia*...
- Standardowo dane wysyłane są na terminal, z którego został uruchomiony program.
- Strumień może być „przekierowany" (*redirection*).
- Nie wszystkie programy z niego korzystają.
- Służy do wyprowadzania informacji z programu.
- Deskryptor 1.

Standardowy strumień błędów

- Standardowo dane wysyłane są na terminal, z którego został uruchomiony program.
- Strumień może być „przekierowany” (*redirection*).
- Dobrze napisane programy kierują tam informacje o błędach i komunikaty diagnostyczne.
- Służy do informowania operatora o błędach.
- Deskryptor 2.

Podstawowe przekierowania I

- Przekierowania to funkcja środowiska, w którym uruchamiany jest program.
- stdout do pliku (poprzednia zawartość pliku ulega zniszczeniu)

```
program >a.txt
```

- stdout do pliku (w trybie dopisywania)

```
program >>a.txt
```

- stderr do pliku

```
program 2>a.txt
```

- stderr oraz stdin do tego samego pliku

```
program >a.txt 2>&1
```

(najpierw stdout łączy z plikiem, następnie stderr łączy z **aktualnym** stdout)

Podstawowe przekierowania II

- Z pliku do stdin

```
program < b.txt
```

- Z stdout (jednego programu) do stdin (drugiego)

```
program1 | program2
```

Dziwny przykład

1. Mamy obrazek

Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki

Dziwny przykład

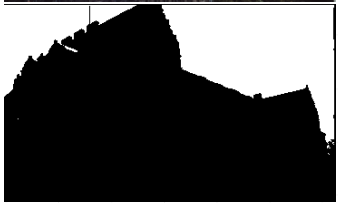
1. Mamy obrazek
2. Chcemy dodać chmurki

Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...

Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...



Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...

Dziwny przykład

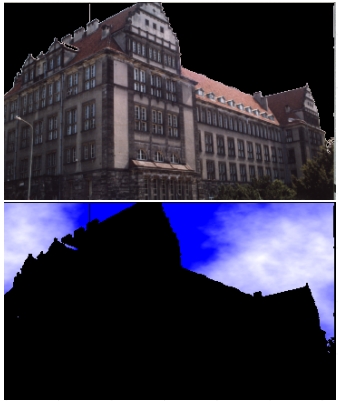
1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...

Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...
5. ...i na chmurki

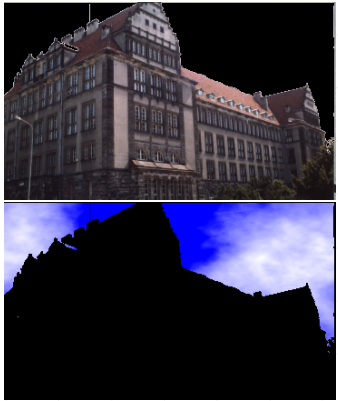
Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...
5. ...i na chmurki



Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...
5. ...i na chmurki
6. Otrzymane obrazki sumujemy



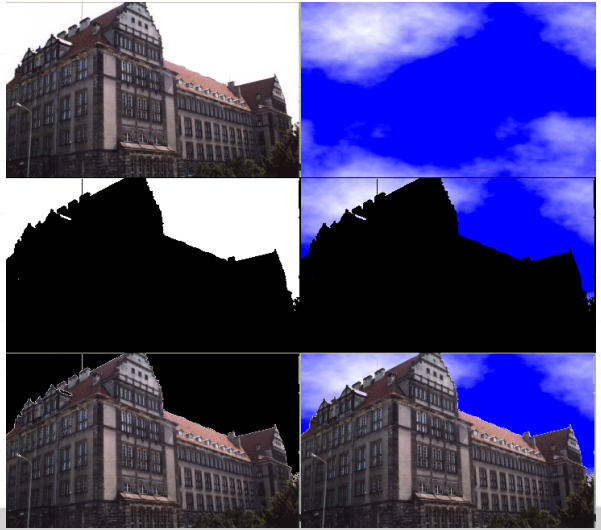
Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę”...
4. ...którą nakładamy na obrazek...
5. ...i na chmurki
6. Otrzymane obrazki sumujemy

Jak to jest zrobione?

```
1  echo pbmmask
2  pngtopnm obj.png >obj.ppm
3  ppmforge -clouds -width 366 -height 218 >dest.ppm
4  ppmtopgm obj.ppm | pgmtopbm -threshold -v 0.85| pbmmask\
5   |pnminvert> objmask.pbm
6  pnmarith -multiply dest.ppm objmask.pbm > t1.ppm
7  pnminvert objmask.pbm | pnmarith -multiply obj.ppm - > t2.ppm
8  pnmarith -add t1.ppm t2.ppm >t3.ppm
9  pnmcat -lr obj.ppm dest.ppm >a1.ppm
10 pnmcat -lr objmask.pbm t1.ppm >a2.ppm
11 pnmcat -lr t2.ppm t3.ppm >a3.ppm
12 pnmcat -tb a1.ppm a2.ppm a3.ppm >przyklad2.ppm
13 rm a1.ppm
14 rm a2.ppm
15 rm a3.ppm
16 rm objmask.pbm
17 rm t1.ppm
18 rm t2.ppm
19 rm t3.ppm
20 rm obj.ppm
21 rm dest.ppm
22 pnmtopng przyklad2.ppm >przyklad2.png
```

Jak to jest zrobione cd



Podstawowe instrukcje wyjścia I

1. **#include**<stdio.h>
2. **printf()**

```
#include <stdio.h>
int main(void)
{
   printf("Hello␣world!\n");
   return 0;
}
```

Wersja ogólna:

```
printf(format, argument1, argument2, ...);
```

Podstawowe instrukcje wyjścia II

Format to napis ujęty w cudzysłowy określający sposób wyświetlania informacji. Format wyświetlany jest tak jak go zapiszemy z wyjątkiem pewnych specjalnych znaków, które są zamieniane na coś innego. Znak % ma znaczenie specjalne (podobnie jak znak \).

```
printf("Procent: %% Backslash: \\");
```

Najczęstsze użycie printf():

- printf("%d", i); gdy i jest typu **int**; zamiast %d można też użyć %i,
- printf("%f", i); gdy i jest typu **float** lub **double**,
- printf("%Lf", i); gdy jest typu **long double**,
- printf("%c", i); gdy i jest typu **char** (i chcemy wydrukować znak)
- printf("%s", i); gdy i jest napisem (typu **char***)

Format może być zmienną!

Funkcja zwraca liczbę znaków w tekście (nie licząc znaku \0 kończącego tekst) w przypadku sukcesu lub znak EOF w przypadku błędu.

Podstawowe instrukcje wyjścia III

3. **puts()**

```c
#include <stdio.h>

int main(void)
{
  puts("Hello world!");
  return 0;
}
```

Funkcja po prostu kopiuje tekst zawarty w argumencie (może być zmienna!) do standardowego strumienia wyjścia dodając na końcu znak przejścia do nowej linii. Funkcja zwraca liczbę nieujemną w przypadku sukcesu lub EOF w przypadku błędu.

Podstawowe instrukcje wyjścia IV

4. **putchar()**
   Funkcja służy do wyprowadzenia pojedynczego znaku do strumienia stdio.
   Funkcja zwraca kod znaku traktowany jako unsigned char przekształcony do typu int; w przypadku błędu funkcja zwraca wartość EOF.

```
#include<stdio.h>
int main (void)
{
    int i;
    for (i = 'a'; i <= 'z'; ++i)
        putchar (i);
    return 0;
}
```

Podstawowe instrukcje wejścia I

1. **#include** <stdio.h>
2. **scanf()**

```
#include <stdio.h>
int main()
{
    int liczba = 0;
    printf("Podaj␣liczbe:␣");
    scanf("%d", &liczba);
    printf("%d*%d=%d\n", liczba, liczba,
           liczba * liczba);
    return 0;
}
```

Zwracam uwagę na znak & przy drugim argumencie funkcji!

Podstawowe instrukcje wejścia II

Typowe użycie:

- scanf("%i", &liczba); wczytuje liczbę typu int,
- scanf("%f", &liczba); — liczbę typu float, (również %e, %g)
- scanf("%lf", &liczba); — liczbę typu double, (również %le, %lg)
- scanf("%s", tablica_znakow); ciąg znaków.

Zwracam uwagę na brak znaku & w ostatnim przypadku — gdy **nazwa tablicy** pojawia się jako argument funkcji automatycznie przekazywany jest adres.
Funkcja zwraca liczbę poprawnie wczytanych zmiennych lub EOF jeżeli nie ma już danych w strumieniu lub nastąpił błąd.

Podstawowe instrukcje wejścia III

```c
#include <stdio.h>
int main(void)
{
    int a, b;
    while (scanf("%d␣%d", &a, &b) == 2) {
        printf("%d\n", a + b);
    }
    return 0;
}
```

Podstawowe instrukcje wejścia IV

Co robi ten program:

```c
#include <stdio.h>
int main(void)
{
    int result, n;
    do {
        result = scanf("%d", &n);
        if (result == 1) {
            printf("%d\n", n * n * n);
        } else if (!result) { /* !result to to
                               samo co result == 0 */
            result = scanf("%*s");
        }
    } while (result != EOF);
    return 0;
}
```

Podstawowe instrukcje wejścia V

Oto wynik działania programu

```
ala ma kota
2
8
ola ma s3
3derft
27
sdsdsd
pi
1234pies
1879080904
```

Podstawowe instrukcje wejścia VI

3. **gets()**
   - Funkcja służy do wczytywania linii tekstu.
   - Nie należy jej używać...
   - ...gdyż funkcja nie sprawdza, czy jest miejsce do zapisu w tablicy

```
#include <stdio.h>
int main(void)
{
   char napis[50], *n;
   n = gets(napis);/* jesli pierwsza linia
                      ma wiecej niz 49
                      znakow nastapi
                      przepelnienie bufora */
   if(n != NULL)
       printf("%s\n",napis);
   else
```

Podstawowe instrukcje wejścia VII

```
        printf("blad␣odczytu");
    return 0;
}
```

4. **getchar()**

```
#include <stdio.h>
int main(void)
{
  int c;
  while ((c = getchar()) != EOF) {
    if (c == '␣') {
      c = '_';
    }
    putchar(c);
  }
```

Podstawowe instrukcje wejścia VIII

```
  return 0;
}
```

Bardzo prosta funkcja czytająca (pobierająca) jeden znak z stdio.

- Ze względu na specyfikę komunikacji (istnienie bufora) funkcja nie jest w stanie „zauważyć" pojedynczego naciśnięcia klawisza; zwraca informację gdy naciśnięty zostanie klawisz Enter lub bufor się przepełni.
- Kod naciśniętego klawisza Enter też trafia do bufora!
- Gdy nastąpi błąd lub nie ma już danych funkcja zwraca EOF.
- EOF to zazwyczaj -1
- Funkcja zwraca kod pobranego znaku traktowany jako **unsigned char** przekształcony do typu **int**

Podstawowe instrukcje wejścia IX

```
#include<stdio.h>
int main()
{
    int i;
    i = getchar();
    printf("przeczytano␣znak␣o␣numerze␣%d", i);
    return 0;
}
```

Pliki

1. Właściwie nie ma wielkiej różnicy w (podstawowej) komunikacji ze standardowymi strumieniami wejścia i wyjścia a plikami.
2. Język C zna dwa sposoby komunikowania z plikami:
   - wysokopoziomowy,
   - niskopoziomowy.
3. Nazwy procedur pierwszej grupy zaczynają się na literę „f" (fopen(), fread(), fclose())
4. Identyfikatorem pliku jest wskaźnik do struktury danych deklarowanej jako FILE
5. Niskopoziomowe operacje to read(), open(), write() i close()
6. Identyfikatorem pliku jest liczba całkowita jednoznacznie identyfikująca plik (w systemie Unix — deskryptor pliku)
7. Nie należy tych funkcji mieszać!

Wskaźnik FILE

1. Plik stdio.h zawiera definicję (typedef) typu FILE.
2. W pierwszej kolejności musimy zadeklarować zmienną, która będzie przechowywałą adres początkowy odpowiedniej struktury danych:

```
FILE * fp;
```

3. Zmienne tego typu przechowują wszystkie informacje na temat pliku.
4. Na ogół nie ma potrzeby odwoływać się do poszczególnych pól tej struktury danych bezpośrednio.
5. Funkcje zdefiniowane w stdio.h powinny w zupełności wystarczyć.

Otwarcie pliku I

1. Przed skorzystaniem z pliku należy go „otworzyć" (*open*).
2. Standardowa biblioteka (stdio.h) zawiera trzy funkcje fopen, freopen i fclose.
3. Pierwsze dwie służą do „skojarzenia" (powiązania) pliku z odpowiednią strukturą danych.
4. Ostatnia — powiązanie to likwiduje.
5. Wszystkie czynności związane z otwieraniem i zamykaniem plików realizowane są przez System Operacyjny.

Otwarcie pliku II

1. Funkcja **fopen()** otwiera plik o podanej nazwie we wskazanym trybie, tworzy strukturę danych opisującą go i zwraca do niej adres. W przypadku błędu — zwraca NULL.

```
fp = fopen("plik.txt", "r")
```

2. Drugi parametr funkcji to tryb otwarcia pliku:
   - "r" — odczyt
   - "w" — zapis
   - "r+" — odczyt i zapis (najpierw czytamy)
   - "w+" — zapis i odczyt (najpierw piszemy)
   - "a" — zapis na końcy pliku (*append*, dodawanie)
   - "a+" — zapis i odczyt (w trybie dopisywania)

Otwarcie pliku III

3. Funkcja **freopen()** najpierw zamyka otwarty wcześniej plik i otwiera go ponownie we wskazanym trybie, zwracając wskaźnik do struktury danych opisujących strumień. W przypadku błędu — zwraca NULL.

```
fp = freopen("plik.txt", "r+", fp)
```

4. Funkcja **fclose()** zamyka wskazany plik. W przypadku błędu — zwraca NULL.

```
#include <stdio.h>
int fclose(FILE *stream);
```

stream to wskaźnik do struktury danych opisujących strumień danych.

```
fclose( fp );
```

Funkcje pomocnicze I

Poniższe funkcje są nieco mniej istotne (na tym poziomie nauki języka C)

1. **fflush()**

```
#include <stdio.h>
int fflush(FILE *stream);
```

Funkcja powoduje zapis w pliku (związanym ze strumieniem wskazywanym przez stream) wszystkich zbuforowanych danych.

2. **setvbuf()**

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, \
                                    size_t size);
```

- funkcji można użyć zaraz po otwarciu pliku, ale przed wykonaniem jakiejkolwiek innej operacji na pliku

Funkcje pomocnicze II

- zmienna mode określa sposób buforowania danych (_IOFBF — pełne buforowanie, _IOLBF — buforowanie linii, _IONBF — buforowanie wyłączone)
- pozostałe parametry (buf i size) określają — jeżeli buf nie jest NULL — obszar (i jego wielkość) przeznaczony do buforowania danych. Jeżeli buf jest równe NULL funkcja przydziela taki obszar automatycznie. Funkcja zwraca zero gdy zakończy się normalnie i wartość różną od zera w przypadku błędów.

Pozycja w pliku I

1. **fgetpos()** i **fsetpos()**

```c
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

- Funkcja **fgetpos** zapisuje aktualną wartość wskaźnika pozycji związanego ze strumieniem stream w obiekcie wskazywanym przez pos
- Funkcja **fsetpos** ustala aktualną wartość wskaźnika pozycji strumienia strem na wartość z obiektu wskazywanego przez pos. Wartość ta powinna być wcześniej uzyskana za pomocą funkcji fgetpos.

Pozycja w pliku II

2. **fseek() i ftell()**

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, \
                                int whence);
long int ftell(FILE *stream);
```

- Funkcja **ftell** zwraca aktualną wartość wskaźnika pozycji (dla plików binarnych jest to liczba znaków od początku plików; dla plików tekstowych sprawa jest bardziej skomplikowana) strumienia określonego przez stream.
- Funkcja **fseek** dla plików binarnych ustala aktualną wartość wskaźnika pozycji przez sumowanie zmiennej offset z pozycją wskazywaną przez whence; stdio.h zawiera makra SEEK_SET, SEEK_CUR i SEEK_END określające odpowiednio początek pliku, bieżącą pozycję i koniec pliku. Dla plików tekstowych wartość offset powinna być albo zero albo wartością uzyskana wcześniej przez odwołanie do funkcji **ftell**.

Pozycja w pliku III

3. **rewind()**

```
#include <stdio.h>
void rewind(FILE *stream);
```

Funkcja ustawia wskaźnik pozycji w pliku na początek pliku.

Czytanie z pliku I

1. **fget()**

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Funkcja zwraca następny znak (jako unsigned char zamieniony na int) ze strumienia wskazywango przez stream. Jeżeli odczyt dotrze do końca pliku fukcja zwraca EOF. W przypadku błędu ustawiany jest wskaźnik błędu i funkcja zwraca EOF.

2. **fgets()**

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Czytanie z pliku II

Funkcja odczytuje co najwyżej n-1 znaków z pliku wskazywanego przez stream i zapisuje je w tablicy wskazywanej przez s. Odczyt kończy się z chwilą napotkania znaku nowej linii, dojścia do końca strumienia lub przeczytania n-1 znaków.
Uwaga: Znak nowej linii to \n dla Unixa, \r\n dla DOS/Windows i \r dla MAC (przed OS X).

3. **getc()** — odpowiednik fgetc, może być zaimplementowana jako makro.
4. **getchar()** — odpowiednik getc ale dla stdin
5. **ungetc()**

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Funkcja „zwraca" znak c do wskazanego strumienia. Operacja może się nie powieść, jeżeli zwracamy w ten sposób zbyt wiele znaków bez wykonania operacji czytania lub pozycjonowania pliku. Zawsze działa dla jednego znaku.
Plik wejściowy nie ulega zmianie!

Odczyt formatowany I

1. Funkcje typu **scanf**

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

Dane czytane są ze wskazanego strumienia (stream) i interpretowane zgodnie z użytym formatem. W formacie powinna wystąpić wystarczająca liczba specyfikacji aby dostarczyć dane dla wszystkich argumentów.

- fscanf zwraca liczbę przeczytanych wartości lub wartość EOF
- scanf jest odpowiednikiem fscanf, ale dotyczy strumienia stdin
- sscanf jest odpowiednikiem fscanf, z tym, że dane „czytane" są z tablicy znakowej; dotarcie do końca tabeli jest równoważne z dotarciem do końca pliku

Wejście: specyfikacja formatu I

Na specyfikację formatu składają się:

- odstępy
- znaki (różne od % i odstępów). Jeżeli taki znak wystąpi w specyfikacji musi się pojawić w strumieniu wejściowym na odpowiednim miejscu!
- specyfikacje konwersji (rozpoczynające się znakiem %)

Po znaku % wystąpić może:

- nieobowiązkowy znak $*$ (oznaczający, że zinterpretowana wartość ma być zignorowana)
- nieobowiązkowa specyfikacja długości pola (określa maksymalną liczbę znaków pola)
- jeden z nieobowiązkowych znaków **h**, **l** (mała litera „el") lub **L** mówiących jak ma być interpretowana czytana wielkość (h — short, l — long albo double w przypadku float, L — long double)
- znak określający typ konwersji

Wejście: specyfikacja formatu II

Po znaku procent (%) wystąpić może jeden ze znaków

- d liczba całkowita
- i liczba całkowita
- o liczba całkowita kodowana ósemkowo
- u liczba całkowita bez znaku (unsigned int)
- x liczba całkowita kodowana szesnastkowo
- e, f, g liczba typu float
- s ciąg znaków (na końcu zostanie dodany znak NULL)
- c znak (lub ciąg znaków gdy wyspecyfikowano szerokość pola), nie jest dodawane zero na końcu
- n do zmiennej odpowiadającej tej specyfikacji konwersji wpisana zostanie liczba znaków przetworzonych przez fscanf
- % znak %

Wejście: specyfikacja formatu III

Przykład

```
#include <stdio.h>
int main(void)
{
    int x, y, n;
    x = scanf("%*i␣%i␣%n", &y, &n);
    printf("y=␣%i,␣n=␣%i\n", y, n);
    printf("x=␣%i\n", x);
    return 0;
}
```

Wejście: specyfikacja formatu IV

Dane i wyniki

```
10 20 30 40
y= 20, n= 6
x= 1

1␣␣␣␣␣2222␣ala␣ma␣kota
y=␣2222,␣n=␣11
x=␣1
```

Wejście: specyfikacja formatu V

Kolejny przykład

```
#include <stdio.h>
int main(void)
{
    int x, y, z, n;
    x = scanf("%4i␣%5i␣%n", &y, &z, &n);
    printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
    printf("x=␣%i\n", x);
    return 0;
}
```

Wejście: specyfikacja formatu VI

Dane i wyniki

```
1234567890
y= 1234, z= 56789, n= 9
x= 2
```

Wejście: specyfikacja formatu VII

I jeszcze jedne przykład

```
#include <stdio.h>
int main(void)
{
    int x, y, z, n;
    x = scanf("%4i␣a␣%5i␣%n", &y, &z, &n);
    printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
    printf("x=␣%i\n", x);
    return 0;
}
```

Wejście: specyfikacja formatu VIII

Dane i wyniki

```
123b123b
y=␣123,␣z=␣32767,␣n=␣2023244192
x=␣1

1a2␣3␣4␣5␣6␣7␣8␣9
y=␣1,␣z=␣2,␣n=␣4
x=␣2
```

Wejście: specyfikacja formatu IX

```c
#include <stdio.h>
int main(void)
{
    int x, y, z, n;
    x = scanf("%4ia%5i%n", &y, &z, &n);
    printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
    printf("x=␣%i\n", x);
    return 0;
}
```

```
123a123
y= 123, z= 123, n= 7
x= 2
```

Pisanie do pliku I

1. **fputc()**

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Funkcja zapisuje podany znak do pliku (czy też dodaje go do strumienia wyjściowego). Wersja **putc** dodaje znak do stumienia stdout. **putchar** jest odpowiednikiem **putc**.

2. **fputs()**

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Funkcja dodaje wskazany ciąg znaków do strumienia wyjściowego. **puts** jest odpowiednikiem działającym na strumieniu stdout.

Rodzina polecen fprintf I

Wszystkie funkcje realizują formatowane wyprowadzanie informacji.

```
#include <stdarg.h>
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
```

- **fprintf** to podstawowa wersja funkcji
- **printf** używa stdout jako strumienia wyjściowego.
- **sprintf** przekazuje sformatowane informacje do tablicy znakowej (odpowiedniej długości)

Rodzina polecen fprintf II

- warianty o nazwie rozpoczynającej się na v (vfprintf, fprintf, vsprintf) używają specyficznej metody przekazywania listy parametrów zmiennej długości — nie będziemy się nimi zajmować.

Podobnie jak w przypadku formatowanego wprowadzania danych, zmienna lub stała tekstowa zawiera informacje o sposobie wyprowadzania danych. Zasadą jest, że znak % rozpoczyna specyfikację konwersji, a pozostałe znaki są wyprowadzane w postaci takiej jak w formacie.
Po znaku % wystąpić może:

- zero lub więcej wskaźników modyfikujących sposób konwersji,
- nieobowiązkową specyfikację minimalnej długości pola wyjściowego; gdy wyprowadzana wartość jest krótsza niż pole zostanie uzupełniona odstępami (z lewej lub prawej strony w zależności od specyfikacji)

Rodzina polecen fprintf III

- Dokładność (podana jako liczba cyfr) dla specyfikacji **d**, **i**, **o**, **u**, **x** oraz **X**, liczba cyfr po przecinku (dla specyfikacji konwersji **a**, **A**, **e**, **E**, **f** oraz **F**) maksymalna liczba cyfr znaczących (dla specyfikacji **g** i **G**) lub liczba znaków (dla specyfikacji **s**). W przypadku liczb „zmiennoprzecinkowych" precyzja ma postać **.** (kropki) po której jest albo ∗ albo liczba określająca liczbę cyfr po kropce dziesiętnej. Liczby podczas wyprowadzania są zaokrąglane czyli printf ("%1.1f\n", 1.19); spowoduje wyprowadzenie 1.2

Specjalne wskaźniki modyfikujące sposób konwersji to:

–	wynik będzie justowany do lewej strony (standardowo do prawej)
+	Liczby będą zapisywane zawsze ze znakiem
odstęp	odstęp będzie zawsze dodawany przed liczbą (chyba, że liczba poprzedzona jest znakiem)
#	Wynik jest konwertowany do postaci „alternatywnej" (zależnej od typu konwersji)
0	Liczby będą poprzedzone wiodącymi zerami