

PROGRAMOWANIE PROCEDURALNE

Marek Grochowski

Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK

<http://www.fizyka.umk.pl/~grochu/pp>
grochu@is.umk.pl

5 listopada 2018

Programowanie to wszechstronny proces prowadzący od problemu obliczeniowego do jego rozwiązania w postaci programu.

Celem programowania jest odnalezienie sekwencji instrukcji, które w sposób automatyczny wykonują pewne zadanie.

Programowanie proceduralne to paradygmat programowania zalecający dzielenie kodu na procedury, czyli fragmenty wykonujące ściśle określone operacje.

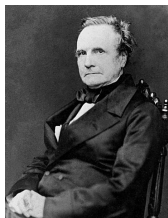
- Podstawy programowania w języku C
- *Case study* - przykłady programów, demonstracje
Problem → Algorytm → Program → Rozwiązanie
- Paradygmat programowania proceduralnego,
algorytmy wydzielone w postaci uniwersalnych funkcji
- Reprezentacja danych w komputerze:
typy proste, złożone, struktury dynamiczne, ...
- Elementy inżynierii oprogramowania: model, projekt, analiza,
implementacja, wykrywanie błędów, testowanie
- Laboratorium: język C, środowisko MS Visual Studio
- Zaliczenie wykładu: TEST

JAK UCZYĆ SIĘ PROGRAMOWANIA?

- pytaj
- czytaj
- podglądaj
- programuj, programuj, programuj, ...

-  Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*, WNT, Warszawa, 2000.
-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  D. Harel, *Rzecz o istocie informatyki. Algorytmika.*, WNT, Warszawa, 1992.
-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  J. Bentley, *Perlełki oprogramowania*, WNT, Warszawa, 2001.

CHARLES BABBAGE (1791–1871)



1822 Projekt maszyny różnicowej.

1837 (Parowa) **maszyna analityczna** - sterowanie sekwencyjne, pętle, odgałęzienia, projekt niedokończony

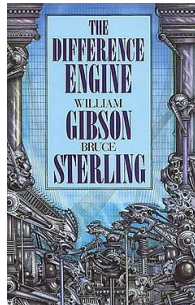
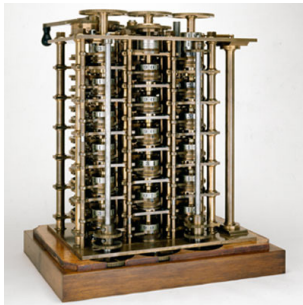
AUGUSTA ADA LOVELACE (1815-1852)



1842 „Note G”, algorytm wyznaczania liczb Bernoulliego. **Pierwszy program komputerowy.**

1979 Język ADA.

- 1849 Difference Engine No. 2, dokładność 31 cyfr, wydruk na wyjściu
- 2002 Realizacja projektu,  Computer History Museum.
- 2011 Rozpoczęto 10 letni projekt rekonstrukcji maszyny analitycznej



<http://www.computerhistory.org/>

0 mechaniczne, przekaźnikowe (do 1945)

Z3 (Berlin, 1941), Harvard Mark 1 (USA, 1944), GAM-1 (Warszawa, 1950), PARK (AGH, 1957)

1 lampy elektronowe (1945-59)

ABC Atanasoff-Berry Computer (USA, 1942), COLOSSUS (UK, 1943), ENIAC (USA, 1945),

XYZ (Warszawa, 1957/1958)

2 tranzystory i pamięci ferrytowe (1959-64)

PDP-1 (USA, 1960), ZAM-41(Polska, 1961), Odra 1204 (Polska, 1967)

3 układy scalone o małej skali integracji SSI (1965-70)

IBM 360 (1965), Odra 1305 (Polska, 1973)

4 układy o wysokiej skali integracji LSI i VLSI, mikroprocesory

mikroprocesor Intel 4004 z częstotliwością taktowania 0,1 MHz (1971), IBM 5150 PC (1981)

5 Komputery przyszłości: kwantowe, optyczne, biologiczne ?



KOMPUTERY GENERACJI 0

MECHANICZNE I PRZEKAŹNIKOWE

Konrad Zuse

1936 Mechaniczny **Z1**,
liczby zmiennopozycyjne.

1941  Przełącznikowy **Z3**,
pierwszy działający
programowalny komputer
5.3Hz,
64 słowa 22 bitowe (176 B).



INNE KOMPUTERY 0 GENERACJI

1939-44 Harvard Mark 1 Howarda Aikena (IBM ASCC)

1950 GAM-1, Państwowy Instytut Matematyczny w Warszawie

1957 PARK (Programowany Automat Rachunków Krakowianowych), AGH

Język
Plankalkül
1943



KOMPUTERY 1 GENERACJI (1945-59)

LAMPY PRÓŻNIOWE

- Zastosowanie do obliczeń numerycznych (łamanie szyfrów, balistyka).
- Wejście: karty dziurkowane, taśmy papierowe
- Wyjście: wydruk, dalekopis, lampy
- Pamięć: dane przechowywane na dyskach magnetycznych, rtęciowe linie opóźniające
- Program: głównie język maszynowy

1949 (prawie) pierwszy assembler (EDSAC)

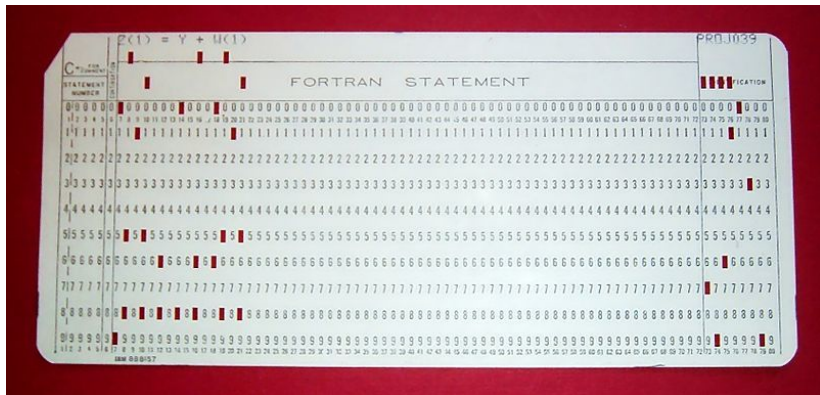
1952 Grace Hopper, pierwszy kompilator A-0 (UNIVAC I)

1954 Język Fortran



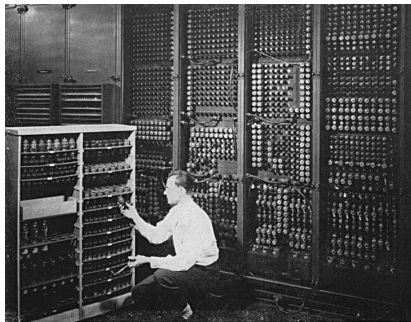
KARTA PERFOROWANA

80 KOLUMN, 10 WIERSZY NUMERYCZNYCH + 2 WIERSZE STREFOWE (IBM, 1928)



ENIAC (1946)

ELEKTRONICZNE URZĄDZENIE NUMERYCZNE CAŁKUJĄCE I LICZĄCE



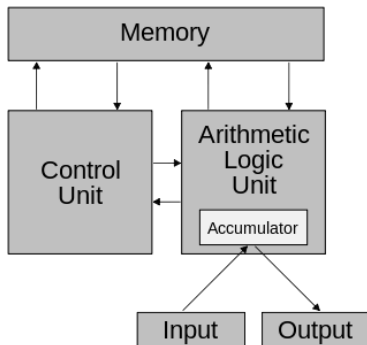
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

18 000 lamp,
30 ton, 170 m², moc 160kW,
5000 operacji dodawania na
sekundę
system dziesiętny,
ręczne programowanie przez
ustawianie przełączników (6k) i
wtykanie kabli

WSPÓŁCZESNA KONCEPCJA KOMPUTERA

JOHN VON NEUMANN, 1945

Pamięć używana zarówno do przechowywania danych jak i samego programu, każda komórka pamięci ma unikatowy identyfikator (adres).



Konrad Zuse
postulował
to w swoich
patentach
w 1936 r.!!!

1949 EDVAC, Electronic Discrete Variable Computer, współpracuje już z dyskami magnetycznymi.

KOMPUTERY 2 GENERACJI (1959-64)

TRANZYSTORY I PAMIĘĆ FERRYTOWA

1960 PDP-1, pierwszy dostępny w sprzedaży minikomputer z monitorem i klawiaturą.



Pierwsza gra wideo „Spacewar!” (Steve Russel), pierwszy edytor tekstu, interaktywny debugger, komputerowa muzyka.

KOMPUTERY 3 GENERACJI (1965-70)

UKŁADY SCALONE O MAŁEJ SKALI INTEGRACJI SSI

1970 Minikomputer K-202, szybszy i wydajniejszy od IBM 5150 PC z roku 1981. opracowany i skonstruowany przez inż. Jacka Karpińskiego.



16 bitów, adresowanie stronicowe do 8MB (konkurencja max. 64kB), modularność, wielodostępowość, 1 mln. operacji/s.

KOMPUTERY 4 GENERACJI (OD 1971)

UKŁADY O WYSOKIEJ SKALI INTEGRACJI LSI i VLSI

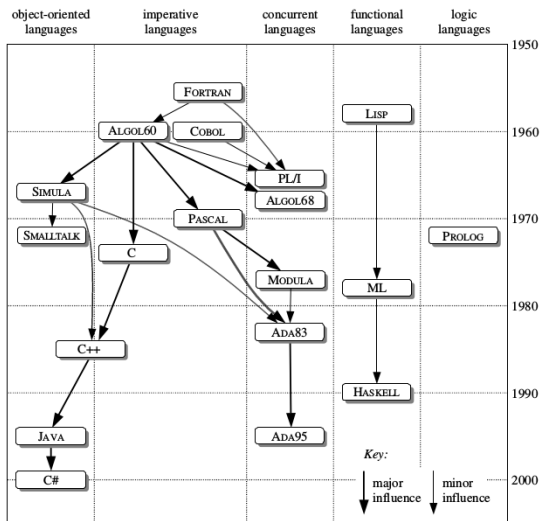
1981 IBM 5150 PC



procesor Intel 8088 (4.77 MHz)
64 kB pamięci ROM
do 640 kB pamięci RAM
brak dysku twardego (taśmy na kasetach,
późniejsze modele dyskietki 5,25 cala)
karta CGA (kolor) lub MGA
(monochromatyczna),
system operacyjny MS-DOS,
dźwięk z PC speakera

- kod maszynowy, assembler
- lata 50-te, języki wysokiego poziomu
Fortran (1955), Lisp (1955), COBOL (1959)
- lata 60-te, rozwój języków specjalistycznych
Simula I (1960, el. obiektowości), Lisp, COBOL
Pierwsze próby stworzenia języków ogólnych
Algol (58/60), PL/1 (1964).
- lata 70-te, początek pojedynku: Pascal vs. C
Zalążki obiektowości: Smalltalk (1972)
- lata 80-te, Dominują: C, Pascal, Basic
Powstają: C++ (1980), Matlab (1984)
- lata 90-te, era internetu, programowanie obiektowe
Java (1995), Python (1991), PHP (1995), JS (1995), .NET (C#, 2001)

Źródło:  *History and Evolution of Programming Languages*



Źródło: David A. Watt, "Programming Language Design Concepts"

- kod maszynowy, języki symboliczne
- wysokiego i niskiego poziomu
- paradygmaty programowania: proceduralne, strukturalne, obiektowe, funkcyjne, logiczne, uniwersalne, ...
- 📖 Lista 2500 języków komputerowych, Bill Kinnersley
- 📖 Lista języków na Wikipedii

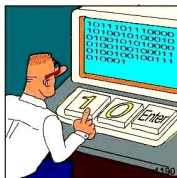


Pieter Bruegel (starszy), 1563

KOD MASZYNOWY

ciąg instrukcji w postaci binarnej wykonywanych bezpośrednio przez procesor.

- Rozkazy i dane w postaci binarnej pobierane są z pamięci
- Nie jest przenośny - każdy procesor ma swój specyficzny zestaw instrukcji



REAL Programmers code in BINARY.

JĘZYK ASSEMBLERA

zastępuje rozkazy maszynowe tzw. mnemonikami, zrozumiałymi przez człowieka słowami określającymi konkretną czynność procesora.

```

push    rbp
mov     rbp, rsp
mov
DWORD PTR [rbp-0x4], 0x0
jmp     11 <main+0x11>
add
DWORD PTR [rbp-0x4], 0x1
cmp
DWORD PTR [rbp-0x4], 0x9
jle     d <main+0xd>
pop     rbp
ret

```

- **Assembler** - program tłumaczący język assemblera na kod maszynowy (asemblacja)
- **Deassembler** - proces odwrotny
- Konrada Zuse, 1945 r., maszyna Z4, moduł „Planfertigungsteil” umożliwiał wprowadzanie oraz odczyt rozkazów i adresów w sposób zrozumiały dla człowieka

JĘZYKI WYSOKIEGO POZIOMU

- nie są bezpośrednio wykonywane przez procesor, przez co pozwalają uniezależnić program od platformy sprzętowej i systemowej
- składnia i instrukcje mają za zadanie maksymalizować zrozumienie kodu programu przez człowieka
- pozwalają skupić się na logice zadania
- kara za abstrakcję: kod niskiego poziomu zazwyczaj będzie bardziej efektywny od kodu wyższego poziomu

```
#include<stdio.h>

int main()
{
    puts("Witaj świecie!");
    return 0;
}
```

KOMPILATOR

program tłumaczący kod napisany w jednym języku na równoważny kod w innym języku. Przykłady:

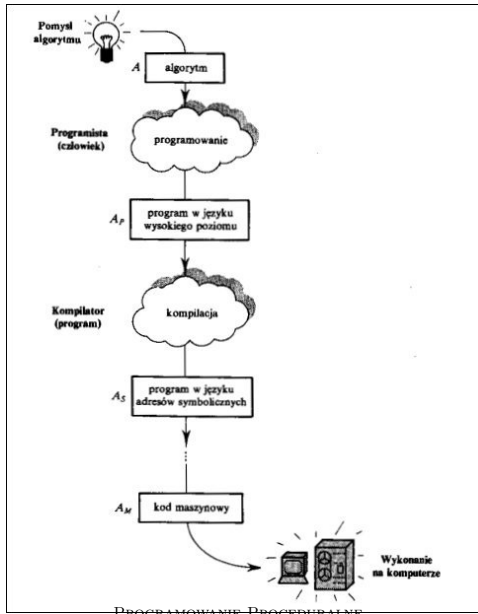
- kod źródłowy → kod maszynowy (C, C++, Pascal, Fortran)
- kod źródłowy → *byte code*, kod pośredni rozumiany lub kompilowany przez maszynę wirtualną (Java, .Net)

INTERPRETER

odczytuje, analizuje i uruchamia instrukcje zawarte w kodzie źródłowym (brak procesu kompilacji).

Języki skryptowe: Bash, Perl, Python

OD POMYSŁU DO PROGRAMU



- Wersja minimalistyczna: *edytor tekstu + kompilator*
 - Linux: GCC (gcc, cc, g++)
`cc hello.c -o hello`
 - Windows: Borland C, Cygwin, MinGW
- IDE, Integrated Development Environment
edytor, kompilator, deasembler, debugger, ...
 - Windows: MS Visual Studio 2010/2012/2013 (Express, Ultimate), Borland C++ Builder
 - Linux: KDevelop, Anjuta
 - Linux/Windows: Eclipse (CDT), NetBeans

Algorytmy

DOKŁADNA NAUKA WARZENIA PIWA.

WEDŁUG METODY ŁATWEJ, STWIERDZONEJ
OSMIOLETNIM DOŚWIADCZENIEM.

DO WYNAŁAZKÓW NAYNOWSZYCH
ZASTOSOWANA,

Z PRZYDANEM OPISANIEM APPARATU DO STUDZENIA, ZASTY-
PIĄCEGO ZWYKŁYCH KILKATOKI, ZA POMOCĄ KTÓREGO PI-
WO WRAŻE, W PRZECIĄGU IEDNEJ MINUTY DO TEMPERATURY
WODY STUDZENNEJ OCHŁODZONE BYDŹ MOŻE;

PRZEDSTAWIONA W SPOSÓBIE PRAKTYCZNYM,

PRZEZ KAROLA WILHELMA SCHMIDT,
AUTORA RÓŻNYCH DZIEŁ TECHNOLOGICZNYCH.

PRZEŁOŻONA Z JĘZYKA NIEMIECKIEGO,
DLA UŻYTKU ZIEMIANY POLSKICH.



WARSZAWA.

NAKLAD I Druk N. GLÜCKSBERGA,
KSIĘGARNIA I TYPOGRAFIA KRÓL: WARSZAW: UNIWERSYT.

1830.

Chambers udziela w swych pismach odmienny spo-
sób robienia Piwa Brunświckiego, odpisany iak twier-
dzi, z pierwotnéj Recepty, zachowywanéj na Ratuszu
miejskim w Brunświku.

Według Recepty téj wziąć należy 200 miarek
zwanych *Kanne* wody, i te tak długo gotować, po-
kąd 1/3 części iéy nie ubędzie. Sypie się potem do
téj wody siedm Szeffłów siodu Pszennego i jeden Sze-
fel drobnéj Fasoli; gdy odleie się do beczki dla wy-
robienia, nie należy zapelniać iéy całkowicie, bo sko-
ro fermentacya nastąpi, kładzie się do beczki we-
wnętrznej kory z Sosnińy funtów trzy, pączków Brzo-
zowych i kotków (pączków) Sosnowych po funcie
jednym, trzy garści ziela *Cārdi-Benedicti*, garść je-
dną lub dwie kwiatu zwanego *Sonnenthaublütthe*, *Pim-
pinelli*, *Betoniki*, *Majeranu*, ziela zwanego *Poley* i dzi-
kiego *Tymianku*, każdego pół garści, albo garść ca-
łą, kwiatu *Bzowego* garści dwie, owocu róży polnéj
uncyi 30, utłuczonych.— Po nakładzeniu tych zapraw
do beczki, dolewa się ta do pełności, zaczęm skutkiem
fermentacyi Piwo naciągnie nieco zapachu i smaku,
z zapraw przydanych.— Przy ukończeniu fermentacyi
wybiją się do beczki io jay świeżo zniesionych i ta
szpuntuje się. Wę dwa roky dopiero po zaszpuntowa-
niu ściaga się Piwo do picia.

- składniki (dane wejściowe): woda, słód, itd.
- wynik: beczka piwa
- sprzęt: beczka, piwowar (mielcarz)
- przepis: oprogramowanie, algorytm
- instrukcje: dodawanie składników, gotowanie, odlewanie, dolewanie, fermentacja, szpuntowanie

Chambers udziela w swych pismach odmienny sposób robienia Piwa Brunświckiego, odpisany iak twierdzi, z pierwotnéy Recepty, zachowywanéy na Ratuszu mieyskim w Brunświku.

Według Recepty téy wziąć należy 200 miarek zwanych *Kanne* wody, i te tak długo gotować, po kąd 1/3 części iéy nie ubędzie. Sypie się potem do téy wody siedm Szeflów siodu Pszennego i jeden Szeffel drobnéy Fasoli; gdy odleie się do beczki dla wyrobienia, nie należy zapełniać iéy całkowicie, bo skoro fermentacyia nastąpi, kładzie się do beczki wewnętrzny kory z Sośnin y funtów trzy, pączków Brzozowych i kotków (pączków) Sosnowych po funcie jednym, trzy garści ziela Cărdi-Benedicti, garść jedną lub dwie kwiatu zwanego *Sonnenhaublütthe*, Pimpinelli, Betoniki, Majeranu, ziela zwanego Poley i dzikiego Tymianku, każdego pół garści, albo garść całą, kwiatu Bzowego garści dwie, owocu róży polnéy uncyi 30, utłuczonych. — Po nakładzeniu tych zapraw do beczki, dolewa się ta do pełności, zaczém skutkiem fermentacyi Piwo naciągnie nieco zapachu i smaku, z zapraw przydanych. — Przy ukończeniu fermentacyi wybiją się do beczki io jay świeżo zniesionych i ta szpuntuje się. Wę dwa roki dopiero po zaszpuntowaniu ściaga się Piwo do picia.

ALGORYTM

jednoznacznie zdefiniowany ciąg operacji prowadzący w skończonej liczbie kroków do rozwiązania zadania.

ALGORYTM EUKLIDESA, OK. 300 P.N.E.

algorytm znajdowania największego wspólnego dzielnika (NWD) dwóch liczb całkowitych dodatnich. Uznawany za pierwszy kiedykolwiek wymyślony niebanalny algorytm.

Algorytmy to rozwiązywanie pewnych zadań - **zadań algorytmicznych**.

SPECYFIKACJA ZADANIA ALGORYTMICZNEGO

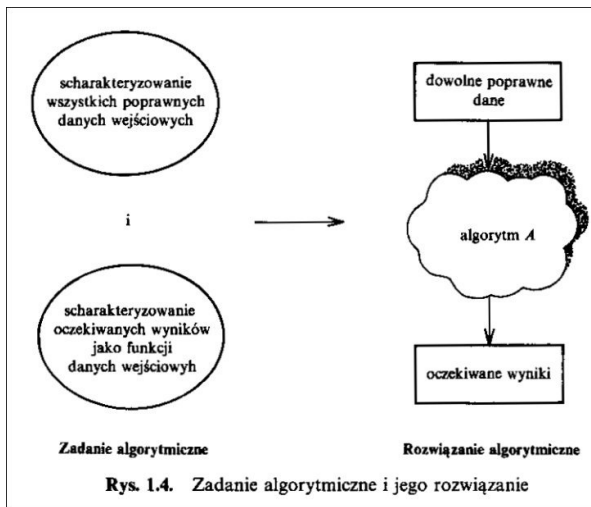
- warunki jakie muszą spełniać dane wejściowe
- określenie oczekiwanych wyników jako funkcji danych wejściowych

Dodatkowe ograniczenia algorytmu, np. dotyczące ilości operacji.

PRZYKŁAD: ALGORYTM EUKLIDESA

Dane wejściowe: liczby całkowite $x, y > 0$

Wynik: $NWD(x, y)$



CECHY ALGORYTMU

- **skończoność** - ograniczona liczba kroków
- **poprawność** - zgodny ze specyfikacją
- **uniwersalność** - poprawny dla klasy problemów
- **efektywność** - niska złożoność to gwarancja użyteczności
- **określoność** - zrozumiałe polecenia, możliwe do wykonania w jednoznacznej kolejności
- określony stan początkowy i wyróżniony koniec

Ciąg struktur sterujących definiuje kolejność wykonywanych operacji.

- bezpośrednie następstwo: *wykonaj A, potem B*
- wybór warunkowy (rozgałęzienie): *jeśli Q, to wykonaj A, w przeciwnym razie wykonaj B*
- iteracja ograniczona: *wykonaj A dokładnie N razy*
- iteracja warunkowa: *dopóki Q, wykonuj A*
- instrukcja skoku: *skocz do G*
- podprogram - wyodrębniony fragment programu, funkcja

Struktury sterujące można dowolnie składać: np. pętle zagnieżdżone. „Nie ma granic zawitości algorytmów”.

- Opis językiem naturalnym
- Lista kroków
- Schematy blokowe
- Pseudo-języki
- Języki wysokiego poziomu

PROBLEM:

znalezienie największej liczby całkowitej dzielącej bez reszty liczby całkowite dodatnie a i b

POMYŚL:

Zauważmy, że

$$NWD(a, b) = k \implies a = nk, b = mk \implies a - b = (m - n)k$$

Stąd dla $a > b$ zachodzi

$$NWD(a, b) = NWD(a - b, b) = NWD(a - b, a)$$

.

ALGORYTM EUKLIDESA

Dane są dwie liczby całkowite. Odejmij od większej liczby mniejszą liczbę a większą liczbę zastąp uzyskaną różnicą. Powtarzaj tę czynność tak długo aż obie liczby będą równe. Otrzymana liczba jest największym wspólnym dzielnikiem liczb wejściowych.

SPECYFIKACJA

Dane wejściowe: liczby całkowite $a, b > 0$

Wynik: liczba całkowita a stanowiąca największy wspólny dzielnik

LISTA KROKÓW

- 1 jeśli a jest równe b to jest to największy dzielnik
- 2 jeśli $a > b$ to zastąp a wartością $a - b$ i wróć do punktu 1
- 3 jeśli $a < b$ to zastąp b wartością $b - a$ i wróć do punktu 1

Algorytm zakłada istnienie operacji $-$, $=$ (porównanie) oraz $>$.

Stan

Blok graniczny: start, stop, przerwanie, opóźnienie.

Instrukcje

Blok operacyjny: zmiana wartości, postaci lub miejsca zapisu danych.

Decyzja

Blok decyzyjny, rozgałęzienie.

Wejście/
wyjście

Wprowadzanie danych i wyprowadzenia wyników.

Łącznik

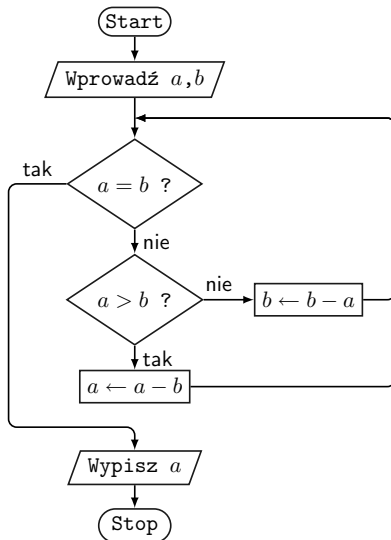
Połączenie z innym fragmentem diagramu.

Podprogram

Wywołanie podprogramu.

PRZYKŁAD: SCHEMAT BLOKOWY

ALGORYTM EUKLIDESA Z ODEJMOWANIEM



Algorytm 1 Algorytm Euklidesa

dopóki $a \neq b$ **wykonuj**

jeżeli $a > b$ **wykonaj**

$a \leftarrow a - b$

w przeciwnym wypadku

$b \leftarrow b - a$

- struktura kodu języka wysokiego poziomu (często Pascal)
- uproszczona składnia na rzecz prostoty i czytelności
- formuły matematyczne, język naturalny, podprogramy
- nie zawiera szczegółów implementacji
 „Dla ludzi, nie dla maszyn”.

```
1  /*  Algorytm Euklidesa.  */  
2  
3  #include <stdio.h>  
4  
5  int main()  
6  {  
7      int a, b;  
8  
9      printf("Podaj dwie liczby calkowite: ");  
10     scanf("%d %d", &a, &b);  
11  
12     while (a != b)  
13         if (a > b) a = a - b;  
14         else     b = b - a;  
15  
16     printf("NWD = %d\n", a);  
17  
18     return 0;  
19 }
```



```
1  program NWD(input,output);
2  { Algorytm Euklidesa. }
3  var
4      A, B : Integer;
5  begin
6      Writeln('Podaj dwie liczby calkowite: ');
7      Readln(a,b);
8
9      while a <> b do
10     begin
11         if a > b then a := a - b
12         else b := b - a;
13     end;
14     Writeln('NWD = ', a);
15 end.
```

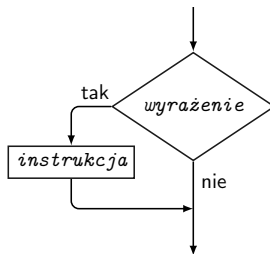
PROGRAM W JĘZYKU FORTRAN 77

```
1      PROGRAM EUCLID1
2  c    Algorytm Euklidesa w jezyku Fortran 77
3
4      WRITE (*,*) 'Podaj dwie liczby calkowite: '
5      READ (*,*) N, M
6
7      DO WHILE ( N .NE. M )
8          IF ( N .GT. M ) THEN
9              N = N - M
10         ELSE
11             M = M - N
12         ENDIF
13     ENDDO
14     WRITE (*,*) 'NWD= ', N
15     END
```

 euclid1.f

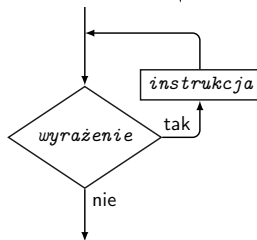
WARUNEK IF (JEŻELI)

```
if ( wyrażenie )  
    instrukcja
```



PĘTLA WHILE (DOPÓKI)

```
while ( wyrażenie )  
    instrukcja
```



- Czy algorytm jest poprawny? Dla jakich danych?
- Problem stopu.
- Efektywność algorytmu. Ile iteracji należy się spodziewać dla różnych danych?

Algorytm 2 Algorytm Euklidesa

1: **dopóki** $b \neq 0$ **wykonuj**
2: $c \leftarrow a \bmod b$
3: $a \leftarrow b$
4: $b \leftarrow c$

- wymaga operacji dzielenie modulo oraz $\neq$
- złożoność: dla $a > b \geq 0$ co najwyżej $2 \log_2(a + b)$ iteracji

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, c;
6
7      printf("Podaj dwie liczby calkowite: ");
8      scanf("%d %d", &a, &b);
9      printf("NWD(%d,%d) = ", a, b);
10
11     while (b != 0)
12     {
13         c = a % b;
14         a = b;
15         b = c;
16     }
17     printf("%d\n", a);
18
19     return 0;
20 }
```

ESENCJA PROGRAMOWANIA PROCEDURALNEGO

- **Podprogram** - funkcja lub procedura, wydzielona część programu, która może być wielokrotnie użyta
- ekonomiczność - ujednolica powtarzające się bloki programu - mniej kodowania
- przejrzystość, nawet przy złożonych i obszernych algorytmach
- podprogram staje się nową instrukcją elementarną
w języku C brak wbudowanych funkcji, tylko `main()`
- uproszczenie problemu poprzez rozbiecie na mniejsze pod-problemy
 - programowanie zstępujące (*top-down*)
 - programowanie wstępujące (*bottom-up*)
- podprogram uruchamiający sam siebie - rekurencja

```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
```

 euclid3.c

```
1  int nwd(int a, int b)
2  {
3      if (b == 0) return a;
4      return nwd(b, a % b);
5  }
```

 euclid3rekurencja.c

Algorytm 3 Sortowanie przez wybieranie (selection sort)

Dane wejściowe: ciąg n liczb $A = \{a_1, a_2, \dots, a_n\}$

Wynik: uporządkowany ciąg $a_1 \leq a_2 \leq \dots \leq a_n$

1: $i \leftarrow 1$

2: **dopóki** $i < n$ **wykonuj**

3: $k \leftarrow \text{minind}(\{a_i, a_{i+1}, \dots, a_n\})$

▷ Podprogram

4: $a_i \longleftrightarrow a_k$

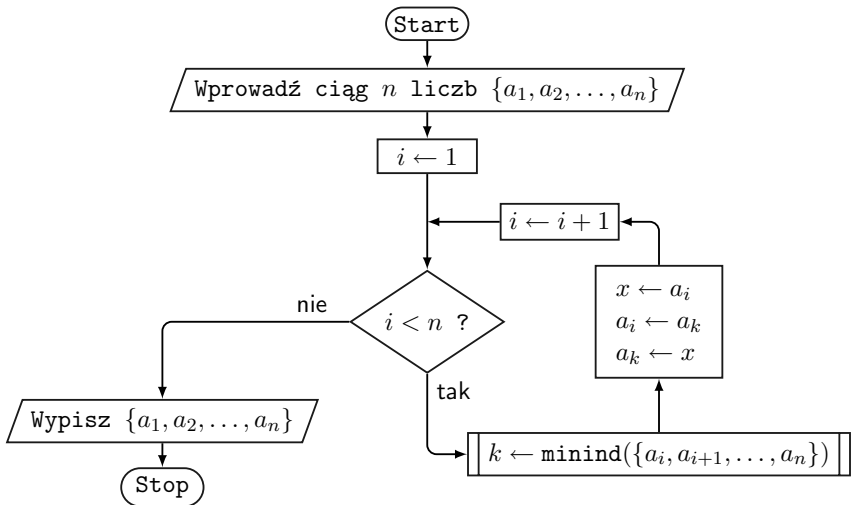
5: $i \leftarrow i + 1$

`minind()` w podanym ciągu wyszukuje indeks elementu o najmniejszej wartości.

Wizualizacja algorytmów sortowania:  AlgoRythmics

PRZYKŁAD: SCHEMAT BLOKOWY

SORTOWANIE PRZEZ WYBIERANIE



ZŁOŻONOŚĆ OBLICZENIOWA

liczba operacji wykonywanych przez algorytm. Zazwyczaj wyznaczana względem ilości danych lub ich rozmiaru.

Ile operacji wymaga ...

- porównanie dwóch liczb całkowitych?
- obliczenie $NWD(a, b)$?
- znalezienie pewnego elementu wśród N elementów?
- ile operacji wymaga posortowanie listy N elementów?
- Problem komiwojażera: znalezienie najkrótszej drogi łączącej wszystkie miasta?

PROBLEM TSP

TRAVELLING SALESMAN PROBLEM



Tabela 1.1. Czasy znalezienia przez komputer, wykonujący 100 miliardów operacji na sekundę, najkrótszej trasy podróży premiera (zob. ćwiczenie 1.3)

Liczba województw	Czas znajdowania najkrótszej trasy
17	3.5 minuty
25	$2 \cdot 10^5$ lat
49	$4 \cdot 10^{42}$ lat

Rysunek 1.1. Najkrótsza trasa premiera przebiegająca przez wszystkie miasta wojewódzkie w podziale administracyjnym z 1975 roku (A. Adrabiński)

[1] M.M. Sysło, „Algorytmy”

PROBLEMY O „ROZSĄDNYCH” ROZWIĄZANIACH

PROBLEMY P I NP

Funkcja \ N	10	50	100	300	1000
$5N$	50	250	500	1500	5000
$N \times \log_2 N$	33	282	665	2469	9966
N^2	100	2500	10 000	90 000	1 milion (7 cyfr)
N^3	1000	125 000	1 milion (7 cyfr)	27 milionów (8 cyfr)	1 miliard (10 cyfr)
2^N	1024	liczba 16-cyfrowa	liczba 31-cyfrowa	liczba 91-cyfrowa	liczba 302-cyfrowa
$N!$	3,6 miliona (7 cyfr)	liczba 65-cyfrowa	liczba 161-cyfrowa	liczba 623-cyfrowa	niewyobrażal- nie duża
N^N	10 miliardów (11 cyfr)	liczba 85-cyfrowa	liczba 201-cyfrowa	liczba 744-cyfrowa	niewyobrażal- nie duża

Dla porównania: liczba protonów w znanym wszechświecie ma 126 cyfr,
liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry.

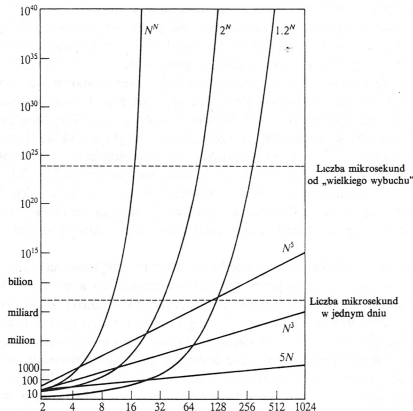
Rys. 7.3. Niektóre wartości pewnych funkcji

[2] D. Harel, *Rzecz o istocie informatyki. Algorytmika.*

PROBLEMY O „ROZSĄDNYCH” ROZWIĄZANIACH

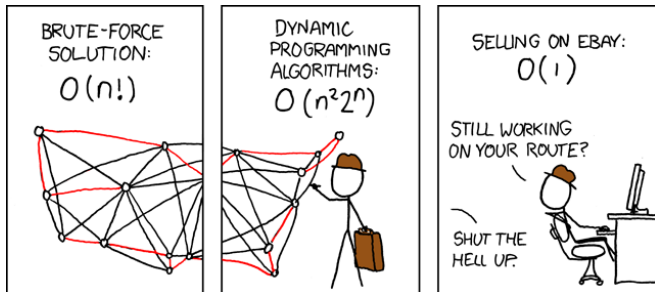
PROBLEMY P I NP

- Rozsądne rozwiązania, wykonywalne w czasie wielomianowym
 $\log N, N, N \log N, N^7 + N^3 + 2$
- Nerozsądne, niepraktyczne, czas ponad-wielomianowy
 $2^N, 1.001^N + N^7, N^N, N!$



Rys. 7.4. Tempo wzrostu pewnych funkcji

[2] D. Harel, *Rzecz o istocie informatyki. Algorytmika.*



ALGORYTMY MANIPULUJĄ DANymi



Reprezentacja danych kształtuje algorytm!

- zmienne: liczby, znaki, $a = 3$
- tablice, wektory, listy, $a[3]=3$
- tablice wielowymiarowe, tablice tablic, $a[3,3]=3$ $a[3][3]=3$
- rekordy, struktury, $a.b=3$ $a.c='a'$
- kolejki, stosy, drzewa
- pliki, bazy danych

Złożoność pamięciowa algorytmu - miara ilości wykorzystanej pamięci

OD PROBLEMU DO ROZWIĄZANIA

- analiza problemu
- specyfikacja zadania:
 - dopuszczalny zestaw danych wejściowych
 - pożądane wyniki jako funkcja danych wejściowych
- algorytm - rozwiązanie zadania
- poprawność algorytmu:
 - względem specyfikacji
 - problem stopu
 - efektywność (złożoność): „Każda akcja zajmuje czas!”
- implementacja algorytmu $\Rightarrow$ program

-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  D. Harel, *Rzecz o istocie informatyki. Algorytmika.*, WNT, Warszawa, 1992.
-  Fulmański Piotr, Sobieski Ścibór, „*Wstęp do informatyki*”, Uniwersytet Łódzki, 2004

Język ANSI C

Pierwsze starcie.



- 1972 Dennis Ritchie (Bell Labs., New Jersey), projekt języka C na bazie języka B
- 1973 UNIX, jądro w C, pierwszy przenośny system operacyjny
- 1978 D. Ritchie, Brian Kernighan, „*The C Programming Language*”
- 1983 Bjarne Stroustrup, Język C++
- 1989 Standard ANSI C, standardowe C, „pure C”, C89, C90
- 1999 Standard C99
- 2011 Standard C11

```
#include <stdio.h>
#define PI 3.1415
```

Dyrektywy preprocesora

```
int main()
{
```

Funkcja główna

```
    int i;
    float x;
```

Deklaracje zmiennych

```
    i = 10 * PI;
    x = 1.0;
```

Instrukcje programu

```
    return 0;
```

```
}
```

Najkrótszy program w C

```
main() {}
```

Witaj świecie

```
#include<stdio.h>

int main()
{
    puts("Witaj świecie!");
    return 0;
}
```

- mały język ale duże możliwości, ważna rola bibliotek
- pliki źródłowe *.c
- pliki nagłówkowe *.h, zawierają deklaracje typów i funkcji, nie zawierają instrukcji
- biblioteki: zbiory typów danych, stałych i funkcji
- biblioteki standardowe, np.: `stdio.h`, `math.h`
- dostęp do pamięci i rejestrów
- zwięzły kod - ale nie wolno przesadzać
- C nie chroni programisty przed nim samym

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Instrukcje zaczynające się od znaku #. Nie są słowami języka C i wykonywane są przed właściwą kompilacją.

- `include`
dołączenie pliku nagłówkowego zawierającego definicje funkcji, typów i stałych

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

- `define`
pozwala zdefiniować stałe lub przezwać słowa kluczowe

```
#define PI 3.1415  
#define TRUE 1  
#define FALSE 0  
#define real float
```

- **preprocesor** wykonuje instrukcje zaczynające się znakiem # (dyrektywy preprocesora). Przygotowuje pliki do kompilacji.
pliki źródłowe (*.c, *.h) $\Rightarrow$ przetworzone pliki
`#include<stdio.h>`
`#define PI 3.14`
- **kompilacja** tłumaczy instrukcje C na kod maszynowy
przetworzone pliki $\Rightarrow$ pliki obiektowe (*.obj, *.o)
- **konsolidacja (linkowanie)** łączy pliki obiektowe w aplikację
pliki obiektowe + biblioteki $\Rightarrow$ program (*.exe , a.out)

- **zmienne** są określone przez typ i unikatową nazwę
- każda zmienna zajmuje pamięć: 1B, 2B, 4B, 8B, ...
- adres zmiennej określa jej położenie w pamięci
- binarna reprezentacja zmiennych
- skończoność - liczby reprezentowane poprawnie w pewnym przedziale
- rozdzielczość - liczby rzeczywiste reprezentowane z pewną dokładnością

`char a='C';`

`&a`

0	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

 = 67

typ	reprezentacja
char	znak (<i>character</i>), kod ASCII zakres wartości $[-127, 128]$
int	liczby całkowite (<i>integer</i>) zakres $[-2^{31}, 2^{31}]$
float	liczby rzeczywiste, zmiennoprzecinkowa (<i>floating point</i>) zakres $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$ najmniejsza dodatnia wartość 1.17×10^{-38}
logiczny	brak typu logicznego wartość całkowita 0 to fałsz a wartość różna od 0 to prawda
void	typ pusty, brak typu

DEKLARACJA

powiązanie zmiennej, stałej lub funkcji danego typu z identyfikatorem (unikatową nazwą).

IDENTYFIKATORY

- dozwolone znaki: litery a-z, A-Z, cyfry 0-9, podkreślnik _
- cyfra nie może być pierwszym znakiem
- małe i duże litery są rozróżniane: a $\neq$ A
- zarezerwowane słowa: zadeklarowane wcześniej identyfikatory, słowa kluczowe

Przykłady deklaracji zmiennych:

```
int a,b,c;  
float srednica_kola;  
char PewienZnak;  
float x1, x2, x3;
```

```
int 0abc;  
float średnica;  
char pewien znak;  
int wazna-zmienna;
```

Źle!

INSTRUKCJA PROSTA

wyrażenie ;

```
a = x + 1;
```

Instrukcja prosta jest zawsze zakończona średnikiem.

INSTRUKCJA ZŁOŻONA (BLOK INSTRUKCJI)

```
{  
    instrukcja 1  
    instrukcja 2  
    instrukcja 3  
}
```

```
{  
    float x, y;  
    x = 1.0;    y = 2.4;  
    x = x + y;  
}
```

INSTRUKCJE STERUJĄCE

if else do while for goto

OPERATORY ARYTMETYCZNE

* / % + -

OPERATORY RELACJI

< > <= >= == !=

OPERATORY LOGICZNE

! && ||

OPERATOR PRZYPISANIA

=

OPERACJA PRZYPISANIA WARTOŚCI

zmienna = wyrażenie;

```
int i, j, k;
```

Deklaracje zmiennych

```
float x = 1.5;
```

```
float y = 1.5e-5;
```

Inicjalizacja

```
char znak = 'A';
```

```
i = i + 3;
```

brak inicjalizacji

```
3 = x;
```

zła kolejność

```
x + y = x - y;
```

```
znak = 65;
```

błąd

	przypisanie	porównanie
C	$a = 3$	$a == 3$
Pascal	$a := 3$	$a = 3$
pseudo-kod	$a \leftarrow 3$	$a = 3$

*	mnożenie	$x * y$	$x \cdot y$
/	dzielenie	x / y	$\frac{x}{y}$
+	dodawanie	$x + y$	$x + y$
-	odejmowanie	$x - y$	$x - y$
%	reszta z dzielenia (modulo)	$x \% y$	$x \bmod y$

- Reszta z dzielenia (modulo, %) określona jest tylko dla argumentów całkowitych
- Operator dzielenia / dla argumentów całkowitych realizuje **dzielenie bez reszty!**

$$1/2 \rightarrow 0$$

$$1/2.0 \rightarrow 0.5$$

PRIORYTETY OPERATORÓW

*	/	%	wyższy priorytet
+	-		niższy priorytet

Dla operatorów o takim samym priorytecie obliczenia są wykonywane od lewej do prawej.

$$z = \frac{x}{2y}$$

```
z = x / 2 * y;      /* złe */
z = x / (2 * y);
```

$$z = \frac{x - y}{x + y}$$

```
z = x - y / x + y;  /* złe */
z = (x - y) / (x + y);
```

W razie wątpliwości użyj nawiasów okrągłych.

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

stdio.h - **Standard input/output**, biblioteka obsługująca komunikację ze standardowym wejściem i wyjściem aplikacji.

FUNKCJA PRINTF()

formatowane wyjście (wyświetlanie w terminalu)

```
printf("format", arg1, arg2, ... )
```

FUNKCJA SCANF()

formatowane wejście (odczyt z klawiatury)

```
scanf("format", adres1, adres2, ... )
```

FORMAT

specyfikator	znaczenie	przykład	wynik
%f	zmiennoprzecinkowa	<code>printf("%f",3.14);</code>	3.140000
%.2f	2 miejsca po przecinku	<code>printf("%.2f",3.14);</code>	3.14
%d	dziesiętna	<code>printf("%d",75);</code>	75
%c	znak	<code>printf("%c",75);</code>	K
%x	szesnastkowy	<code>printf("%x",75);</code>	4b

Przykład z większą liczbą argumentów

```
int a=2;
int b=3;
printf("Liczba %d plus %d wynosi %d\n", a, b , a + b);
```

Wynik:

Liczba 2 plus 3 wynosi 5

SYMBOLE SPECJALNE

symbol	znaczenie	przykład	wynik
\n	nowa linia	printf("n\nn")	n n
\t	tabulator	printf("t\tt")	t t
\"	"	printf("raz \"dwa\"")	raz "dwa"
\\	ukośnik \	printf("C:\\Users\\")	C:\Users\
%%	%	printf("%d%", 5)	5%
\?	?	printf("Jeszcze raz\?")	Jeszcze raz?

Przykład

```
printf("P\tj\tz\nr\te\ta\no\ts\tb\ng\tt\ta\nr\t\t");
printf("w\ na\t \tn\nm\t \te\no\nw\ na\nn\ni\ne\n");
```

- specyfikator formatu powinien pasować do typu zmiennej

```
int a;  
scanf("%f",&a);
```

problem !!!

- drugim argumentem funkcji `scanf` jest adres zmiennej (pamiętaj o `&`)
- formaty `%f` i `%d` pomijają poprzedzające liczbę białe znaki
- znak niepasujący do formatu przerywa wczytywanie i pozostaje w strumieniu wejściowym

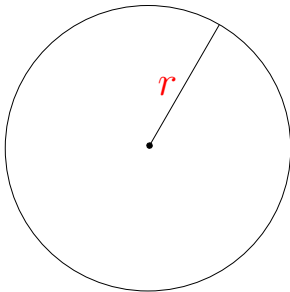
```
char a;  
float x,y;  
scanf("%f%f", &x, &y);  
scanf("%c",&a);
```

wczytywanie 2 wartości

czyta pojedynczy znak

PRZYKŁAD: POLE I OBWÓD KOŁA

Problem: wyznacz pole i obwód koła o promieniu r .



$$P_{\circ} = \pi r^2 \quad O_{\circ} = 2\pi r$$

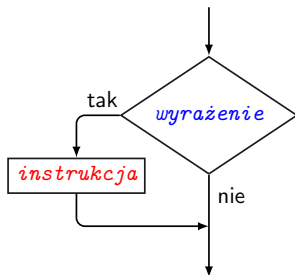
PRZYKŁAD: POLE I OBWÓD KOŁA

```
1  #include <stdio.h>
2  #define PI 3.14159
3
4  int main()
5  {
6      float r,pole,obw;
7
8      printf("Podaj promien kola\nr= ");
9      scanf("%f",&r);
10
11     pole = PI*r*r;
12     obw  = 2*PI*r;
13
14     printf("Pole kola o promieniu %f wynosi %f\n",r,pole);
15     printf("Obwod kola o promieniu %f wynosi %f\n",r,obw);
16
17     return 0;
18 }
```


INSTRUKCJA WARUNKOWA IF ELSE

WARUNEK IF (JEŻELI)

```
if ( wyrażenie )  
    instrukcja
```



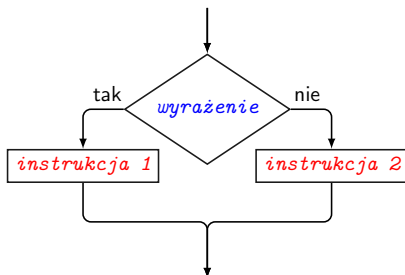
PRZYKŁAD:

```
if( x > 0 ) printf("liczba dodatnia");  
  
if ( x % 2 == 0 )  
{  
    printf("liczba parzysta");  
    x=x-1;  
}
```

INSTRUKCJA WARUNKOWA IF ELSE

WARUNEK IF ELSE

```
if ( wyrażenie )  
    instrukcja 1  
else  
    instrukcja 2
```



PRZYKŁAD:

```
if( x % 2 ) printf("liczba nieparzysta");  
else printf("liczba parzysta");
```

```
if ( x > 0 )  
{  
    printf("liczba dodatnia");  
    x=x-1;  
}  
else x=0;
```

operator	znaczenie	przykład	mat.
<	mniejszy niż	$x < y$	$x < y$
>	większy niż	$x > y$	$x > y$
<=	mniejszy lub równy	$x <= y$	$x \leq y$
>=	większy lub równy	$x >= y$	$x \geq y$
==	równy	$x == y$	$x = y$
!=	różny	$x != y$	$x \neq y$

operator	znaczenie	przykład	mat.
!	negacja (NOT)	!x	$\neg x$
&&	koniunkcja (AND)	x>1 && y<2	$x > 1 \wedge y < 2$
	alternatywa (OR)	x<1 y>2	$x < 1 \vee y > 2$

x	!x
0	1
1	0

x	y	x && y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x y
0	0	0
0	1	1
1	0	1
1	1	1

```
if ( x > 0 )
    if ( x < 10 )
        printf("liczba wieksza od 0 i mniejsza niz 10");

if ( x > 0 && x < 10 )
    printf("liczba wieksza od 0 i mniejsza niz 10");

if (!(x > 0)) printf("liczba ujemna lub zero");

if ( !x > 0 ) printf("\?");
if ( x+1 > y-1 ) printf("\?");
if ( x || ! y && z ) printf("\?");
```

Kolejność operatorów: !, arytmetyczne, relacji, &&, ||
W razie wątpliwości użyj nawiasów okrągłych.

PRZYKŁAD: RÓWNANIE Z JEDNĄ NIEWIADOMĄ

Problem: znajdź miejsce zerowe funkcji liniowej

$$f(x) = ax + b$$

Algorytm 4 Równanie z jedną niewiadomą

Dane wejściowe: współczynniki $a, b \in \mathbb{R}$

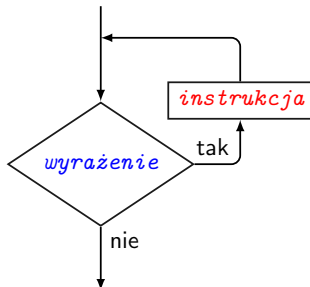
Wynik: miejsce zerowe $x_0 \in \mathbb{R}$ lub informacja o braku rozwiązania

- 1: **jeżeli** $a \neq 0$ **wykonaj**
 - 2: $x_0 \leftarrow -\frac{b}{a}$
 - 3: **wypisz:** x_0
 - 4: **w przeciwnym wypadku**
 - 5: **wypisz:** Brak rozwiązania
-

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float a, b, x0;
6
7      printf("Podaj współczynniki równania\n");
8      printf("a = "); scanf("%f",&a);
9      printf("b = "); scanf("%f",&b);
10
11     if( a != 0.0 )
12     {
13         x0=-b/a;
14         printf("x0 = %.4f\n",x0);
15     }
16     else printf("Brak rozwiązań\n");
17
18     return 0;
19 }
```

PĘTLA WHILE (DOPÓKI)

```
while ( wyrażenie )
    instrukcja
```



PRZYKŁAD

```
int n = 10;
while( n > 0 )
{
    printf("%d\n", n);
    n = n - 1;
}
```

PĘTLA NIESKONCZONA

```
while(1) printf("C");
```


PRZYKŁAD: WYZNACZANIE SILNI $n!$

Problem: wyznaczenie wartości silni $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$

Algorytm 5 Silnia

Dane wejściowe: liczba całkowita $n \geq 0$

Wynik: wartość $x = n!$

1: $i \leftarrow 2$

2: $x \leftarrow 1$

3: **dopóki** $i \leq n$ **wykonuj**

4: $x \leftarrow x \cdot i$

5: $i \leftarrow i + 1$

6: **wypisz** x

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int x, i, n;
6
7     printf("n = ");   scanf("%d",&n);
8
9     if(n<0)
10    {
11        printf("Zle dane: n<0\n");
12    }
13    else
14    {
15        i=2;  x=1;
16        while( i <= n )
17        {
18            x = x * i;
19            i = i + 1;
20        }
21        printf("%d! = %d\n",n,x);
22    }
23    return 0;
24 }
```

TABLICA

przechowuje elementy tego samego typu, elementy identyfikowane liczbami (indeksem).

PRZYKŁAD

```
int tab[4];
```

Deklaracja tablicy 4 elementowej

```
tab[0] = 13;
```

```
tab[1] = 4;
```

```
tab[2] = -3;
```

```
tab[3] = tab[0] - 1;
```

```
tab[4] = -1;
```

Tablice w C są indeksowane od 0

Źle ! Poza zakresem.

	0	1	2	3
tab	13	4	-3	12

PRZYKŁAD: ODWRACANIE KOLEJNOŚCI

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int tab[100];
6      int i = 0;
7
8      printf("Podaj sekwencje liczb calkowitych.\n");
9      printf("Aby zakonczyc podaj 0.\n");
10
11     while((tab[i-1] != 0 || i == 0 ) && i < 100)
12     {
13         scanf("%d", &(tab[i]));
14         i = i + 1;
15     }
16
17     printf("Podales %d liczb.\n", i);
18
19     while(i > 0)
20     {
21         i = i - 1;
22         printf("%d\n", tab[i]);
23     }
24
25     return 0;
26 }
```

```

1  #include<studio.h>;
2
3  char main()
4  {
5      int n
6
7      printf("Podaj liczbe calkowita wieksza od zera: ");
8      scanf("%f",&n);
9
10     if(n <= 0)
11         if ( n==0 ) printf("To jest zero!\n");
12     else
13     {
14         printf("Dzielniki liczby %d:\n",&n);
15         int i;
16         while( i<n );
17         {
18             if( n % i ) printf("%c/n",i);
19             i = i + 1;
20         }
21     }
22     return 0;
23 }
```

```

1  #include<stdio.h>                                /* studio.h, srednik */
2
3  int main()                                         /* int */
4  {
5      int n,i=1;                                    /* srednik , deklaracja , inicjalizacja */
6
7      printf("Podaj liczbe calkowita wieksza od zera : ");
8      scanf("%d",&n);                               /* format, adres */
9
10     if(n <= 0)
11     {                                              /* nawiasy */
12         if ( n==0 ) printf("To jest zero!\n");    /* == */
13     }
14     else
15     {
16         printf("Dzielniki liczby %d:\n",n);
17         while( i<=n )                             /* srednik , p. nieskonczona */
18         {
19             if( n % i == 0 ) printf("%d\n",i);     /* %d, \n, == */
20             i = i + 1;
21         }
22     }
23     return 0;
24 }

```

```
1 #include <stdio.h>
2 int nwd(int a,int b){int c;
3 while(b!=0){c=a%b;a=b;b=c;}
4 return a;} int main(){
5 int a,b; printf("Podaj dwie li"
6 "czby calkowite: "); scanf("%d %d",
7 &a,&b); printf("NWD(%d,%d) = %d\n",
8 a,b,nwd(a,b));return 0;}
```

 nwd-balagan.c

- Czytelność przede wszystkim
-  The International Obfuscated C Code Contest

```
1  #include<stdio.h>
2  int nwd(int a,int b)
3  {
4  int c;
5  while (b!=0)
6  {
7  c=a%b;
8  a=b;
9  b=c;
10 }
11 return a;
12 }
13 int main()
14 {
15 int a,b;
16 printf("Podaj dwie liczby calkowite: ");
17 scanf("%d %d",&a,&b);
18 printf("NWD(%d,%d) = %d\n",a,b,nwd(a,b));
19 return 0;
20 }
```

 nwd-balagan2.c

STYL ALLMANA (BSD)

```
int main()
{
    int i;

    scanf("%d", &n);
    i=0;
    while (i < n)
    {
        if( i % 2 )
        {
            printf("%d\n", i);
        }
        i = i + 1;
    }
    return 0;
}
```

STYL K&R (GNU)

```
int main()
{
    int i, n = 100;

    scanf("%d", &n);
    i=0;
    while (i < n){
        if( i % 2 ){
            printf("%d\n", i);
        }
        i = i + 1;
    }
    return 0;
}
```

- Wewnętrzne bloki instrukcji wcięte względem zewnętrznych
- Instrukcje w jednym bloku zaczynają się w tej samej kolumnie
- Nie przesadzaj z długością linii (max. 78 znaków)
- Oddzielaj deklaracje zmiennych od instrukcji lub grupy spójnych instrukcji pustymi liniami
- Długie ciągi instrukcji warto rozbić na kilka linii i otoczyć nawiasami

```
while ( a < b && wpłata(x) != -1 ) {  
    if ( w != NULL ) {  
        a = a - 1 + sin(PI * 2);  
    }  
}
```

- Python - wcięcia elementem składni języka

- Komentarz blokowy umieszcza się pomiędzy /* a */. Zawierają informacje dla czytających kod (nie są kompilowane).

```
int main()  
{  
    /* Wszystko co tutaj jest napisane  
       jest komentarzem i nie zostanie skompilowane */  
  
    int x;    /* Bardzo ważna zmienna */  
    int y;    // Komentarz do końca linii  
}
```

- Nie komentuj oczywistych rzeczy, raczej opis sensu operacji

```
i = i + 1;    /* Zwiększenie licznika o 1 */  
i = i + k;    /* Ustawienie indeksu na ostatni element */
```

- Komentarz liniowy // niedostępny w C89 !

STANDARD C99

- funkcje inline
- deklaracje zmiennych w dowolnym miejscu w programie
- typ logiczny (bool), long long int
- tablice o zmiennej liczbie elementów
- komentarze w stylu C++ // to jest komentarz
- biblioteki, np.: complex.h, stdbool.h

Uwaga: nie wszystkie kompilatory wspierają pełny standard C99
dlatego ANSI C daje największą szansę na przenośność.

- każdą zmienną trzeba zadeklarować (określić typ i nazwę)
- przejrzystość: czytelne nazwy zmiennych i wcięcia
- nie zapomnij o średniku na końcu instrukcji
- operator przypisania `a=b` a operator porównania `a==b`
- najpierw deklaracja potem użycie
- inicjuj zmienne

-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>
-  „C Programming Tutorial”, Tutorials Point,
<http://www.tutorialspoint.com/cprogramming/>

Instrukcje sterujące

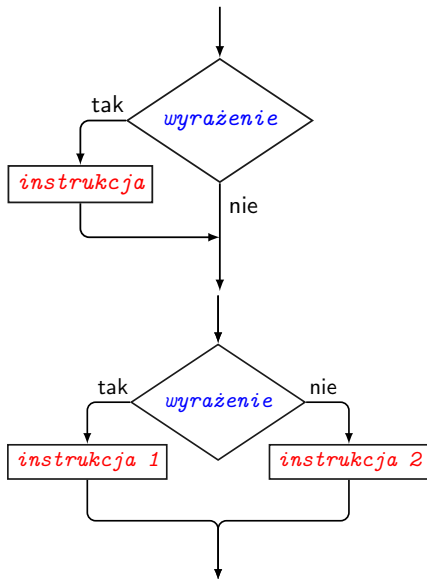
PRZYPOMNIENIE: OPERATORY

Operator przypisania			
=	przypisanie	$x = y$	$x \leftarrow y$
Operatory arytmetyczne			
*	mnożenie	$x * y$	$x \cdot y$
/	dzielenie	x / y	$\frac{x}{y}$
+	dodawanie	$x + y$	$x + y$
-	odejmowanie	$x - y$	$x - y$
%	reszta z dzielenia (modulo)	$x \% y$	$x \bmod y$
++	inkrementacja	$x++$	$x \leftarrow x + 1$
--	dekrementacja	$x--$	$x \leftarrow x - 1$
Operatory relacji			
<	mniejszy niż	$x < y$	$x < y$
>	większy niż	$x > y$	$x > y$
<=	mniejszy lub równy	$x \leq y$	$x \leq y$
>=	większy lub równy	$x \geq y$	$x \geq y$
==	równy	$x == y$	$x = y$
!=	różny	$x != y$	$x \neq y$
Operatory logiczne			
!	negacja (NOT)	$!x$	$\neg x$
&&	koniunkcja (AND)	$x > 1 \ \&\& \ y < 2$	$x > 1 \wedge y < 2$
	alternatywa (OR)	$x < 1 \ \ \ y > 2$	$x < 1 \vee y > 2$

INSTRUKCJA WARUNKOWA IF ELSE

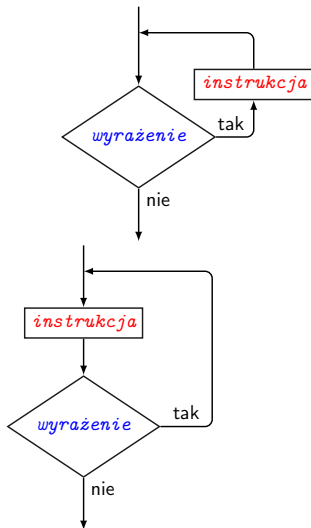
```
if ( wyrażenie )  
    instrukcja
```

```
if ( wyrażenie )  
    instrukcja 1  
else  
    instrukcja 2
```



```
while ( wyrażenie )  
    instrukcja
```

```
do  
    instrukcja  
while ( wyrażenie );
```



Problem: wyznaczenie wartości pierwiastka $\sqrt{a}$

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right) \xrightarrow{n \rightarrow \infty} \sqrt{a}$$

Algorytm 6 Algorytm Herona

Dane wejściowe: liczba $a \geq 0$, wartość początkowa $x_0 > 0$, dokładność obliczeń $\epsilon > 0$

Wynik: wartość przybliżona $x \approx \sqrt{a}$ z dokładnością ϵ

1: $x \leftarrow x_0$

2: **wykonuj**

3: $x_0 \leftarrow x$

4: $x \leftarrow \frac{1}{2} \left(x_0 + \frac{a}{x_0} \right)$

5: **dopóki** $|x - x_0| > \epsilon$

6: **wypisz** x

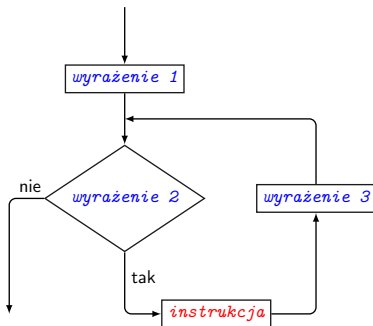
```

1  #include<stdio.h>
2
3  int main()
4  {
5      const float eps=1e-4;
6      float x,a,x0;
7
8      printf("a = "); scanf("%f",&a);
9
10     if( a < 0 ) printf("Zle dane: a < 0\n");
11     else
12     {
13         x0 = 1;
14         x = x0;
15         do
16         {
17             x0 = x;
18             x = (x0 + a/x0)/2;
19         }while(x - x0 > eps || x - x0 < -eps);
20
21         printf("pierwiastek z %f wynosi %f (eps= %f)\n",a,x,eps);
22     }
23     return 0;
24 }

```

for (*wyrażenie 1* ; *wyrażenie 2* ; *wyrażenie 3*)
instrukcja

```
int s=1, n=100, i;  
  
for(i=1; i<n; i++)  
{  
    s = s * i;  
}
```



```

A
while ( B )
{
    instrukcja
    C
}

```

```

i=0;
while( i<n )
{
    printf( "%d\n", i );
    i=i+1;
}

```

```

for ( A ; B ; C )
{
    instrukcja
}

```

```

for(i=0; i<n; i=i+1)
    printf( "%d\n", i );

```

Polecenie break przerywa działanie pętli while, do while, for.

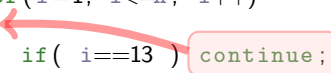
```
1  #include<stdio.h>
2
3  int main()
4  {
5      char c;
6
7      while(1)
8      {
9          printf("Czy przerwac [t/n]? ");
10         scanf(" %c", &c);
11         if( c == 't' || c == 'T' ) break;
12         printf("Przmysl to!\n");
13     }
14     printf("Koniec.\n");
15 }
16 }
```

Czy przerwac [t/n]? n
Przmysl to!
Czy przerwac [t/n]? N
Przmysl to!
Czy przerwac [t/n]? y
Przmysl to!
Czy przerwac [t/n]? t
Koniec.

KOLEJNA ITERACJA: CONTINUE

Polecenie `continue` przechodzi do następnej iteracji (pomija wszystkie instrukcje do końca bloku pętli).

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int i, n=20;
6
7      for(i=1; i<=n; i++)
8      {
9          if( i==13 ) continue;
10         printf("%d\n",i);
11     }
12
13     return 0;
14 }
```



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

 `continue.c`

Instrukcja skoku goto przenosi sterowanie w miejsce kodu oznaczone etykietą.

etykieta:

instrukcje

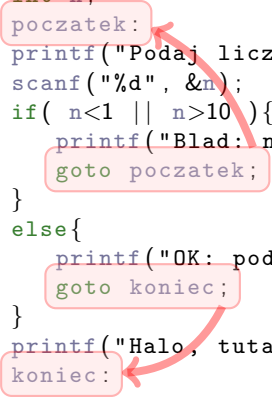
...

goto *etykieta*;

Główne przykazanie **programowania strukturalnego**:

Nie używaj instrukcji skoku!

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6      poczatek:
7      printf("Podaj liczbe z zakresu od 1 do 10\n");
8      scanf("%d", &n);
9      if( n<1 || n>10 ){
10         printf("Blad: niepoprawna wartosc\n");
11         goto poczatek;
12     }
13     else{
14         printf("OK: podales liczbe %d\n", n);
15         goto koniec;
16     }
17     printf("Halo, tutaj jestem!\n");
18     koniec:
19
20     return 0;
21 }
```



- Algorytm zawierający wiele instrukcji skoku, przenoszących sterowanie do przodu lub do tyłu, jest nieczytelny i trudny do zrozumienia a ewentualne modyfikacje często prowadzą do powstawania błędów
- Trudności techniczne, niektóre skoki są niewykonalne:
skoki do wnętrza pętli, z pętli do pętli, z podprogramu (funkcji) do podprogramu, itp. ...
- `break` i `continue` - podobne wątpliwości
- Program używający instrukcji skoku zawsze można przepisać do postaci nie zawierającej tej instrukcji
- Używanie instrukcji skoku to zły styl programowania

POPRZEDNI PRZYKŁAD BEZ SKOKU

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6      int stop = 0;
7
8      while( stop == 0 )
9      {
10         printf("Podaj liczbe z zakresu od 1 do 10\n");
11         scanf("%d", &n);
12         if( n >= 1 && n<=10 ) stop=1;
13         else printf("Blad: niepoprawna wartosc\n");
14     }
15
16     printf("OK: podales liczbe %d\n", n);
17     return 0;
18 }
```

```
1  #include <stdio.h>
2
3  int main ()
4  {
5      int i, j, n;
6
7      printf("Podaj zakres: ");
8      scanf("%d", &n);
9
10     printf("Liczby pierwsze w zakresie od 1 do %d\n", n);
11     for(i=2; i<n; i++) {
12         for(j=2; j <= (i/j); j++)
13             if(!(i%j)) break; /* nie jest l. pierwsza */
14         if(j > (i/j)) printf("%d\n", i);
15     }
16
17     return 0;
18 }
```

PĘTLE ZAGNIEŹDZONE I INSTRUKCJA SKOKU

```
1  int i, j, k;
2
3  for(i =0; i < 100; i = i + 1)
4  {
5      for(j =0; j < i; j = j + 1)
6      {
7          for( k=0; k < j; k = k + 1 )
8          {
9              if ( i + j + k > 42 ) goto koniec;
10             }
11         }
12     }
13
14     koniec:
```

 goto4.c

- przykładowe uzasadnienie użycia instrukcji skoku - zakończenie zagnieżdżonej pętli
- ale zagnieżdżone pętle to też oznaka złych praktyk



Instrukcja warunkowa switch porównuje wyrażenie z listą wartości i wykonuje instrukcje pasującego przypadku (case).

```
switch( wyrażenie )  
{  
    case wartość 1:  
        instrukcja 1  
        break;  
    case wartość 2:  
        instrukcja 2  
        break;  
    ...  
    default :  
        instrukcja n  
}
```



```

1  #include<stdio.h>
2
3  int main()
4  {
5      float x, y;
6      char op;
7
8      scanf("%f %c%f",&x, &op, &y);
9
10     switch(op)
11     {
12         case '+':
13             printf("%f\n", x+y);
14         case '-':
15             printf("%f\n", x-y);
16         case '*':
17             printf("%f\n", x*y);
18             break;
19         case '/':
20             printf("%f\n", x/y);
21             break;
22         default:
23             printf("Nieznana operacja: %c\n", op);
24     }
25     return 0;
26 }

```

3.14 + 5

8.140000

3.14+5

8.140000

3.14

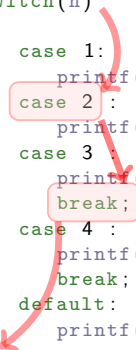
+

5

8.140000

- Dopasowanie przypadków tylko dla zmiennych całkowitych (int, char, enum)
- break kończy działanie instrukcji switch
- instrukcje break i default są opcjonalne
- Po dopasowaniu właściwego przypadku wykonywane są WSZYSTKIE instrukcje aż do napotkania pierwszego wystąpienia break (lub do końca bloku switch). Pominięcie instrukcji break spowoduje więc wykonanie instrukcji dla kolejnego przypadku.

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int n;
6
7      printf("Podaj liczbe od 1 do 4\n");
8      scanf("%d", &n);
9
10     switch(n)
11     {
12         case 1:
13             printf("Przypadek 1\n");
14         case 2 :
15             printf("Przypadek 2\n");
16         case 3 :
17             printf("Przypadek 3\n");
18             break;
19         case 4 :
20             printf("Przypadek 4\n");
21             break;
22         default:
23             printf("Nieznana operacja.\n");
24     }
25     return 0;
26 }
```



Podaj liczbe od 1 do 4
2
Przypadek 2
Przypadek 3

- Instrukcje warunkowe: `if`, `else`, `switch`
- Pętle: `while`, `do while`, `for`
- Operator inkrementacji `++` i dekrementacji `--`
- Instrukcja skoku `goto` (lepiej nie używać)
- Instrukcja przerywania `break` oraz kontynuacji pętli `continue`
- Instrukcja wielokrotnego wyboru `switch`

 David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.

 „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>

Funkcje

czyli jak programować proceduralne.

STRUKTURA PROGRAMU W C

```
#include <stdio.h>
#define PI 3.1415
```

Dyrektywy preprocesora

```
float g = 2.5;
```

Zmienne globalne

```
float kwadrat(float x)
{
    return x*x;
}
```

Definicja funkcji

```
int main()
{
```

Funkcja główna

```
    float x, y;
```

Zmienne lokalne

```
    x = PI * g;
    y = kwadrat(x);
```

Instrukcje programu

```
    return 0;
```

```
}
```

PODPROGRAM

wydzielony fragment programu (kodu) zawierający instrukcje do wielokrotnego użytku

- Dekompozycja złożonego problemu na prostsze
- Przezrzystość
- Unikanie powtórzeń kodu
- Uniwersalność - jak najmniejszy związek z konkretnym kodem
- Biblioteki funkcji
- Nic za darmo - wywołanie funkcji to dodatkowy koszt czasu i pamięci
- Rekurencja(Rekurencja(Rekurencja(Rekurencja(...))))

DEKLARACJA FUNKCJI (PROTOTYP)

Zapowiedź użycia funkcji - określenie nazwy funkcji, typów argumentów i typu wartości zwracanej

```
typ identyfikator(typ arg1, typ arg2, ... );
```

DEFINICJA FUNKCJI

Deklaracja parametrów i instrukcje podprogramu (ciało funkcji).

```
typ identyfikator(typ arg1, typ arg2, ... )  
{  
    deklaracje zmiennych lokalnych  
    instrukcje  
    return wartość;  
}
```

DEKLARACJE

```
float sin(float x);  
  
float pierwiastek(float x);  
  
void wypiszmenu(void);  
  
int random(void);  
  
int nwd(int a, int b);  
  
char getchar(void);  
  
int fff(int z, char z, float a);
```

WYWOŁANIE

```
y = sin(x);  
  
y = pierwiastek(5.0);  
  
wypiszmenu();  
  
i = random();  
  
i = nwd(144, a + 1);  
  
z = getchar();  
  
i = fff(3, 'A', 3.14);
```

```

1  #include <stdio.h>
2
3  int nwd(int a, int b)
4  {
5      int c;
6      while (b != 0)
7      {
8          c = a % b;
9          a = b;
10         b = c;
11     }
12     return a;
13 }
14
15 int main()
16 {
17     int a, b;
18
19     printf("Podaj dwie liczby calkowite: ");
20     scanf("%d %d", &a, &b);
21     printf("NWD(%d,%d) = %d\n", a, b, nwd(a,b));
22
23     return 0;
24 }

```

PARAMETRY FORMALNE I AKTUALNE FUNKCJI

PARAMETRY FORMALNE

```
int nwd(int a, int b)
{
    int c;
    ...
}
```

PARAMETRY AKTUALNE

```
float x,y;
int a=1, b=2, c;

x = sin(1);
c = nwd(144, b);
y = sin(x * nwd(1+a, nwd(100, b))));
```

W języku C argumenty są przekazywane wyłącznie przez wartość.

FUNKCJA BEZ WARTOŚCI ZWRACANEJ - PROCEDURA

```
void wypisz(int n)
{
    while( n>0 )
    {
        printf("Programowaie proceduralne\n");
        n = n - 1;
    }
}
```

FUNKCJA Z WARTOŚCIĄ ZWRACANĄ

```
int silnia(int n)
{
    int i=2;  x=1;
    while( i <= n )
    {
        x = x * i;
        i = i + 1;
    }
    return x;
}
```

- return przerywa działanie funkcji i zwraca wartość z funkcji
- Może pojawić się w dowolnym miejscu wewnątrz funkcji

```
int jest_pierwsza(int n)
{
    int i=2;
    while( i<n )
    {
        if( n % i == 0 ) return 0;
        i = i + 1;
    }
    return 1;
}
```

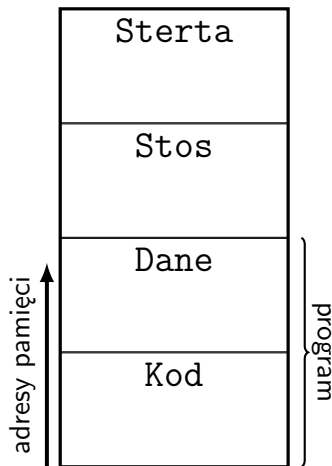
Wartość zwracana jest podstawiona w miejsce wywołania

```
if(jest_pierwsza(a)) printf("%d jest pierwsza\n",a);
```

DEKLARACJA I DEFINICJA FUNKCJI C.D

```
1 #include <stdio.h>
2
3 /* Deklaracja funkcji */
4 int jest_pierwsza ( int n );
5
6 int main()
7 {
8     int n, i=1;
9     printf("n = ");
10    scanf("%d", &n);
11    while(i<=n)
12    {
13        if(jest_pierwsza(i) == 1) printf("%d\n",i);
14        i++;
15    }
16    return 0;
17 }
18
19 /* Definicja funkcji */
20 int jest_pierwsza ( int n )
21 {
22     int i =2;
23     while ( i<=n/i )
24     {
25         if ( n % i == 0 ) return 0 ;
26         i = i + 1;
27     }
28     return 1 ;
29 }
```

W momencie kompilacji musi być znana przynajmniej deklaracja funkcji (jej nazwa i typy argumentów).



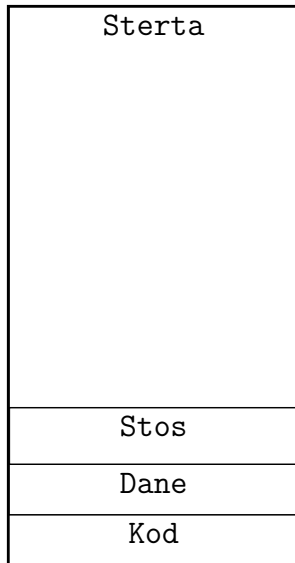
pamięć dostępna do alokacji

zmienne lokalne funkcji wkładane
na stos przy wywołaniu funkcji

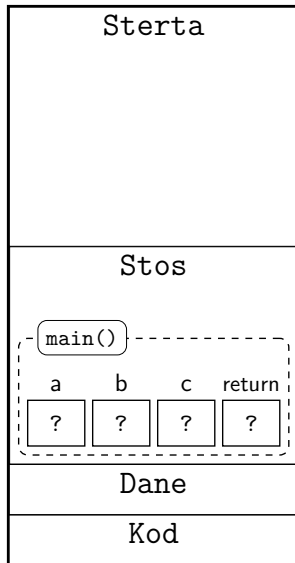
stałe oraz **zmienne globalne** i
statyczne inicjowane przy
uruchomieniu

tekst programu
tylko do odczytu

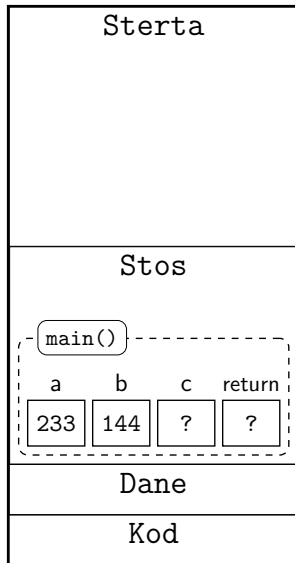

```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



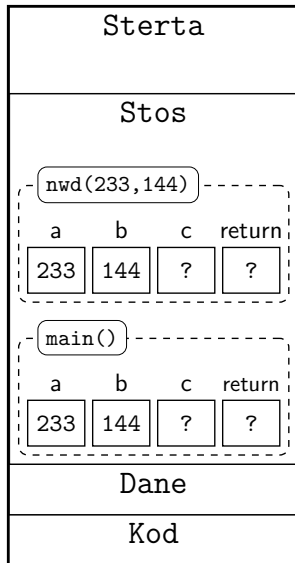
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```

1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }

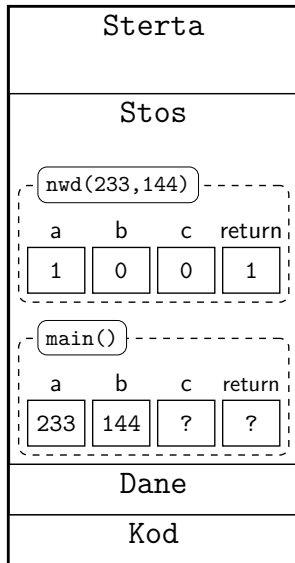
```



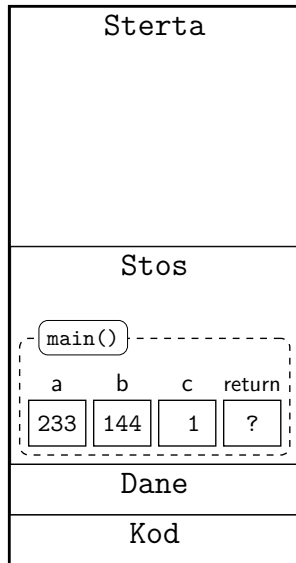
```

1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }

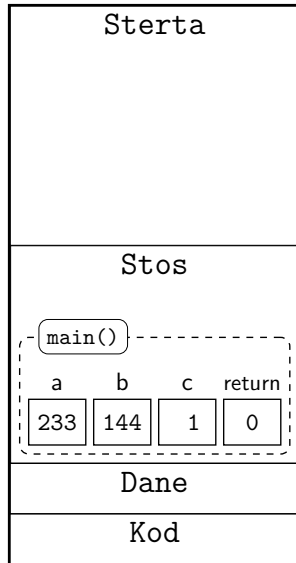
```



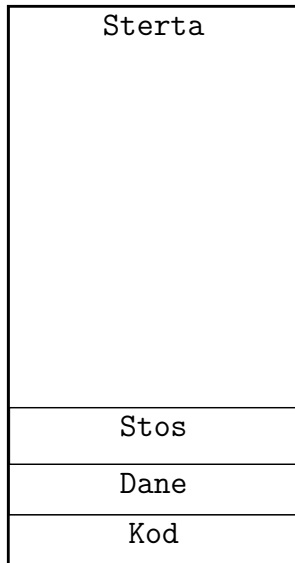
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



ZMIENNA GLOBALNA

dostępna dla wszystkich funkcji programu. Zmienna istnieje przez cały czas działania programu. Deklaracja zmiennej nie znajduje się wewnątrz żadnej funkcji.

```
int a;  
void funkcja()  
{  
    a = a + 1;  
}
```

ZMIENNA LOKALNA (AUTOMATYCZNA)

wewnętrzna zmienna funkcji. Czas życia ograniczony czasem działania funkcji.

```
void funkcja(int a)  
{  
    a = a + 1;  
}
```

```
1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10     printf("%d\t %d\t f\n", globalna, lokalna);
11 }
12
13 int main()
14 {
15     int lokalna = 13;
16
17     printf("globalna lokalna funkcja\n");
18     printf("%d\t %d\t main\n", globalna, lokalna);
19     f();
20     printf("%d\t %d\t main\n", globalna, lokalna);
21     f();
22     printf("%d\t %d\t main\n", globalna, lokalna);
23     return 0;
24 }
```

 globalne.c

```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10     printf("%d\t %d\t f\n", globalna, lokalna);
11 }
12
13 int main()
14 {
15     int lokalna = 13;
16
17     printf("globalna lokalna funkcja\n");
18     printf("%d\t %d\t main\n", globalna, lokalna);
19     f();
20     printf("%d\t %d\t main\n", globalna, lokalna);
21     f();
22     printf("%d\t %d\t main\n", globalna, lokalna);
23     return 0;
24 }

```

Wynik działania:

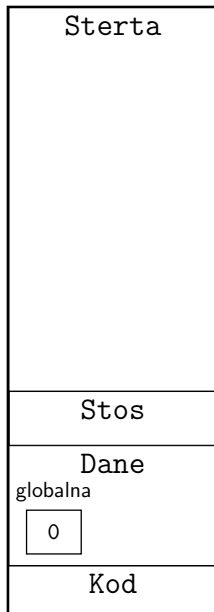
globalna	lokalna	funkcja
0	13	main
1	1	f
1	13	main
2	1	f
2	13	main

 globalne.c

```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

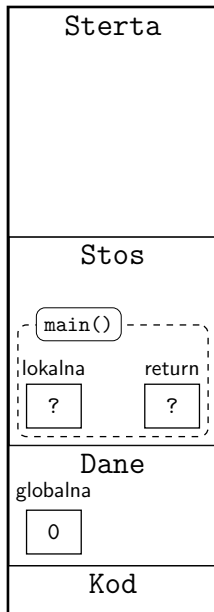
```



```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main() ←
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

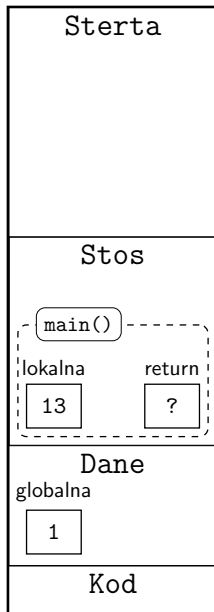
```



```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

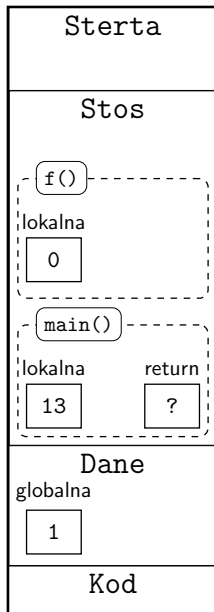
```



```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

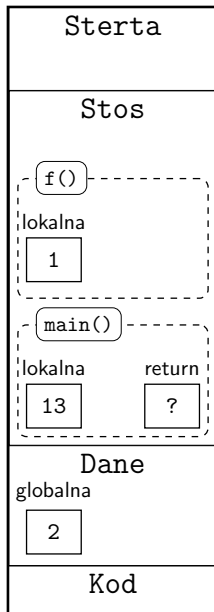
```



```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

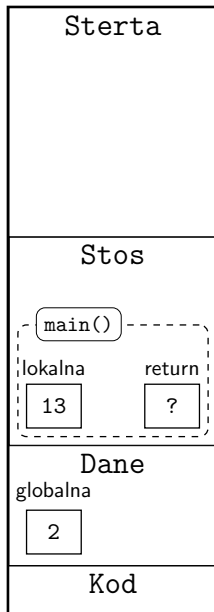
```


```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

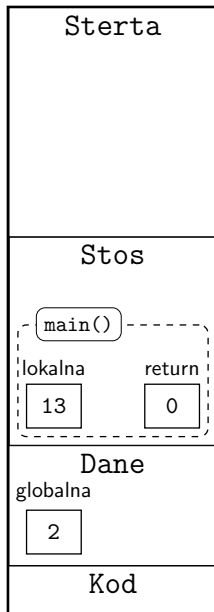
```



```

1  #include<stdio.h>
2
3  int globalna = 0;
4
5  void f(void)
6  {
7      int lokalna = 0;
8      globalna = globalna + 1;
9      lokalna = lokalna + 1;
10 }
11
12 int main()
13 {
14     int lokalna = 13;
15     globalna = globalna + 1;
16     f();
17
18     return 0;
19 }

```

W programowaniu proceduralnym nie używamy zmiennych globalnych!

```
#include<stdio.h>
int x=1, y=1;

int main()
{
    f();
    zzz();
    pewna_funkcja();

    printf("%d %d\n", x, y);
}
```

Źle

```
#include<stdio.h>

int main()
{
    int x=1, y=1;
    y = f(x, 10, x * 2);
    zzz();
    x = pewna_funkcja(x);

    printf("%d %d\n", x, y);
}
```

Dobrze

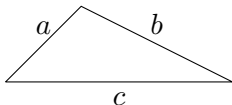
<code><assert.h></code>	Asercje - diagnostyka kodu
<code><ctype.h></code>	Klasyfikacja znaków
<code><float.h></code>	Ograniczenia typów zmiennopozycyjnych
<code><limits.h></code>	Ograniczenia typów całkowitych
<code><signal.h></code>	Obsługa sygnałów
<code><stddef.h></code>	Standardowe definicje (makra)
<code><stdio.h></code>	Operacje wejścia/wyjścia
<code><math.h></code>	Funkcje matematyczne
<code><stdlib.h></code>	Zestaw podstawowych narzędzi, np. <code>rand()</code>
<code><string.h></code>	Obsługa łańcuchów znakowych
<code><time.h></code>	Funkcje obsługi czasu

<code>sin</code>	<code>cos</code>	<code>tan</code>	funkcje trygonometryczne
<code>asin</code>	<code>acos</code>	<code>atan</code>	
<code>sinh</code>	<code>cosh</code>	<code>tanh</code>	funkcje hiperboliczne
<code>exp</code>			f. eksponencjalna e^x
<code>ceil</code>	<code>floor</code>		zaokrąglenia: <i>sufit</i> , <i>podłoga</i>
<code>sqrt</code>			pierwiastek $\sqrt{x}$
<code>pow</code>			potęga x^y
<code>log</code>			logarytm naturalny $\ln(x) = \log_e x$
<code>log10</code>			logarytm dziesiętny $\log_{10} x$
<code>fabs</code>			wartość bezwzględna $ x $
<code>fmod</code>			reszta z dzielenia (zmiennoprzecinkowe)
<code>HUGE_VAL</code>			bardzo duża wartość double

Uwaga: korzystając z kompilatora GCC należy dodać opcję `-lm`
`gcc -lm euclid.c`

PRZYKŁAD: WZÓR HERONA

Problem: pole trójkąta dla danych długości boków a , b i c .



WZÓR HERONA (60 N.E.)

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

gdzie

$$p = \frac{1}{2}(a+b+c)$$

W jakiej sytuacji wyrażenie pod pierwiastkiem jest mniejsze od 0 ?

PRZYKŁAD: POLE TRÓJKĄTA

```
1  #include <math.h>
2
3  /*  Funkcja wyznacza pole trojkata z wzoru Heroana.
4      *  Argumenty a, b i c to dlugosci bokow.
5      *  Jezeli boki a, b, i c nie tworza trojkata
6      *  zwracana jest wartosc -1. */
7  float heron(float a, float b, float c)
8  {
9      float p = (a+b+c)/2;
10     p = p*(p-a)*(p-b)*(p-c);
11     if(p<0) return -1;
12     return sqrt(p);
13 }
```

 heron2.c

```
1  int main()
2  {
3      float a, b, c, pole;
4
5      printf("Podaj dlugosci bokow trojkata: a, b, c > 0\n");
6      scanf("%f%f%f",&a,&b,&c);
7
8      if ( a <= 0 || b <= 0 || c<=0 )
9      {
10         printf("Zle dane: wartosci musza byc dodatnie.\n");
11         return 1;
12     }
13
14     pole = heron(a,b,c);
15     if( pole < 0 )
16     {
17         printf("Zle dane: to nie sa boki trojkata\n");
18         return 2;
19     }
20     printf("Pole wynosi: %f\n", pole);
21
22     return 0;
23 }
```

 heron2.c

Funkcje mogą wywoływać same siebie - **funkcje rekurencyjne**.

```
#include<stdio.h>

void funkcja()
{
    /* cialo funkcji */
    funkcja();
}

int main()
{
    funkcja();
}
```

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n",n,x);
14     return 0;
15 }
```

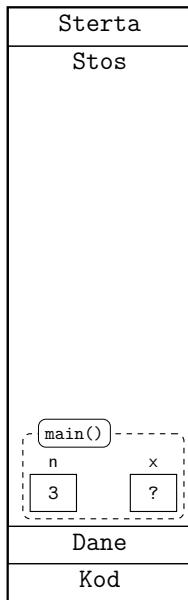
Sterna
Stos
Dane
Kod

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf("%d!=%d\n",n,x);
14     return 0;
15 }

```

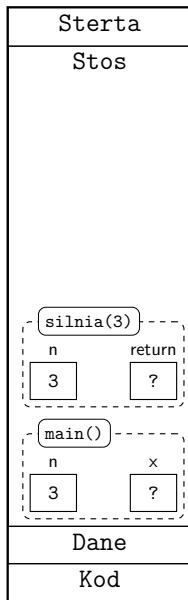


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

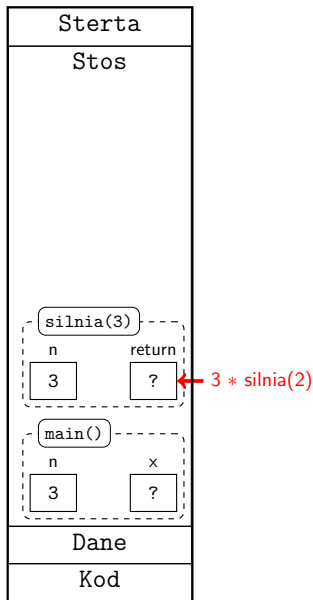


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

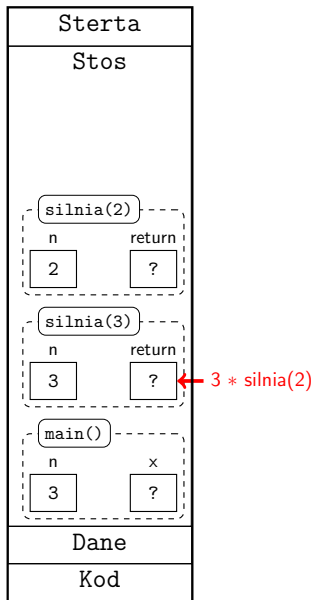


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

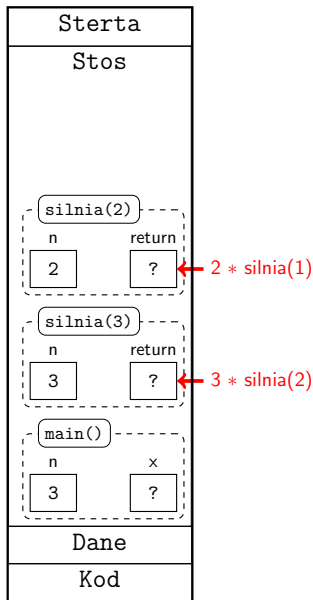


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

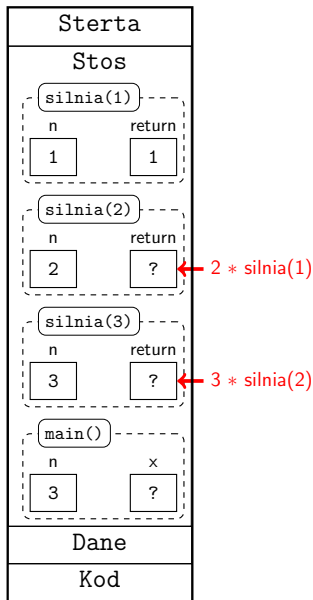


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n",n,x);
14     return 0;
15 }

```

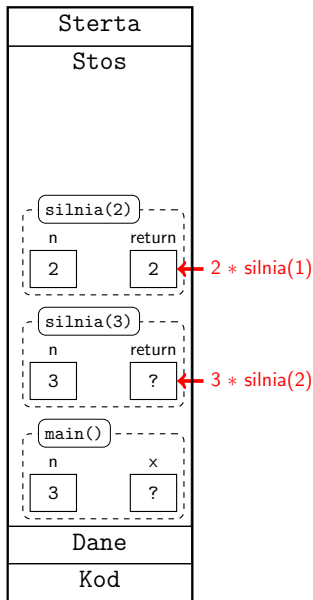


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

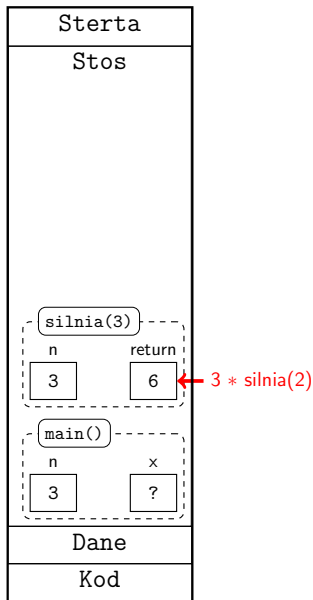


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```

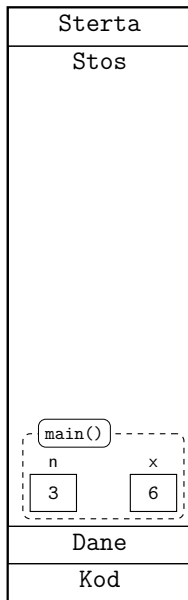


$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n \cdot (n - 1)!$$

```

1  #include<stdio.h>
2
3  int silnia(int n)
4  {
5      if( n<=1 ) return 1;
6      return n * silnia(n-1);
7  }
8
9  int main()
10 {
11     int x, n=3;
12     x = silnia(n);
13     printf ("%d!=%d\n", n, x);
14     return 0;
15 }

```



Problem: zadanie o rozmnażaniu się królików.

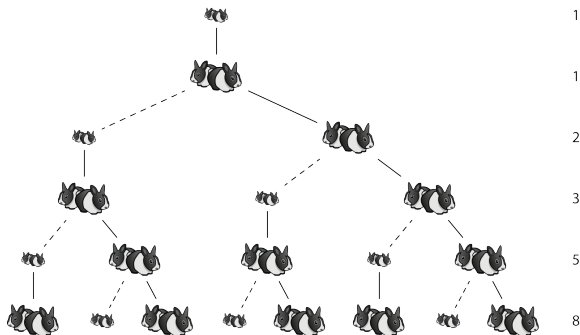
Populacja królików rozmnaża się wg. poniższych zasad:

- rozpoczynamy od pojedynczej pary nowonarodzonych królików,
- króliki stają się płodne po upływie miesiąca życia,
- każda płodna para wydaje na świat parę królików co miesiąc,
- króliki nigdy nie umierają.

Ile par królików będzie w populacji po roku?

ZADANIE O ROZMNAŻANIU KRÓLIKÓW

Ile par królików będzie w populacji po n miesiącach?



Źródło: Jens Bossaert, Curiosa Mathematica,
<http://curiosamathematica.tumblr.com/>

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...,

CIĄG FIBINACCIEGO

$$F_n = \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ F_{n-1} + F_{n-2} & \text{dla } n > 1 \end{cases}$$

REKURENCJA

```
1 int fibonacci(int n)
2 {
3     if( n==0 ) return 1;
4     if( n==1 ) return 1;
5     return fibonacci(n-1) + fibonacci(n-2);
6 }
```

 fib1.c

ITERACJA

```
1 int fibonacci(int n)
2 {
3     int f=1, fp=1, k;
4
5     while( n>1 )
6     {
7         k = f + fp;
8         fp = f;
9         f = k;
10        n--;
11    }
12    return f;
13 }
```

 fib2.c

- Zmienne globalne mają zasięg całego pliku, zmienne lokalne istnieją tylko wewnątrz funkcji
- Programowanie proceduralne: funkcje nie powinny korzystać ze zmiennych globalnych
- Deklaracja funkcji określa nazwę funkcji, typy argumentów i typ wartości zwracanej
- Definicja funkcji to deklaracja + ciało funkcji (implementacja)
- Funkcja przed użyciem musi być przynajmniej zadeklarowana
- `return` zwraca pojedynczą wartość z funkcji
- Pytanie: jak zwrócić z funkcji więcej wartości?
Np. funkcja wyznaczająca miejsca zerowe paraboli.

-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>
-  „C Reference”, <http://en.cppreference.com/w/c>

Tablice i struktury

czyli złożone typy danych.

TABLICA

przechowuje elementy tego samego typu

struktura jednorodna, homogeniczna

Elementy identyfikowane liczbami (indeksem).

8	1	-6	3	5	7	4	9	2	10	-4	88	6	3	1	3	332	2
---	---	----	---	---	---	---	---	---	----	----	----	---	---	---	---	-----	---

STRUKTURA

przechowuje elementy dowolnego typu

struktura niejednorodna, heterogeniczna

Elementy identyfikowane przez nazwy.

"Hans"		
"Kloss"		
4	11	2013
'M'		
181.5		

- wszystkie elementy są tego samego typu
- elementy identyfikowane przez liczbę całkowitą (indeks)
- tablice jednowymiarowe
- rozmiar musi być znany w momencie kompilacji
tablice statyczne
- swobodny dostęp (*random acces*) do elementów
- operator dostępu []
- w C tablice są indeksowane od 0

0	1	2	3	4	5	6	7	8	9
8	1	-6	3	5	7	4	9	2	10

DEKLARACJA TABLICY

`typ identyfikator[rozmiar];`

PRZYKŁAD

```
#define MAX 1000
const int n=200;

int a[10];
float tablica[MAX];
char napis[n];
```

- operator [] daje dostęp do i-tego elementu
- indeksowanie wartościami całkowitymi
- brak kontroli zakresu tablicy podczas kompilacji

PRZYKŁADY

```
tablica[0] = 1.3;  
napis[3] = 'x';  
i = a[i];  
a[i] = a[i] + 5;  
tablica[i-1] = tablica[i];  
tablica[maxind(x)] = tablica[0];
```

```
float t[10];
```

t[0]	t[1]	t[2]	t[2]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]
------	------	------	------	------	------	------	------	------	------

WCZYTYWANIE WARTOŚCI

```
float t[10];
int i=0;

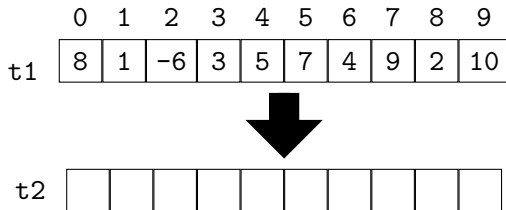
while(i<10)
{
    printf("t[%d]= ", i);
    scanf("%f", &t[i]);
    i = i + 1;
}
```

ZEROWANIE WARTOŚCI

```
float t[10];
int i;

for(i=0; i<10; i++)
{
    t[i] = 0;
}
```

KOPIOWANIE TABLIC

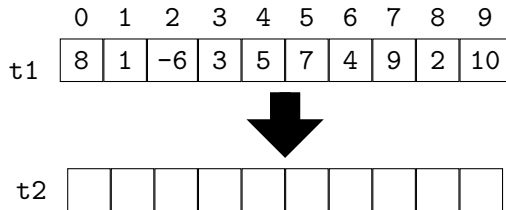


```
int t1[10];
int t2[10];
```

```
t1 = t2;
```

Źle !

KOPIOWANIE TABLIC



```
int i;
int t1[10];
int t2[10];

i = 0;
while( i < 10 )
{
    t2[i] = t1[i];
    i++;
}
```

TABLICE JAKO PARAMETRY FUNKCJI

Tablica jednowymiarowa w argumentach funkcji podawana bez rozmiaru (informacja o rozmiarze jest ignorowana i nie jest dostępna wewnątrz funkcji).

```
1 float max(float t[], int n)
2 {
3     float m = t[0];
4     while( n > 1 )
5     {
6         n = n - 1;
7         if( m < t[n] ) m = t[n];
8     }
9     return m;
10 }
```

 max.c

PRZYKŁADOWE DEKLARACJE FUNKCJI

```
void wczytaj(float tab[], int n);  
float max(float t[], int n);
```

Dobrze

```
float max(float t[10]);  
float max(float t, int n);  
float max(float t[]);
```

Źle

TABLICE JAKO PARAMETRY FUNKCJI

Zawartość tablicy przekazana do funkcji nie jest kopiowana, zaś funkcja może dowolnie modyfikować elementy przekazanej tablicy.

```
1  #include<stdio.h>
2
3  void funkcja(const int tab[], int x)
4  {
5      tab[0]++;
6      x++;
7  }
8
9  int main()
10 {
11     int tab[10], x = 5;
12
13     tab[0] = x;
14     funkcja(tab, x);
15     printf("tab[0]=%d, x=%d\n", tab[0], x);
16 }
```

TABLICE JAKO PARAMETRY FUNKCJI

Modyfikator `const` pozwala zaznaczyć, że funkcja nie zmienia wartości elementów tablicy.

PRZYKŁAD:

```
float max(const float t[], int n);
```

```
void funkcja(const int tab[], int x)
{
    tab[0]++;
    x++;
}
```

Błąd! Nie można modyfikować.

TABLICE JAKO PARAMETRY FUNKCJI

Parametrem aktualnym funkcji (w momencie wywołania) jest nazwa tablicy

```
float max(const float t[], int n);  
void wczytaj(float t[], int n);
```

```
int main()  
{  
    float t[10], x;  
    int n=10;  
  
    wczytaj(t,n);  
    x = max(t,n);  
  
    x = max(t[10], 10);  
    x = max(t[], 10);  
    x = max(float t[], int n);  
}
```

Dobrze

Źle

Problem: w zbiorze zawierającym n elementów odnajdź element x .

Algorytm 7 Przeszukiwanie liniowe

Dane wejściowe: ciąg $\{t_0, t_1, \dots, t_{n-1}\}$ zawierający n elementów,
szukany element x , pozycja początku przeszukiwania i

Wynik: pozycja pierwszego znalezionej elementu x w ciągu lub
wartość -1 jeśli nie znaleziono

- 1: **dopóki** $i < n$ **wykonuj**
 - 2: **jeżeli** $t_i = x$ **wykonaj**
 - 3: **zwróć** i
 - 4: $i \leftarrow i + 1$
 - 5: **zwróć** -1
-

PRZYKŁAD W C: PRZESZUKIWANIE LINIOWE

```
1 int szukaj(const int t[], int n, int x, int i)
2 {
3     while( i < n )
4     {
5         if( t[i]== x ) return i;
6         i = i + 1;
7     }
8     return -1;
9 }
```

 przeszukiwanie.c

- Ile porównań należy wykonać w najgorszym przypadku?
- Czy istnieje szybszy sposób przeszukania ciągu elementów?

Algorytm 8 Przeszukiwanie liniowe z wartownikiem

Dane wejściowe: ciąg $\{t_0, t_1, \dots, t_{n-1}\}$ zawierający n elementów, szukany element x , pozycja początku przeszukiwania i

Wynik: pozycja pierwszego znalezionej elementu x w ciągu lub wartość -1 jeśli nie znaleziono

- 1: $t_n \leftarrow x$
 - 2: **dopóki** $t_i \neq x$ **wykonuj**
 - 3: $i \leftarrow i + 1$
 - 4: **jeżeli** $i = n$ **wykonaj**
 - 5: **zwróć** -1
 - 6: **w przeciwnym wypadku**
 - 7: **zwróć** i
-

PRZYKŁAD W C: PRZESZUKIWANIE LINIOWE Z WARTOWNIKIEM

```
1 int szukaj2(const int t[], int n, int x, int i)
2 {
3     t[n] = x;
4     while( t[i] != x ) i = i + 1;
5     if( i != n ) return i;
6     return -1;
7 }
```

 przeszukiwanie2.c

- przechowuje zmienne dowolnego typu
- pole struktury - pojedyncza składowa
- elementy (pola) są identyfikowane nazwami
- operator dostępu bezpośredniego .
- struktury ułatwiają organizację danych → załączek obiektowości
- jak każda zmienna struktura może być argumentem funkcji oraz wartością zwracaną z funkcji

"Bond"
"James"
007
3.2

DEKLARACJA STRUKTURY

```
struct nazwa  
{  
    typ pole1;  
    typ pole2;  
    ...  
};
```

```
struct student  
{  
    char nazwisko[30];  
    char imie[30];  
    int indeks;  
    float srednia;  
};
```

UTWORZENIE ZMIENNEJ (DEFINICJA)

```
struct nazwa identyfikator;
```

```
struct student s;
```

DOSTĘP DO PÓL STRUKTURY

```
identyfikator.pole
```

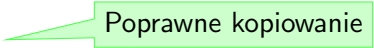
```
s.srednia = 5.0;
```

```
#include <stdio.h>

struct zespolona
{
    float re;
    float im;
};

int main()
{
    struct zespolona z1 ,z2;
    z1.re = 2.5;
    z1.im = -2.2;

    z2 = z1;
}
```



Poprawne kopiowanie

DEKLARACJE FUNKCJI

```
struct zespolona iloczyn(struct zespolona z1, struct
    zespolona z2);
void wyswietl(struct student s);

int main()
{
    struct zespolona z1, z2, z3;
    struct student s;

    z3 = iloczyn(z1, z2);
    wyswietl(s);

    return 0;
}
```

Inicjalizacja elementów tablicy

```
int tab[10] = { 5, 3, 7};
```

tab	5	3	7	?	?	?	?	?	?
-----	---	---	---	---	---	---	---	---	---

Gdy pominiemy rozmiar tablicy to jest on wyznaczany automatycznie

```
int tab[] = { 5, 3, 7};
```

tab	5	3	7
-----	---	---	---

Tablica znaków

```
int tab[] = { 'A', 'B', 'C' };
```

tab

A	B	C
---	---	---

Napis (łańcuch znakowy)

```
int tab[] = "ABC";
```

tab

A	B	C	\0
---	---	---	----

Inicjalizacja wartości struktur

```
struct student
{
    int numer;
    char nazwisko[5];
};
```

```
struct student janek = { 13, "ABC"};
struct student franek = { 5, { 'A', 'B', 'C' } };
```

janek

13				
A	B	C	\0	?

franek

5				
A	B	C	?	?

- tablice wielowymiarowe, macierze, tablice tablic
`t[10][2][3]`
- struktury zawierające struktury
`s.data.dzien = 1`
- tablice struktur
`s[1].wiek = 31`
- struktury zawierające tablice
`punkt.wsp[1] = 1.2`

Problem: wyznaczyć położenie środka masy dla n punktów materialnych.

PUNKT MATERIALNY

$$p_i = \{m_i, \vec{r}_i\}, \quad \vec{r}_i = [x_i, y_i, z_i]$$

ŚRODEK MASY DWÓCH PUNKTÓW

$$\vec{r}_{12} = \frac{m_1 \vec{r}_1 + m_2 \vec{r}_2}{m_1 + m_2}, \quad m_{12} = m_1 + m_2$$

ŚRODEK MASY n PUNKTÓW

$$\vec{r}_0 = \frac{\sum_{i=1}^n m_i \vec{r}_i}{\sum_{i=1}^n m_i}, \quad m_0 = \sum_{i=1}^n m_i$$

TABLICA 4 ELEMENTOWA

konwencja: 0,1,2 - współrzędne kartezjańskie, 3 - masa

```
float punkt[4];
```

STRUKTURA

```
struct punkt
{
    float m;
    float x;
    float y;
    float z;
};
```

```
struct punkt
{
    float m;
    float wsp[3];
};
```

```
float* srodek(const float p1[], const float p2[])
{
    float sm[4];
    int i=0;

    sm[3] = p1[3] + p2[3];

    for(i=0; i<3; i++)
        sm[i] = (p1[3] * p1[i] + p2[3] * p2[i])/sm[3];

    return sm;
}
```

```
float* srodek(const float p1[], const float p2[])
{
    float sm[4];
    int i=0;

    sm[3] = p1[3] + p2[3];

    for(i=0; i<3; i++)
        sm[i] = (p1[3] * p1[i] + p2[3] * p2[i])/sm[3];

    return sm;
}
```

zmienna lokalna

brak kopiowania tablic

Źle!

```
1 void srodek(const float p1[], const float p2[], float sm[])
2 {
3     int i=0;
4
5     sm[3] = p1[3] + p2[3];
6
7     for(i=0; i<3; i++)
8         sm[i] = (p1[3] * p1[i] + p2[3] * p2[i]) / sm[3];
9 }
```

 sm1.c


```
1 struct punkt
2 {
3     float x, y, z;
4     float m;
5 };
6
7 struct punkt srodek(struct punkt p1, struct punkt p2)
8 {
9     struct punkt sm;
10    sm.m = p1.m + p2.m;
11    sm.x = ( p1.m * p1.x + p2.m * p2.x ) / sm.m;
12    sm.y = ( p1.m * p1.y + p2.m * p2.y ) / sm.m;
13    sm.z = ( p1.m * p1.z + p2.m * p2.z ) / sm.m;
14    return sm;
15 }
```

ŚRODEK MASY 2 PUNKTÓW

STRUKTURY C.D.

```
1  struct punkt
2  {
3      float wsp[3];
4      float m;
5  };
6
7  struct punkt srodek(struct punkt p1, struct punkt p2)
8  {
9      struct punkt sm;
10     int i=0;
11     sm.m=p1.m+p2.m;
12     while(i<3)
13     {
14         sm.wsp[i]=(p1.m*p1.wsp[i]+p2.m*p2.wsp[i])/sm.m;
15         i = i + 1;
16     }
17     return sm;
18 }
```

Algorytm 9 Środek masy n punktów materialnych

Dane wejściowe: zestaw punktów $\{p_1, p_2, \dots, p_n\}$ określonych przez masę i współrzędne kartezjańskie $p_i = \{x_i, y_i, z_i, m_i\}$

Wynik: $p_0 = \{x_0, y_0, z_0, m_0\}$ położenie środka masy i masa całkowita układu

- 1: $p_0 \leftarrow \{0, 0, 0, 0\}$
 - 2: $i \leftarrow 0$
 - 3: **dopóki** $i < n$ **wykonuj**
 - 4: **wczytaj** p_i
 - 5: $p_0 \leftarrow \text{srodek}(p_i, p_0)$
 - 6: $i \leftarrow i + 1$
 - 7: **wypisz** p_0
-

ŚRODEK MASY n PUNKTÓW

```
1  int main()
2  {
3      struct punkt p, p1;
4      char dalej='t';
5
6      p.m=0.0;  p.x = 0.0;  p.y = 0.0;  p.z = 0.0;
7
8      do
9      {
10         p1 = wczytaj();
11         p = srodek(p1,p);
12
13         printf("Czy dodac kolejny punkt [t/n] ? ");
14         scanf(" %c",&dalej);
15     }while(dalej != 'n' );
16
17     printf("Srodek masy:\n");
18     wypisz(p);
19
20     return 0;
21 }
```

TABLICA TABLIC

```
float chmura[1000][4];
```

```
m42 = chmura[42][3]
```

TABLICA STRUKTUR

```
struct punkt chmura[1000];
```

```
m42 = chmura[42].m
```

STRUKTURA Z TABLICAMI

```
struct chmura {  
    int n;  
    float x[1000];  
    float y[1000];  
    float z[1000];  
    float m[1000];  
};
```

```
struct chmura c;  
m42 = c.m[42]
```

```
struct chmura {  
    int n;  
    struct punkt p[1000];  
};
```

```
struct chmura c;  
m42 = c.p[42].m
```

```
1  struct punkt srodek(struct punkt p[], int n)
2  {
3      struct punkt sm;
4      int i=0;
5
6      sm.m = 0.0; sm.x = 0.0; sm.y = 0.0; sm.z = 0.0;
7
8      while(i<n)
9      {
10         sm.m = sm.m + p[i].m;
11         sm.x = sm.x + p[i].x * p[i].m;
12         sm.y = sm.y + p[i].y * p[i].m;
13         sm.z = sm.z + p[i].z * p[i].m;
14         i = i + 1;
15     }
16     sm.x = sm.x/sm.m;
17     sm.y = sm.y/sm.m;
18     sm.z = sm.z/sm.m;
19     return sm;
20 }
```

```
1  int main()  
2  {  
3      struct punkt chmura[MAX];  
4      int i=0;  
5  
6      do  
7      {  
8          chmura[i] = wczytaj();  
9          i = i + 1;  
10     }while(czy_dalej() == 1 && i < MAX );  
11  
12     printf("Srodek masy:\n");  
13     wypisz(srodek(chmura,i));  
14  
15     return 0;  
16 }
```

ŚRODEK MASY n PUNKTÓW

STRUKTURA Z TABLICĄ PUNKTÓW

```
1  int main()
2  {
3      struct chmura c;
4      int i=0;
5
6      c.n = 0;
7
8      do
9      {
10         c = dodaj(c, wczytaj());
11         i = i + 1;
12     }while( czy_dalej() && i < MAX );
13
14     printf("Aktualny zbior punktow:\n");
15     wypisz_chmure(c);
16     printf("Srodek masy:\n");
17     wypisz_punkt(srodek(c));
18
19     return 0;
20 }
```


- **Tablica** - zbiór elementów tego samego typu indeksowanych od 0
- **Struktura** - zbiór elementów różnych typów o nazwanych polach
- Tablice trzeba kopiować „ręcznie” element po elemencie
- Właściwie dobrana reprezentacja danych może istotnie ułatwić realizację rozwiązania. Reprezentacja danych ma wpływ na czytelność kodu i efektywność programu
- Dobra zasada: twórz funkcje do manipulowania złożonymi typami danych

-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>

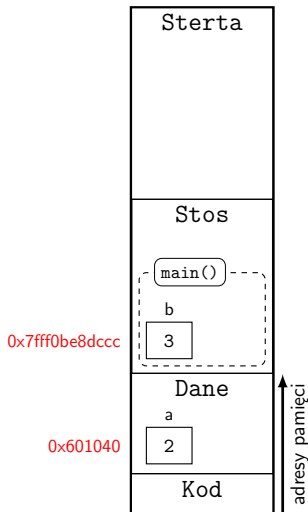
Wskaźniki

```

1  #include<stdio.h>
2
3  int a = 2;
4
5  int main()
6  {
7      int b = 3;
8
9      printf("adres zmiennej a %p\n", &a);
10     printf("adres zmiennej b %p\n", &b);
11
12     return 0;
13 }

```

adres zmiennej a 0x601040
 adres zmiennej b 0x7fff0be8dccc



WSKAŹNIK (*pointer*)

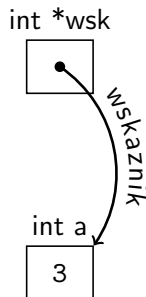
adres zmiennej w pamięci (np. `&a`)

```
int a = 5;  
printf("%p\n", &a);
```

ZMIENNA WSKAŹNIKOWA

zmienna przechowująca adres

```
int *wsk;  
wsk = &a;
```



TYP WSKAŹNIKOWY

typ zmiennej znajdujący się pod wskazanym adresem (np. `int`)

DEKLARACJA

```
typ *identyfikator;
```

PRZYKŁAD

```
int *wa;           /* wskaznik na zmienna typu int */  
float *wx;         /* wskaznik na zmienna typu float */  
char *wz;          /* wskaznik na zmienna typu char */  
void *w;           /* wskaznik na zmienna dowolnego typu */  
int *t[10];        /* tablica zmiennych wskaznikowych */  
int **ww;          /* wskaznik na zmienna wskaznikowa */
```

Operator referencji & zwraca adres zmiennej

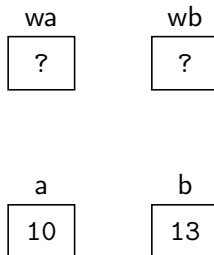
PRZYKŁAD

```
1  int a=10;
2  int b=13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```

Operator referencji & zwraca adres zmiennej

PRZYKŁAD

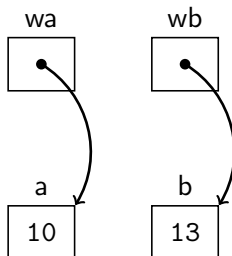
```
1  int a=10;
2  int b=13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```



Operator referencji & zwraca adres zmiennej

PRZYKŁAD

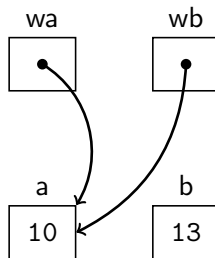
```
1  int a=10;  
2  int b=13;  
3  int *wa;  
4  int *wb;  
5  
6  wa = &a;  
7  wb = &b;  
8  
9  wb = wa;
```



Operator referencji & zwraca adres zmiennej

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa;  
4  int *wb;  
5  
6  wa = &a;  
7  wb = &b;  
8  
9  wb = wa;
```



Operator dereferencji * daje dostęp do wskazanego adresu

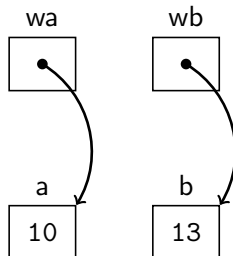
PRZYKŁAD

```
1  int a=10;
2  int b=13;
3  int *wa = &a;
4  int *wb = &b;
5
6  *wb = 5;
7
8  *wa = *wb;
9
10 wa = wb;
11 *wb = *wb + 1;
```

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

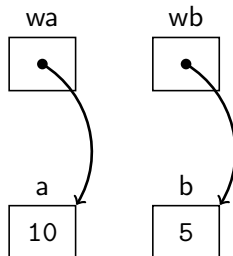
```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

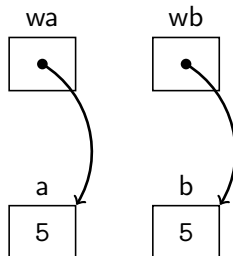
```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;   
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

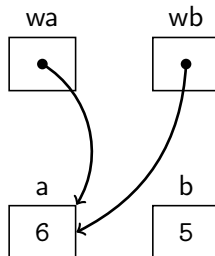
```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```



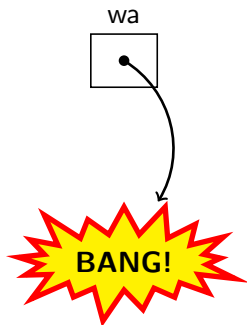
Uważaj na niezainicjowane zmienne wskaźnikowe.

PRZYKŁAD

```
int *wa;  
*wa = 5;
```

NULL TO ADRES 0

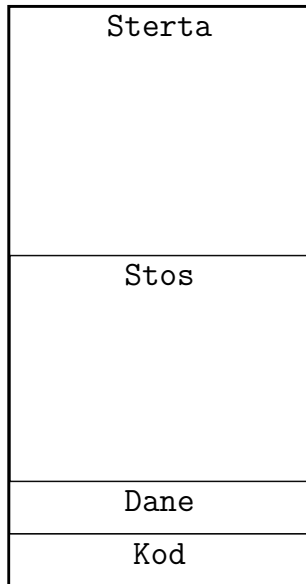
```
int *wa;  
wa = 0;  
wa = NULL;      /* stdlib.h */
```



- Nigdy nie używaj operatora `*` na niezainicjowanej zmiennej.
- Wskazanie puste, adres 0, `NULL` - informacja, że wskaźnik nic nie pokazuje

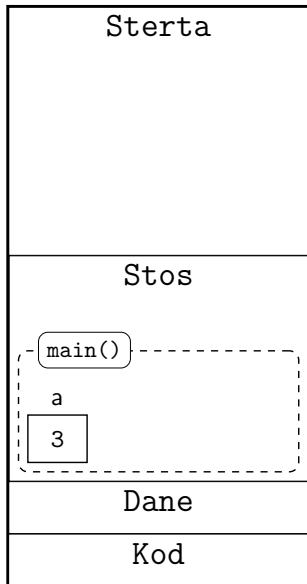
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n",a);
15
16     return 0;
17 }
```



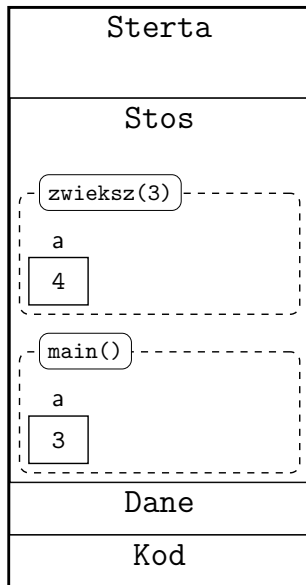
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



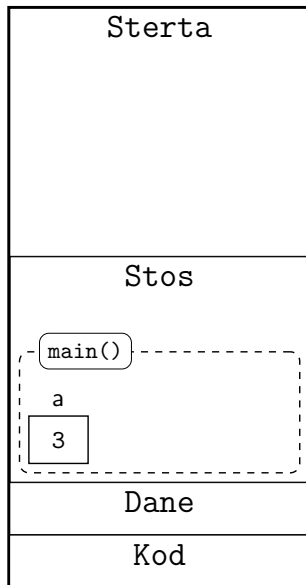
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



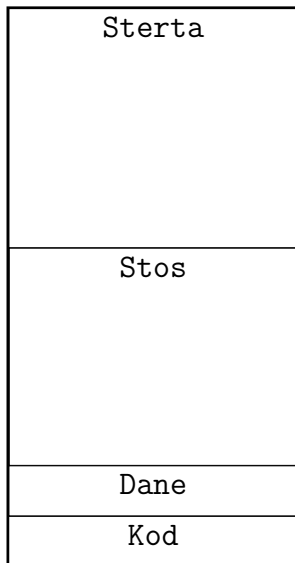
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n",a);
15
16     return 0;
17 }
```

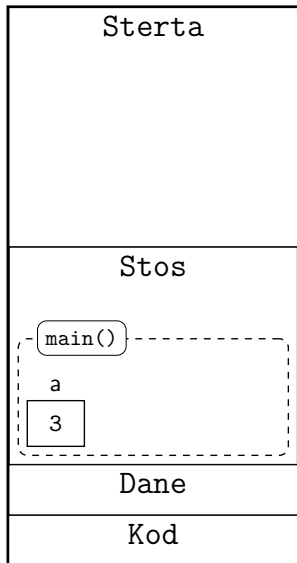


WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n",a);
15
16     return 0;
17 }
```



0x7fff0be8

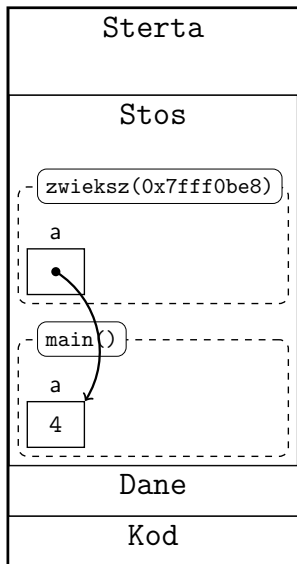


WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



0x7fff0be8

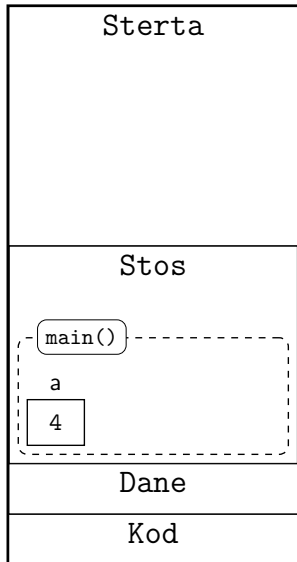


WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



0x7fff0be8



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

- poprzez wskaźnik (adres zmiennej) funkcja może zwrócić dodatkową wartość
- `scanf("%d", &x) ←` funkcja modyfikuje zmienną `x`, stąd argumentem musi być adres



Problem: znajdź wartość minimalną i maksymalną w zbiorze liczb

Algorytm 10 Wyznaczanie maksimum i minimum

Dane wejściowe: ciąg n liczb $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość x_{min} i x_{max} , odpowiednio najmniejszy i największy element podanego ciągu

- 1: $x_{min} \leftarrow x_1$
 - 2: $x_{max} \leftarrow x_1$
 - 3: **dla każdego** $x \in \{x_2, \dots, x_n\}$ **wykonuj**
 - 4: **jeżeli** $x < x_{min}$ **wykonaj**
 - 5: $x_{min} \leftarrow x$
 - 6: **jeżeli** $x > x_{max}$ **wykonaj**
 - 7: $x_{max} \leftarrow x$
 - 8: **zwróć** x_{min}, x_{max}
-

```
1 void minmax(float t[], int n, float *min, float *max)
2 {
3     int i = 1;
4     *min=t[0];
5     *max=t[0];
6
7     while( i < n)
8     {
9         if( *min > t[i] ) *min = t[i];
10        if( *max < t[i] ) *max = t[i];
11        i = i + 1;
12    }
13 }
```

 minmax1.c

ELEMENT MINIMALNY I MAKSYMALNY

PRZYKŁAD W C

```
1  int main()  
2  {  
3      int n;  
4      float t[MAX], max, min;  
5  
6      n = wczytaj(t, MAX);  
7      minmax(t, n, &min, &max);  
8      printf("min=%f\nmax=%f\n", min, max);  
9  
10     return 0;  
11 }
```

 minmax1.c

- Ilość porównań: $2(n - 1)$
- Czy istnieje szybszy sposób?
- Dziel i zwyciężaj !

Algorytm 11 Wyznaczanie maksimum i minimum

Dane wejściowe: ciąg n liczb $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość x_{min} i x_{max} , odpowiednio najmniejszy i największy element podanego ciągu

- 1: $\mathcal{A} \leftarrow \emptyset, \quad \mathcal{B} \leftarrow \emptyset$
 - 2: **dla** $i = 1, 2, \dots, \lfloor \frac{n}{2} \rfloor$ **wykonuj**
 - 3: **jeżeli** $x_{2i-1} < x_{2i}$ **wykonaj**
 - 4: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i-1}\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i}\}$
 - 5: **w przeciwnym wypadku**
 - 6: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i}\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i-1}\}$
 - 7: **jeżeli** n jest nieparzyste **wykonaj**
 - 8: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_n\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_n\}$
 - 9: $x_{min} \leftarrow \min(\mathcal{A})$
 - 10: $x_{max} \leftarrow \max(\mathcal{B})$
 - 11: **zwróć** x_{min}, x_{max}
-

```

1 void minmax(float t[], int n, float *min, float *max)
2 {
3     int i;
4     float tmin, tmax;
5
6     if( n==1 ){
7         *min=t[0];
8         *max=t[0];
9         return;
10    }
11
12    if( t[0]<t[1] ){
13        *min=t[0];
14        *max=t[1];
15    }
16    else{
17        *min=t[1];
18        *max=t[0];
19    }
20
21    for( i=2; i < n-1; i=i+2 ){
22        if( t[i]<t[i+1] ) {

```

```

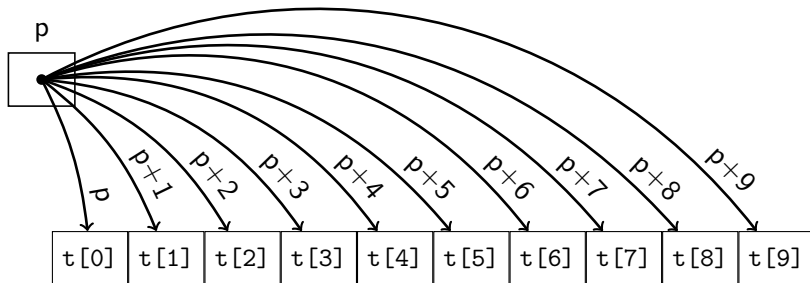
23         tmin=t[i];
24         tmax=t[i+1];
25     }
26     else {
27         tmin=t[i+1];
28         tmax=t[i];
29     }
30     if( *max<tmax ) *max=tmax;
31     if( *min>tmin ) *min=tmin;
32 }
33 if( i == n-1 ){
34     if( *max<t[i] ) *max=t[i];
35     if( *min>t[i] ) *min=t[i];
36 }
37 }

```

 minmax2.c

WSKAŹNIKI DO ELEMENTÓW TABLIC

```
p = &t[0];
```



```
1  int    a;  
2  char   b;  
3  int    *pa = &a;  
4  char   *pb = &b;  
5  
6  printf("pa    = %p %lu\n", pa    , pa    );  
7  printf("pa+1 = %p %lu\n", pa+1  , pa+1);  
8  printf("pb    = %p %lu\n", pb    , pb    );  
9  printf("pb+1 = %p %lu\n", pb+1  , pb+1);
```

 pointer.c

Przykładowy wynik:

```
pa    = 0x7fff44cb239c 140734347551644  
pa+1  = 0x7fff44cb23a0 140734347551648  
pb    = 0x7fff44cb239b 140734347551643  
pb+1  = 0x7fff44cb239c 140734347551644
```

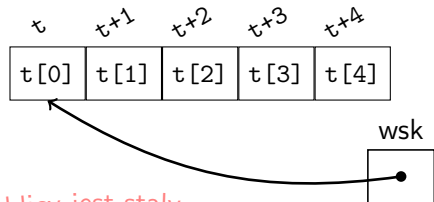
Tablica jest wskaźnikiem !

i -ty element $t[i]$ $\Leftrightarrow$ $*(t+i)$
 adres i -tego elementu $\&(t[i])$ $\Leftrightarrow$ $t+i$

PRZYKŁAD

```

1  int t[5];
2  int *wsk;
3
4  wsk=t;
5  *t = 5;
6  *(t+2) = 6;
7  wsk = wsk + 1;
8  t = t + 1;
    
```



Adres tablicy jest stały

Źle !

PONOWNIE ŚRODEK MASY 2 PUNKTÓW

```
float* srodek(float p1[], float p2[])
{
    float sm[4];
    int i=0;
    sm[3]=p1[3]+p2[3];
    while(i<3)
    {
        sm[i]=(p1[3]*p1[i]+p2[3]*p2[i])/sm[3];
        i = i + 1;
    }
    return sm;
}
```

zmienna lokalna

zwracany adres zmiennej lokalnej

Źle!

PONOWNIE ŚRODEK MASY 2 PUNKTÓW

- Deklaracja `const` zmiennej wskaźnikowej - kompilator wykryje próbę modyfikacji.
- Zmienna `sm` zawiera adres zmiennej, która zostanie zmodyfikowana przez funkcję `srodek()`.

```
void srodek(const float p1[], const float p2[], float sm[])
{
    int i=0;
    sm[3]=p1[3]+p2[3];
    while(i<3)
    {
        sm[i]=(p1[3]*p1[i]+p2[3]*p2[i])/sm[3];
        i = i + 1;
    }
}
```

Wskaźnik



WSKAŹNIK JAKO WARTOŚĆ ZWRACANA Z FUNKCJI

```
1 float *wczytaj(float *t, int n)
2 {
3     float *p=t;
4     printf("Wprowadz %d liczb\n", n);
5     while( n >0 )
6     {
7         scanf("%f",t);
8         t = t + 1;
9         n = n - 1;
10    }
11    return p;
12 }
13
14 int main()
15 {
16     float t[100];
17     float min,max;
18     minmax(wczytaj(t,10), &min, &max);
19 }
```

PRZYKŁADOWE DEKLARACJE FUNKCJI

```
/* Miejsca zerowe paraboli */
int pierwiastki(float a, float b, float c, float *x1, float *x2);

/* Wyszukiwanie binarne */
int szukaj(const int *t, int n, int x);

/* Srodek masy ukladu punktow */
struct punkt srodek(const struct ogromna_chmura_punktow *u);

/* Dekompozycja liczby zmiennopozycyjnej (math.h) */
float modf(float num, float *i);

/* Alokacja pamieci (stdlib.h) */
void* malloc(int size);

/* Kopiowanie tablic (stdlib.h) */
void* memcpy(void *dest, const void *src, int count);

/* Otwieranie pliku (stdio.h) */
FILE *fopen( const char *filename, const char *mode );
```

PRZYKŁAD: WYSZUKIWANIE DOMINANTY

Problem: znajdź wartość występującą najwięcej razy w zbiorze

8	1	-6	3	5	7	4	9	2	10	-4	88	6	3	1	3	332	2
---	---	----	---	---	---	---	---	---	----	----	----	---	---	---	---	-----	---

Algorytm 12 Wyznaczanie dominanty (mody) - algorytm naiwny

Dane wejściowe: ciąg n elementów $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość dominanty x_{moda} oraz ilość wystąpień l_{moda} . Jeżeli istnieje więcej niż jedna wartość dominującą to zwracana jest pierwsza znaleziona.

- 1: $l_{moda} \leftarrow 0$
 - 2: **dla każdego** $x \in \{x_1, \dots, x_n\}$ **wykonuj**
 - 3: $k \leftarrow 0$
 - 4: **dla każdego** $y \in \{x_1, \dots, x_n\}$ **wykonuj**
 - 5: **jeżeli** $x = y$ **wykonaj**
 - 6: $k \leftarrow k + 1$
 - 7: **jeżeli** $k > l_{moda}$ **wykonaj**
 - 8: $l_{moda} \leftarrow k$
 - 9: $x_{moda} \leftarrow x$
 - 10: **zwróć** x_{moda}, l_{moda}
-

```

1  int dominanta(const int *t, int n, int *c)
2  {
3      int i, j, k, x;
4
5      *c = 0;
6      i=0;
7      while( i<n )
8      {
9          k=0;
10         j=0;
11         while( j<n )
12         {
13             if ( t[i]==t[j] ) k = k + 1;
14             j = j + 1;
15         }
16         if( k > *c )
17         {
18             *c = k;
19             x = t[i];
20         }
21         i = i + 1;
22     }
23     return x;
24 }

```

ZŁOŻONOŚĆ ALGORYTMU

- ilość operacji rzędu n^2
- jeżeli pewien element został aktualnie zaznaczony jako dominujący to nie musimy powtarzać dla niego obliczeń
- zliczanie można rozpocząć od $i + 1$ miejsca, jeżeli wartość dominująca pojawiła się wcześniej to już została policzona
- jeśli aktualna wartość dominująca ma k wystąpień, to szukanie możemy przerwać na pozycji $n - k$ w zbiorze

```

1  int dominanta2(const int *t, int n, int *c)
2  {
3      int i, j, k, x;
4
5      *c=0;
6      x=t[0]-1;
7      i=0;
8      while( i<n-*c )
9      {
10         if( t[i]!=x )
11         {
12             k=1;
13             j=i+1;
14             while( j<n )
15             {
16                 if ( t[i]==t[j] ) k = k + 1;
17                 j = j + 1;
18             }
19             if( k > *c )
20             {
21                 *c = k;
22                 x = t[i];
23             }
24         }
25         i = i + 1;
26     }
27     return x;
28 }

```

ALOKACJA PAMIĘCI

```
void *malloc(int rozmiar);
```

Funkcja `malloc` zwraca adres przydzielonego bloku pamięci lub wartość 0 (NULL) w przypadku niepowodzenia.

ZWOLNIENIE PRZYDZIELONEJ PAMIĘCI

```
void free(void *wskaznik);
```

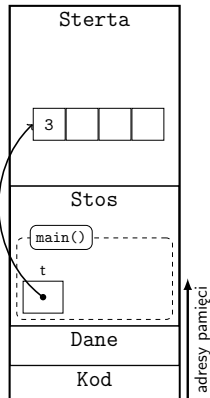
Argumentem funkcji `free` jest adres uzyskany wcześniej z funkcji `malloc`.

Funkcja `malloc` i `free` zadeklarowane są w pliku `stdlib.h`.

```

1  #include<stdlib.h>
2  #include<stdio.h>
3
4  int main()
5  {
6      float *t;
7
8      t = malloc( 4 * sizeof(int));
9      if( t == NULL ){
10         printf("Bład alokacji pamieci.\n");
11         exit(1);
12     }
13
14     *t = 3;
15
16     free(t);
17
18     return 0;
19 }

```



- Operator referencji (adresowania) &
&a to adres zmiennej a
- Operator dereferencji (wyłuskania) *
*b daje dostęp do wartości wskazywanej przez zmienną b
- Deklaracja zmiennej wskaźnikowej również zawiera znak *
int *wsk;
float* wsk2;
- Tablice to wskaźniki $t[i] \Leftrightarrow *(t+i)$
- Przekazanie wskaźnika do funkcji pozwala na modyfikację zmiennej wskazywanej
- Uważaj na co wskazujesz!

-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>

Reprezentacja symboli w komputerze.

Liczby całkowite i zmiennoprzecinkowe.

- **Bit** (*binary digit*) najmniejsza ilość informacji
 $\{0, 1\}$, wysokie/niskie napięcie
- **Kod binarny** - grupa bitów reprezentująca zbiór symboli
 - 1b $\rightarrow$ 2 symbole $\{0, 1\}$
 - 2b $\rightarrow$ 4 symbole $\{00, 01, 10, 11\}$
 - 8b $\rightarrow$ 256 = 2^8 symboli
 - n bitów $\rightarrow$ 2^n symboli
- Do zakodowania n symboli potrzeba co najmniej $\lceil \log_2 n \rceil$ bitów
- Ile bitów potrzeba do zakodowania alfabetu polskiego?

- **Bajt** (*byte*) $1\text{B} = 8\text{b}$ (zazwyczaj ...)
Wg. standardu C: najmniejsza ilość adresowalnej pamięci
Stała `CHAR_BIT` z pliku `limits.h`
Najmniejsza porcja danych, którą może „ugryźć” komputer.
Utożsamiany ze znakiem (typ `char`).
- Ile pamięci można zaadresować?
 $16\text{b} \rightarrow 2^{16}\text{B} = 64\text{KB}$
 $32\text{b} \rightarrow 2^{32}\text{B} = 4294967296\text{B} \sim 4\text{GB}$
 $64\text{b} \rightarrow 2^{64}\text{B} \sim 16\text{EB}$

- układ SI: $1k = 10^3 = 1000 \neq 1024 = 2^{10} = 1K$
- 1997 EIC, przedrostki dwójkowe - raczej nieużywane

IEC		podstawa						SI	
nazwa	symbol	2	16	różnica	10			nazwa	symbol
kibi	Ki	2^{10}	$16^{2.5}$	400_{16}	2,40%	1 024	$> 10^3$	kilo	k
mebi	Mi	2^{20}	16^5	$10\ 0000_{16}$	4,86%	1 048 576	$> 10^6$	mega	M
gibi	Gi	2^{30}	$16^{7.5}$	$4000\ 0000_{16}$	7,37%	1 073 741 824	$> 10^9$	giga	G
tebi	Ti	2^{40}	16^{10}	$100\ 0000\ 0000_{16}$	9,95%	1 099 511 627 776	$> 10^{12}$	tera	T
pebi	Pi	2^{50}	$16^{12.5}$	$4\ 0000\ 0000\ 0000_{16}$	12,59%	1 125 899 906 842 624	$> 10^{15}$	peta	P
eksbi	Ei	2^{60}	16^{15}	$1000\ 0000\ 0000\ 0000_{16}$	15,29%	1 152 921 504 606 846 976	$> 10^{18}$	eksa	E
zebi	Zi	2^{70}	$16^{17.5}$	$40\ 0000\ 0000\ 0000\ 0000_{16}$	18,06%	1 180 591 620 717 411 303 424	$> 10^{21}$	zetta	Z
jobi	Yi	2^{80}	16^{20}	$1\ 0000\ 0000\ 0000\ 0000\ 0000_{16}$	20,89%	1 208 925 819 614 629 174 706 176	$> 10^{24}$	jotta	Y

<http://www.wikipedia.org>

Operator sizeof oblicza rozmiar w bajtach typu lub zmiennej.

SKŁADNIA

`sizeof (typ)`

`sizeof zmienna`

PRZYKŁAD

```
int x = 10;
printf("%d\n", sizeof (int));
printf("%d\n", sizeof x);
```

TYPOWE UŻYCIE

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

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  struct s {
5      int a;
6      char b[100];
7  };
8
9  int main()
10 {
11     int tab[5];
12
13     printf("CHAR_BIT           = %d\n", CHAR_BIT);
14     printf("sizeof (char)      = %d\n", sizeof (char));
15     printf("sizeof (int)               = %d\n", sizeof (int));
16     printf("sizeof (long)              = %d\n", sizeof (long));
17     printf("sizeof (float)             = %d\n", sizeof (float));
18     printf("sizeof (double)            = %d\n", sizeof (double));
19     printf("sizeof (int *)              = %d\n", sizeof (int*));
20     printf("sizeof (char *)             = %d\n", sizeof (char*));
21     printf("sizeof (struct s)          = %d\n", sizeof (struct s));
22     printf("sizeof tab                  = %d\n", sizeof tab);
23
24     return 0;
25 }

```

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  struct s {
5      int a;
6      char b[100];
7  };
8
9  int main()
10 {
11     int tab[5];
12
13     printf("CHAR_BIT          = %d\n", CHAR_BIT);
14     printf("sizeof (char)      = %d\n", sizeof (char));
15     printf("sizeof (int)               = %d\n", sizeof (int));
16     printf("sizeof (long)              = %d\n", sizeof (long));
17     printf("sizeof (float)             = %d\n", sizeof (float));
18     printf("sizeof (double)            = %d\n", sizeof (double));
19     printf("sizeof (int *)              = %d\n", sizeof (int*));
20     printf("sizeof (char *)             = %d\n", sizeof (char*));
21     printf("sizeof (struct s)           = %d\n", sizeof (struct s));
22     printf("sizeof tab                   = %d\n", sizeof tab);
23
24     return 0;
25 }

```

Przykładowy wynik. Wyniki mogą być inne na różnych architekturach.

CHAR_BIT	= 8
sizeof (char)	= 1
sizeof (int)	= 4
sizeof (long)	= 8
sizeof (float)	= 4
sizeof (double)	= 8
sizeof (int *)	= 8
sizeof (char *)	= 8
sizeof (struct s)	= 104
sizeof tab	= 20

$$x_{n-1}x_n \dots x_0 = \sum_{i=0}^{n-1} x_i a^i$$

a - baza (podstawa) systemu

DZIESIĘTNY

$$46532_{(10)} = 4 \cdot 10^4 + 6 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0$$

DWÓJKOWY (BINARNY)

$$10011_{(2)} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19_{(10)}$$

ÓSEMKOWY (OKTALNY)

$$351_{(8)} = 3 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 233_{(10)}$$

SZESNASTKOWY (HEKSADECYMALNY)

$$3FC_{(16)} = 3 \cdot 16^2 + 15 \cdot 16^1 + 12 \cdot 16^0 = 1020_{(10)}$$

A=10, B=11, C=12, D=13, E=14, F=15

$$1111110101111110_{(2)} = \text{FD7E}_{(16)} = 176576_{(8)} = 64894_{(10)}$$

BINARNY NA SZESNASTKOWY

1 cyfrze odpowiadają 4 bity

1111	1101	0111	1110
F	D	7	E

BINARNY NA ÓSEMKOWY

1 cyfrze odpowiadają 3 bity

1	111	110	101	111	110
1	7	6	5	7	6

LICZBY ÓSEMKOWE I SZESNASTKOWE W C

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 10;          /* dec */
6      int b = 010;         /* oct */
7      int c = 0x10;        /* hex */
8
9      printf("dec: %d %d %d\n", a, b, c);
10     printf("oct: %o %o %o\n", a, b, c);
11     printf("hex: %x %x %x\n", a, b, c);
12
13     return 0;
14 }
```

PRZELICZANIE NA INNY SYSTEM POZYCYJNY

Problem: wyrazić liczbę $x_{(10)}$ w dowolnym systemie pozycyjnym o podstawie $p \in \{2, \dots, 10\}$.

$$\begin{array}{r|l} 139 \% 2 & 1 \\ 69 \% 2 & 1 \\ 34 \% 2 & 0 \\ 17 \% 2 & 1 \\ 8 \% 2 & 0 \\ 4 \% 2 & 0 \\ 2 \% 2 & 0 \\ 1 \% 2 & 1 \end{array}$$

$$\begin{array}{r|l} 139 \% 8 & 3 \\ 17 \% 8 & 1 \\ 2 \% 8 & 2 \end{array}$$

$$\begin{array}{r|ll} 139 \% 16 & 11 & \text{B} \\ 8 \% 16 & 8 & \end{array}$$

$$139_{(10)} = 10001011_{(2)} = 213_{(8)} = 8\text{B}_{(16)}$$

Algorytm 13 Zapis liczby dziesiętnej w innej podstawie

Dane wejściowe: liczba całkowita przeliczana $x \geq 0$ oraz p podstawa docelowego systemu

Wynik: ciąg $\{t_{n-1}, \dots, t_1, t_0\}$ reprezentujący zapis w nowym systemie

1: $i \leftarrow 0$

2: **dopóki** $x \neq 0$ **wykonuj**

3: $t_i \leftarrow x \bmod p$

4: $x \leftarrow \lfloor \frac{x}{p} \rfloor$

5: $i \leftarrow i + 1$

6: **zwróć** $\{t_{i-1}, \dots, t_1, t_0\}$

▷ w odwrotnej kolejności

```
1  int zmien_podstawe(int x, int *t, int p)
2  {
3      int i=0;
4
5      while( x != 0 )
6      {
7          t[i] = x % p;
8          x = x / p;
9          i = i + 1;
10     }
11     return i;
12 }
```

 dec2all.c

Operator / dla liczb całkowitych dzieli bez reszty!

$$b_{n-1} \dots b_1 b_0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

- liczby całkowite bez znaku
- zakres $[0, 2^n - 1]$
- operacje arytmetyczne przeprowadza się podobnie jak w systemie dziesiętnym
- Typy całkowite bez znaku w C:

<code>unsigned char</code>	
<code>unsigned short</code>	<code>unsigned short int</code>
<code>unsigned int</code>	
<code>unsigned long</code>	<code>unsigned long int</code>
<code>unsigned long long</code>	<code>unsigned long long int</code> <code>/* standard C99 */</code>

- John Napier, XVI w.

typ	rozmiar [B]	zakres
unsigned char	1	[0, 255]
unsigned short int	2	[0, 65535]
unsigned int	4 (lub 2)	[0, 4294967295]
unsigned long int	8 (lub 4)	[0, 18446744073709551615]
unsigned long long int	8	[0, 18446744073709551615]

- Rozmiary typów zależą od implementacji
- Zakresy typów całkowitych są określone w `limits.h`
`UINT_MAX`, `ULLONG_MAX`

SPECYFIKATOR FORMATU PRINTF/SCANF

dec	oct	hex	typ
u	o	x	unsigned int
lu	lo	lx	unsigned long int
llu	llo	llx	unsigned long long int

PRZYKŁAD

```
unsigned int x = 4294967295;
unsigned long y = 4294967296;
printf("%u\n", x);
printf("%d\n", x);
printf("%o\n", x);
printf("%x\n", x);
printf("%u\n", y);
printf("%lu\n", y);
```

```
4294967295
-1
3777777777
ffffff
0
4294967296
```



```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6      printf("sizeof\n");
7      printf("unsigned char          = %lu\n", sizeof (unsigned char));
8      printf("unsigned short         = %lu\n", sizeof (unsigned short));
9      printf("unsigned int           = %lu\n", sizeof (unsigned int));
10     printf("unsigned long int      = %lu\n", sizeof (unsigned long int));
11     printf("unsigned long long int = %lu\n", sizeof (unsigned long long int));
12     printf("Zakres:\n");
13     printf("UCHAR_MAX              = %hu\n", UCHAR_MAX);
14     printf("USHRT_MAX              = %hu\n", USHRT_MAX);
15     printf("UINT_MAX               = %u\n",  UINT_MAX);
16     printf("ULONG_MAX              = %lu\n",  ULONG_MAX);
17     printf("ULLONG_MAX             = %llu\n", ULLONG_MAX);
18
19     return 0;
20 }

```

 uint1.c

Wartości mogą się różnić zależnie od implementacji.

sizeof

unsigned char = 1

unsigned short = 2

unsigned int = 4

unsigned long int = 8

unsigned long long int = 8

Zakres:

UCHAR_MAX = 255

USHRT_MAX = 65535

UINT_MAX = 4294967295

ULONG_MAX = 18446744073709551615

ULLONG_MAX = 18446744073709551615

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6      printf("sizeof\n");
7      printf("unsigned char\n");
8      printf("unsigned short\n");
9      printf("unsigned int\n");
10     printf("unsigned long int\n");
11     printf("unsigned long long int\n");
12     printf("Zakres:\n");
13     printf("UCHAR_MAX\n");
14     printf("USHRT_MAX\n");
15     printf("UINT_MAX\n");
16     printf("ULONG_MAX\n");
17     printf("ULLONG_MAX\n");
18
19     return 0;
20 }
```

```
printf("UCHAR_MAX\n", UCHAR_MAX);
printf("USHRT_MAX\n", USHRT_MAX);
printf("UINT_MAX\n", UINT_MAX);
printf("ULONG_MAX\n", ULONG_MAX);
printf("ULLONG_MAX\n", ULLONG_MAX);
```

 uint1.c

U2, KOD UZUPEŁNIEŃ DO DWÓCH

$$b_{n-1} \dots b_1 b_0 = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

- liczby ze znakiem, najstarszy bit b_{n-1} odpowiada za znak
- zakres $[-2^{n-1}, 2^{n-1} - 1]$
- typy całkowite ze znakiem w C:

char	
short	short int
int	
long	long int
long long	long long int /* standard C99 */

- można także użyć słowa signed, np.: signed long int

bity	NKB	U2
00000000	0	0
00000001	1	1
⋮		
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
⋮		
11111110	254	-2
11111111	255	-1

$$\begin{array}{r}
 127 \\
 \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\
 + \quad \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \\
 \hline
 \end{array}
 =
 \begin{array}{r}
 -128 \\
 \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0}
 \end{array}$$

- **Nadmiar** (*overflow*) - przekroczenie zakresu wynikające ze skończonej liczby bitów reprezentujących liczbę .
- Zazwyczaj błąd nie jest sygnalizowany.
- Dodawanie dużych liczb dodatnich (lub odejmowanie liczb ujemnych). W U2 widoczna zmiana znaku.
- Próba zapisu liczby typu bardziej pojemnego do typu mniej pojemnego (np. float do char)
- Dodawanie liczby ujemnej i dodatniej nie powoduje nadmiaru.
- 4 czerwiec 1996, pierwszy lot rakiety Ariane 5 zakończony zniszczeniem w skutek nadmiaru, 370 mln \$.

typ	rozmiar [B]	zakres
char	1	[−128, 127]
short int	2	[−32768, 32767]
int	4 (lub 2)	[−2147483648, 2147483647]
long int	8 (lub 4)	[−9223372036854775808, 9223372036854775807]
long long int	8	

- Zakresy typów całkowitych są określone w `limits.h`
- Specyfikacja formatu `printf`: `d`, `ld`, `lld`

```
long int x = 4294967295;
printf("%d\n", x);           /* ZLE */
printf("%ld\n", x);          /* OK  */
```

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6      printf("sizeof:\n");
7      printf("char          = %lu\n", sizeof (char));
8      printf("short         = %lu\n", sizeof (short));
9      printf("int           = %lu\n", sizeof (int));
10     printf("long int      = %lu\n", sizeof (long int));
11     printf("long long int = %lu\n", sizeof (long long int));
12     printf("Zakresy:\n");
13     printf("CHAR_MIN      = %hhd\n", CHAR_MIN);
14     printf("CHAR_MAX      = %hhd\n", CHAR_MAX);
15     printf("SHRT_MIN      = %hd\n", SHRT_MIN);
16     printf("SHRT_MAX      = %hd\n", SHRT_MAX);
17     printf("INT_MIN       = %d\n", INT_MIN);
18     printf("INT_MAX       = %d\n", INT_MAX);
19     printf("LONG_MIN      = %ld\n", LONG_MIN);
20     printf("LONG_MAX      = %ld\n", LONG_MAX);
21     printf("LLONG_MIN     = %lld\n", LLONG_MIN);
22     printf("LLONG_MAX     = %lld\n", LLONG_MAX);
23
24     return 0;
25 }

```

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6      printf("sizeof:\n");
7      printf("char\n");
8      printf("short\n");
9      printf("int\n");
10     printf("long int\n");
11     printf("long long int\n");
12     printf("Zakresy:\n");
13     printf("CHAR_MIN\n");
14     printf("CHAR_MAX\n");
15     printf("SHRT_MIN\n");
16     printf("SHRT_MAX\n");
17     printf("INT_MIN\n");
18     printf("INT_MAX\n");
19     printf("LONG_MIN\n");
20     printf("LONG_MAX\n");
21     printf("LLONG_MIN\n");
22     printf("LLONG_MAX\n");
23
24     return 0;
25 }

```

Wartości mogą się różnić zależnie od implementacji.

sizeof:

char	= 1
short	= 2
int	= 4
long int	= 8
long long int	= 8

Zakresy:

CHAR_MIN	= -128
CHAR_MAX	= 127
SHRT_MIN	= -32768
SHRT_MAX	= 32767
INT_MIN	= -2147483648
INT_MAX	= 2147483647
LONG_MIN	= -9223372036854775808
LONG_MAX	= 9223372036854775807
LLONG_MIN	= -9223372036854775808
LLONG_MAX	= 9223372036854775807

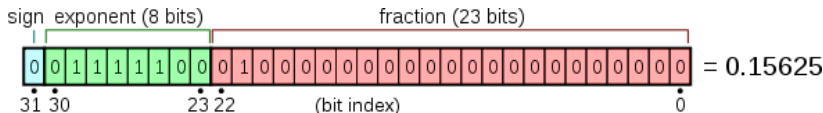
```

= %lld\n", LLONG_MIN),
= %lld\n", LLONG_MAX);

```


pojedyncza precyzja (*single precision*), float

$$x = (-1)^s \cdot 1.m \cdot 2^{(c-127)}$$



$$s = 0$$

$$1.m = \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) = 1 + 2^{-2} = 1.25$$

$$2^{(c-127)} = 2^{(124-127)} = 2^{-3} = 0.125$$

Podobne rozwiązanie stosował Konrad Zuse, 1936 r.

http://en.wikipedia.org/wiki/Single-precision_floating-point_format

typ	rozmiar	zakres	cyfry znaczące
float	32 b	$\pm 3.4 \cdot 10^{38}$	6
double	64 b	$\pm 1.8 \cdot 10^{308}$	15
long double	80 b	$\pm 1.2 \cdot 10^{4932}$	18

- Zakresy typów zmiennopozycyjnych są zdefiniowane w pliku nagłówkowym `float.h`
np.: `FLT_MAX`, `DBL_MAX`
- Wartości `long double` zależą od implementacji (w MS VC++ tożsame z `double`)

FORMATOWENIE LICZB ZMIENNOPOZYCYJNYCH

Formatowanie liczb zmiennopozycyjnych za pomocą printf

specyfikator	znaczenie	przykład
f	dziesiętnie	123.45678
e, E	notacja naukowa	1.2345678e+2
g	krótszy zapis %f lub %e	123.4657
Lf, Le, Lg	dla typu long double	1.2345e+400
.2f	dokładność 2 miejsc po przecinku	123.46

PRZYKŁAD

```
float x = 10.14;  
double y = 5e-3;
```

```
printf("%f %e %g\n", x, x, x);  
printf("%f %e %g\n", y, y, y);
```

10.140000	1.014000e+01	10.14
0.005000	5.000000e-03	0.005

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x = 0.1;
6
7      if(x == 0.1 ) printf ("OK, %f jest rowne 0.1\n", x);
8      else printf("Nie OK, %f nie jest rowne 0.1\n", x);
9
10     return 0;
11 }
```

 prec.c

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x = 0.1;
6
7      if(x == 0.1 ) printf ("OK, %f jest rowne 0.1\n", x);
8      else printf("Nie OK, %f nie jest rowne 0.1\n", x);
9
10     return 0;
11 }
```

 prec.c

- Niektóre liczby nie będą dokładnie reprezentowane
- Liczb zmiennoprzecinkowych nie należy przyrównywać do dokładnych wartości. Lepiej $|a - b| < \epsilon$

NIEDOKŁADNOŚĆ REPREZENTACJI

```
1  #include <stdio.h>
2  #include <math.h>
3  #define EPS 0.000001
4
5  int main()
6  {
7      float x = 0.1;
8
9      if( fabs(x-0.1) < EPS )
10         printf ("OK, %f jest rowne 0.1\n", x);
11     else
12         printf("Nie OK, %f nie jest rowne 0.1\n", x);
13
14     return 0;
15 }
```

 prec2.c

```

1  #include <stdio.h>
2  #include <float.h>
3
4  int main()
5  {
6      printf("Limity typu float:\n");
7      printf("sizeof(float)      : %ld\n", sizeof(float));
8      printf("Najwieksza wartosc   : %e\n", FLT_MAX );
9      printf("Najmniejsza dodatnia  : %e\n", FLT_MIN );
10     printf("Epsilon maszynowy    : %e\n", FLT_EPSILON );
11     printf("Cyfry znaczace      : %d\n", FLT_DIG );
12
13     printf("\nLimity typu double:\n");
14     printf("sizeof(double)       : %ld\n", sizeof(double));
15     printf("Najwieksza wartosc   : %e\n", DBL_MAX );
16     printf("Najmniejsza dodatnia  : %e\n", DBL_MIN );
17     printf("Epsilon maszynowy    : %e\n", DBL_EPSILON );
18     printf("Cyfry znaczace      : %d\n", DBL_DIG );
19
20     printf("\nLimity typu long double:\n");
21     printf("sizeof(long double)  : %ld\n", sizeof(long double));
22     printf("Najwieksza wartosc   : %Le\n", LDBL_MAX );
23     printf("Najmniejsza dodatnia  : %Le\n", LDBL_MIN );
24     printf("Epsilon maszynowy    : %Le\n", LDBL_EPSILON );
25     printf("Cyfry znaczace      : %d \n", LDBL_DIG );
26
27     return 0;
28 }

```

```

1 #include <stdio.h>
2 #include <float.h>
3
4 int main()
5 {
6     printf("Limity typu float:\n");
7     printf("sizeof(float)      : %ld\n", sizeof(float));
8     printf("Najwieksza wartosc   : %e\n", FLT_MAX);
9     printf("Najmniejsza dodatnia    : %e\n", FLT_MIN);
10    printf("Epsilon maszynowy      : %e\n", FLT_EPSILON);
11    printf("Cyfry znaczące           : %d\n", FLT_DIG);
12
13    printf("\nLimity typu double:\n");
14    printf("sizeof(double)       : %ld\n", sizeof(double));
15    printf("Najwieksza wartosc   : %e\n", DBL_MAX);
16    printf("Najmniejsza dodatnia : %e\n", DBL_MIN);
17    printf("Epsilon maszynowy      : %e\n", DBL_EPSILON);
18    printf("Cyfry znaczące       : %d\n", DBL_DIG);
19
20    printf("\nLimity typu long double:\n");
21    printf("sizeof(long double)  : %ld\n", sizeof(long double));
22    printf("Najwieksza wartosc   : %Le\n", LDBL_MAX);
23    printf("Najmniejsza dodatnia : %Le\n", LDBL_MIN);
24    printf("Epsilon maszynowy      : %Le\n", LDBL_EPSILON);
25    printf("Cyfry znaczące       : %d\n", LDBL_DIG);
26
27    return 0;
28 }

```

Wartości mogą się różnić zależnie od implementacji

Limity typu float:

```

sizeof(float)      : 4
Najwieksza wartosc : 3.402823e+38
Najmniejsza dodatnia : 1.175494e-38
Epsilon maszynowy   : 1.192093e-07
Cyfry znaczące     : 6

```

Limity typu double:

```

sizeof(double)     : 8
Najwieksza wartosc : 1.797693e+308
Najmniejsza dodatnia : 2.225074e-308
Epsilon maszynowy   : 2.220446e-16
Cyfry znaczące     : 15

```

Limity typu long double:

```

sizeof(long double) : 16
Najwieksza wartosc   : 1.189731e+4932
Najmniejsza dodatnia : 3.362103e-4932
Epsilon maszynowy     : 1.084202e-19
Cyfry znaczące       : 18

```

 float.c

- **nadmiar** (*overflow*) - przekroczenie zakresu (+Inf, -Inf)
- **niedomiar** (*underflow*) - zaokrąglenie bardzo małej liczby do 0
- redukcja cyfr przy odejmowaniu bardzo bliskich sobie liczb
- dodawanie (lub odejmowanie) dużej i małej liczby
- kolejność operacji może mieć wpływ na wynik
 $(a + b) + c \neq a + (b + c)$
- zaokrąglenia, np. liczby 0.1 nie można dokładnie reprezentować w systemie binarnym
- 25 luty 1991 r., wojna w zatoce perskiej, awaria systemu antyrakietowego Patriot (zegar rakiety tykał co 0.1 s.), zginęło 28 amerykańskich żołnierzy a 100 zostało rannych.

Problem: wyznaczyć sumę pierwszych n elementów szeregu harmonicznego

$$\sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \xrightarrow{n \rightarrow \infty} \infty$$

```

1  #include <stdio.h>
2
3  int main()
4  {
5      float x=0.0, y=0.0;
6      int i=1, n;
7
8      printf("n="); scanf("%d", &n);
9
10     while(i <= n)
11     {
12         x = x + 1.0/i;
13         y = y + 1.0/(n-i+1);
14         i = i + 1;
15     }
16
17     printf("x=%f\\ny=%f\\n", x, y);
18     return 0;
19 }

```

n=150
x=5.591181
y=5.591181

n=600
x=6.974980
y=6.974978

n=2000
x=8.178369
y=8.178370

n=500000
x=13.690692
y=13.699607

 szereg.c

- Typy całkowite: `char`, `short`, `int`, `long`
- Typy całkowite bez znaku: `unsigned`
- Typy zmiennopozycyjne: `float`, `double`
- Nadmiar, niedomiar i inne efekty wynikające z bitowej reprezentacji liczb
- $3/2$ wynosi 1, zaś $3/2.0$ wynosi 1.5
- Porównywanie wartości zmiennopozycyjnych $fabs(a-b) < EPS$

-  Jerzy Wałaszek, „*Binarne kodowanie liczb*”,
http://edu.i-lo.tarnow.pl/inf/alg/006_bin/index.php
-  Thomas Huckle, „*Collection of Software Bugs*”,
<http://www5.in.tum.de/~huckle/bugse.html>
-  Piotr Krzyżanowski, Leszek Plaskota, Materiały do wykładu
„*Metody numeryczne*”, Uniwersytet Warszawski, WMliM,
<http://wazniak.mimuw.edu.pl/>
-  WikiBooks, „*Kursie programowania w języku C*”,
Zaawansowane operacje matematyczne,
http://pl.wikibooks.org/wiki/C/Zaawansowane_operacje_matematyczne/

Reprezentacja symboli w komputerze.

Znaki alfabetu i łańcuchy znakowe.

- 7 bitów, liczby z zakresu 0-127
- litery (alfabet angielski) [a-zA-Z]
- cyfry [0-9],
- znaki przestankowe, np. , . ; '
- inne symbole drukowalne, np. * ^ \$
- białe znaki: spacja, tabulacja, ...
- polecenia sterujące: znak nowej linii \n, nowej strony, koniec tekstu, koniec transmisji,...
- 8-bitowe rozszerzenia ASCII: cp1250, latin2, ...

n	znak	n	znak	n	znak	n	znak	n	znak	n	znak	
0	NUL	'\0'	22	SYN	44	,	66	B	88	X	110	n
1	SOH		23	ETB	45	-	67	C	89	Y	111	o
2	STX		24	CAN	46	.	68	D	90	Z	112	p
3	ETX		25	EM	47	/	69	E	91	[	113	q
4	EOT		26	SUB	48	0	70	F	92	\	114	r
5	ENQ		27	ESC	49	1	71	G	93	]	115	s
6	ACK		28	FS	50	2	72	H	94	^	116	t
7	BEL	'\a'	29	GS	51	3	73	I	95	_	117	u
8	BS	'\b'	30	RS	52	4	74	J	96	'	118	v
9	HT	'\t'	31	US	53	5	75	K	97	a	119	w
10	LF	'\n'	32	SPACE	54	6	76	L	98	b	120	x
11	VT	'\v'	33	!	55	7	77	M	99	c	121	y
12	FF	'\f'	34	"	56	8	78	N	100	d	122	z
13	CR	'\r'	35	#	57	9	79	O	101	e	123	{
14	SO		36	\$	58	:	80	P	102	f	124	
15	SI		37	%	59	;	81	Q	103	g	125	}
16	DLE		38	&	60	<	82	R	104	h	126	~
17	DC1		39	'	61	=	83	S	105	i	127	DEL
18	DC2		40	(	62	>	84	T	106	j		
19	DC3		41	)	63	?	85	U	107	k		
20	DC4		42	*	64	@	86	V	108	l		
21	NAK		43	+	65	A	87	W	109	m		

- typ char, 1 bajt, liczba całkowita $[-128, 127]$
- **stała znakowa** - znak zapisany w apostrofach, np.:
 - 'A' to liczba 65,
 - 'A'+1 to liczba 66 (kod litery 'B'),
 - nowy wiersz '\n' to liczba 10,
 - tabulator '\t' to liczba 9,
 - znak '\0' ma wartość 0
 - powrót karetki '\r' (w MS Windows koniec linii to \r\n)
 - znaki o specjalnym zapisie:
 - ukośnik w lewo '\\',
 - apostrof '\'',
 - cudzysłów '\"',
 - znak zapytania '\?'
- 🖱️ man 7 ascii
- Uwaga: "A" nie jest pojedynczym znakiem lecz tablicą znaków!

```
1  #include<stdio.h>
2
3  int main()
4  {
5      char a;
6
7      printf("Podaj znak: ");
8      scanf("%c",&a);
9
10     printf("znak %c, dec %d, oct %o, hex %x\n",a,a,a,a);
11     return 0;
12 }
```

 char.c

WYKRYWANIE MAŁEJ LITERY

```
int islower(int x)
{
    if(x >= 'a' && x <= 'z' ) return 1;
    return 0;
}
```

ZAMIANA MAŁEJ LITERY NA DUŻĄ

```
int toupper(int x)
{
    if( islower(x) ) x = x + 'a' - 'A';
    return x;
}
```

Plik nagłówkowy: `ctype.h`

KLASYFIKACJA ZNAKÓW

- `isalnum()` - litera alfabetu lub cyfra [A-Za-z0-9]
- `isalpha()` - litera alfabetu [A-Za-z]
- `islower()`, `isupper` - mała / duża litera alfabetu
- `isdigit()` - cyfra
- `isgraph()` - znaki drukowalne (ze spacją)
- `isspace()` - znaki białe
- `isprint()` - znaki drukowalne (bez spacji)
- `ispunct()` - znaki przestankowe (drukowalne bez liter i cyfr)

MANIPULACJE NA ZNAKACH

- `tolower` , `toupper` - zamiana wielkości liter alfabetu

PRZYKŁADY W C: ZAMIANA WIELKOŚCI ZNAKÓW

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main()
5  {
6      int a;
7
8      do
9      {
10         a=getchar();
11         putchar(toupper(a));
12     }while( a != EOF );
13
14     return 0;
15 }
```

 toupper2.c

- funkcja `getchar()`
zwraca pojedynczy znak
ze standardowego wejścia
lub EOF
- funkcja `putchar()`
wypisuje znak
- EOF - koniec pliku
(*end of file*)
- `toupper2 < plik.txt`
- EOF w terminalu:
CTRL+Z (Windows)
CTRL+D (Unix/Linux)

- **Strona kodowa:** kodowanie binarne znaków pisarskich
- Polskie ogonki, 8 bitowe rozszerzenia ASCII:
 - ☞ ISO 8859-2 (latin2), alfabet łaciński dla Europy środkowej i wschodniej, UNIX/GNU Linux
 - ☞ CP-1250, języki środkowoeuropejskie, MS Windows
 - ☞ CP-852, MS DOS
- ☞ Kodowanie polskich znaków diakrycznych

- **Unicode** - standard z założenia obejmujący wszystkie języki świata (obecnie ponad 110k znaków ze 100 grup językowych).
- Określa przydział przestrzeni numeracyjnej poszczególnym grupom znaków
- Kodowanie znaków: UCS (Universal Character Set), UTF (Unicode Transformation Format), UTF-8, UTF-16, UTF-32
- UTF-8 - znaki kodowane za pomocą 1, 2, 3 bajtów
- Każdy tekst w ASCII jest tekstem w UTF-8
- Znaki ASCII od 0 do 127 są mapowane jako jeden bajt.
- Znaki spoza ASCII (np. polskie znaki) są mapowane jako dwa bajty (lub więcej).
- Ten sam znak można zapisać na kilka sposobów - wybieramy najkrótszy zapis.

- **Łańcuch** (*string*) - skończony ciąg symboli alfabetu
- W języku C brak typu `string`, występuje w C++
- Łańcuch to tablica zawierająca ciąg znaków zakończony znakiem `\0` (wartość zero)

A	l	a		m	a		k	o	t	a	\0
---	---	---	--	---	---	--	---	---	---	---	----

- Dostęp do znaku jak w tablicach: `t[i]`
- Stała napisowa (literał znakowy): `"Ała ma kota"`
- stała znakowa `'A'` to liczba 65
- stała napisowa `"A"` to tablica

A	\0
---	----


```
#include<stdio.h>

int main()
{
    char *s1="nieciekawy fragment tekstu.";
    char s2[]="Ala ma kota";

    s1[0]='N';
    s2[0]='E';

    printf(s1);
    printf(" %s\n",s2);

    printf("\nBardzo %s\n", s1+3);

    return 0;
}
```

s1 to stały napis

Źle!

```
scanf("%s",...);
```

```
#include<stdio.h>
int main()
{
    char text[10];
    scanf("%s", text);
}
```

- niebezpieczeństwo przepełnienia tablicy
- `scanf("%10s",text);`
- wczyta tylko pierwszy wyraz, nie wczyta całego łańcucha z odstępami: "Ala ma kota"

```
char *gets(char *s);
```

- funkcja `gets()` czyta linię tekstu ze standardowego wejścia i umieszcza ją w tablicy `s`
- Uwaga: brak możliwości zabezpieczenia się przez przepełnieniem tablicy

```
#include<stdio.h>
int main()
{
    char text[10];
    gets(text);
}
```

```
char *fgets(char *s, int n, FILE *f);
```

- czyta linię tekstu, ale nie więcej niż `n` znaków, ze strumienia `f` i umieszcza tekst w tablicy `s`
- obowiązkowe podanie parametru `n`
- standardowy strumień wejściowy to `stdin`

```
#include<stdio.h>
int main()
{
    char text[10];
    fgets(text, 10, stdin);
}
```

```
int puts(char *s);
```

- wypisuje łańcuch i dodaje znak nowej linii

```
int printf(char *s, ...);
```

- wypisuje napis zgodnie z zadany formatem

The diagram illustrates the mapping between the format specifiers in a `printf` statement and the resulting output. It shows the input code: `printf("Color %s, Number %d, Float %.2f", "red", 123456, 3.14;)`. Arrows point from each format specifier to its corresponding output value: `%s` maps to `red`, `%d` maps to `123456`, and `%.2f` maps to `3.14`. Additionally, arrows point from the opening and closing parentheses of the format string to the opening and closing parentheses of the argument list, respectively.

Input: `printf("Color %s, Number %d, Float %.2f", "red", 123456, 3.14;)`

Output: Color red, Number 123456, Float 3.14

SPECYFIKACJA FORMATU

`%[flagi] [szerokość] [.precyzja] [długość] specyfikator`

%[flagi] [szerokość] [.precyzja] [długość]specyfikator

specyfikator	znaczenie	przykład
d lub i	liczba całkowita ze znakiem	-123
u	liczba całkowita bez znaku	123
o	liczba całkowita bez znaku ósemkowa	123
x lub X	liczba całkowita bez znaku szesnastkowo	fa1 FA1
f lub F	zmiennopozycyjna w zapisie dziesiętnym	123.45678
e lub E	notacja naukowa	1.2345678e+2
g lub G	krótszy zapis %f lub %e	123.4657
c	pojedynczy znak (ASCII)	a
s	łańcuch znakowy	Ala ma kota
p	adres (wskaźnik), szesnastkowo	ff01ffab
%	wypisuje znak %	%

%[flagi] [szerokość] [.precyzja] [długość]specyfikator

flaga	znaczenie	przykład
-	wyrównanie do lewej (względem podanej szerokości)	123
+	liczby dodatnie poprzedzone znakiem +	+123
<i>spacja</i>	wstawia spację zamiast znaku +	123
0	wypełnia podaną szerokość znakiem 0	000123
#	liczby ósemkowe i szesnastkowe poprzedza 0, 0x, 0X	0123 0xfa1

szerokość	znaczenie
<i>liczba</i>	określa minimalną ilość znaków wypisanych (szerokość pola), brakujące miejsca są dopełniane spacjami. Jeżeli szerokość pola jest za mała to wynik nie jest obcinany.
*	szerokość nie jest dana w formacie lecz poprzez argument funkcji printf

`%[flagi] [szerokość] [.precyzja] [długość]specyfikator`

precyzja	znaczenie
<i>.liczba</i>	ilość cyfr wypisywanych po przecinku
<i>.*</i>	precyzja liczby zmiennopozycyjnej nie jest podawana w formacie lecz poprzez argument funkcji <code>printf</code> . W przypadku łańcuchów znakowych oznacza maksymalną liczbę wypisanych znaków (łańcuch jest obcinany)

długość	znaczenie
<i>brak</i>	liczby <code>int</code> , <code>double</code> , <code>float</code>
<i>l</i>	liczby <code>long int</code>
<i>ll</i>	liczby <code>long long int</code>
<i>L</i>	liczby <code>long double</code>

```

1  #include <stdio.h>
2
3  int main()
4  {
5      printf ("Znaki                : %c %c \n", 'A', 65);
6      printf ("Liczby calkowite     : %d \n", 123);
7      printf ("   ze znakiem              : %+d \n", 123);
8      printf ("   szesnastkowo             : %x %#x \n", 123, 123);
9      printf ("   dopelnienie spacjami    : %20d \n", 123);
10     printf ("   dopelnienie zerami      : %020d \n", 123);
11     printf ("Zmienopozycyjne           : %f \n", 3.1416);
12     printf ("   notacja naukowa         : %e \n", 3.1416);
13     printf ("   precyzja                : %.3f \n", 3.1416);
14     printf ("   dopelnienie             : %20.3f \n", 3.1416);
15     printf ("Szerokosc pola      *      : %*d \n", 20, 123);
16     printf ("Precyzja      .*         : %20.*f \n", 3, 3.1416);
17     printf ("Napis                  : %s \n", "Ala ma kota");
18     printf ("   dopelnienie          : %20s \n", "Ala ma kota");
19     printf ("   obciecie             : %20.5s \n", "Ala ma kota");
20
21     return 0;
22 }

```

```

1  #include <stdio.h>
2
3  int main()
4  {
5      printf ("Znaki                : %c %c \n", 'A', 65);
6      printf ("Liczby calkowite      : %d \n", 123);
7      printf ("   ze znakiem              : %+d \n", 123);
8      printf ("   szesnastkowo              : %x %#x \n", 123, 123);
9      printf ("   dopelnienie spacjami      : %20d \n", 123);
10     printf ("   dopelnienie zerami        : %020d \n", 123);
11     printf ("Zmienopozycyjne             : %f \n", 3.1416);
12     printf ("   notacja naukowa           : %e \n", 3.1416);
13     printf ("   precyzja                   : %.3f \n", 3.1416);
14     printf ("   dopelnienie                : %20.3f \n", 3.1416);
15     printf ("Szerokosc pola *             : %*d \n", 20, 123);
16     printf ("Precyzja                     : Znaki                : A A
17     printf ("Napis                         : Liczby calkowite      : 123
18     printf ("   dopelnienie               : ze znakiem              : +123
19     printf ("   obciecie                   : szesnastkowo              : 7b 0x7b
20                                     dopelnienie spacjami      : 123
21     return 0;                                     dopelnienie zerami      : 000000000000000000123
22 }                                     Zmienopozycyjne             : 3.141600
                                     notacja naukowa           : 3.141600e+00
                                     precyzja                   : 3.142
                                     ...

```

```

1  #include <stdio.h>
2
3  int main()
4  {
5      printf ("Znaki
6      printf ("Liczby call
7      printf (" ze znakie
8      printf (" szesnastk
9      printf (" dopelnier
10     printf (" dopelnienie zerami
11     printf ("Zmienopozycyjne : %f \n", 3.1416);
12     printf (" notacja naukowa : %e \n", 3.1416);
13     printf (" precyzja : %.3f \n", 3.1416);
14     printf (" dopelnienie : %20.3f \n", 3.1416);
15     printf ("Szerokosc pola * : %*d \n", 20, 123);
16     printf ("Precyzja .* : %20.*f \n", 3, 3.1416);
17     printf ("Napis : %s \n", "Ala ma kota");
18     printf (" dopelnienie : %20s \n", "Ala ma kota");
19     printf (" obciecie : %20.5s \n", "Ala ma kota");
20
21     return 0;
22 }

```

```

...
Zmienopozycyjne      : 3.141600
notacja naukowa      : 3.141600e+00
precyzja              : 3.142
dopelnienie           :                3.142
Szerokosc pola *     :                123
Precyzja .*           :                3.142
Napis                 : Ala ma kota
dopelnienie           :                Ala ma kota
obciecie              :                Ala m

```

- plik nagłówkowy `string.h`
- długość łańcucha
`int strlen(const char *s);`
- łączenie dwóch łańcuchów
`char *strcat(char *dest, const char *src);`
- porównywanie łańcuchów
`int strcmp(const char *s1, const char *s2);`
- kopiowanie łańcuchów
`char *strcpy(char *dest, const char *src);`
- wyszukiwanie wzorca
`char *strstr(const char *napis, const char *wzor);`

Problem: wyszukiwanie podciągu (*pattern matching*).

W ciągu $\mathcal{T}$ znajdź wystąpienie wzorca $\mathcal{W}$.

Tekst $\mathcal{T}$ = "programowanie"

p	r	o	g	r	a	m	o	w	a	n	i	e	\0
---	---	---	---	---	---	---	---	---	---	---	---	---	----

Wzorzec $\mathcal{W}$ = "gra"

g	r	a	\0
---	---	---	----

Algorytm 14 Naiwne wyszukiwanie wzorca

Dane wejściowe: łańcuch znaków $\mathcal{T}$, wzorzec $\mathcal{W}$

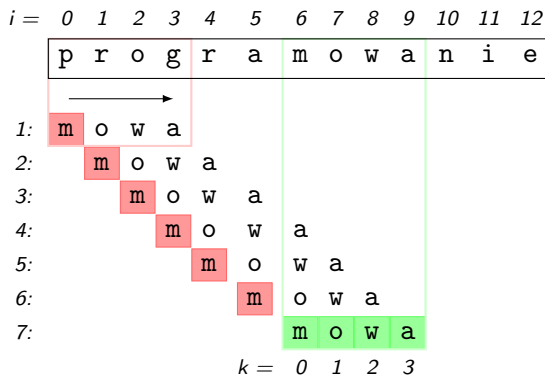
Wynik: pozycja tekstu $\mathcal{W}$ w $\mathcal{T}$ lub wartość -1, gdy brak

- 1: **dla każdego** $i = 0, 1, 2, \dots, |\mathcal{T}| - |\mathcal{W}|$ **wykonuj**
 - 2: $k \leftarrow 0$
 - 3: **dopóki** $\mathcal{T}[i + k] = \mathcal{W}[k]$ **i** $k < |\mathcal{W}|$ **wykonuj**
 - 4: $k \leftarrow k + 1$
 - 5: **jeżeli** $k = |\mathcal{W}|$ **wykonaj**
 - 6: **zwróć** i
 - 7: **zwróć** -1
-

$|\mathcal{W}|$ oznacza długość łańcucha $\mathcal{W}$

Tekst $\mathcal{T}$ = "programowanie"

Wzorzec $\mathcal{W}$ = "mowa"




```
1  int strindex(char t[], char w[])
2  {
3      int i, k;
4
5      i=0;
6      while(t[i] != '\0')
7      {
8          k=0;
9          while(t[i+k]==w[k] && w[k] != '\0')
10             k = k + 1;
11             if(w[k]=='\0') return i;
12             i = i + 1;
13     }
14     return -1;
15 }
```

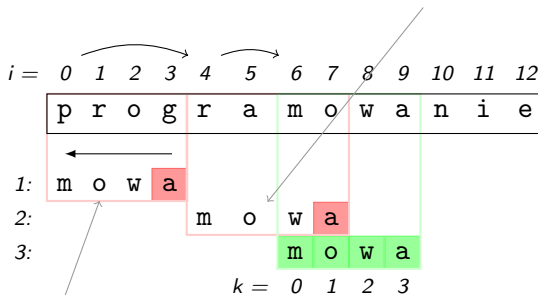
 strindex.c

UPROSZCZONY ALGORYTM BOYERA-MOORE'A

Tekst $\mathcal{T}$ = "programowanie"

Wzorzec $\mathcal{W}$ = "mowa"

'o' występuje we wzorcu na pozycji 1
przesunięcie okna o $k-1 = 2$



'g' nie występuje we wzorcu
przesunięcie okna o $k+1 = 4$

TABLICA PRZESUNIĘĆ

- informacja o tym czy znak występuje we wzorcu
- rozmiar tablicy odpowiada liczbie znaków danego alfabetu, dla kodu ASCII wystarczy 128 elementów
- przechowuje pozycję ostatniego wystąpienia znaku we wzorcu lub -1 gdy znak nie występuje
- $tp[w[k]] = k$
- przesunięcie okna o wartość $\max(k - tp[w[k]], 1)$

```
1 void utworz_tp(int tp[], char w[])
2 {
3     int i=0;
4
5     while(i<128)
6     {
7         tp[i]=-1;
8         i=i+1;
9     }
10
11     i=0;
12     while(w[i] != '\0')
13     {
14         tp[w[i]]=i;
15         i=i+1;
16     }
17 }
```

Algorytm 15 Uproszczony algorytm Boyera-Moore'a

Dane wejściowe: łańcuch znaków $\mathcal{T}$, wzorzec $\mathcal{W}$

Wynik: pozycja tekstu $\mathcal{W}$ w $\mathcal{T}$ lub wartość -1, gdy brak

- 1: $\mathcal{P} \leftarrow \text{utworz_tp}(\mathcal{W})$ ▷ tworzenie tablicy przesunięć
 - 2: $i \leftarrow 0$
 - 3: **dopóki** $i \leq |\mathcal{T}| - |\mathcal{W}|$ **wykonuj**
 - 4: $k \leftarrow |\mathcal{W}|$ ▷ porównywanie od tyłu
 - 5: **dopóki** $k \geq 0$ **i** $\mathcal{T}[i + k] = \mathcal{W}[k]$ **wykonuj**
 - 6: $k \leftarrow k - 1$
 - 7: **jeżeli** $k = -1$ **wykonaj**
 - 8: **zwróć** i
 - 9: **w przeciwnym wypadku**
 - 10: $i \leftarrow i + \max(1, k - \mathcal{P}[\text{kod}(\mathcal{T}[i + k])])$
 - 11: **zwróć** -1
-

gdzie $\text{kod}(x)$ oznacza kod liczbowy znaku x .

```

1  int strindex2(char t[], char w[])
2  {
3      int i, k, z, tl, wl;
4      int tp[128];
5
6      utworz_tp(tp, w);
7      tl=strlen(t);
8      wl=strlen(w);
9
10     i=0;
11     while(i <= tl-wl)
12     {
13         k=wl-1;
14         while(k>=0 && t[i+k]==w[k] ) k=k-1;
15         if(k== -1) return i;
16         z=k-tp[t[i+k]];
17         if(z<1) z=1;
18         i = i + z;
19     }
20     return -1;
21 }

```

- Typ znakowy jest liczbą całkowitą
- Łańcuch to tablica znaków zakończona znakiem `'\0'`
- Stała znakowa w apostrofach `'A'`
- Stała napisowa w cudzysłowach `"A"` jest tablicą
- Porównywanie łańcuchów: `t == "napis"` źle, trzeba znak po znaku (funkcja `strcmp()`)
- Kopiowanie napisów: `t = "napis"` źle, kopiowanie tablic (funkcja `strcpy()`)

-  Linux Programmer's Manual
man 7 ascii unicode codepages iso_8859-2
<http://www.unix.com/man-page/>
-  Wikipedia:  Kodowanie polskich znaków diakrycznych
-  Jerzy Wałaszek, „*Algorytmy. Struktury danych.*”,
 Łańcuchy znakowe.
-  R.S. Boyer, J. Strother Moore, *A Fast String Searching Algorithm* Communications of the Association for Computing Machinery, 20(10), 1977, pp. 762-772.
www.cs.utexas.edu/~moore/publications/fstrpos.pdf

Operatory

Operatory bitowe i uzupełnienie informacji o pozostałych operatorach.

PRZYPOMNIENIE: OPERATORY

Operator przypisania			
=	przypisanie	$x = y$	$x \leftarrow y$
Operatory arytmetyczne			
*	mnożenie	$x * y$	$x \cdot y$
/	dzielenie	x / y	$\frac{x}{y}$
+	dodawanie	$x + y$	$x + y$
-	odejmowanie	$x - y$	$x - y$
%	reszta z dzielenia (modulo)	$x \% y$	$x \bmod y$
++	inkrementacja	$x++$	$x \leftarrow x + 1$
--	dekrementacja	$x--$	$x \leftarrow x - 1$
Operatory relacji			
<	mniejszy niż	$x < y$	$x < y$
>	wiekszy niż	$x > y$	$x > y$
<=	mniejszy lub równy	$x \leq y$	$x \leq y$
>=	wiekszy lub równy	$x \geq y$	$x \geq y$
==	równy	$x == y$	$x = y$
!=	różny	$x != y$	$x \neq y$
Operatory logiczne			
!	negacja (NOT)	$!x$	$\neg x$
&&	koniunkcja (AND)	$x > 1 \ \&\& \ y < 2$	$x > 1 \wedge y < 2$
	alternatywa (OR)	$x < 1 \ \ \ y > 2$	$x < 1 \vee y > 2$
Operatory wskaźnikowe			
&	referencja (pobranie adresu)	$\&x$	
*	dereferencja (dostęp pod adres)	$*x$	

operator	znaczenie	przykład
~	negacja bitowa (NOT)	~x
&	koniunkcja bitowa (AND)	x & y
	alternatywa bitowa (OR)	x y
^	alternatywa rozłączna (XOR)	x ^ y

- działają na pojedynczych bitach
- zdefiniowane dla liczb całkowitych
- najbezpieczniej używać wyłącznie dla liczb całkowitych bez znaku (unsigned)

NEGACJA

$\sim$	0	1
	1	0

KONIUNKCJA (AND)

$\&$	0	1
0	0	0
1	0	1

ALTERNATYWA (OR)

$\mid$	0	1
0	0	1
1	1	1

ALTERNATYWA ROZŁĄCZNA (XOR)

$\wedge$	0	1
0	0	1
1	1	0

```

1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int a = 9;
6      unsigned int b = 12;
7
8      printf("a      = %u\nb      = %u\n", a, b);
9      printf("~a      = %u\n~b      = %u\n", ~a, ~b);
10     printf("a & b = %u\n", a & b);
11     printf("a | b = %u\n", a | b);
12     printf("a ^ b = %u\n", a ^ b);
13
14     return 0;
15 }

```

a	=	9
b	=	12
~a	=	4294967286
~b	=	4294967283
a & b	=	8
a b	=	13
a ^ b	=	5

[illegible]

Operacja XOR jest wykorzystywana np. w funkcjach haszujących i algorytmach szyfrujących.

$$A \oplus B \oplus B \longrightarrow A$$

```

1  #include<stdio.h>
2
3  int main()
4  {
5      unsigned char klucz = 13;
6      char znak;
7
8      while( (znak=getchar()) != EOF )
9          putchar( znak ^ klucz );
10
11     return 0;
12 }
```

operator	znaczenie	przykład
<<	przesunięcie bitowe w lewo	$x \ll y$
>>	przesunięcie bitowe w prawo	$x \gg y$

- pozycje wszystkich bitów są przesuwane
- bity skrajne są tracone, puste miejsca są zastępowane zerami
- odpowiada do mnożeniu/dzieleniu całkowitemu przez 2


```
1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int a = 5;
6
7      printf("a          = %u\n", a);
8      printf("a << 1 = %u\n", a << 1);
9      printf("a << 2 = %u\n", a << 2);
10     printf("a >> 1 = %u\n", a >> 1);
11     printf("a >> 2 = %u\n", a >> 2);
12
13     return 0;
14 }
```

a	=	5
a << 1	=	10
a << 2	=	20
a >> 1	=	2
a >> 2	=	1

Maska bitowa: słowo (wartość) używane w operacjach bitowych do ustawienia, zmiany lub odczytu poszczególnych bitów.

Ustawienie pojedynczego bitu na 1 operatorem OR

10011101	10010101
00001000	00001000
= 10011101	10011101

```
int zapal_bit(int x, int n)
{
    return x | 1 << n;
}
```

Włączenie wielu bitów zadaną maską

$x = x \mid \text{MASKA}$

Ustawienie pojedynczego bitu na 0 operatorem AND i negacją

10011101	10010101
& 11110111	11110111
= 10010101	10010101

```
int zgas_bit(int x, int n)
{
    return x & ~(1 << n);
}
```

Wyzerowanie wielu bitów zadaną maską

`x = x & ~MASKA`

Sprawdzenie wartości bitu operatorem AND

10011101	10010101
& 00001000	00001000
= 00001000	00000000

```
int sprawdz_bit(int x, int n)
{
    return x & 1 << n;
}
```

Sprawdzenie czy bity pokrywają się z maską

```
if ( (x & MASKA) == MASKA ) { ... }
```

Zamiana wartości bitu za pomocą operatora XOR

10011101	10010101
$\wedge$ 00001000	00001000
= 10010101	10011101

```
int zamien_bit(int x, int n)
{
    return x ^ 1 << n;
}
```

Zamiana wielu bitów zadaną maską

$x = x \wedge \text{MASKA}$

Rzutowanie: konwersja wartości z jednego typu na inny typ

- **niejawne**, dokonywane automatycznie

```
int a = 3.14;  
float x = a;
```

- **jawne**, wykonane przy pomocy operatora rzutowania `()`
(typ)wartość

```
int a = (int)3.14;  
float b = 3/(float)2;  
char *w = (char *)malloc(10);
```

Przy rzutowaniu z typu bardziej pojemnego do mniej pojemnego możliwa utrata dokładności (nadmiar, *overflow*), taka sytuacja nie jest (zazwyczaj) sygnalizowana

Skrócony zapis operacji podstawienia:

$$a = a \bigcirc b \quad \Leftrightarrow \quad a \bigcirc = b$$

$a += b$	$a = a + b$	$a = b$	$a = a b$
$a -= b$	$a = a - b$	$a \&= b$	$a = a \& b$
$a *= b$	$a = a * b$	$a ^= b$	$a = a ^ b$
$a /= b$	$a = a / b$	$a <<= b$	$a = a << b$
$a \% = b$	$a = a \% b$	$a >>= b$	$a = a >> b$

Uwaga na kolejność, zamiana nie zawsze powoduje błąd:

<code>a -=b;</code>	<code>/* OK */</code>
<code>a =-b;</code>	<code>/* czy poprawnie? */</code>

- Jednoargumentowy operator inkrementacji ++ i operator dekrementacji --

++a	pre-inkrementacja, to samo co a=a+1
a++	post-inkrementacja
--a	pre-dekrementacja, to samo co a=a-1
a--	post-dekrementacja

- Operator post-inkrementacji i post-dekrementacji zwracają wartość poprzednią (przed zwiększeniem/zmniejszeniem). Zmiana wartości argumentu odbywa się później (po wyliczeniu całego wyrażenia).

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 1;
6      int b;
7
8      b = ++a;
9      printf("a=%d, b=%d\n", a, b);
10
11     b = a++;
12     printf("a=%d, b=%d\n", a, b);
13
14     b = --a;
15     printf("a=%d, b=%d\n", a, b);
16
17     b = a--;
18     printf("a=%d, b=%d\n", a, b);
19
20     return 0;
21 }
```

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 1;
6      int b;
7
8      b = ++a;
9      printf("a=%d, b=%d\n", a, b);
10
11     b = a++;
12     printf("a=%d, b=%d\n", a, b);
13
14     b = --a;
15     printf("a=%d, b=%d\n", a, b);
16
17     b = a--;
18     printf("a=%d, b=%d\n", a, b);
19
20     return 0;
21 }
```

a=2, b=2
a=3, b=2
a=2, b=2
a=1, b=2

Kopiowanie napisu z tablicy a do b

```
void strcpy(char *a, char *b)
{
    while( *a != '\0' )
    {
        *b = *a;
        a = a + 1;
        b = a + 1;
    }
    *b = '\0';
}
```

Dokładnie to samo

```
void strcpy(char *a, char *b)
{
    while(*b++ = *a++);
}
```

OPERATOR WARUNKOWY ?:

wyrażenie ? wartość 1 : wartość 2

- jeżeli *wyrażenie* jest prawdą to wynikiem jest *wartość 1*, w przeciwnym wypadku wynikiem jest *wartość 2*
- jedyny operator trzyargumentowy

PRZYKŁAD

Wyznaczanie większej wartości z 2 liczb:

$a = (b > c) ? b : c;$

Wartość absolutna:

$a = (a < 0) ? -a : a;$

- przecinek, wyrażenie rozdzielone przecinkiem wykonywane są od lewej do prawej. Wynikiem jest ostatnia instrukcja.

```
a = 1, b = 2, c = 3;
```

- dostęp do elementów tablicy []

```
a = t[i+1];
```

- wywołanie funkcji ()
- grupowanie wyrażeń ()

```
a = (b + c)/(b - c);
```

- dostęp do pól struktur . (kropka) i ->

```
a = student.nazwisko;  
b = wskaznik->nazwisko
```

- **Priorytet** operatora decyduje o kolejności wykonywania operacji, np. mnożenie przed dodawaniem.

```
a = 3;  
x = a + a * a;
```

- **Łączność** operatora (lewostronna lub prawostronna) decyduje o kolejności wykonywania obliczeń dla operatorów o takim samym priorytecie

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

Operator		Łączność
 priorytet	()	
	[] . -> () ++ --	lewostronna
	! ~ + - * & sizeof() ++ -- ()	prawostronna
	* / %	lewostronna
	+ -	lewostronna
	<< >>	lewostronna
	<<= >>=	lewostronna
	== !=	lewostronna
	&	lewostronna
	^	lewostronna
		lewostronna
	&&	lewostronna
		lewostronna
	? :	prawostronna
	= += -= *= /= %= &= = ^=	prawostronna
	,	lewostronna


```
int a, b=1, c=2;  
a = -b++ + ++c << 3 / 2 * (int)2.5 ;  
printf("a=%d, b=%d, c=%d\n", a ,b ,c);
```

Jaki będzie wynik?

```
int a, b=1, c=2;  
a = -b++ + ++c << 3 / 2 * (int)2.5 ;  
printf("a=%d, b=%d, c=%d\n", a ,b ,c);
```

Jaki będzie wynik?

a=8, b=2, c=3

- ❶ `b++` zwiększa wartość `b` na 2 ale zwraca 1
`a = -1 + ++c << 3 / 2 * (int)2.5 ;`
- ❷ `++c` zwiększa wartość `c` na 3, rzutowanie `(int)2.5` daje 2
`a = -1 + 3 << 3 / 2 * 2 ;`
- ❸ dzielenie `3/2` zwraca 1, które następnie jest mnożone przez 2 (łączność lewostronna)
`a = -1 + 3 << 2 ;`
- ❹ `a = 2 << 2 ;`
- ❺ `a = 8 ;`

W razie wątpliwości co do kolejności obliczeń używaj nawiasów grupujących (mają najwyższy priorytet)

-  Stephan Brumme, „*the bit twiddler*”,
<http://bits.stephan-brumme.com/>
-  Alex Aliain, „*Bitwise Operators in C and C++: A Tutorial*”,
http://www.cprogramming.com/tutorial/bitwise_operators.html

Typy złożone

Struktury, pola bitowe i unie.

- Typy całkowite:
 - char
 - short
 - int
 - long
- Typy zmiennopozycyjne
 - float
 - double
- Modyfikatory : unsigned, signed
- Typ wskaźnikowy

- Typ wyliczeniowy `enum`
- Tablice
- Struktury `struct`
- Unie `union`
- Pola bitowe
- Wskaźniki na funkcje i wskaźniki na typy złożone

DEKLARACJA

```
enum nazwa_typu
{
    element1,
    element2,
    ...
};
```

Przykład

```
enum kolory { czerwony, niebieski, zielony, czarny };

int main()
{
    enum kolory n;
    n = 0;
    if ( n == czerwony ) printf("To kolor czerwony\n");
}
```


- Typ wyliczeniowy enum to liczba całkowita (prawie to samo co int)
- Operacje arytmetyczne takie same jak na typie całkowitym

```
enum kolory n=zielony;  
n++;  
n=n+100;
```

- Poprawia czytelność kodu
- Może być zastąpiony przez odpowiednie dyrektywy, np.:

```
#define ZIELONY 1  
#define CZERWONY 2  
int kolor = ZIELONY;
```

```
1 #include <stdio.h>
2
3 enum kolory { zielony, niebieski, czerwony, czarny };
4
5 int main()
6 {
7     enum kolory n;
8
9     for(n=zielony; n<=czarny; n++)
10         printf("%d\\n", n);
11
12     return 0;
13 }
```

0
1
2
3

 enum1.c

```
1 #include <stdio.h>
2
3 enum kolory { zielony=3, niebieski=7, czerwony, czarny };
4
5 int main()
6 {
7     enum kolory n;
8
9     printf("sizeof = %d\n", sizeof(n));
10
11     for(n=zielony; n<=czarny; n++)
12         printf("%d\n", n);
13
14     return 0;
```

sizeof = 4

3

4

5

6

7

8

9

 enum2.c

- Polecenie typedef pozwala nadać własne nazwy dowolnym typom
- Deklaracja typu wygląda identycznie jak deklaracja zmiennej, należy tylko dodać słowo typedef na początku

```
typedef float liczba;
typedef float punkt[4];
typedef enum { false, true } bool;
typedef float *wskaznik;

int main()
{
    liczba x = 3.14;
    punkt y;
    bool czy_koniec = false;
    wskaznik w = &x;
}
```

- Polecenie typedef pozwala uniknąć powtarzania słowa struct lub enum

```
struct student
{
    char nazwisko[100];
    int indeks;
};

typedef struct student student_s;

void wypisz_student(student_s s)
{
    printf("Nazwisko: %s\n", s.nazwisko);
}
```

Operator -> umożliwia dostęp do pola struktury za pomocą wskaźnika.
wskaźnik->pole jest równoważne z (*wskaźnik).pole

```
1 #include <stdio.h>
2
3 typedef enum { M, K } plec;
4
5 typedef struct
6 {
7     char nazwisko[100];
8     int indeks;
9     plec p;
10 } student;
11
12 void wypisz_studenta(const student *s)
13 {
14     printf("Nazwisko: %s\n", s->nazwisko);
15     printf("Indeks   : %d\n", (*s).indeks);
16     printf("Plec     : %s\n", s->p == M ? "M" : "K");
17 }
18
19 int main()
20 {
21     student s = {"Zenon Adamski", 123456, M};
22     wypisz_studenta(&s);
23     return 0;
24 }
```

- Unie, podobnie jak struktury, przechowują w polach zmienne dowolnego typu
- W odróżnieniu od struktur, wszystkie zmienne umieszczone są pod tym samym adresem (przechowywana jest pojedyncza wartość).
- Modyfikacja jednego pola zmienia wartość pozostałych.

```
1 union ufloat
2 {
3     float f;
4     unsigned int i;
5     unsigned char c[4];
6 };
```

 union1.c

```
1 #include <stdio.h>
2
3 union liczba
4 {
5     float f;
6     unsigned int i;
7     unsigned char c;
8 };
9
10 int main()
11 {
12     union liczba x;
13
14     x.f = 3.14;
15
16     printf("sizeof = %d\n", sizeof(union liczba));
17
18     printf("adres pola f = %p\n", &x.f);
19     printf("adres pola i = %p\n", &x.i);
20     printf("adres pola c = %p\n", &x.c);
21
22     printf("wartosc pola f = %f\n", x.f);
23     printf("wartosc pola i = %x\n", x.i);
24     printf("wartosc pola c = %x\n", x.c);
25
26 }
```


- Pola bitowe to pola struktury o określonej liczbie bitów
- Pozwalają zoptymalizować zużycie pamięci w reprezentacji zmiennych całkowitych
- Wykorzystywane m. in. przy obsłudze urządzeń zewnętrznych przez porty

```
struct data
{
    unsigned int dzien    : 5 ;
    unsigned int miesiac  : 4 ;
    unsigned int rok      : 11 ;
};
```

```
1  #include<stdio.h>
2
3  typedef struct
4  {
5      unsigned int  dzien    :   5  ;
6      unsigned int  miesiac  :   4  ;
7      unsigned int  rok      :  11  ;
8  } data;
9
10 int main()
11 {
12     data dzis = { 24, 1, 2017 };
13
14     printf("Data: %u-%u-%u\n",
15           dzis.dzien, dzis.miesiac, dzis.rok);
16
17     printf("sizeof(data)=%ld\n", sizeof(data));
18
19     return 0;
```

Pola bitowe pozwalają poprawić czytelność kodu zawierającego operacje na bitach, jednak działają wolniej niż operatory bitowe.

```
typedef struct
{
    int prawo_odczytu: 1;
    int prawo_zapisu : 1;
    int prawo_wykonania : 1;
} atrybuty_pliku;

int main()
{
    atrybuty_pliku plik;

    if ( plik.prawo_odczytu )
    {
        /*    ...    */
    }
}
```

- deklarowanie własnych typów poleceniem typedef
- struktury, unie i pola bitowe
- typ wyliczeniowy enum

Strumienie i pliki.

- **Plik** - ciąg bajtów o skończonej długości
- Nawa pliku nie stanowi jego części
- Położenie pliku określone przez ścieżkę dostępu
- Pliki są opatrzone **atrybutami**: uprawnienia, własności pliku, np. plik ukryty, itp.
- Plik tekstowy a plik binarny
- W Unix/Linux wszystko jest plikiem

<http://pl.wikipedia.org/wiki/Plik>

Format pliku: określa rodzaj i sposób zapisu danych w pliku.

- Rozszerzenia plików (DOS, Winows)
txt html c csv bmp mp3 jpg
- Metadane zawarte w pliku, sygnatura formatu zakodowana najczęściej na początku pliku: nagłówek pliku, liczba magiczna
FF D8 FF dla formatu jpg
D0 CF 11 E0 dokumenty MS Office
- Metadane zewnętrzne, np. umieszczone w systemie plików.
Typy **MIME**, Multipurpose Internet Mail Extensions
Content-Type: text/plain
Content-Type: audio/mpeg:

http://en.wikipedia.org/wiki/List_of_file_signatures

http://en.wikipedia.org/wiki/File_format

System plików: sposób przechowywania plików na nośniku.

- fizyczny zapis danych, blokowa struktura danych (sektory, klastry)
- logiczna struktura widoczna dla użytkownika
- DOS/Windows: FAT, FAT32, NTFS
- Linux: ext2, ext3, ext4
- inne: HFS (Mac OS), NFS (sieć), ISO9660 (CD-ROM)
- Hierarchia systemów plików: katalogi, podkatalogi i pliki.

Dostęp do pliku: ścieżka + nazwa pliku

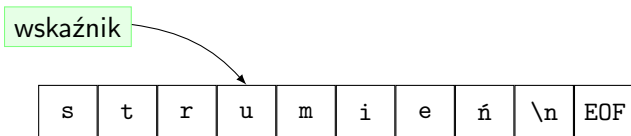
C:\Documents and Settings\user\moje dokumenty\plik.txt
/home/user/doc/plik.txt

http://en.wikipedia.org/wiki/List_of_file_systems

- nazwa pliku
- rozmiar
- data utworzenia, data modyfikacji, dostępu
- uprawnienia: właściciel, grupy i ich prawa (odczyt/zapis)
- DOS/Windows: ukryty, tylko do odczytu, archiwalny, systemowy, zaszyfrowany (NTFS), skompresowany (NTFS)
- rodzaje plików: plik zwykły, katalog, dowiązanie, FIFO, blokowe, ...
- położenie pliku, wskaźnik do miejsca na nośniku (często tablica wskaźników FAT), Linux i-węzły

- W C nie ma typu plikowego
- Pascal: pliki tekstowe, amorficzne (niezredagowane), pliki zdefiniowane (jednorodne)
- **Deskryptor pliku:** liczba całkowita, niskopoziomowa reprezentacja
Potrzebne funkcje systemowe, np. biblioteka `unistd.h`
- **Strumień:** ogólny sposób komunikacji między plikami, urządzeniami i procesami
- Wykorzystanie strumieni: dostęp do plików, komunikacja między procesami, komunikacja sieciowa, komunikacja z urządzeniami, łańcuchy jako strumień.
- Plik nagłówkowy `stdio.h`, obsługa wejścia i wyjścia (***Standard input output***)

- Strumień to zmienna typu FILE*
wskaźnik do pliku, „uchwyt” do pliku
- Struktura zawierająca informacje o pliku: nazwa, deskryptor pliku, rodzaj dostępu, znacznik pozycji, adres bufora, ...
- Strumień udostępnia dane jako sekwencję bajtów zakończoną wartością końca pliku EOF
- Sekwencyjny odczyt i zapis, automatyczne buforowanie
- Wskaźnik bieżącej pozycji - miejsce odczytu/zapisu
- Obsługa błędów i końca pliku



1. Deklaracja zmiennej typu FILE*

```
FILE* plik;
```

2. Uzyskanie dostępu do strumienia fopen()

```
plik = fopen("plik.txt", "w");
```

3. Odczyt, zapis lub zmiana pozycji w strumieniu, np.:

```
fprintf(plik, "Hello world!\n");  
fscanf(plik2, "%d", &x);
```

4. Zamknięcie strumienia fclose()

```
fclose(plik);
```

Nie zapomnij o obsłudze błędów.
Każda operacja na pliku może zakończyć się niepowodzeniem.

```
FILE *fopen(char *ścieżka, char *tryb);
```

- ścieżka do pliku:
 - względem bieżącego katalogu "dane.xml", "../plik.txt"
 - bezwzględna: "C:\\plik.txt", "/home/user/plik". Nie najlepszy pomysł.
- tryb otwarcia:
 - "w" zapis (*write*), plik powstaje od początku
 - "r" odczyt (*read*) pliku istniejącego
 - "a" dopisanie (*append*) na końcu pliku istniejącego
- tryb binarny: "rb", "wb", "ab". W systemie Linux nie ma różnicy pomiędzy trybem tekstowym i binarnym.

```
FILE *odczyt, *zapis;  
odczyt = fopen("plik1.txt", "r");  
zapis = fopen("plik2.txt", "w");
```

- W przypadku niepowodzenia `fopen()` zwraca wartość `NULL`
- Wówczas nie ma możliwości wykonania jakiejkolwiek operacji na strumieniu
- Zawsze sprawdzaj czy udało się uzyskać dostęp do strumienia

```
FILE *plik;  
plik=fopen("plik.txt","r");  
if ( plik != NULL ) { /* ... */ }
```

- Możliwe przyczyny: zła nazwa pliku, zła ścieżka, brak nośnika, uszkodzenie nośnika, brak uprawnień do odczytu lub zapisu, za duża liczba otworzonych strumieni, itp...

FORMATOWANY ZAPIS I ODCZYT

```
int fprintf(FILE *plik, char *format, ...);  
int fscanf(FILE *plik, char *format, ...);
```

ZAPIS I ODCZYT POJEDYNCZEGO BAJTU (ZNAKU)

```
int fgetc(FILE *plik);  
int fputc(int c, FILE *plik);
```

ODCZYT ŁAŃCUCHA

```
char *fgets(char *tablica, int rozmiar, FILE *strumien);
```

- Wartość zwracana może informować o błędach lub końcu pliku
- Odczyt i zapis strumieni binarnych: `fread`, `fwrite`
- Więcej informacji w dokumentacji biblioteki `stdio.h`

TYPOWY SCHEMAT: ZAPIS DO PLIKU

```
1  #include<stdio.h>
2
3  int main()
4  {
5      FILE *plik = NULL;
6      float pi = 3.1415;
7
8      plik = fopen( "plik.txt", "w" );
9      if(plik != NULL )
10     {
11         /* Tutaj operacje na pliku */
12         fprintf(plik,"Witaj swiecie!\nPI=%f\n", pi);
13         fclose( plik );
14     }
15     else
16     {
17         /* Obsluga bledu otwarcia pliku */
18         printf("Blad otwarcia pliku %s\n", "plik.txt" );
19     }
20     return 0;
21 }
```


KONIEC STRUMIENIA

```
int feof(FILE* plik);
```

Wartość niezerowa gdy wystąpił koniec pliku.

Funkcje odczytu: `fscanf()`, `fgetc()`, `fgets()` zwracają EOF

INNE BŁĘDY OPERACJI WEJŚCIA-WYJŚCIA

```
int ferror(FILE *stream);
```

Wartość niezerowa gdy wystąpił błąd.

TYPOWY SCHEMAT: ODCZYT ZNAKÓW Z PLIKU

```
1  #include<stdio.h>
2
3  int main()
4  {
5      FILE *plik = NULL;
6      int znak;
7
8      plik = fopen( "plik.txt", "r" );
9      if( plik == NULL )
10     {
11         perror("Wystapil blad");
12         return 1;
13     }
14
15     while( feof(plik) == 0 )
16     {
17         znak = fgetc(plik);
18         if (znak != EOF) printf("%c\n", znak);
19     }
20     fclose( plik );
21
22     return 0;
23 }
```

STANDARDOWE WEJŚCIE I WYJŚCIE PROGRAMU

```
cat < plik1.txt > plik2.txt
```

< przekierowanie strumienia wejściowego

> przekierowanie strumienia wyjściowego

POTOKI

```
ls | wc
```

- Strumień to podstawowy sposób komunikacji między procesami.
- Unix/Linux: zbiór małych narzędzi o wielkich możliwościach

Strumienie FILE* dostępne w `stdio.h`

0 `stdin`

standardowe wejście, domyślnie klawiatura
`getchar()`, `scanf()`, ...

1 `stdout`

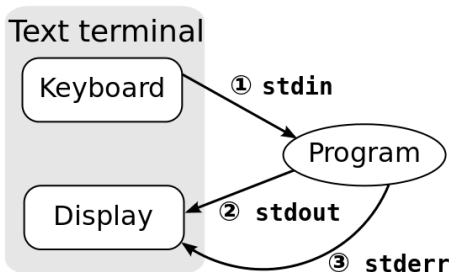
standardowe wyjście, domyślnie ekran
`putchar()`, `printf()`, ...

2 `stderr`

standardowe wejście diagnostyczne, domyślnie ekran
`perror()`, ...

```
z=getchar();  
putchar(z);  
printf("x=%d\n",x);  
scanf("%f",&y);  
gets(tab);
```

```
z=fgetc(stdin);  
fputc(z, stdout);  
fprintf(stdout, "x=%d\n",x);  
fscanf(stdin, "%f",&y);  
fgets(tab, n, stdin);
```



- DOS/Windows: konwersja plików tekstowych
Nowy wiersz: `\r\n` $\iff$ `\n`
- Niepoprawny format danych dla `scanf()`, `fscanf()`
Wartość zwracana to ilość wczytanych elementów.

```
if( scanf("%d", &x) > 0 ) { /* OK */ }
```

- Ostrożnie z mieszaniem odczytu formatowanego z nieformatowanym

```
scanf("%d", &x);  
getchar();      /* wczyta co zostawił scanf() */
```

- Jednoczesny zapis i odczyt może wymagać dodatkowych zabiegów (np. czyszczenie bufora)
- W Unix/Linux wielkość liter w ścieżkach ma znaczenie
`plik.txt` `PLIK.TXT` `Plik.txt` `Plik.TXT`
- W razie niepewności zajrzyj do dokumentacji

- 🌐 Wikipedia:  Plik,  List of file signatures,  File format.
 System plików,  List of file systems,  Comparison of file systems,  Standardowe strumienie
- 🌐 The GNU C Library:  Input/Output Overview,
 Input/Output on Streams