Programowanie Proceduralne

Marek Grochowski

Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK

https://www.is.umk.pl/~grochu/pp
https://github.com/IS-UMK/pp_wyklad
grochu@is.umk.pl

Ostatnia aktualizacja: 2 lutego 2026

Wstęp do programowania komputerów

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.

pl.wikipedia.org

Przegląd zagadnień

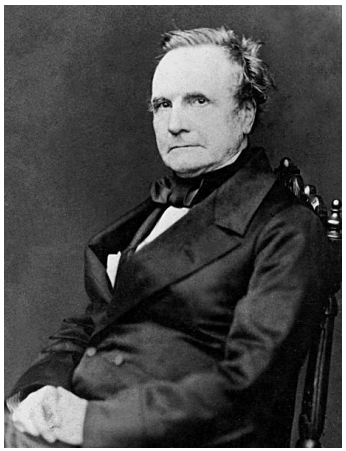
- Podstawy programowania w oparciu o język C
- *Case study* - przykłady programów, demonstracje
  Problem $\rightarrow$ Algorytm $\rightarrow$ Program $\rightarrow$ Rozwiązanie
- Paradygmat programowania proceduralnego,
  modularna struktura, algorytmy w postaci uniwersalnych funkcji
- Reprezentacja danych w komputerze:
  typy proste, złożone, wskaźniki, struktury dynamiczne, ...
- Elementy inżynierii oprogramowania: projektowanie, implementacja, wykrywanie błędów, testowanie, narzędzia
- Laboratorium: język C, rekomendowane środowisko Visual Studio C++
- Zaliczenie wykładu: pozytywna ocena z laboratorium + TEST zaliczeniowy

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.

Charles Babbage (1791–1871)



1822 Projekt maszyny różnicowej.

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

Augusta Ada Lovelace (1815-1852)



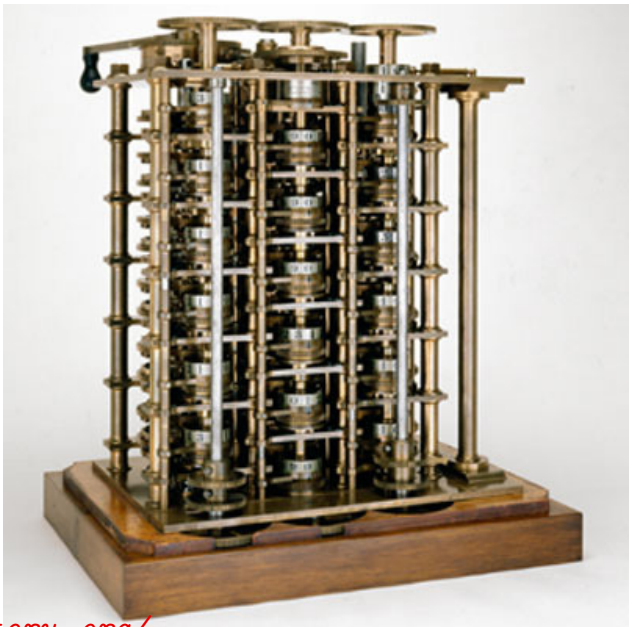
1842 *Note G*, algorytm wyznaczania liczb Bernoullego. **Pierwszy program komputerowy.**

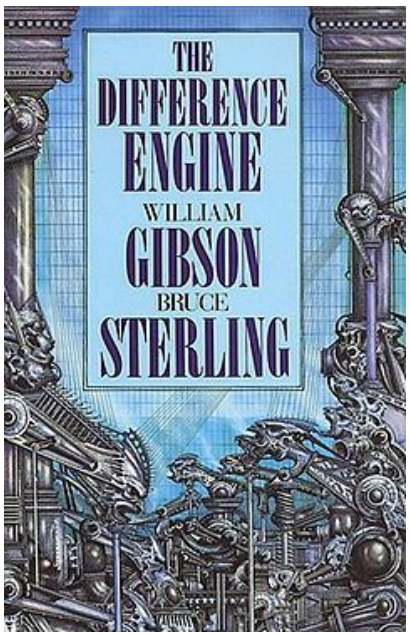
1979 Język ADA.

1849 Difference Engine No. 2, dokładność 31 cyfr, wydruk na wyjściu

2002 Realizacja projektu rekonstrukcji maszyny różnicowej, ☞Computer History Museum.

2011 Początek (10 letniego) projektu rekonstrukcji maszyny analitycznej, ☞Plan 28.





http://www.computerhistory.org/

Generacje komputerów

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 ☞Przekaźnikowy **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

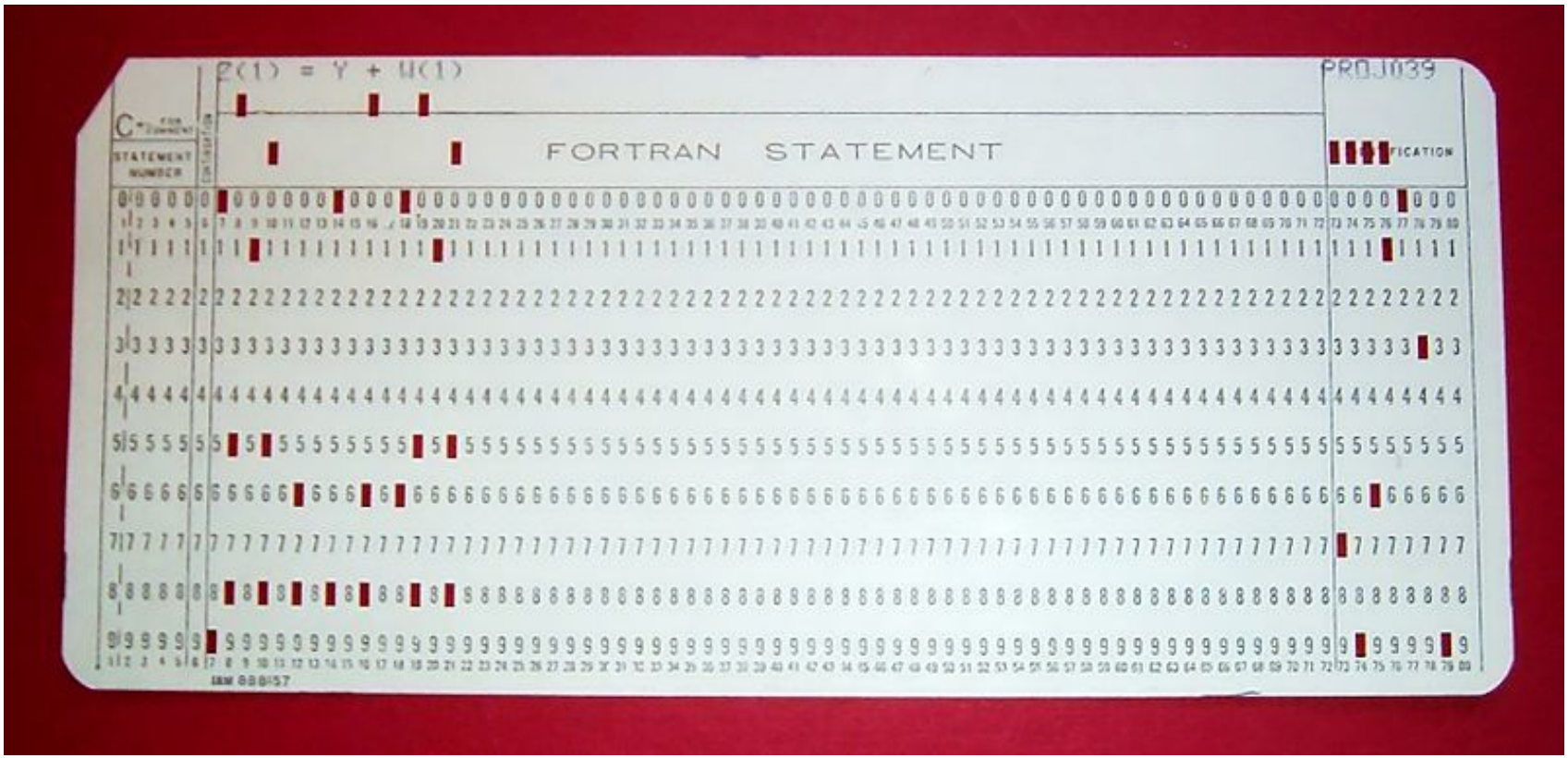
- Zastosowanie do obliczeń numerycznych (łamanie szyfrów, balistyka)
- Wejście: karty dziurkowane, taśmy papierowe
- Wyjście: wydruk, dalekopis, lampy
- Pamięć: dane przechowywane na dyskach magnetycznych, rtęciowe linie opóźniające
- Program: głównie język maszynowy

1947-50 pierwsze assemblery (EDSAC)
1952 pierwszy kompilator A-0 (UNIVAC I)
1954-57 Język **Fortran** (John Backus, IBM)
pierwszy język wysokiego poziomu,
który znalazł powszechne zastosowanie



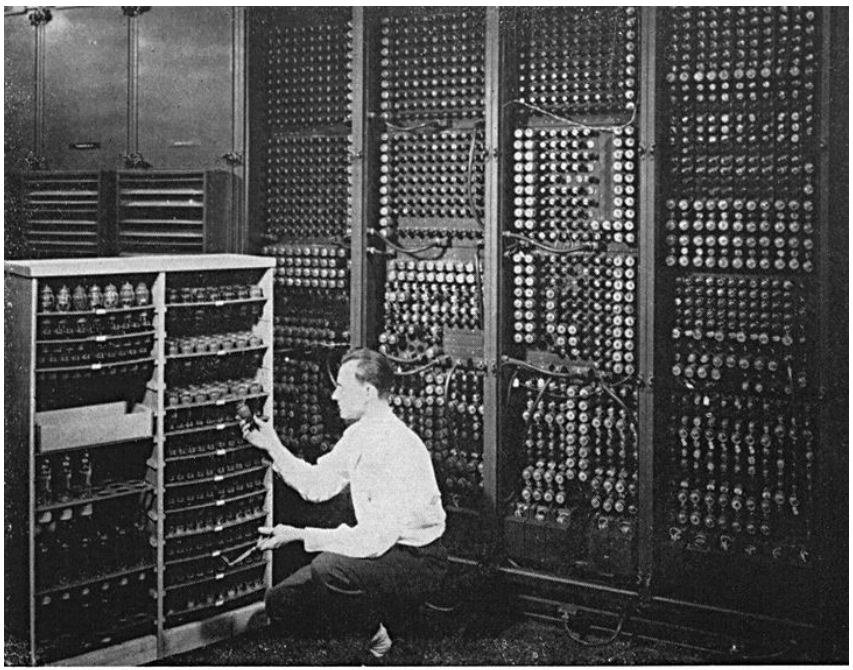
Karta perforowana

80 kolumn, 10 wierszy numerycznych + 2 wiersze strefowe (IBM, 1928)



ENIAC (1946)

Elektroniczne urządzenie numeryczne całkujące i liczące



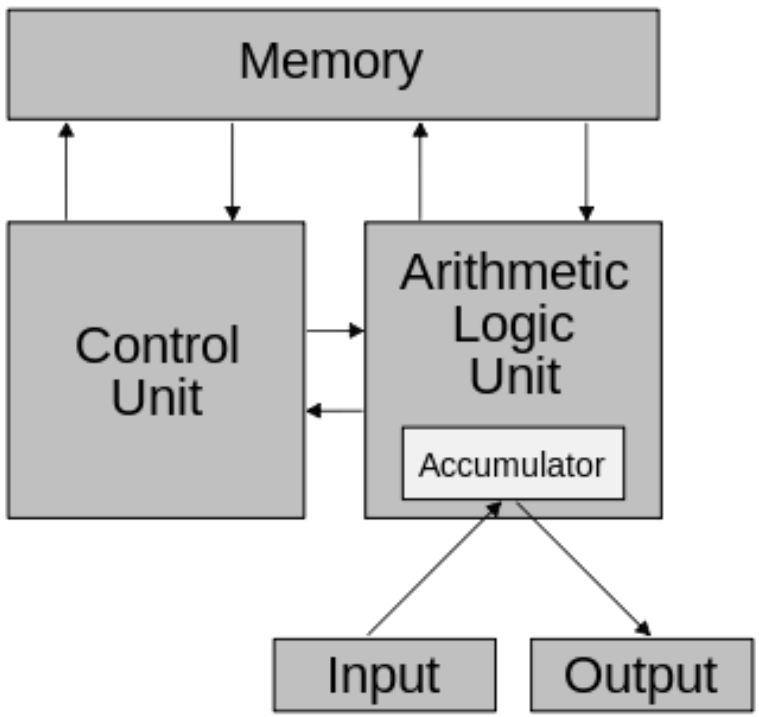
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

18 000 lamp,
30 ton, 170 m^2, moc 160kW,
5000 operacji dodawania / sec.
system dziesiętny,
ręczne programowanie przez ustawianie 6K przełączników 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).





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.

1970 Minikomputer K-202
(potencjalnie) wydajniejszy od IBM 5150 PC (1981 r.)
Opracowany i skonstruowany przez inż. Jacka Karpińskiego.



16 bitów
adresowanie stronicowe do 8MB
(konkurencja max. 64kB)
modularność
wielodostępowość

1 mln. operacji/s.

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

Rozwój języków programowania

- 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

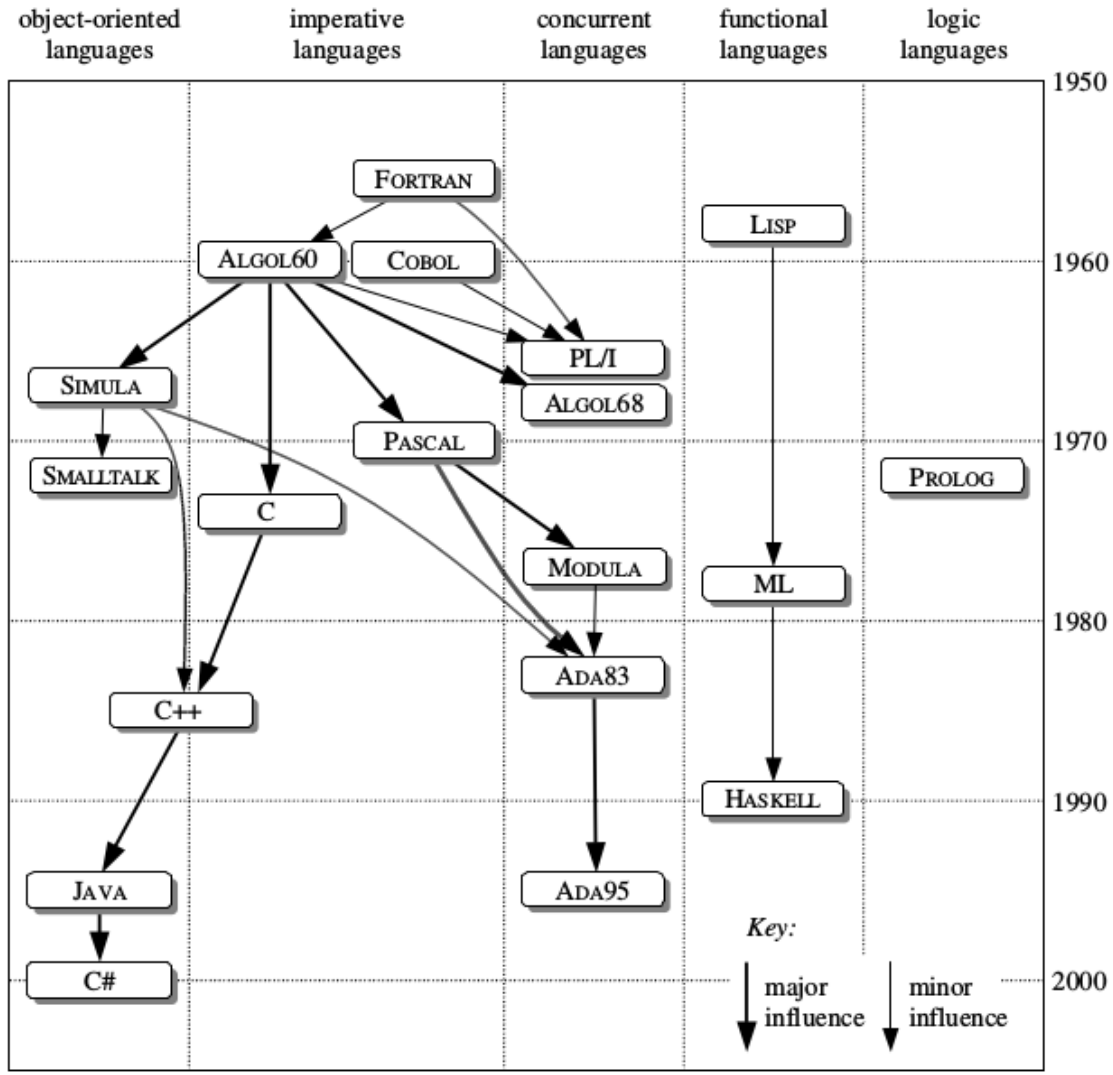
  Algol (58/60), PL/1 (1964).
- lata 70-te, początek pojedynku: Pascal vs. C

  Zalążki obiektowości: Smalltalk (1972)
- lata 80-te, Dominują: C, Pascal, Basic

  Powstają: C++ (1980), Matlab (1984)
- lata 90-te, era internetu, programowanie obiektowe

  Java (1996), Python (1991), PHP (1995), JS (1995), .NET (C#, 2001)

Źródło: ☞ History and Evolution of Programming Languages



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

Głowne typy języków programowania

- **imperatywne**: program określa kolejność instrukcji do wykonania, zmieniając stan programu krok po kroku. Kod określa jak coś zrobić (C, Pascal)
- **obiektowe**: łączą dane i działania (metody) w obiektach. Ułatwiają modelowanie złożonych systemów przez enkapsulację i dziedziczenie (Java, C++, C#)
- **równoległe** (*concurrent*): umożliwiają wyrażanie i zarządzanie współbieżnością, czyli wykonywaniem wielu procesów lub wątków jednocześnie (Go, Erlang)
- **funkcyjne**: opierają się na funkcjach matematycznych, unikają zmiany stanu. Programy opisują co ma być zrobione, a nie jak (Haskell, Lisp)
- **logiczne**: bazują na logice formalnej, definiując fakty i reguły, a język sam dochodzi do wniosków poprzez dedukcję (Prolog).

- 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
  ☞ Porównanie języków

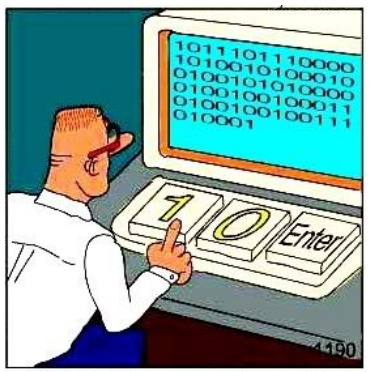


Pieter Bruegel (starszy), 1563

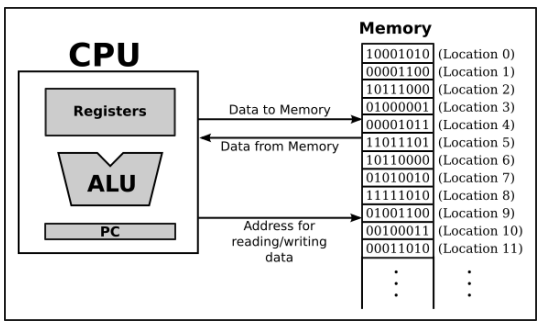
Kod maszynowy

ciąg instrukcji w postaci binarnej wykonywanych bezpośrednio przez procesor

- rozkazy procesora i dane w postaci słów bitowych pobierane są z pamięci do rejestrów procesora
- procesor wykonuje rozkazy, zmieniając stan rejestrów i pamięci
- kod maszynowy nie jest przenośny, każdy procesor ma swój specyficzny zestaw instrukcji
- tworzenie programów bezpośrednio w kodzie maszynowym jest trudne



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 asemblera na kod maszynowy (assemblacja)
- **Deasembler** - program tłumaczący kod maszynowy na jezyk asemblera

Języki wysokiego poziomu

- nie są bezpośrednio wykonywane przez procesor, przez co pozwalają uniezależnić program od platformy sprzętowej i systemowej

```c
#include<stdio.h>

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

- muszą być skompilowane lub interpretowane
- przypominają jezyk naturalny, 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,
- jedna instrukcja w języku wysokiego poziomu może odpowiadać wielu instrukcjom w kodzie maszynowym

Kompilator

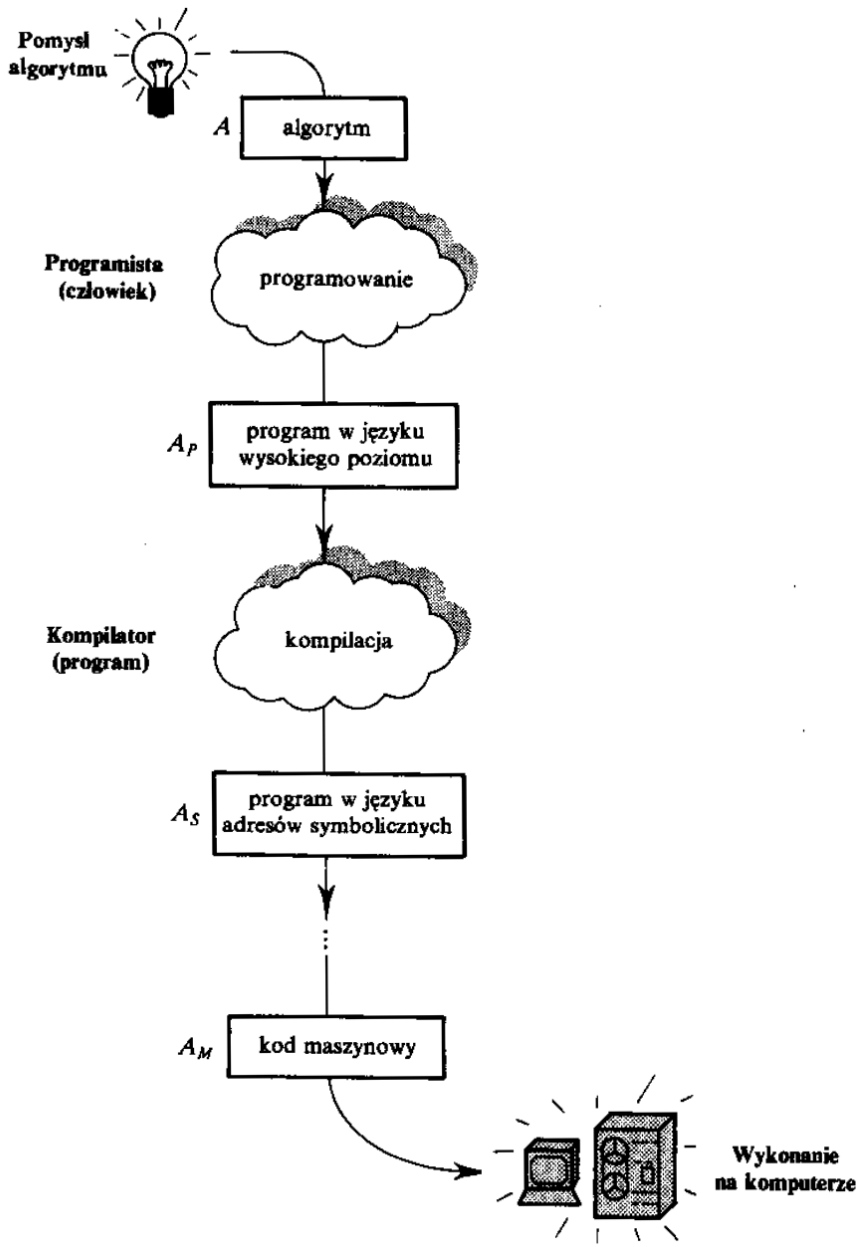
program tłumaczący kod napisany w jednym języku na równoważny kod w innym języku. Przykłady:

- kod źródłowy → kod maszynowy (C, C++, Pascal, Fortran)
- kod źródłowy → *byte code*, kod pośredni, interpretowany 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



D. Harel, [2]

Wersja minimalistyczna: *edytor tekstu* + *kompilator*

- **GCC** (GNU Compiler Collection)
  Linux, MacOS, Windows (Cygwin, MinGW)
  `cc hello.c -o hello`
- **Clang** (LLVM), Linux, MacOS, Windows (Cygwin, MinGW)
  `clang hello.c -o hello`
- **Microsoft Visual C++** (MSVC), Windows
  `cl hello.c /out:hello.exe`

IDE, Integrated Development Environment
kompilator, edytor, deasembler, debugger, ...

Windows	**Linux**	**Mac OS**
MS Visual Studio	KDevelop, Anjuta	Xcode
Visual Studio Code, CodeBlocks		

Algorytmy

DOKŁADNA NAUKA

WARZENIA PIWA.

WEDŁUG METODY ŁATWEY, STWIERDZONEY OSMIOLETNIEM DOSWIADCZENIEM.

DO WYNALAZKÓW NAYNOWSZYCH ZASTOSOWANA,

Z PRZYDANEM OPISANIEM APPARATU DO STUDZENIA, ZASTĘPUIĄCEGO ZWYCZAYNE KILSZTOKI, ZA POMOCĄ KTÓREGO PIWO WRZĄCE, W PRZECIĄGU IEDNEY MINUTY DO TEMPERATURY WODY STUDZIENNEY OCHŁODZONE BYDŹ MOŻE;

PRZEDSTAWIONA W SPOSOBIE PRAKTYCZNYM,

PRZEZ KAROLA WILHELMA SCHMIDT,

AUTORA ROZMAITYCH DZIEŁ TECHNOLOGICZNYCH.

PRZEŁOŻONA Z JĘZYKA NIEMIECKIEGO, DLA UŻYTKU ZIEMIAN POLSKICH.



WARSZAWA.

NAKŁAD I DRUK N. GLÜCKSBERGA,

KSIĘGARZA I TYPOGRAFA KRÓL: WARSZAW: UNIWERSYT.

1830.

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 wziąść należy 200 miarek zwanych *Kanne* wody, i te tak długo gotować, pokąd 1/3 części iéy nie ubędzie. Sypie się potém do téy wody siedm Szeflów slodu Pszennego i jeden Szefel drobnéy Fasoli; gdy odleie się do beczki dla wyrobienia, nie należy zapełniać iéy całkowicie, bo skoro fermentacyia nastąpi, kładzie się do beczki wewnętrznéy kory z Sośniny funtów trzy, pączków Brzozowych i kotków (pączków) Sosnowych po funcie jednym, trzy garści ziela Cardi-Benedicti, garść jedną lub dwie kwiatu zwanego *Sonnenthaubläthe*, 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 10 jay świeżo zniesionych i ta szpuntuie się. We dwa roki dopiero po zaszpuntowaniu ściąga się Piwo do picia.

Przepis

Warzenie piwa Brunświckiego

- 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 wziąść należy 200 miarek zwanych *Kanne* wody, i te tak długo gotować, pokąd 1/3 części iéy nie ubędzie. Sypie się potém do téy wody siedm Szeflów slodu Pszennego i jeden Szefel drobnéy Fasoli; gdy odleie się do beczki dla wyrobienia, nie należy zapełniać iéy całkowicie, bo skoro fermentacyia nastąpi, kładzie się do beczki wewnętrznéy kory z Sośniny funtów trzy, pączków Brzozowych i kotków (pączków) Sosnowych po funcie jednym, trzy garści ziela Cardi-Benedicti, garść jedną lub dwie kwiatu zwanego *Sonnenthaublüthe*, 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 10 jay świeżo zniesionych i ta szpuntuie się. We dwa roki dopiero po zaszpuntowaniu ściąga 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 rozwiązania pewnych zadań - **zadań algorytmicznych**.

Specyfikacja zadania algorytmicznego

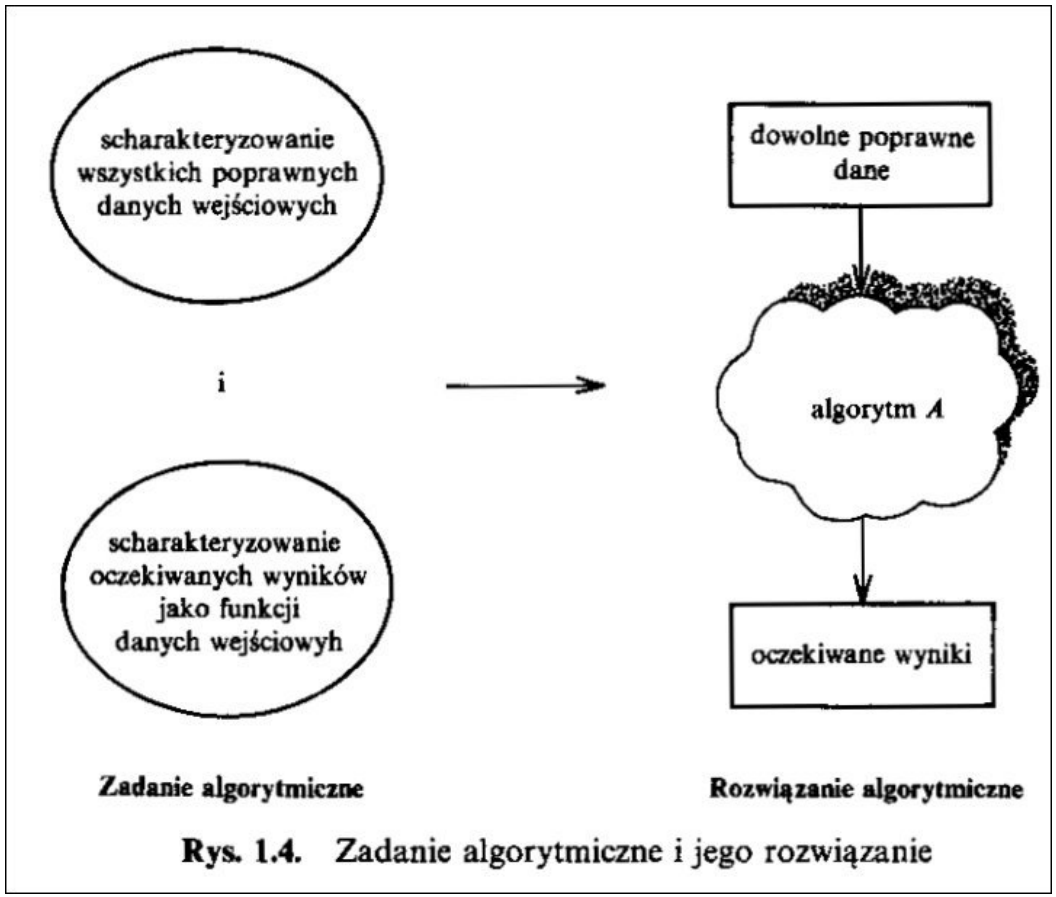
- warunki jakie muszą spełniać dane wejściowe
- określenie oczekiwanych wyników jako funkcji danych wejściowych

Specyfikacja może też definiować ograniczenia algorytmu, np. dotyczące liczby operacji.

Przykład specyfikacji: Algorytm Euklidesa

Dane wejściowe: dwie liczby naturalne a i b
Wynik: liczba naturalna $NWD(a, b)$

Rozwiązanie zadania algorytmicznego to skonstruowanie algorytmu, który dla każdych dopuszczalnych danych wejściowych wyznaczy wynik zgodny ze specyfikacją.



Rys. 1.4. Zadanie algorytmiczne i jego rozwiązanie

D. Harel, [2]

Cechy algorytmu

- **skończoność** - algorytm musi zakończyć działanie po skończonej liczbie kroków
- **poprawność** - algorytm powinien dawać prawidłowy wynik dla wszystkich dopuszczalnych danych wejściowych
- **uniwersalność** - algorytm powinien działać dla wszystkich przypadków spełniających warunki zadania, a nie tylko dla konkretnego przykładu
- **efektywność** - rozwiązanie powinno być możliwe do zrealizowania w rozsądnym czasie i z dostępnymi zasobami
- **określoność** - każdy krok musi być precyzyjnie opisany, bez możliwości różnej interpretacji.

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 zawiłości algorytmów".

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

Problem:

znalezienie największej liczby naturalnej, która dzieli liczby naturalne a i b bez reszty

Pomysł:

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

Specyfikacja

Dane wejściowe: liczby naturalne a i b
Wynik: liczba naturalne 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 $-$, $=$ (równość) oraz $>$.



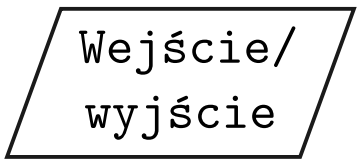
Blok graniczny: start, stop, przerwanie, opóźnienie.



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



Blok decyzyjny, rozgałęzienie.



Wprowadzanie danych i wyprowadzenia wyników.



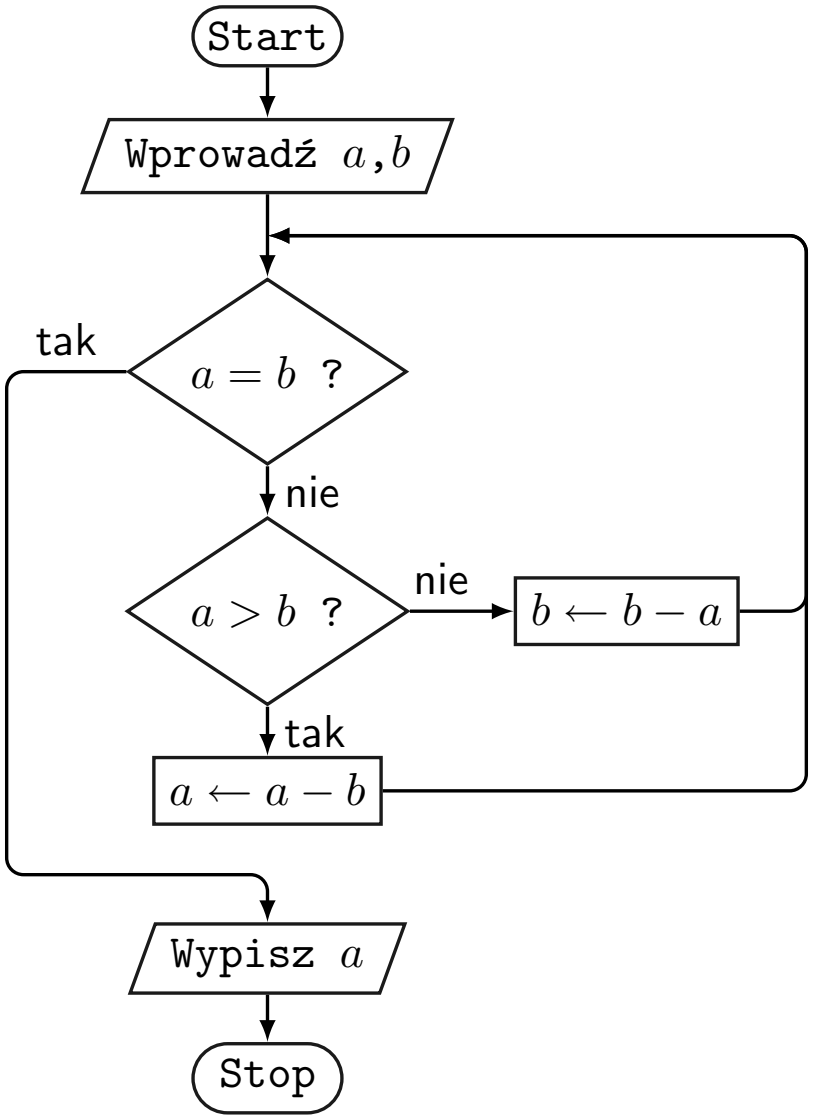
Połączenie z innym fragmentem diagramu.



Wywołanie podprogramu.

Przykład: Schemat blokowy

Algorytm Euklidesa z odejmowaniem



Algorithm 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
- uproszczona składnia na rzecz prostoty i czytelności
- formuły matematyczne, język naturalny, podprogramy
- nie zawiera szczegółów implementacji
  „Dla ludzi, nie dla maszyn”.

```c
/*  Algorytm Euklidesa. */

#include <stdio.h>

int main()
{
    int a, b;

    printf("Podaj dwie liczby calkowite: ");
    scanf("%d %d", &a, &b);

    while (a != b)
        if (a > b) a = a - b;
        else       b = b - a;

    printf("NWD = %d\n", a);

    return 0;
}
```

☞ euclid1.c

```
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.
```

☞ euclid1.pas

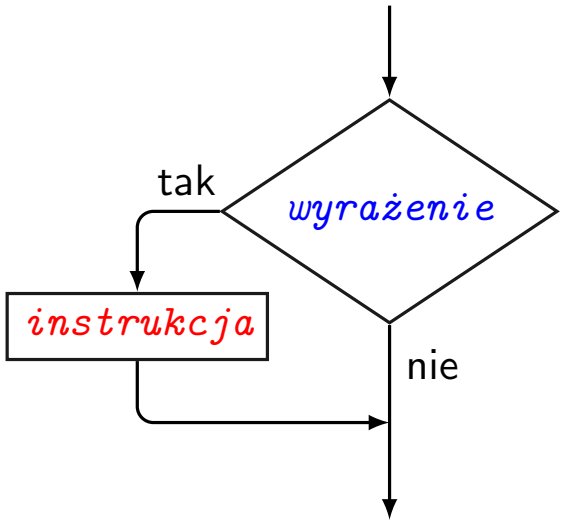
Program w języku Fortran 77

```
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

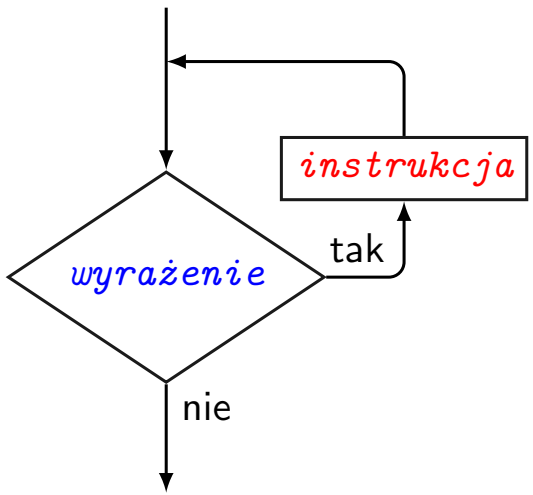
Warunek if (jeżeli)

```
if ( wyrażenie )
    instrukcja
```



Pętla while (dopóki)

```
while ( wyrażenie )
    instrukcja
```



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

Algorytm Euklidesa z operacją modulo

Algorithm Algorytm Euklidesa

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

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

```c
#include <stdio.h>

int main()
{
    int a, b, c;

    printf("Podaj dwie liczby calkowite: ");
    scanf("%d %d", &a, &b);
    printf("NWD(%d,%d) = ", a, b);

    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }
    printf("%d\n", a);

    return 0;
}
```

☞ euclid2.c

Esencja programowania proceduralnego

Podprogram - funkcja lub procedura, wydzielona część programu

- **wielokrotne użycie** - moga być wywołane w różnych miejscach programu
- **ekonomiczność** - ujednolica powtarzające się bloki programu, mniej kodowania,
- **czytelność** - nawet przy złożonych i obszernych algorytmach
- **modularny kod** - podział kodu na mniejsze fragmenty, ułatwia jego rozwój i utrzymanie
- podprogram staje się nową instrukcją elementarną

  w języku C brak wbudowanych funkcji oprócz `main()`
- uproszczenie problemu poprzez rozbicie na mniejsze pod-problemy
  - programowanie zstępujące (*top-down*)
  - programowanie wstępujące (*bottom-up*)
- podprogram może uruchamiać sam siebie - **rekurencja**

```c
int nwd(int a, int b)
{
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

☞ euclid3.c

```c
int nwd(int a, int b)
{
    if (b == 0) return  a;
    return nwd(b, a % b );
}
```

☞ euclid3rekurencja.c

Algorithm Sortowanie przez wybieranie (selection sort)

Dane wejściowe: ciąg n liczb $A = \{a_1, a_2, \ldots, a_n\}$
Wynik: uporządkowany ciąg $a_1 \leqslant a_2 \leqslant, \ldots, \leqslant a_n$

1: $i \leftarrow 1$
2: **dopóki** $i < n$ **wykonuj**
3: $\quad k \leftarrow \texttt{minind}(\{a_i, a_{i+1}, \ldots, a_n\})$ ▷ Podprogram
4: $\quad a_i \longleftrightarrow a_k$
5: $\quad i \leftarrow i + 1$

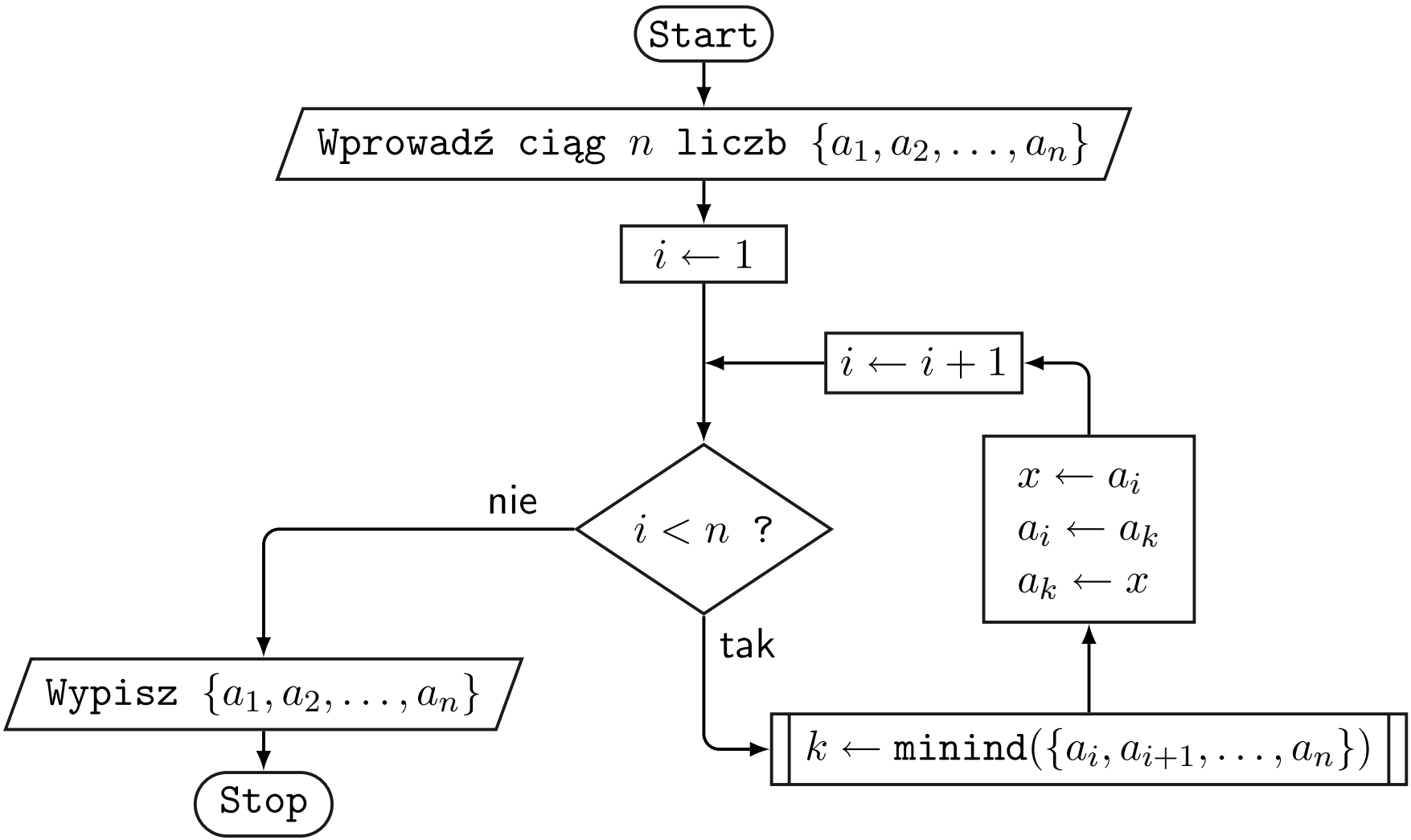
Procedura `minind` wyszukuje położenie najmniejszego elementu

Specyfikacja `minind`
Dane wejściowe: ciąg m liczb $\{x_1, x_2, \ldots, x_m\}$
Wynik: indeks najmniejszej liczby w ciągu

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?
- posortowanie zbioru N elementów?
- problem komiwojażera: znalezienie najkrótszej drogi łączącej miasta?

Problem komiwojażera (TSP)

Travelling salesman problem



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

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

[1] M.M. Sysło, „Algorytmy"

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	niewyobrażalnie duża
N^N	10 miliardów (11 cyfr)	liczba 85-cyfrowa	liczba 201-cyfrowa	liczba 744-cyfrowa	niewyobrażalnie 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

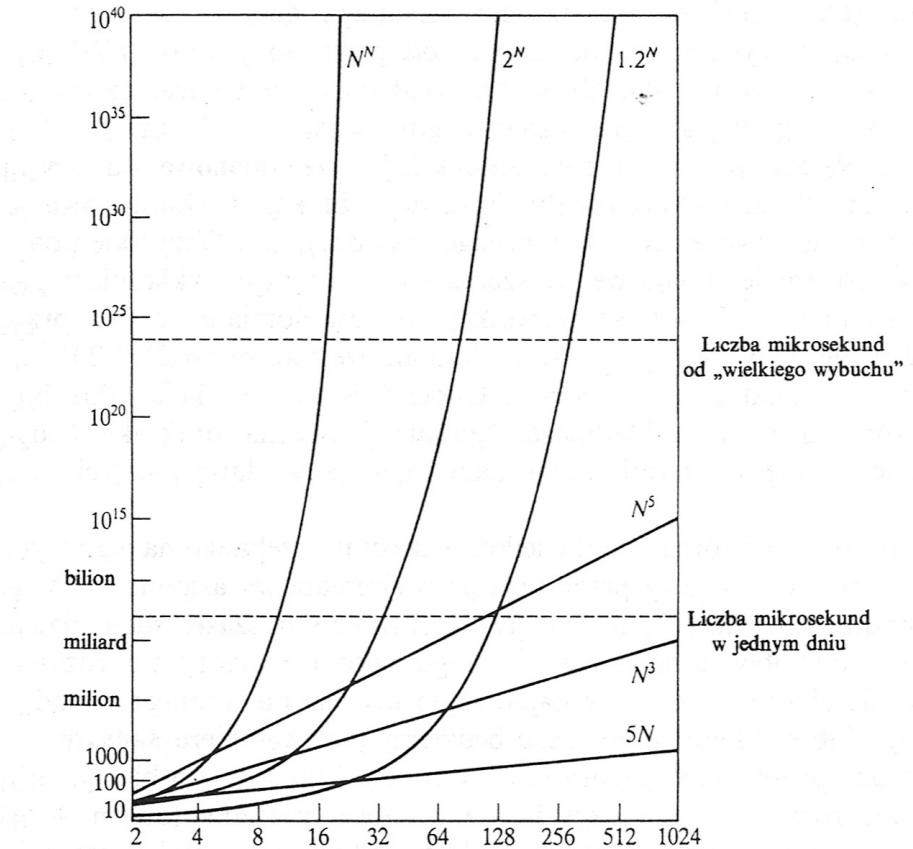
[2] D. Harel, Rzecz o istocie informatyki. Algorytmika.

- Rozsądne rozwiązania, wykonywalne w czasie wielomianowym

  $\log N, \quad N, \quad N \log N, \quad N^7 + N^3 + 2$

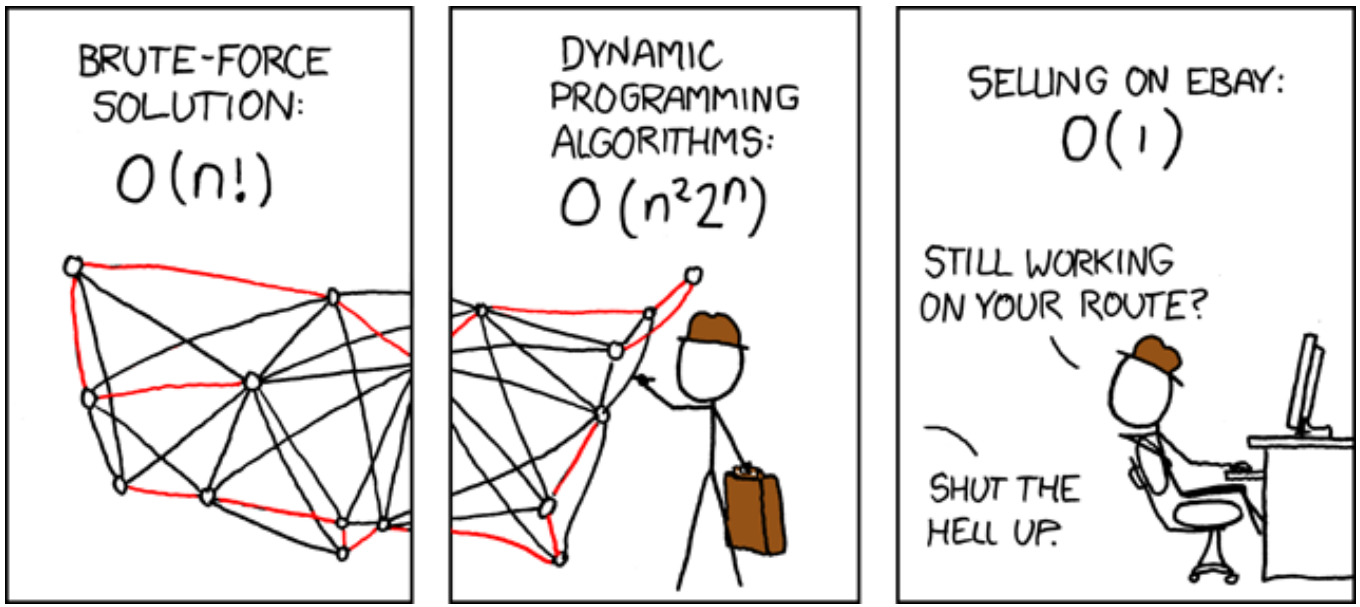
- Nierozsądne, niepraktyczne, czas ponad-wielomianowy

  $2^N, \quad 1.001^N + N^7, \quad N^N, \quad N!$



Rys. 7.4. Tempo wzrostu pewnych funkcji

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



http://xkcd.com

Reprezentacja danych kształtuje algorytm!

Dane wejściowe ⇒ Algorytm ⇒ Dane wyjściowe

- zmienne: liczby, znaki, adresy `a = 3`
- tablice, wektory, listy, `a[3]=3`
- macierze, 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: miara ilości pamięci potrzebnej do wykonania algorytmu

Trade-off: kompromis między złożonością obliczeniową a pamięciową. Często algorytmy pamięciożerne, mogą być szybsze, gdy algorytmy oszczędzające pamięć mogą wymagać więcej obliczeń.

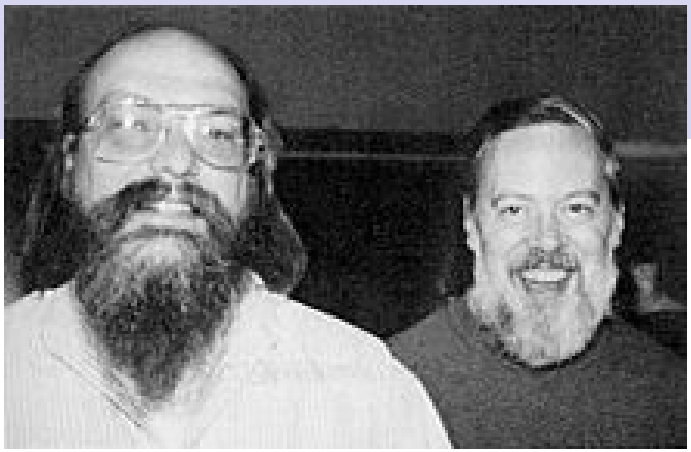
Od problemu do rozwiązania

- 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.



Znowu trochę historii

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

2024 Standard C23

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

int main()
{
    int i;
    float x;

    i = 10 * PI;
    x = 1.0;

    return 0;
}
```

Dyrektywy preprocesora

Funkcja główna

Deklaracje zmiennych

Instrukcje programu

Najkrótszy program w C

```
main(){}
```

Witaj świecie

```
#include<stdio.h>

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

- mały język ale duże możliwości, ważna rola bibliotek
- **pliki źródłowe** `*.c` zawierają instrukcje programu
- **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**

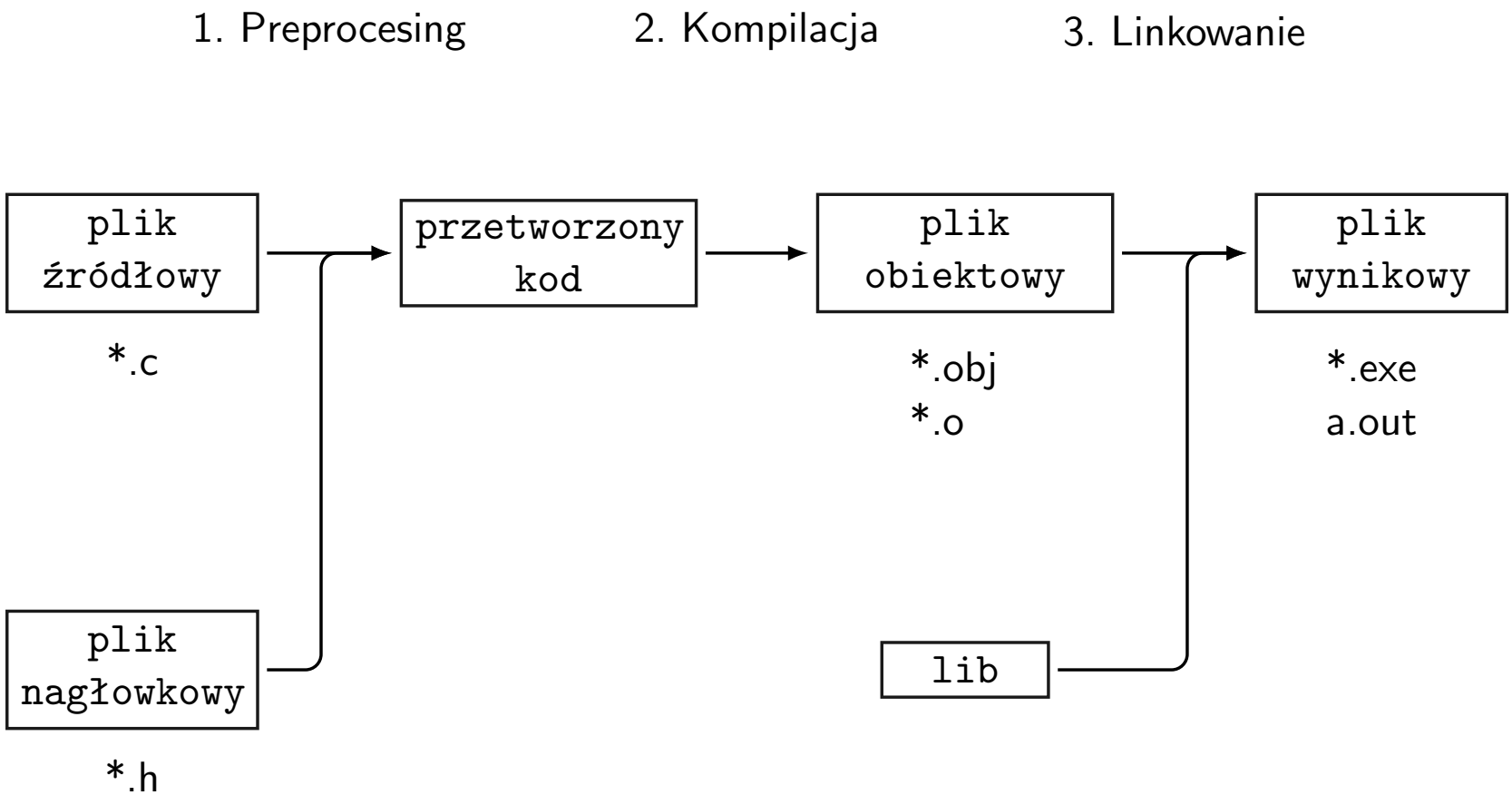
Dyrektywy to instrukcje zaczynające się od znaku #.
Nie są słowami języka C, wykonywane są przed właściwą kompilacją.

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

```
#include <stdio.h>
#include "../plik.h"
```

- #define *nazwa wartość*
  definiuje stałe, tworzy „przezwiska” do słów i wyrażeń (makra)

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


1. Preprocesing
2. Kompilacja
3. Linkowanie
plik źródłowy
*.c
przetworzony kod
plik obiektowy
*.obj
*.o
plik wynikowy
*.exe
a.out
plik nagłowkowy
*.h
lib

Preprocesor wykonuje instrukcje zaczynające się znakiem # (dyrektywy preprocesora). Przygotowuje pliki do kompilacji.

pliki źródłowe (`*.c, *.h`) ⇒ przetworzone pliki

```
#include<stdio.h>
#define PI 3.14
```

Kompilacja tłumaczy instrukcje C na kod maszynowy

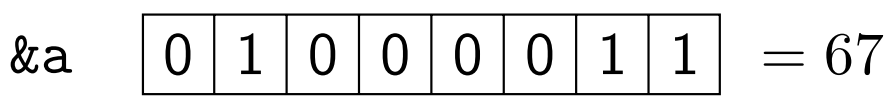
przetworzone pliki ⇒ pliki obiektowe (`*.obj, *.o`)

Konsolidacja (linkowanie) łączy pliki obiektowe w aplikację

pliki obiektowe + biblioteki ⇒ 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 są reprezentowane poprawnie w pewnym przedziale
- **rozdzielczość** - liczby rzeczywiste są reprezentowane z pewną dokładnością

```
char a='C';
```



typ	reprezentacja	zakres wartości
`char`	znaki (*character*), kod ASCII	$[-127, 128]$
`int`	liczby całkowite (*integer*)	$[-2^{31}, 2^{31} - 1]$
`float`	liczby rzeczywiste (*floating point*) reprezentacja zmiennoprzecinkowa najmniejsza dodatnia wartość 1.17×10^{-38}	$[-3.4 \times 10^{38},$ $+3.4 \times 10^{38}]$
logiczny	brak typu logicznego wartość całkowita 0 to fałsz a wartość różna od 0 to prawda	
`void`	typ pusty, brak typu	

typ `bool` wprowadzony w standardzie C23

Deklaracja

występuje na początku bloku instrukcji

typ identyfikator;
typ identyfikator1, identyfikator2, identyfikator3;

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

Przykład:

```
int i;
float x, y;
char c, d, e;
```

- 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 nie mogą wystąpić w roli identyfikatorów:
  - słowa kluczowe języka (np. `while`, `int`, `return`)
  - identyfikatory zadeklarowane wcześniej (zmienne, funkcje)

Przykład:

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

```
int 4abc;
float średnica;
char pewien znak;
int wazna-zmienna;
float return;
```

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`

```
int x = 1;

int main()
{
    int x = 2;

    {
        int x = 3;


    }

}
```

zmienna **globalna**,
dostępna w zakresie całego pliku

zmienna **lokalna**,
dostępna w zakresie funkcji `main`,
przysłania zmienną globalną `x`

zmienna lokalna
dostępna w zakresie bloku instrukcji {}
przysłania zmienną `x` z funkcji `main`

Zmienne deklaruje się na samym początku bloku (czyli przed pierwszą instrukcją).

Od C99 można deklarować w dowolnym miejscu przed użyciem zmiennej

Operatory arytmetyczne

`*` `/` `%` `+` `-`

Operatory relacji

`<` `>` `<=` `>=` `==` `!=`

Operatory logiczne

`!` `&&` `||`

Operator przypisania

`=`

```
zmienna = wyrażenie;
```

```
int i, j, k;
float x = 1.5;
float y = 1.5e-5;
char znak = 'A';

i = i + 3;
3 = x;
x + y = x - y;
znak = 65;
```

deklaracje zmiennych

inicjalizacja

brak inicjalizacji

zła kolejność

błąd

Zmienne lokalne nie są domyślnie inicjowane - jeśli nie przypiszesz wartości to zmienna będzie posiadała pewną (losową) wartość. Dlatego warto zawsze inicjować wartości zmiennych.

	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`	$x \mod 2$

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

```
1 / 2 → 0
1 / 2.0 → 0.5
1.0 / 2 → 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;      /* zle */
z = x / (2 * y);
```

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

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

W razie wątpliwości użyj nawiasów okrągłych.

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

`stdio.h` - **St**an**d**ard **i**nput/**o**utput, biblioteka obsługująca komunikację ze standardowym wejściem i wyjściem aplikacji.

`printf()`

formatowane wyjście (wyświetlanie w terminalu)

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

`scanf()`

formatowane wejście (odczyt z klawiatury)

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

Format

specyfikator	znaczenie	przykład	wynik
%f	zmiennoprzecinkowa	`printf("%f",3.14);`	3.140000
%.2f	2 miejsca po przecinku	`printf("%.2f",3.14);`	3.14
%d	dziesiętna	`printf("%d",75);`	75
%c	znak	`printf("%c",75);`	K
%x	szesnastkowy	`printf("%x",75);`	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%cr\te\ta\no\ts\tb\ng\tt\ta\nr\t ", 10);
printf("\tw\n%x\t \tn\nm\t \te\no\nw\na\nn\ni\ne\n", 10);
```

☞ printf–specjalne.c

- specyfikator formatu powinien pasować do typu zmiennej

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

problem !!!

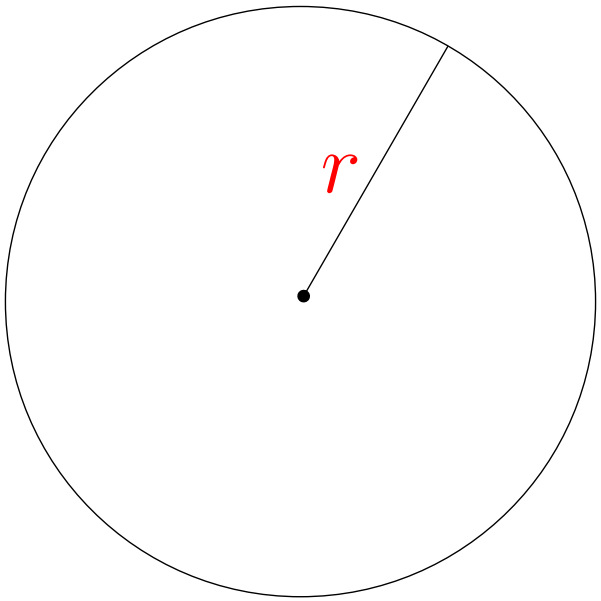
- drugi argument funkcji `scanf` to adres zmiennej `&a`
- formaty `%f` i `%d` pomijają początkowe białe znaki aż do napotkania wartości liczbowej
- znak niepasujący do formatu przerywa wczytywanie i pozostaje w strumieniu wejściowym

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

wczyta 2 liczby oddzielone białymi znakami

czyta pojedynczy znak

Problem: wyznacz pole i obwód koła o promieniu r.



$$P_\circ = \pi r^2 \qquad O_\circ = 2\pi r$$

Przykład: pole i obwód koła

```c
#include <stdio.h>
#define PI 3.14159

int main()
{
    float r, pole, obw;

    printf("Podaj promien kola\nr= ");
    scanf("%f",  &r);

    pole = PI * r * r;
    obw  = 2 * PI * r;

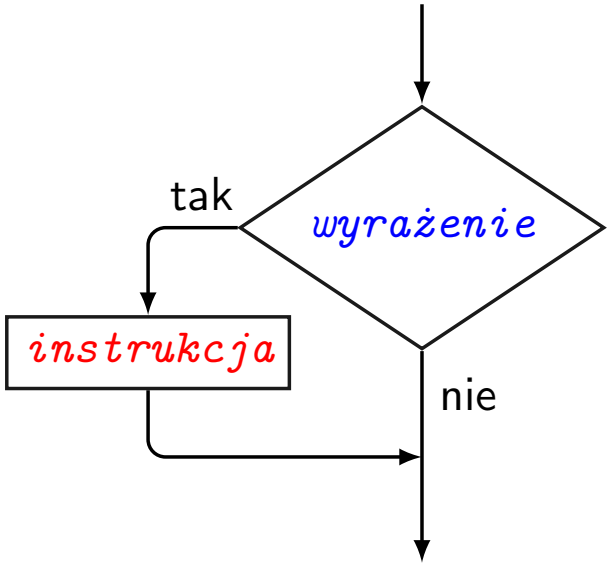
    printf("Pole kola o promieniu %f wynosi %f\n", r, pole);
    printf("Obwod kola o promieniu %f wynosi %f\n", r, obw);

    return 0;
}
```

☞ kolo.c

Składnia instrukcji `if`

```
if ( wyrażenie )
    instrukcja
```



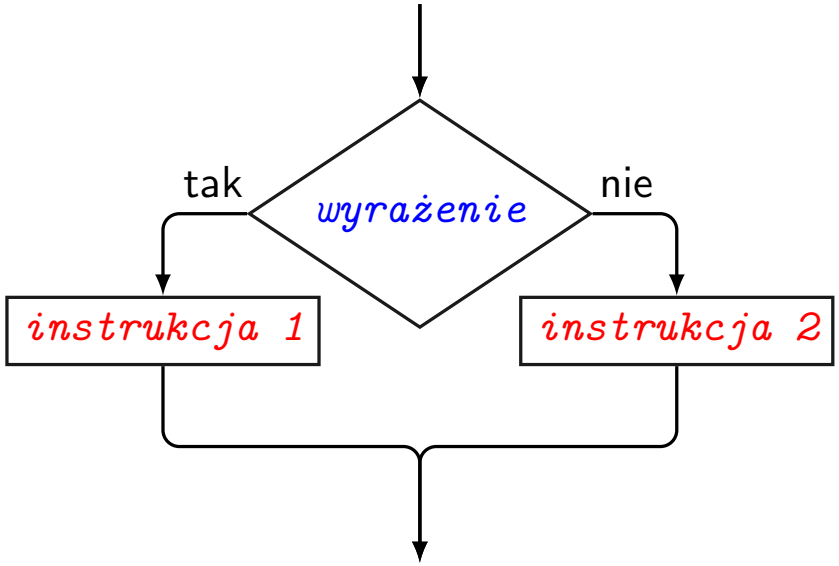
Przykład:

```
if( x > 0 ) printf("liczba dodatnia");

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

Składnia instrukcji `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`	$x \leqslant y$
>=	większy lub równy	`x >= y`	$x \geqslant y$
==	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, relacje, &&, ||
W razie wątpliwości użyj nawiasów okrągłych.

Przykład: Równanie z jedną niewiadomą

Problem: znajdź miejsce zerowe funkcji liniowej

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

Algorithm 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: $\quad x_0 \leftarrow -\frac{b}{a}$
3: $\quad$ **wypisz:** x_0
4: **w przeciwnym wypadku**
5: $\quad$ **wypisz:** `Brak rozwiązania`

```c
#include <stdio.h>

int main()
{
    float a, b, x0;

    printf("Podaj wspolczynniki rownania\n");
    printf("a = "); scanf("%f", &a);
    printf("b = "); scanf("%f", &b);

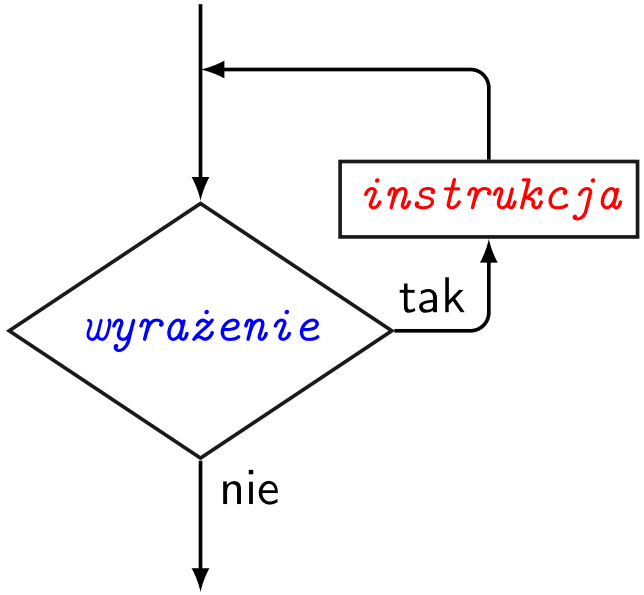
    if( a != 0.0 )
    {
        x0 = -b / a;
        printf("x0 = %.4f\n", x0);
    }
    else printf("Brak rozwiazan\n");

    return 0;
}
```

☞ linia.c

Składnia pętli `while` (dopóki)

```
while ( wyrażenie )
    instrukcja
```



Przykład

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

Pętla nieskończona

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

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

Algorithm Silnia

Dane wejściowe: liczba całkowita $n \geqslant 0$

Wynik: wartość $x = n!$

1: $i \leftarrow 2$

2: $x \leftarrow 1$

3: **dopóki** $i \leqslant n$ **wykonuj**

4: $\quad x \leftarrow x \cdot i$

5: $\quad i \leftarrow i + 1$

6: **wypisz** x

```c
#include<stdio.h>

int main()
{
    int x, i, n;

    printf("n = ");
    scanf("%d", &n);

    if(n<0) printf("Zle dane: n<0\n");
    else
    {
        x=1;
        i=2;
        while( i <= n )
        {
            x = x * i;
            i = i + 1;
        }
        printf("%d! = %d\n", n, x);
    }
    return 0;
}
```

☞ silnia.c

Tablica

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

Przykład

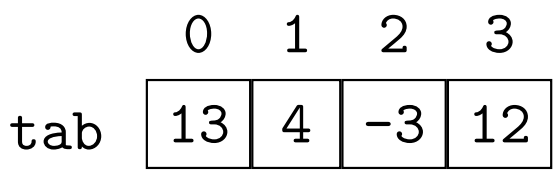
```
int tab[4];

tab[0] = 13;
tab[1] = 4;
tab[2] = -3;
tab[3] = tab[0] - 1;
tab[4] = -1;
```

Deklaracja tablicy 4 elementowej

Tablice w C są indeksowane od 0

Źle ! Poza zakresem.



Przykład: odwracanie kolejności

```c
#include<stdio.h>

int main()
{
    int tab[100];
    int i = 0;
    int a = -1;

    printf("Podaj sekwencje liczb calkowitych.\n");
    printf("Aby zakonczyc podaj 0.\n");

    while( a != 0 && i < 100)
    {
        scanf("%d", &a);
        tab[i] = a;
        i = i + 1;
    }

    printf("Podales %d liczb.\n", i);

    while(i > 0)
    {
        i = i - 1;
        printf("%d\n", tab[i]);
    }

    return 0;
}
```

```
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 }
```

```c
#include<stdio.h>                                   /* studio.h, srednik */

int main()                                                      /* int */
{
   int n,i=1;                        /* srednik, deklaracja, inicjalizacja */

   printf("Podaj liczbe calkowita  wieksza od zera : ");
   scanf("%d",&n);                                    /* format, adres */

   if(n <= 0)
   {                                                          /* nawiasy */
      if ( n==0 ) printf("To jest zero!\n");                      /* == */
   }
   else
   {
      printf("Dzielniki liczby %d:\n",n);
      while( i<=n )                          /* srednik, p. nieskonczona */
      {
         if( n % i == 0 )  printf("%d\n",i);          /* %d, \n, == */
         i = i + 1;
      }
   }
   return 0;
}
```

☞ liczba–ok.c

```
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(){int a,
5 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
- narzędzia dbające o czytelność

  `$ astyle nwd-balagan.c`

```c
#include<stdio.h>
int nwd(int a,int b)
{
int c;
while (b!=0)
{
c=a%b;
a=b;
b=c;
}
return a;
}
int main()
{
int a,b;
printf("Podaj dwie liczby calkowite: ");
scanf("%d %d",&a,&b);
printf("NWD(%d,%d) = %d\n",a,b,nwd(a,b));
return 0;
}
```

☞ **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
- Ogranicz długość linii do 80–120 znaków (szerokość ekranu)
- Rozbijaj długie instrukcje na kilka linii, zachowując czytelność:

```
while ((a < b && wplata(x) != -1) ||
       (c > d && sprawdz(x) == 1)) {
    if (w != NULL) {
        a = a - 1 + sin(PI * 2);
    }
}
```

- Oddzielaj deklaracje zmiennych, funkcje i logiczne bloki kodu pustymi liniami
- Stosuj odstępy (spacje) między operatorami, po przecinkach
- język Python - wcięcia elementem składni języka

Komentarze pozwalają umieścić dodatkowe informacje dla czytających kod (nie są kompilowane)

```
/* komentarz blokowy */
// komentarz liniowy wprowadzony w C99
```

```
int main()
{
    /* Wszystko co tutaj jest napisane
       jest komentarzem i nie zostanie skompilowane */

    int x;    /* Bardzo wazna zmienna */
    int y;    // Komentarz do konca linii
}
```

Pisz **krótkie, klarowne komentarze** wyjaśniające "dlaczego", a nie "co" (kod powinien być samodokumentujący się).
Nie komentuj oczywistych rzeczy.

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

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`

Podsumowanie

- każdą zmienną trzeba zadeklarować (określić typ i nazwę)
- nie zapomnij o średniku na końcu instrukcji
- operator przypisania `a=b`, operator porównania `a==b`
- najpierw deklaracja zmiennej potem użycie (ANSI C)
- inicjuj zmienne wartościami początkowymi
- czytelność kodu bardzo ważna:
  nieczytelny kod najprawdopodobniej jest błędny,
  zrozumiałe nazwy zmiennych, wcięcia, komentarze
- spójny styl w całym projekcie

 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/`

Narzędzia programistyczne

- **kompilator** + edytor tekstu
- **debugger** (odpluskwiacz) - służy do dynamicznej analizy programów (w czasie działania), w celu odnalezienia i identyfikacji zawartych w nich błędów, np. śledzenie wartości zmiennych, stan stosu
- **profilowanie** - badanie zachowania programu, przy użyciu informacji zdobytych podczas jego wykonywania np. graf wywołań, pomiar czasu wykonywania instrukcji, badanie zajętości pamięci
- **analiza statyczna kodu (linter)** - analiza fragmentów kodu (bez konieczności uruchamiania) w calu wykrycia potencjalnych błędów, naruszeń stylu (tzw. *code smells*)
- **systemy kontroli wersji** - śledzenie zmian w kodzie i współdzielenie kodu w zespołach, scalanie zmian
- środowiska IDE - integrują wiele narzędzi

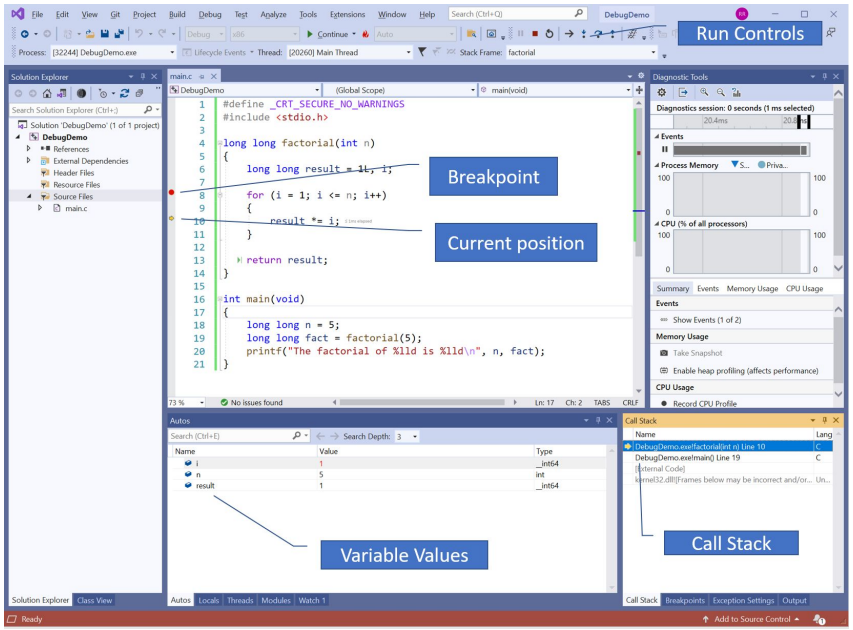
Przydatne opcje kompilatora GCC:

`-Wall`	włacza szereg reguł wykrywających potencjalne błedy, generuje ostrzeżenia
`-Wextra`	włacza dodatkowe reguły dotyczące ostrzeżeń
`-ansi`	używa standardu ANSI C89
`-pedantic`	pilnuje restrykcyjnie norm standardu ISO
`-fanalyzer`	analiza statyczna kodu
`-g`	dodaje do wynikowego kodu informacje niezbędne do debugowania
`-pb`	dodaje do wynikowego kodu informacje niezbędne do profilowania
`-lm`	dołącza bibliotekę matematyczną (`math.h`)

Przykład:

```
$ gcc -Wall -Wextra -ansi -pedantic -fanalyzer liczba-err.c
```

Debugger w Visual Studio



- punkty przerwania (*breakpoints*)
- krokowanie (*step in*, *step over*, *step out*)
- podgląd wartości zmiennych
- podgląd stosu wywołań

GDB - debuger konsolowy projektu GNU

```
$ gcc -g -o program program.c
$ gdb program
(gdb) run
```

Źródło: ☞ Visual Studio Debugging Seneca, Software testing

- **Analiza statyczna kodu** - analiza struktury kodu źródłowego lub kodu skompilowanego bez jego uruchomienia
- **lint, linter** - program do statycznej analizy
- wykorzystywane w automatyzacji weryfikacji jakości kodu
- zawierają reguły i heurystyki wykrywające podatności
- analiza poprawności składni, detekcja luk bezpieczeństwa, weryfikacja stylu kodu, sugestie dotyczące wydajnośc, ...
- czesto koncentrują się na wybranym zestawie podatności, dlatego warto stosować kilka rozwiązań
- uwaga: mogą generować fałszywie pozytywne ostrzeżenia, czyli błednie wskazywać na istnienie problemu

Przykładowe narzędzia analizy statycznej dla języka C/C++:

- **cppcheck** - głównie wykrywanie krytycznych błędów
  zob.: ☞ lista testów
  ```
  $ cppcheck --enable=all liczba-err.c
  ```

- **clang-tidy** - wiele testów i bogate możliwości konfiguracji
  zob.: ☞ lista testów
  ```
  $ clang-tidy -checks='*'  liczba-err.c
  ```

Narzędzia pozwalające dbać o styl w języku C

- ☞ Artistic Style

```
$ astyle  --style=kr nwd-balagan.c
$ astyle  --style=allman nwd-balagan.c
```

- ☞ ClangFormat

```
$ clang-format -style GNU nwd-balagan.c
```

- on-line ☞ formatter.org ☞ Code Beautify

Przykłady wytycznych dotyczących stylu:

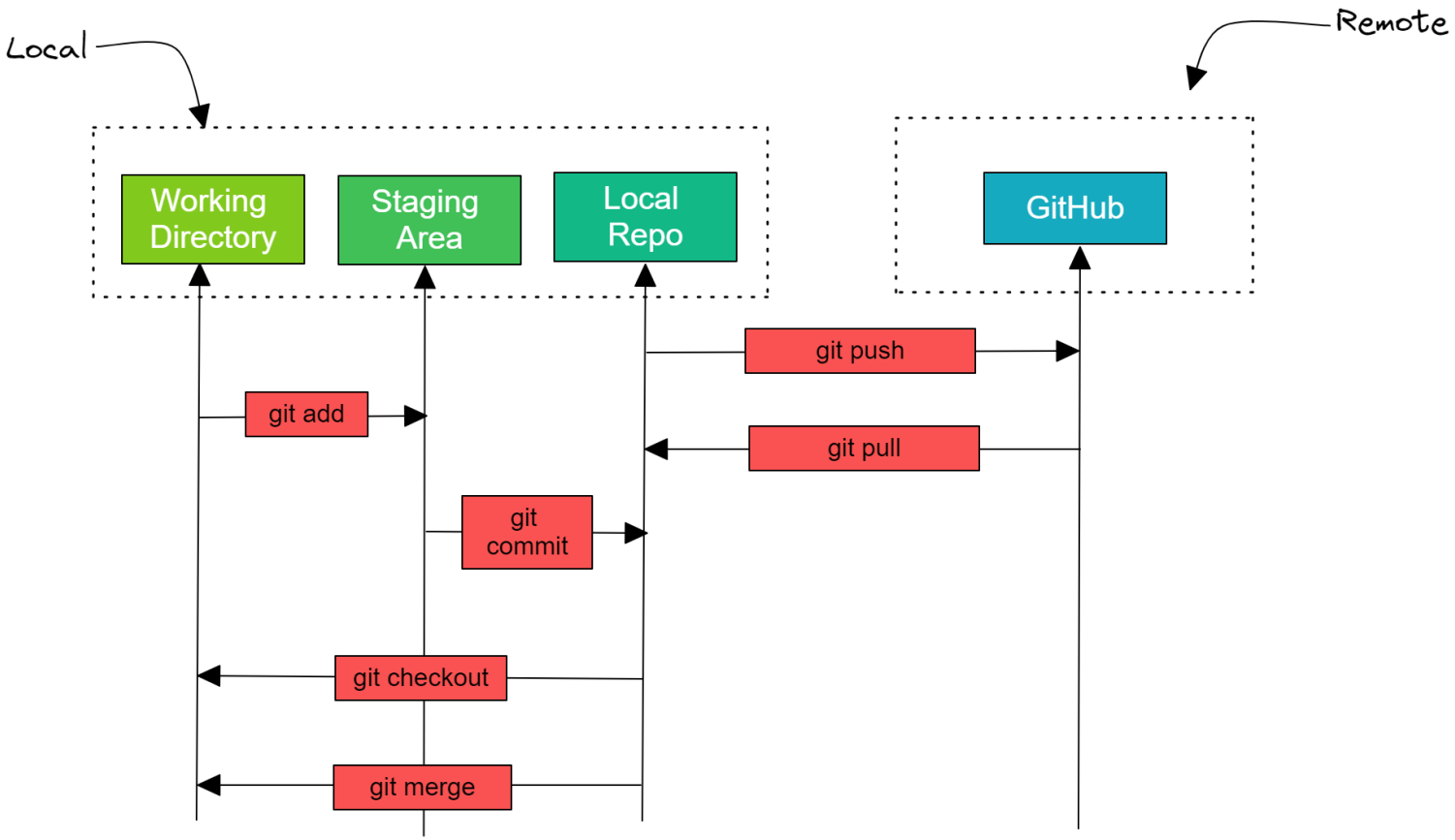
- wytyczne GNU ☞ Making The Best Use of C
- ☞ C Code Style Guidelines
- ☞ Google C++ Style Guide

Git - rozproszony system kontroli wersji służący do zarządzania historią zmian w plikach i kodzie źródłowym projektów

- przechowuje historię zmian w kodzie
- umożliwia współdzielenie kodu w zespołach
- gałęzie, łatwe tworzenie i scalanie gałęzi
- synchronizacja historii pomiędzy repozytoriami
- szybki w działaniu - wiele operacji wykonywanych lokalnie
- elastyczny, wiele aplikacji klienckich, integracja z narzędziami

☞ GitHub - usługa chmurowa udostępniająca repozytoria git
Inne: ☞ GitLab, ☞ BitBucket, ☞ Azure DevOps, ...

`add`	dodanie pliku do rewizji (śledzenie zmian)
`commit`	zatwierdzenie zmian, powstaje nowy węzeł w historii repozytorium (lokalnie)
`push`	wypchnięcie zmian do zdalnego repozytorium
`pull`	pobranie zmian ze zdalnego repozytorium i scalenie z lokalną kopią
`merge`	scalanie zmian, łączenie gałęzi
`log`	historia zmian
`diff`	porównywanie zmian w kodzie
`status`	aktualny stan repozytorium



☞ *GitHub and Git Tutorial for Beginners*

Repozytorium z kodami z wykładu:

☞ https://github.com/IS-UMK/pp_wyklad

Klonowanie repozytorium:

```
$ git clone https://github.com/IS-UMK/pp_wyklad
```

Historia zmian:

```
$ git log
```

```
commit 01cc8a5dfb066e280ace0059ad05af9bbc951fbf (HEAD -> master)
Author: Marek <grochu@is.umk.pl>
Date:   Tue Oct 1 11:08:55 2024 +0200

      Dodanie euclid1.f euclid1.pas

commit 0552f871c029186cca1784270411ce77aad5e266
Author: Marek <grochu@is.umk.pl>
Date:   Tue Oct 1 10:51:51 2024 +0200

      Aktualizacja README
```

Dodanie pliku:

```
$ git add nowy.c
```

Status repozytorium:

```
$ git status
```

```
Na gałęzi master
Twoja gałąź jest na bieżąco z „origin/master".

Zmiany do złożenia:
  (użyj „git restore --staged <plik>...", aby wycofać)
nowy plik:       nowy.c
```

Zapamiętanie zmian:

```
$ git commit -m "Opis wprowadzonych zmian"
```

Wypchnięcie zmian do zdalnego repozytorium:

```
$ git push
```

Pobranie i scalenie zmian pochodzących ze zdalnego repozytorium:

```
$ git pull
```

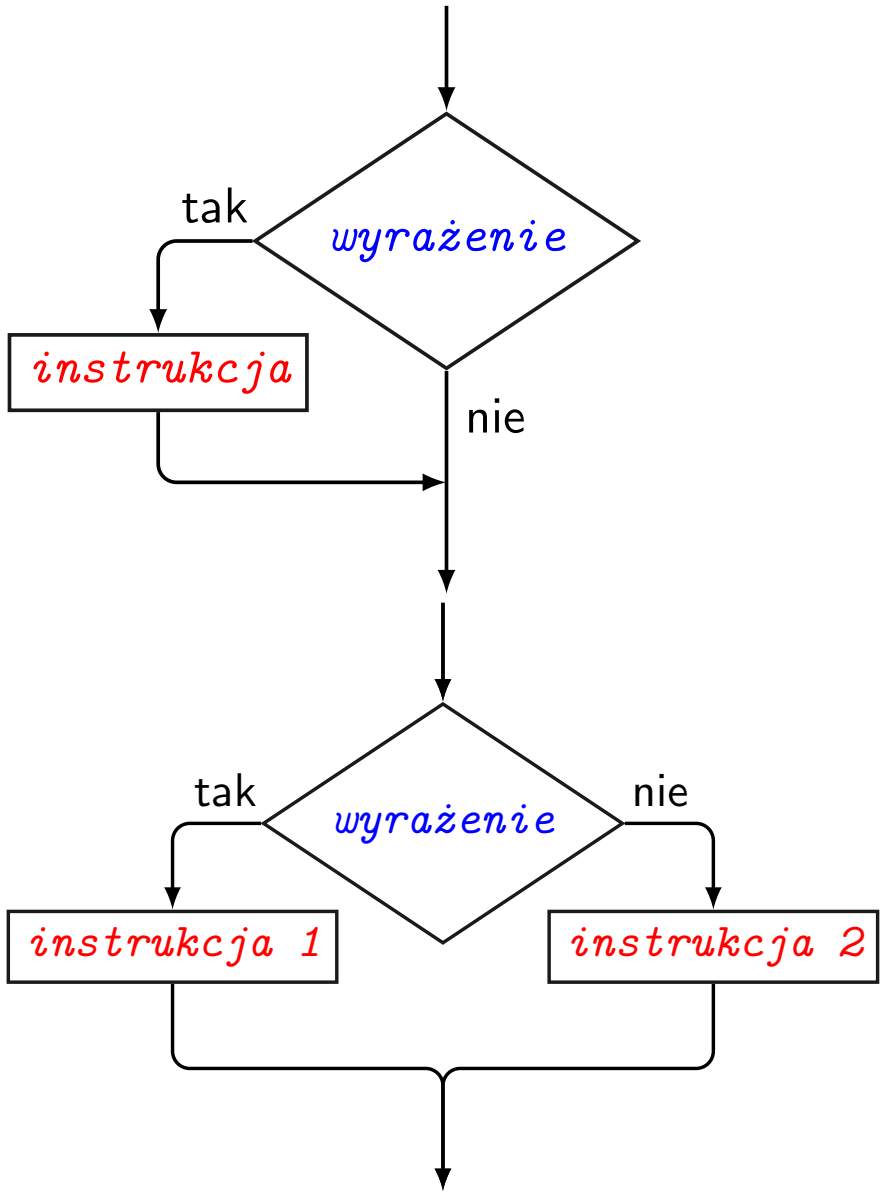
Instrukcje sterujące

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`	$x \mod 2$
++	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 <= y`	$x \leqslant y$
>=	większy lub równy	`x >= y`	$x \geqslant y$
==	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`	$x < 1 \vee y > 2$

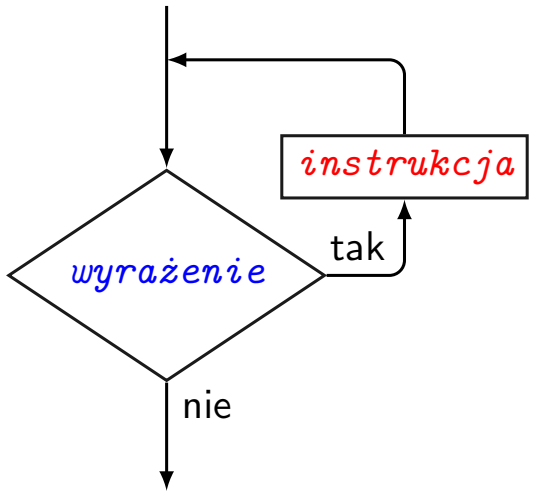
Instrukcja warunkowa if else

```
if ( wyrażenie )
    instrukcja
```

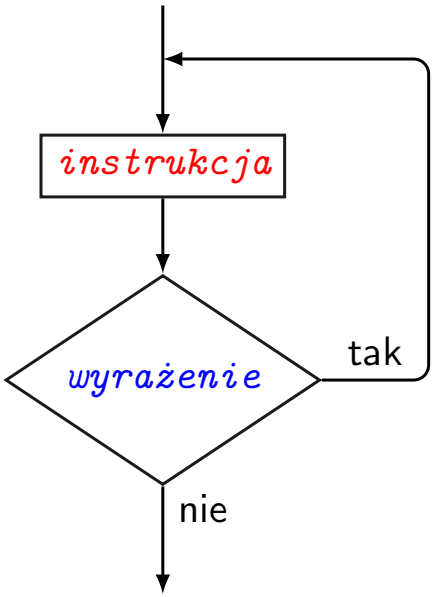
```
if ( wyrażenie )
    instrukcja 1
else
    instrukcja 2
```



```
while ( wyrażenie )
   instrukcja
```




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



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

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

Algorithm Algorytm Herona

Dane wejściowe: liczba $a \geqslant 0$, 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: $\quad x_0 \leftarrow x$
4: $\quad x \leftarrow \frac{1}{2}\left(x_0 + \frac{a}{x_0}\right)$
5: **dopóki** $|x - x_0| > \epsilon$
6: **wypisz** x

```c
#include<stdio.h>

int main()
{
    const float eps = 1e-4;
    float x, a, x0;

    printf("a = ");
    scanf("%f", &a);

    if (a < 0) printf("Zle dane: a < 0\n");
    else
    {
        x0 = 1;
        x = x0;
        do
        {
            x0 = x;
            x = (x0 + a/x0)/2;
        } while (x - x0 > eps || x - x0 < -eps);

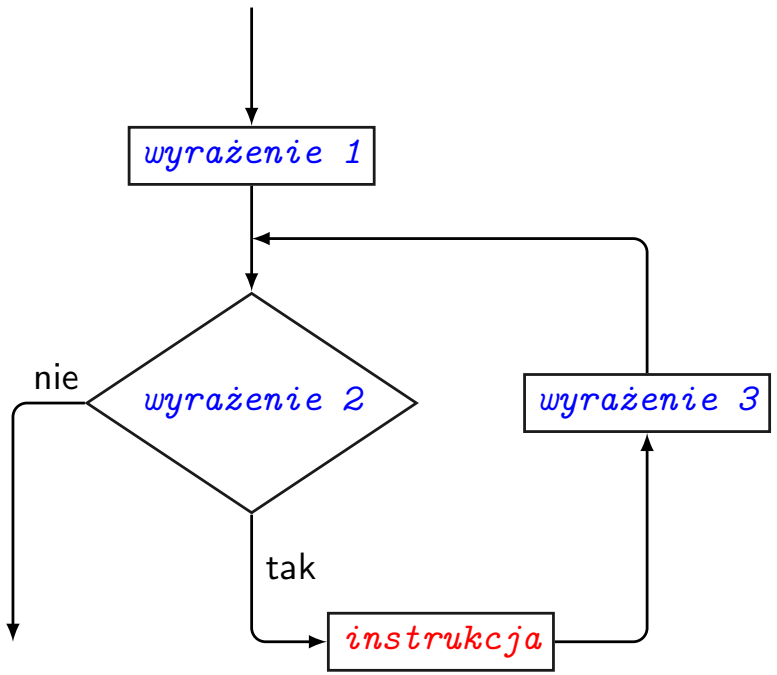
        printf("pierwiastek z %f wynosi %f (eps= %f)\n", a, x, eps);
    }
    return 0;
}
```

☞ heron.c

```
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
}
```

```
for ( A ; B ; C )
{
    instrukcja
}
```

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

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

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
15     printf("Koniec.\n");
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.
```

☞ break.c

Algorithm Metoda siłowa sprawdzania czy liczba jest liczbą pierwszą

Dane wejściowe: liczba całkowita $n > 0$
Wynik: odpowiedź: liczba n jest (lub nie jest) liczbą pierwszą

1: **dla** $i = 2, 3, \ldots, n-1$ **wykonuj**
2: **jeżeli** i jest dzielnikiem n **wykonaj**
3: **wypisz** `nie jest liczbą pierwszą`
4: **przerwij**
5: **jeżeli** nie znaleziono dzielnika **wykonaj**
6: **wypisz** `jest liczbą pierwszą`

```c
#include <stdio.h>

int main()
{
    int n, i = 2;

    printf("n = ");
    scanf("%d", &n);

    while (i < n)
    {
        if (n % i == 0) break;
        i = i + 1;
    }

    if (i == n || n == 1)
        printf("Liczba %d jest liczba pierwsza\n", n);
    else
        printf("Liczba %d nie jest liczba pierwsza\n", n);

    return 0;
}
```

☞ czy–pierwsza.c

Kolejna iteracja: continue

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

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int i, n=16;
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
14
15
16
```

☞ continue.c

Instrukcja skoku goto przenosi sterowanie w miejsce kodu oznaczone etykietą.

```
etykieta:
instrukcje
...
goto etykieta;
```

Główna zasada **programowania strukturalnego**:

Nie używaj instrukcji skoku!

```c
#include <stdio.h>

int main()
{
    int n;
    poczatek:
    printf("Podaj liczbe z zakresu od 1 do 10\n");
    scanf("%d", &n);
    if (n<1 || n>10){
        printf("Blad: niepoprawna wartosc\n");
        goto poczatek;
    }
    else{
        printf("OK: podales liczbe %d\n", n);
        goto koniec;
    }
    printf("Halo, tutaj jestem!\n");
    koniec:

    return 0;
}
```

☞ goto.c

Czemu skok to zazwyczaj zły pomysł?

- instrukcja skoku przerywa **liniowy przepływ programu**, kod staje się **nieczytelny i trudny do zrozumienia** (tzw. *spaghetti code*)
- kod ze skokami jest **trudniejszy do utrzymania i debugowania**, przepływ programu jest **mniej przewidywalny** a ewentualne modyfikacje kodu częściej prowadzą do powstawania błędów
- trudności techniczne, niektóre skoki są niewykonalne:

  skoki do wnętrza pętli, z pętli do pętli, do wnętrza funkcji, itd. ...
- skok omija strukturalne elementy kodu (pętle, bloki warunkowe), jest to sprzeczne z praktykami **programowania strukturalnego**
- kompilatory mogą mieć **trudności z optymalizacją** kodu zawierającego skoki, co może prowadzić do mniej efektywnego działania programu
- `break` i `continue` - podobne wątpliwości, ale to skoki lokalne
- kod ze skokami zawsze można przepisać do postaci bez skoków
- używanie instrukcji skoku świadczy o złym stylu programowania

Poprzedni przykład bez skoku

```c
#include <stdio.h>

int main()
{
    int n;
    int stop = 0;

    while (stop == 0)
    {
        printf("Podaj liczbe z zakresu od 1 do 10\n");
        scanf("%d", &n);
        if (n >= 1 && n <= 10) stop = 1;
        else printf("Blad: niepoprawna wartosc\n");
    }

    printf("OK: podales liczbe %d\n", n);
    return 0;
}
```

☞ goto2.c

```c
#include <stdio.h>

int main ()
{
    int i, j, n;

    printf("Podaj zakres: ");
    scanf("%d", &n);

    printf("Liczby pierwsze w zakresie od 1 do %d\n", n);

    for (i=2; i<n; i++) {
        for (j=2; j <= (i/j); j++)
            if (!(i%j)) break;          /* nie jest l. pierwsza */
        if (j > (i/j)) printf("%d\n", i);
    }

    return 0;
}
```

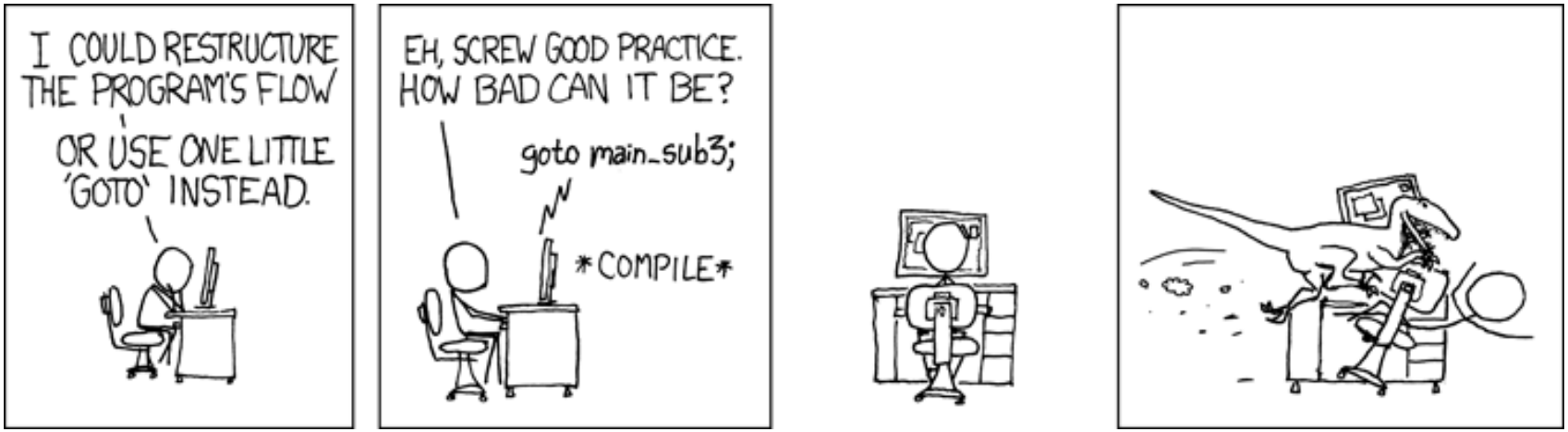
☞ prime.c

Pętle zagnieżdżone 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



http://xkcd.com/292/

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
}
```

```c
#include<stdio.h>

int main()
{
    float x, y;
    char op;

    scanf("%f %c%f", &x, &op, &y);

    switch (op)
    {
        case '+' :
            printf("%f\n", x + y);
            break;
        case '-' :
            printf("%f\n", x - y);
            break;
        case '*' :
            printf("%f\n", x * y);
 //           break;
        case '/' :
            printf("%f\n", x / y);
//          break;
        default:
            printf("Nieznana operacja: %c\n", op);
    }
    return 0;
}
```

```
3.14 + 5
8.140000

3.14+5
8.140000

3.14
+
5
8.140000
```

☞ switch.c

- Dopasowanie przypadków możliwej jest tylko dla wartości 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 `brake` (lub do końca bloku `switch`). Pominięcie instrukcji `break` spowoduje więc wykonanie instrukcji dla kolejengo przypadku.

```c
#include<stdio.h>

int main()
{
    int n;

    printf("Podaj liczbe od 1 do 4\n");
    scanf("%d", &n);

    switch (n)
    {
        case 1:
            printf("Przypadek 1\n");
        case 2 :
            printf("Przypadek 2\n");
        case 3 :
            printf("Przypadek 3\n");
            break;
        case 4 :
            printf("Przypadek 4\n");
            break;
        default:
            printf("Nieznana operacja.\n");
    }
    return 0;
}
```

```
Podaj liczbe od 1 do 4
2
Przypadek 2
Przypadek 3
```

Instrukcje sterujące przepływem programu w języku C:

- Instrukcje warunkowe: `if`, `else`, `switch`
- Pętle: `while`, `do while`, `for`
- Instrukcja wielokrotnego wyboru `switch`
- Instrukcja skoku `goto` (lepiej nie używać, łamie zasady programowania strukturalnego)
- Instrukcja przerwania bieżącej pętli `break`
- Instrukcja kontynuacji bieżącej pętli `continue`

 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.

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

float g = 2.5;

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

int main()
{
    float x, y;

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

    return 0;
}
```

Dyrektywy preprocesora

Zmienne globalne

Definicja funkcji

Funkcja główna

Zmienne lokalne

Instrukcje programu

Podprogram

wydzielony fragment programu (kodu) zawierający instrukcje do wielokrotnego użytku

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

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);
```

Przykład funkcji NWD

```c
#include <stdio.h>

int nwd(int a, int b)
{
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}

int main()
{
    int a, b;

    printf("Podaj dwie liczby calkowite: ");
    scanf("%d %d", &a, &b);
    printf("NWD(%d,%d) = %d\n", a, b, nwd(a,b));

    return 0;
}
```

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 funkcji są przekazywane **wyłącznie przez wartość**, tzn. parametr aktualny jest kopiowany do parametru formalnego. Funkcja opera się na kopii, a nie na oryginalnej zmiennej.

Funkcja bez wartości zwracanej (procedura)

```
void wypisz(int n)
{
    while(n > 0)
    {
            printf("Programowanie 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;
}
```

Instrukcja `return` przerywa działanie funkcji i zwraca wartość z funkcji. Może wystąpić 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 z funkcji jest podstawiona w miejsce wywołania

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

Deklaracja i definicja funkcji c.d

```c
#include <stdio.h>

/* Deklaracja funkcji */
int jest_pierwsza ( int n );

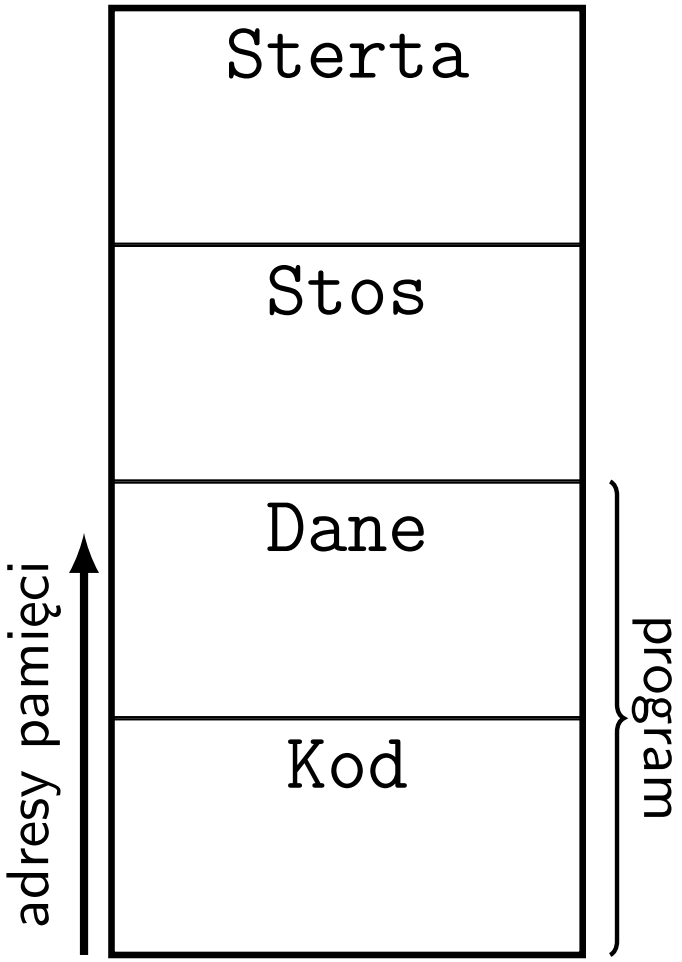
int main()
{
   int n, i=1;
   printf("n = ");
   scanf("%d", &n);
   while(i <= n)
   {
      if(jest_pierwsza(i) == 1) printf("%d\n", i);
      i++;
   }
   return 0;
}

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

☞ pierwsza2.c

W momencie kompilacji pliku muszą być znana przynajmniej deklaracje wszystkich funkcji (nazwy i typy argumentów).

Definicje funkcji mogą wystąpić później, zwykle umieszczane są w osobnym pliku źródłowym.



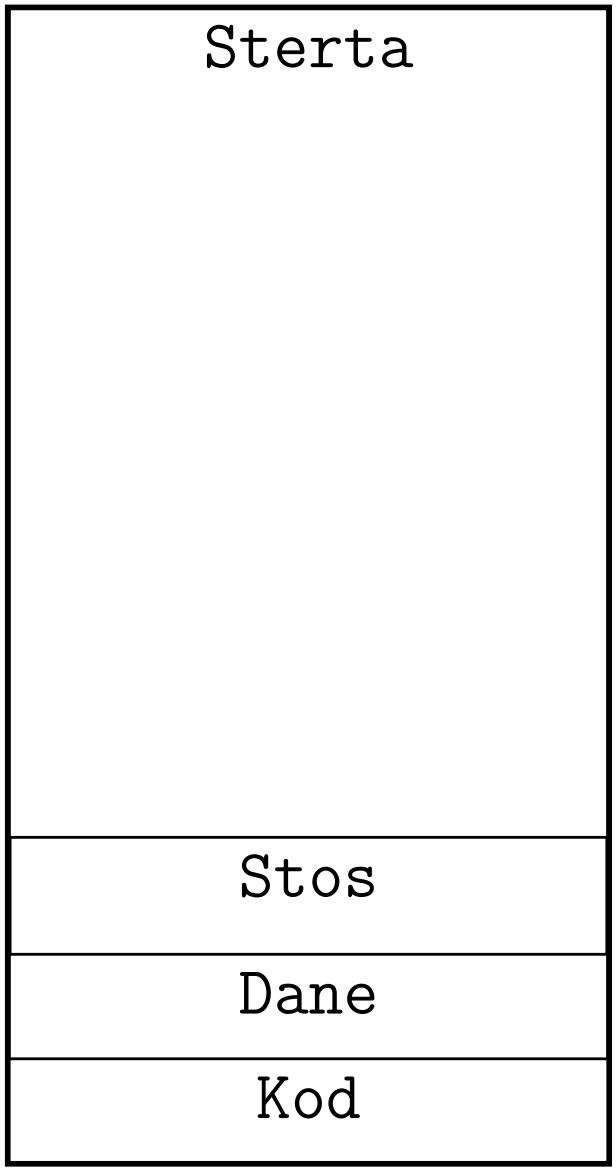
pamięć dostępna do alokacji

zmienne lokalne funkcji wkładane na stos przy wywołaniu funkcji

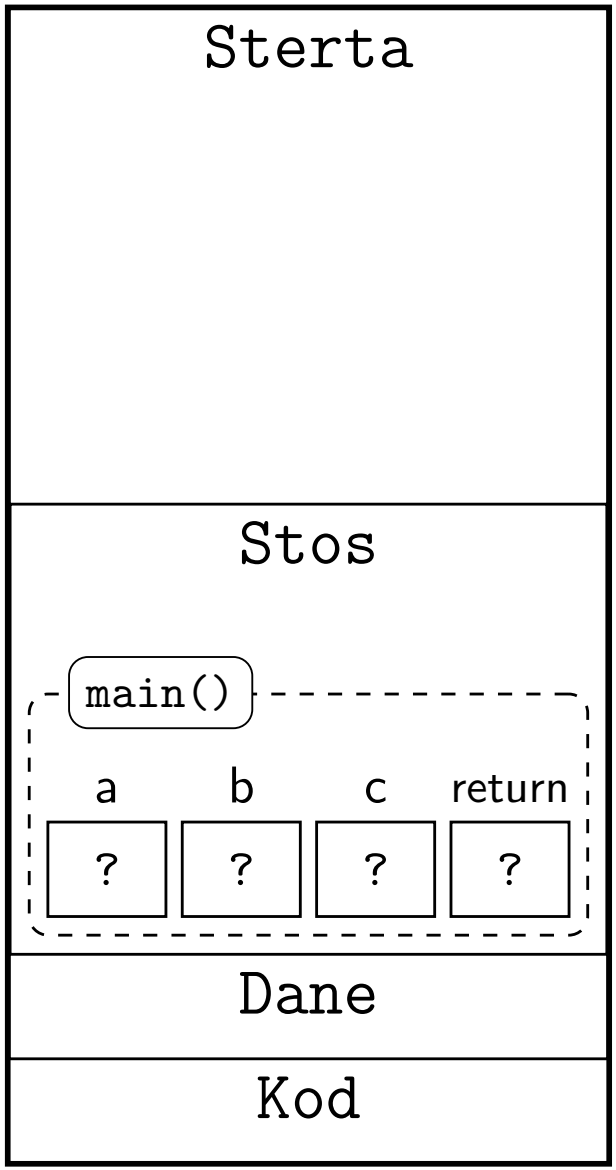
stałe oraz **zmienne globalne** i statyczne inicjowane przy uruchomieniu

tekst programu
tylko do odczytu

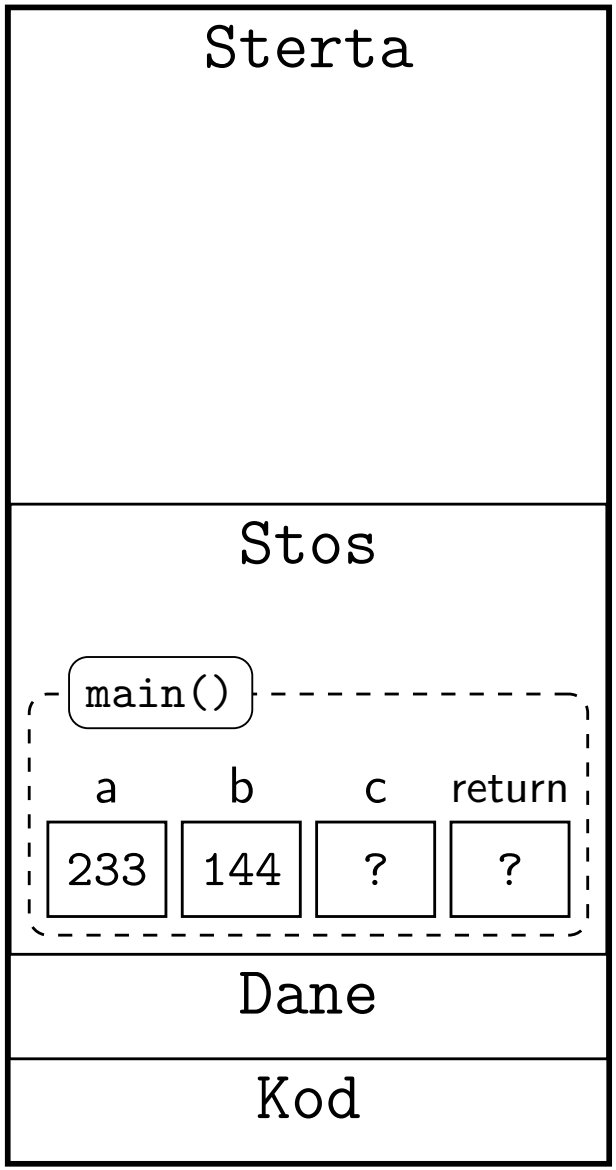
```
 1  int nwd(int a, int b)
 2  {
 3      int c;
 4      while (b != 0)
 5      {
 6          c = a % b;
 7          a = b;
 8          b = c;
 9      }
10      return a;
11  }
12
13  int main()
14  {
15      int a, b, c;
16      a = 233;
17      b = 144;
18      c = nwd(a, b);
19      return 0;
20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



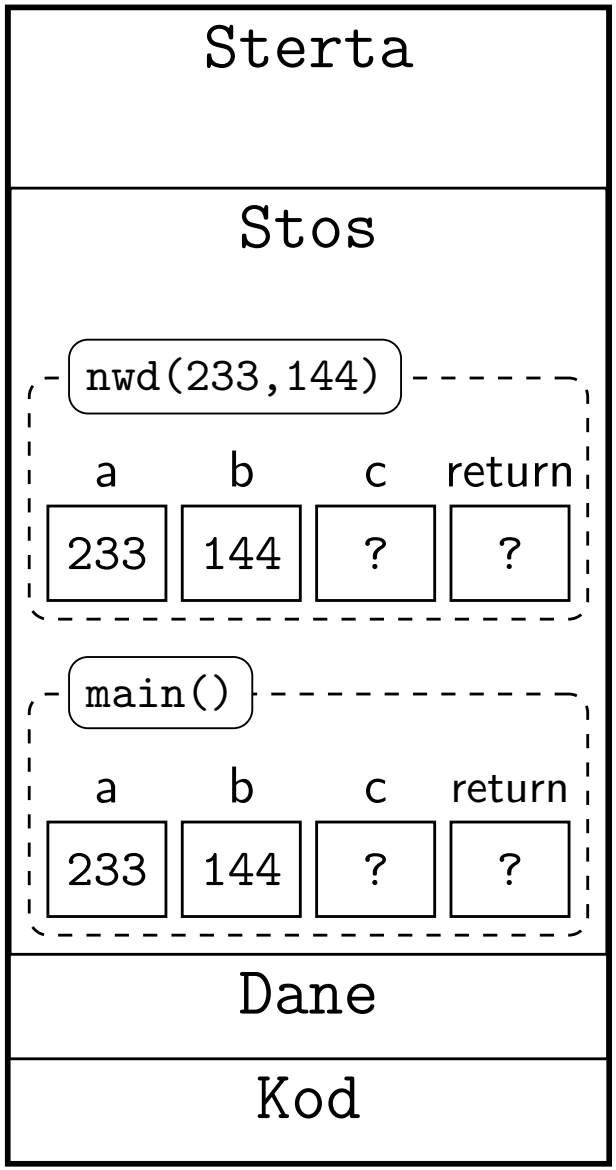
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



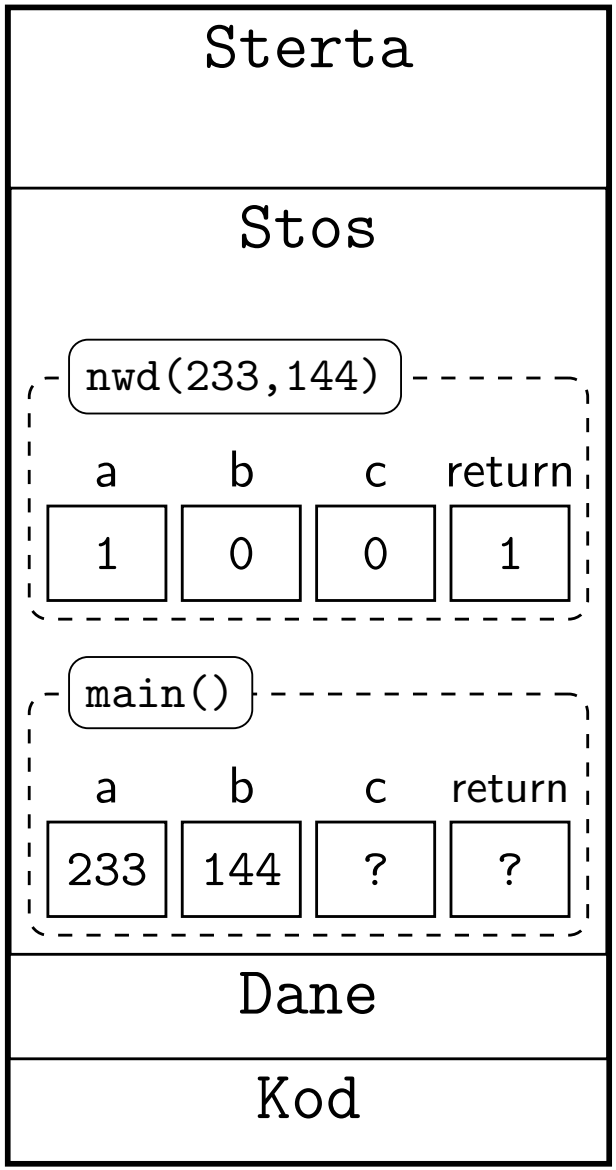
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```
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 }
```

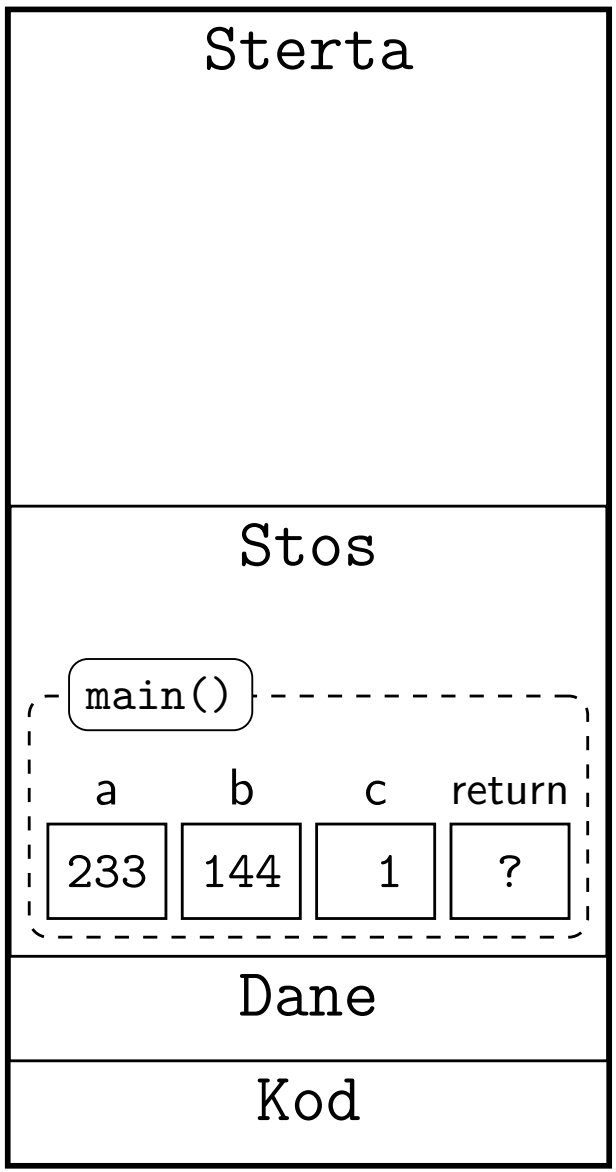


```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```

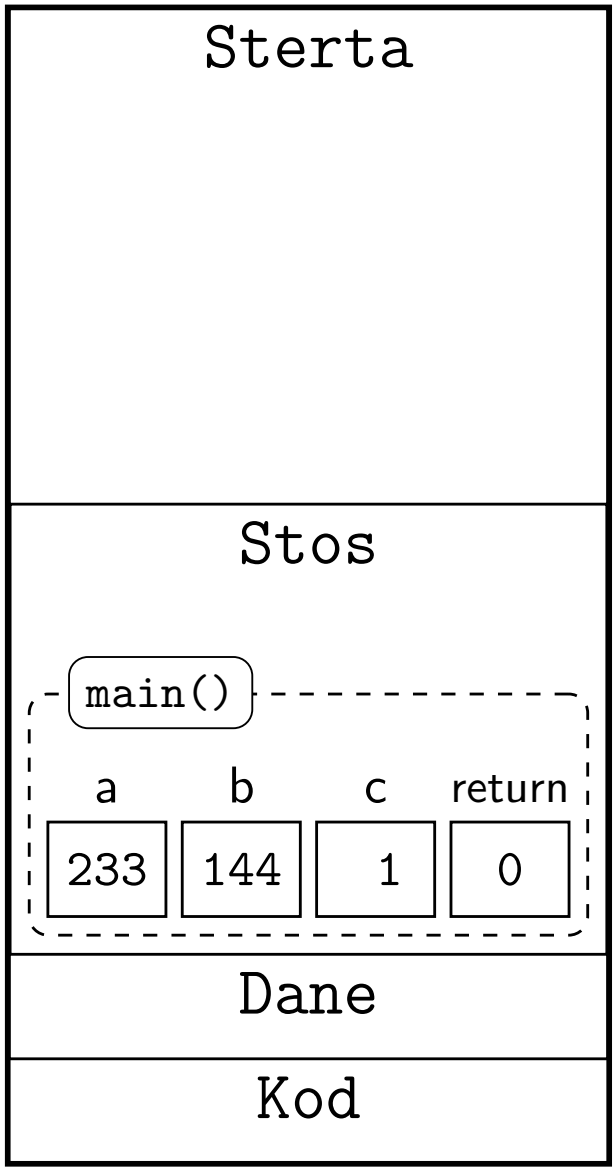


```
 1  int nwd(int a, int b)
 2  {
 3      int c;
 4      while (b != 0)
 5      {
 6          c = a % b;
 7          a = b;
 8          b = c;
 9      }
10      return a;
11  }
12
13  int main()
14  {
15      int a, b, c;
16      a = 233;
17      b = 144;
18      c = nwd(a, b);
19      return 0;
20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

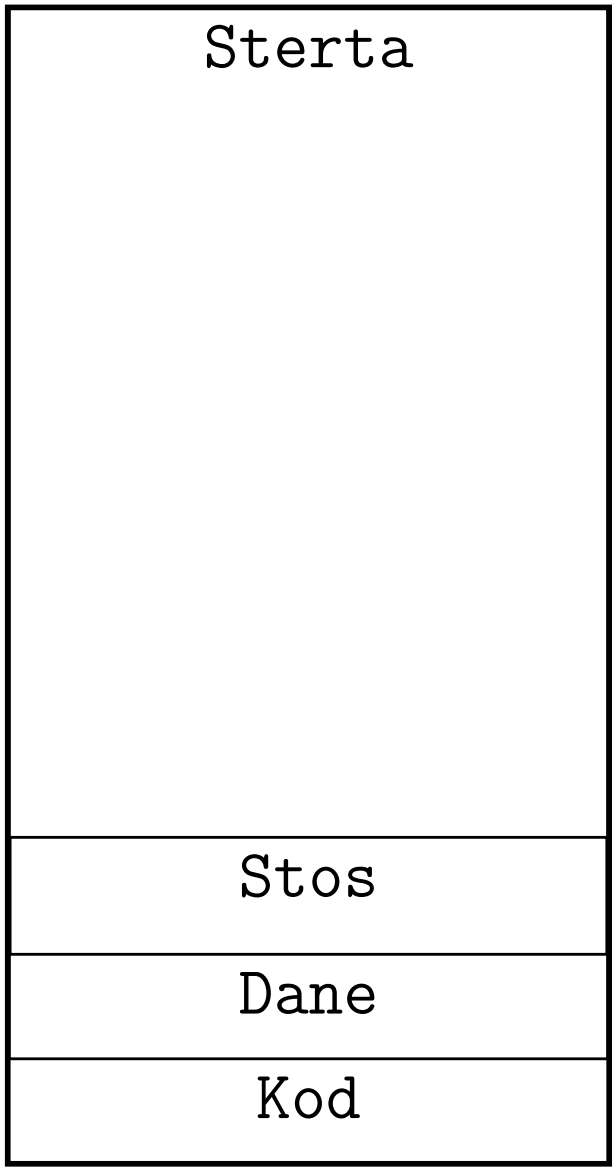




```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



```
 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  }
```



Zasięg zmiennych: globalne i lokalne

Zmienna globalna jest dostępna dla wszystkich funkcji programu (ma zasięg pliku). Istnieje przez cały czas działania programu. Deklaracja zmiennej nie znajduje się wewnątrz żadnej funkcji.

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

Zmienna lokalna (automatyczna) to wewnętrzna zmienna funkcji. Czas życia zmiennej jest ograniczony czasem działania funkcji.

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

```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
   int lokalna = 0;
   globalna = globalna + 1;
   lokalna = lokalna + 1;
   printf("%d          %d       f\n", globalna, lokalna);
}

int main()
{
   int lokalna = 42;

   printf("globalna lokalna funkcja\n");
   printf("%d         %d       main\n", globalna, lokalna);
   f();
   printf("%d         %d       main\n", globalna, lokalna);
   f();
   printf("%d         %d       main\n", globalna, lokalna);
   return 0;
}
```

☞ globalne.c

```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
    printf("%d          %d       f\n", globalna, lokalna);
}

int main()
{
    int lokalna = 42;

    printf("globalna lokalna funkcja\n");
    printf("%d         %d       main\n", globalna, lokalna);
    f();
    printf("%d         %d       main\n", globalna, lokalna);
    f();
    printf("%d         %d       main\n", globalna, lokalna);
    return 0;
}
```

Wynik działania:

```
globalna lokalna funkcja
0         42       main
1          1       f
1         42       main
2          1       f
2         42       main
```

☞ globalne.c

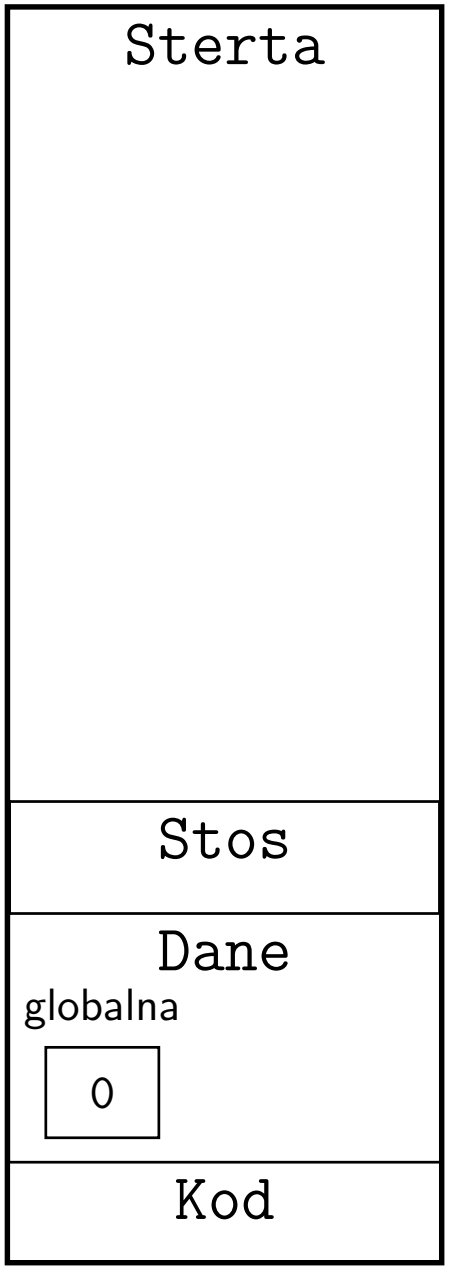
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



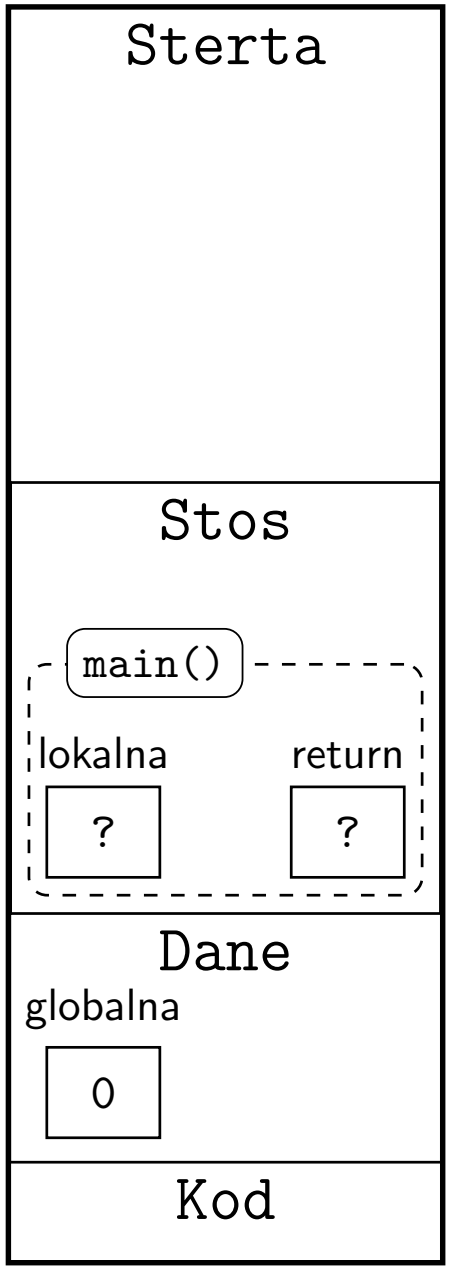
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



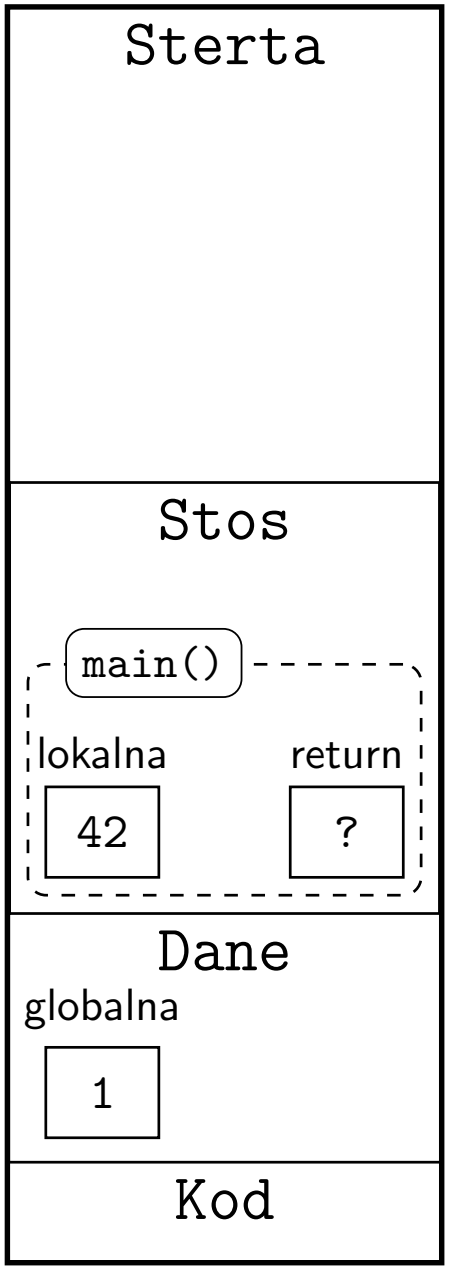
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



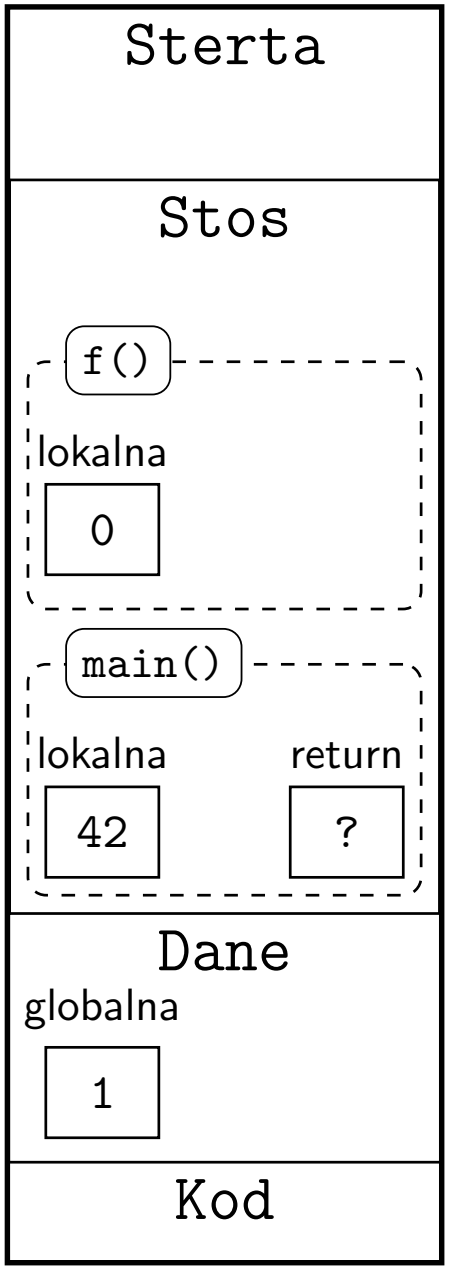
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



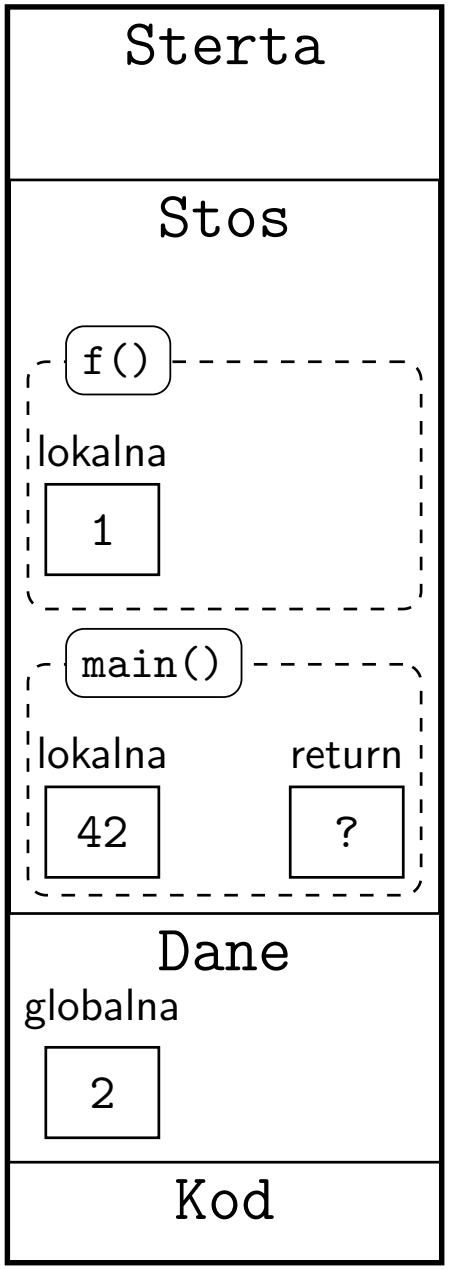
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



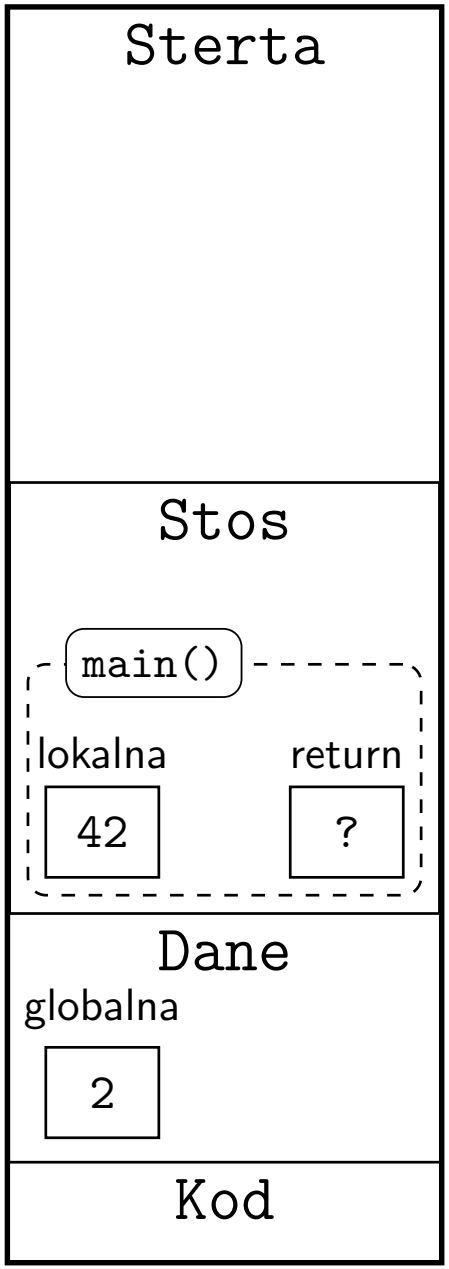
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



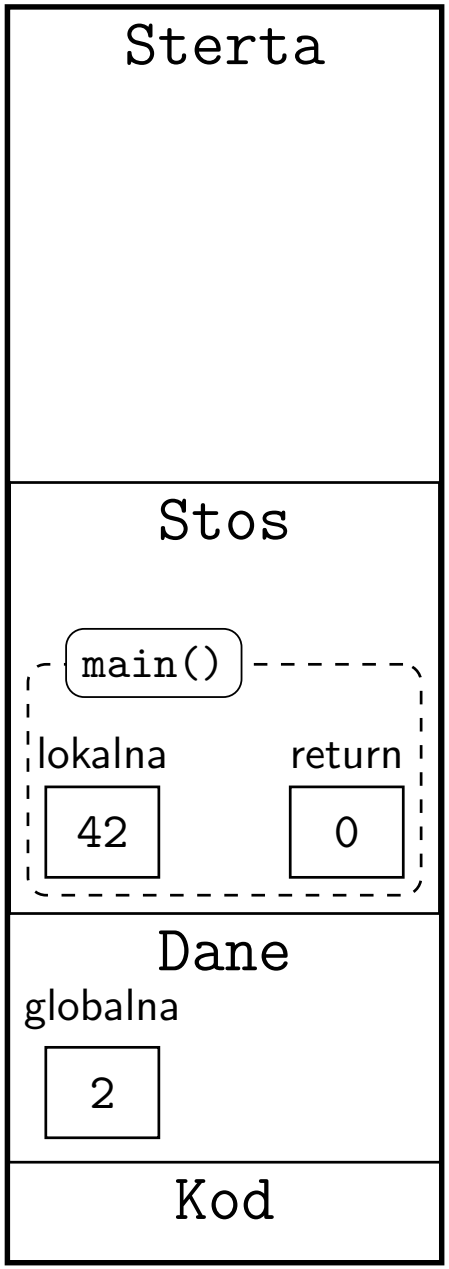
```c
#include<stdio.h>

int globalna = 0;

void f(void)
{
    int lokalna = 0;
    globalna = globalna + 1;
    lokalna = lokalna + 1;
}

int main()
{
    int lokalna = 42;
    globalna = globalna + 1;
    f();

    return 0;
}
```



W programowaniu proceduralnym nie używamy zmiennych globalnych!

Źle

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

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

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

Dobrze

```c
#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);
}
```

`<assert.h>`	asercje (do diagnostyki kodu)
`<ctype.h>`	klasyfikacja znaków
`<errno.h>`	deklaracje kodów błędów
`<float.h>`	ograniczenia typów zmiennopozycyjnych
`<limits.h>`	ograniczenia typów całkowitych
`<signal.h>`	obsługa sygnałów
`<stddef.h>`	standardowe definicje (makra)
`<stdio.h>`	operacje wejścia/wyjścia
`<math.h>`	funkcje matematyczne
`<stdlib.h>`	podstawowe narzędzia (np. `rand()`, `malloc()`)
`<string.h>`	operacje na łańcuchach znaków (napisach)
`<time.h>`	funkcje obsługi czasu
`<wchar.h>`	obsługa ciągów szerokich znaków (napisy Unicode)

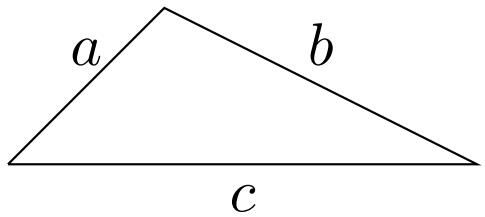
☞ *WikiBook: spis plików nagłówkowych biblioteki standardowej*
☞ *WikiBook: indeks alfabetyczny funkcji z biblioteki standardowej*

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

Uwaga: w kompilatorze GCC dołączenie funkcji z biblioteki matematycznej wymaga użycia opcji `-lm`

```
gcc -lm euclid.c
```

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

```c
#include <math.h>

/*  Funkcja wyznacza pole trojkata z wzoru Heroana.
 *  Argumenty a, b i c to dlugosci bokow.
 *  Jezeli boki a, b, i c nie tworza trojkata
 *  zwracana jest wartosc -1. */
float heron(float a, float b, float c)
{
    float p = (a + b + c) / 2;
    p = p * (p - a) * (p - b) * (p - c);
    if (p < 0) return -1;
    return sqrt(p);
}
```

☞ heron2.c

```c
int main()
{
    float a, b, c, pole;

    printf("Podaj dlugosci bokow trojkata: a, b, c > 0\n");
    scanf("%f%f%f", &a, &b, &c);

    if (a <= 0 || b <= 0 || c <= 0)
    {
        printf("Zle dane: wartosci musza byc dodatnie.\n");
        return 1;
    }

    pole = heron(a, b, c);
    if (pole < 0)
    {
       printf("Zle dane: to nie sa boki trojkata\n");
       return 2;
    }
    printf("Pole wynosi: %f\n", pole);

    return 0;
}
```

☞ heron2.c

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

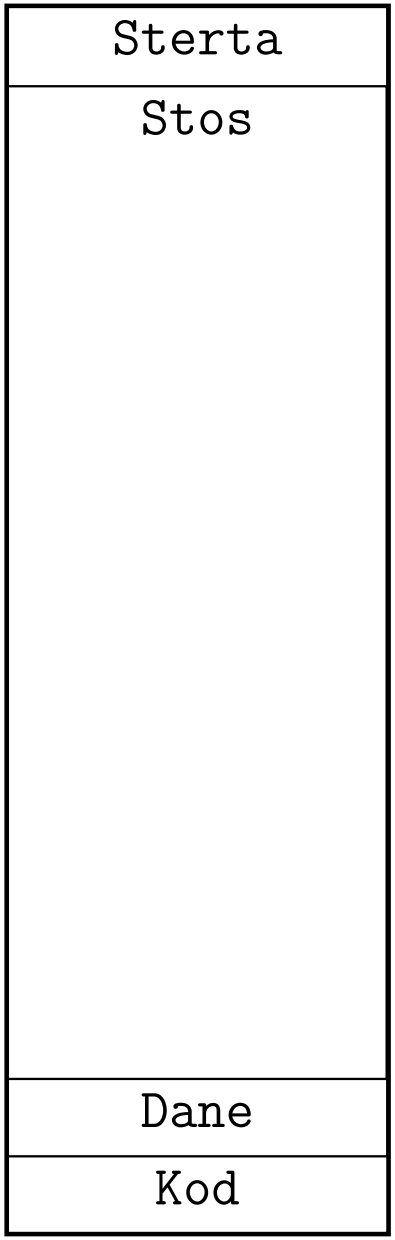
```c
#include<stdio.h>

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

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

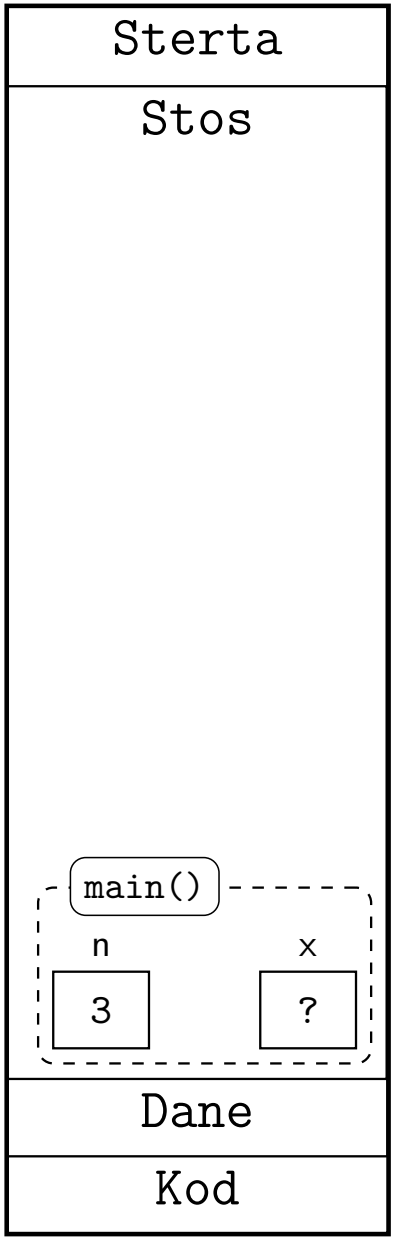
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
 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  }
```



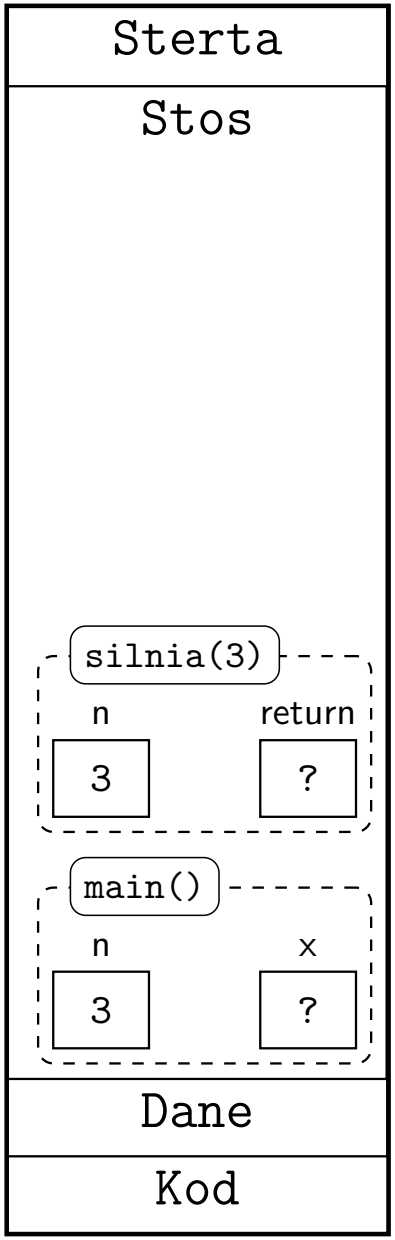
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



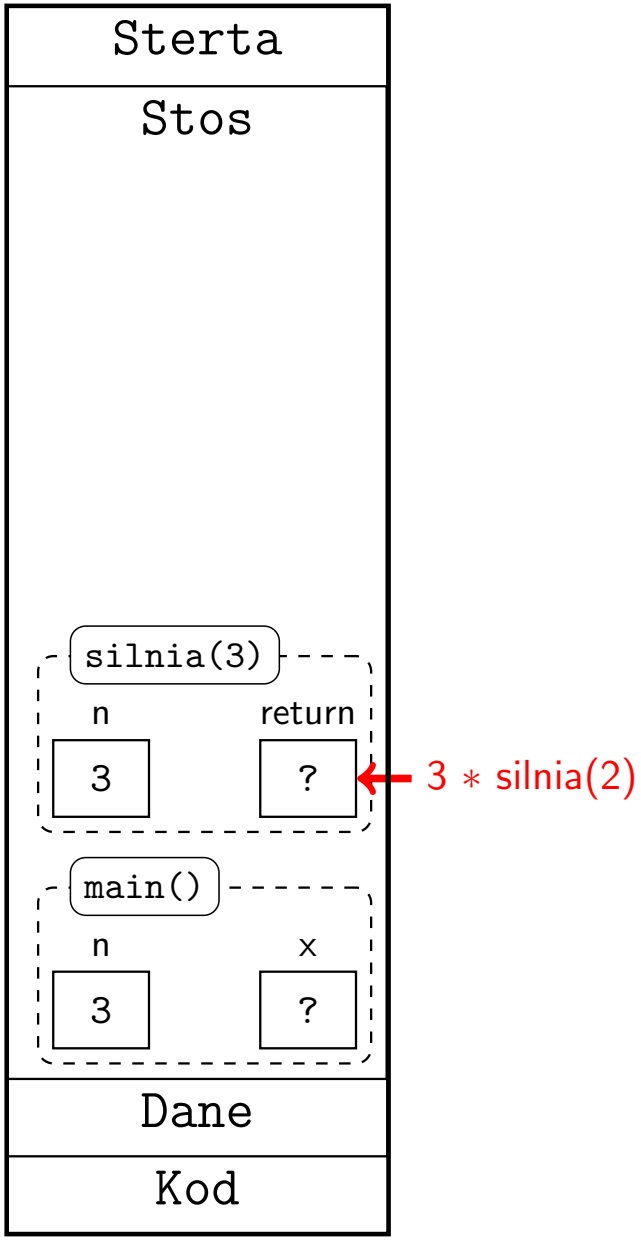
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



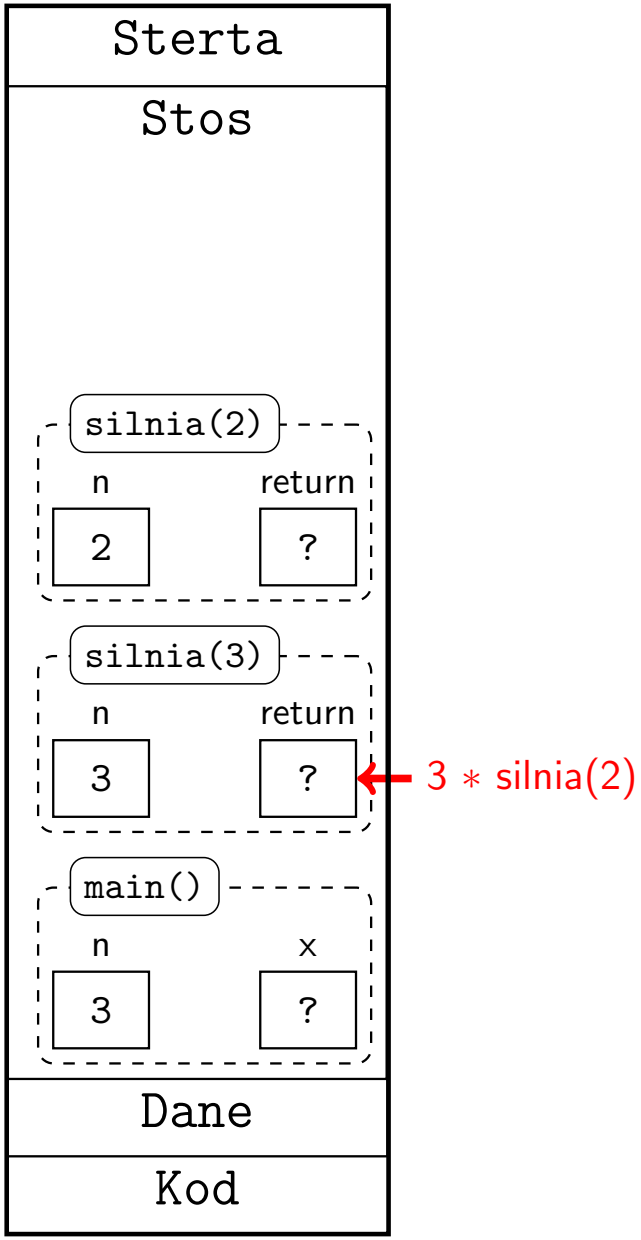
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



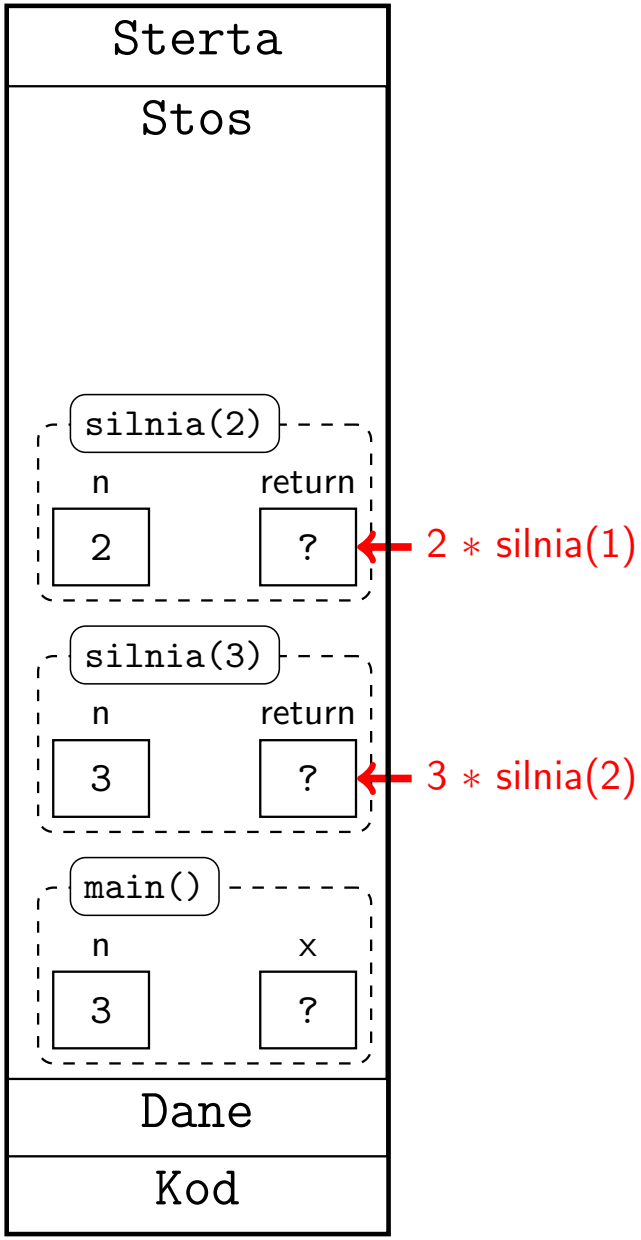
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



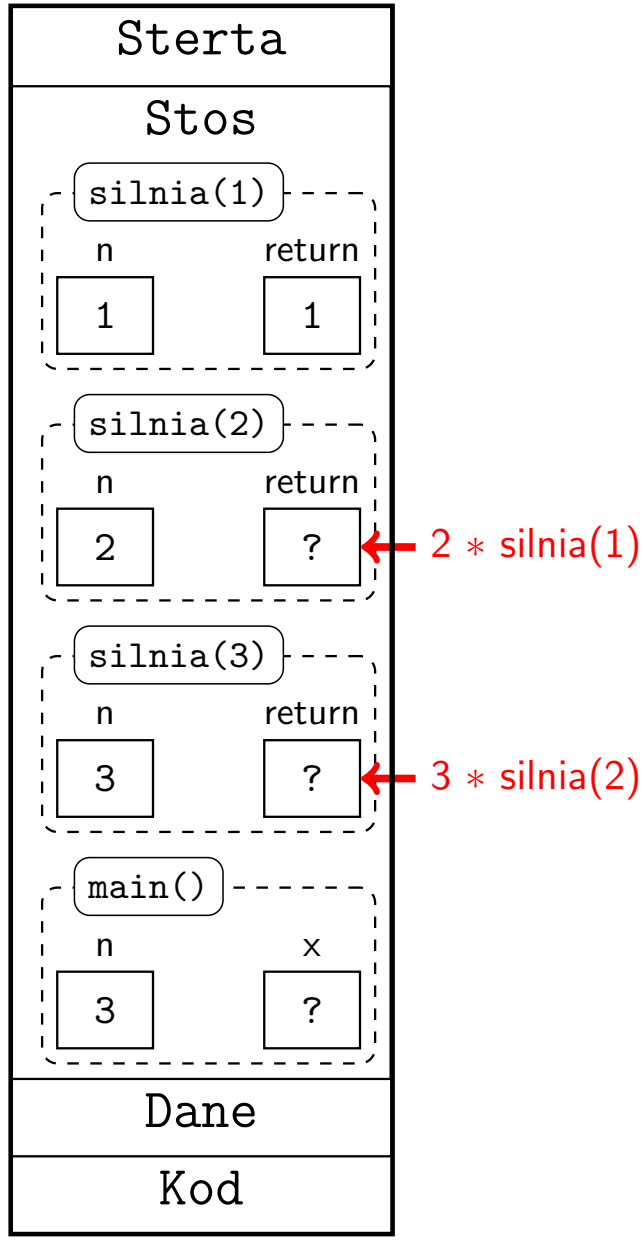
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



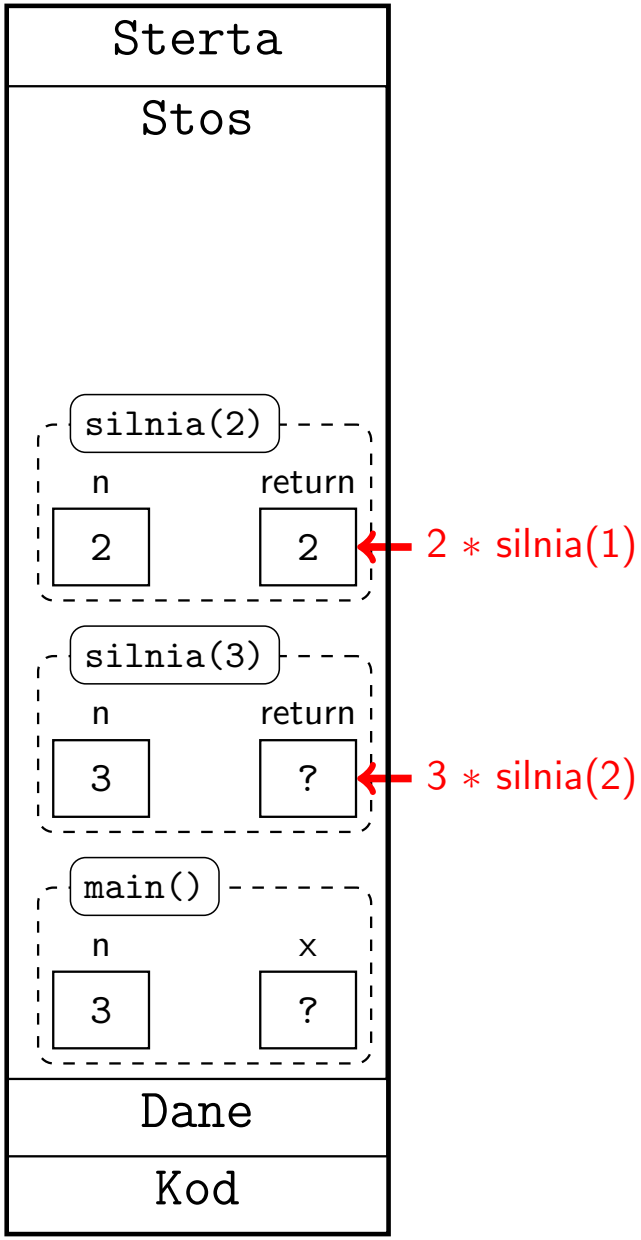
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



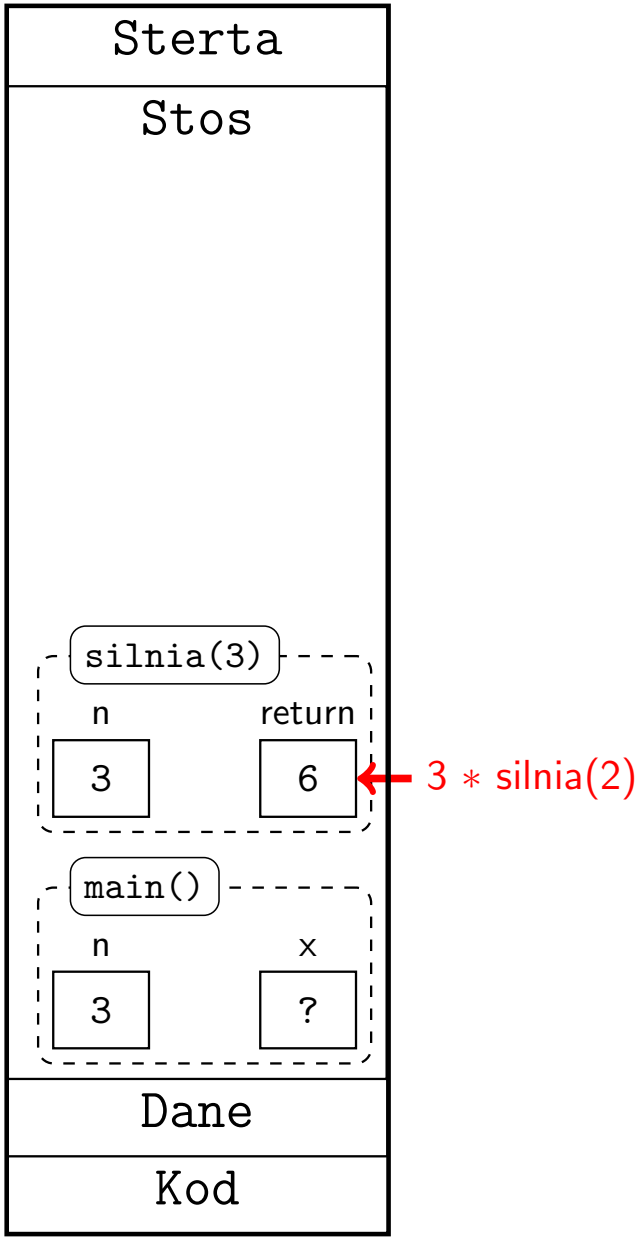
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



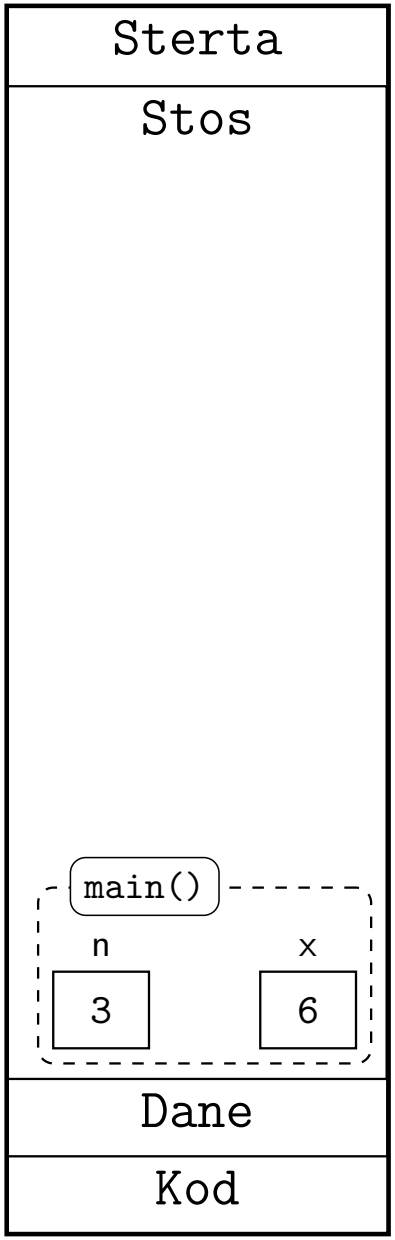
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n \cdot (n-1)!$$

```
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 }
```



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

```
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 }
```



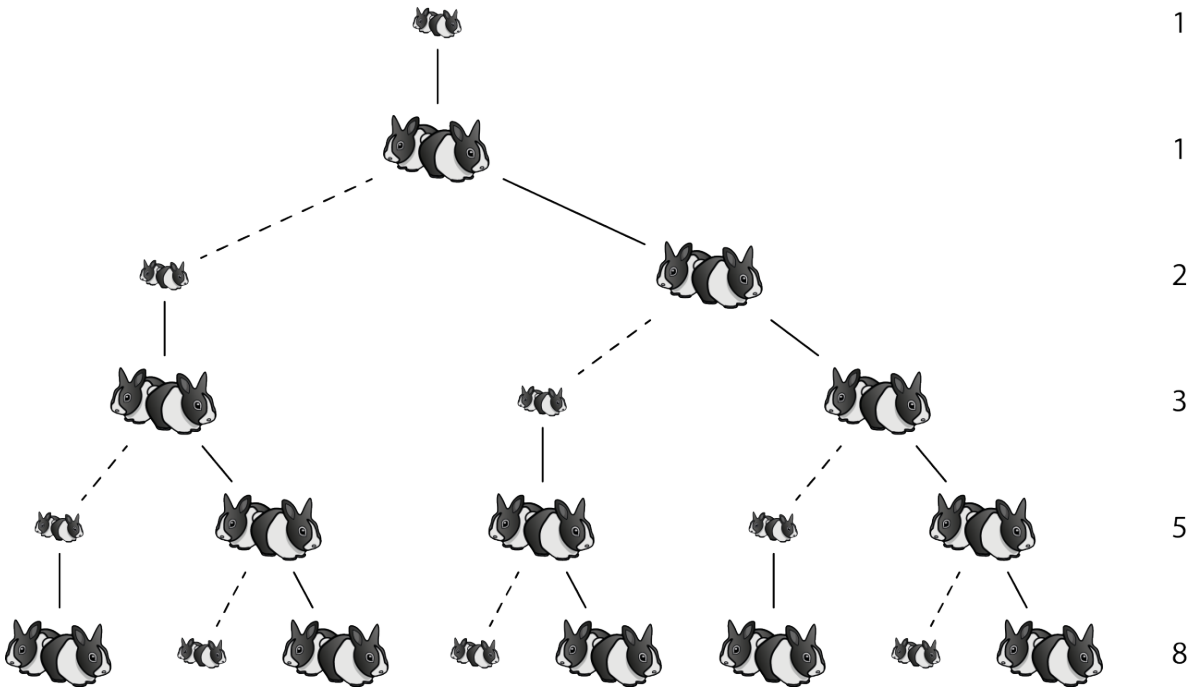
Zadanie o rozmnażaniu królików

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?

Ile par królików będzie w populacji po n miesiącach?



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

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots,$$

Ciąg Fibinacciego

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

Wersja z rekurencją

```
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

Wersja iteracyjna

```
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 dwie lub więcej wartości?
  Np. funkcja wyznaczająca miejsca zerowe paraboli powinna zwracać dwa wyniki x_1 i x_2.

 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

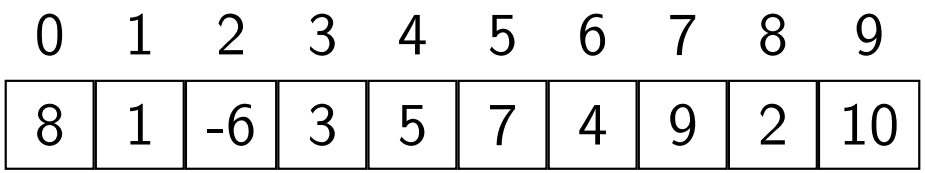
przechowuje elementy dowolnego typu

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 access*) do elementów
- operator dostępu []
- w C tablice są indeksowane od 0



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;

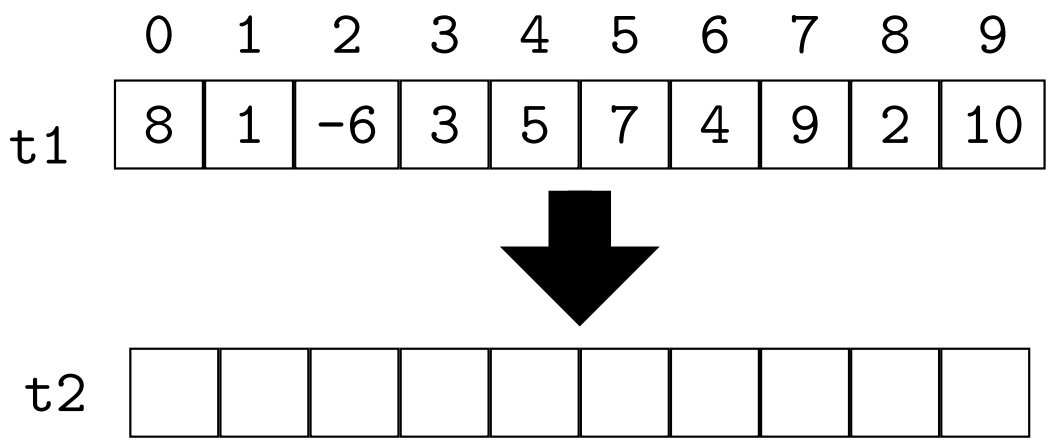
while(i<10)
{
    printf("t[%d]=", i);
    scanf("%f", &t[i]);
    i = i + 1;
}
```

Zerowanie wartości

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

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

Kopiowanie tablic

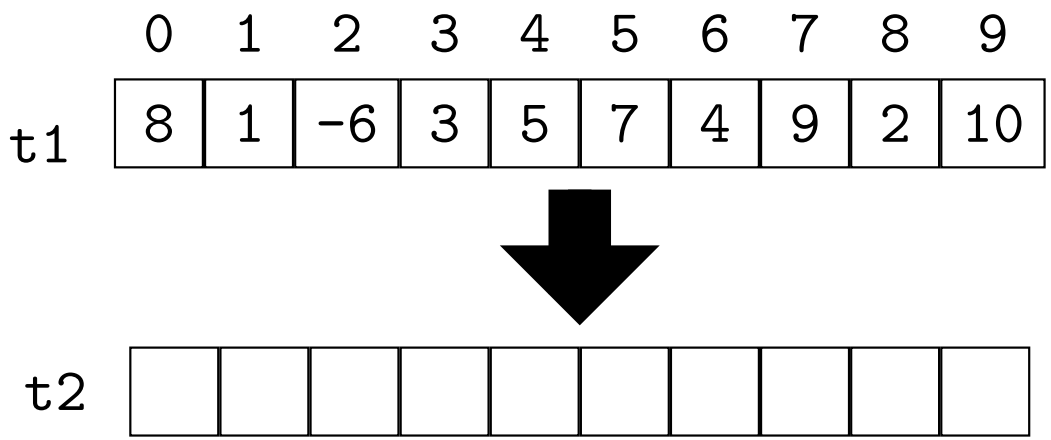



```
int t1[10];
int t2[10];

t1 = t2;
```

Źle !

Kopiowanie tablic




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

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

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

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

```
1  #include<stdio.h>
2
3  void funkcja(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 }
```

☞ funtab.c

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.

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);

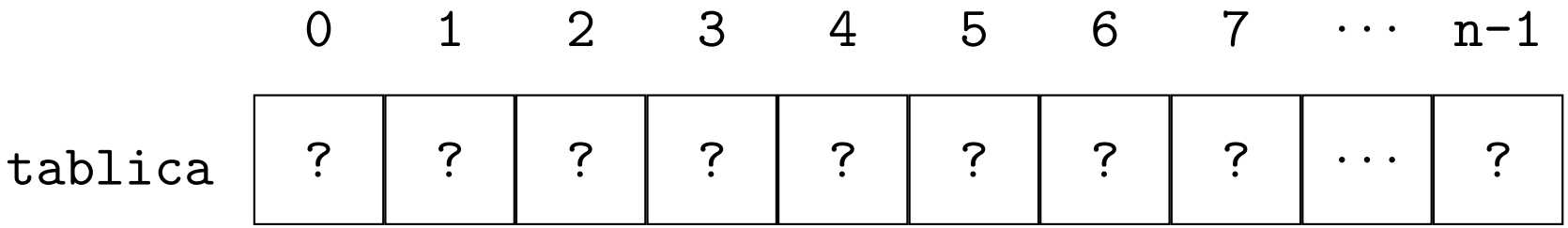
    x = max(t[10], 10);
    x = max(t[], 10);
    x = max(float t[], int n);
}
```

Dobrze

Źle

Przykład: przeszukiwanie liniowe

Problem: w zbiorze zawierającym n elementów odnajdź element x.



- czy x znajduje się w tablicy?
- jeśli tak, to na którym miejscu (indeksie)?
- co jeśli x występuje wielokrotnie?

Algorithm Przeszukiwanie liniowe

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

Wynik: pozycja pierwszego znalezionego 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

```c
int szukaj(const int t[], int n, int x, int i)
{
    while (i < n)
    {
        if (t[i] == x) return i;
        i = i + 1;
    }
    return -1;
}
```

☞ przeszukiwanie.c

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

Algorithm Przeszukiwanie liniowe z wartownikiem

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

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

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

Przykład w C: przeszukiwanie liniowe z wartownikiem

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

☞ przeszukiwanie2.c

Typ złożony `struct`

- przechowuje zmienne dowolnego typu
- **pole** struktury to pojedynczy element składowy
- pola są identyfikowane nazwami
- operator dostępu bezpośredniego `.` (kropka)
- struktury ułatwiają organizację danych $\rightarrow$ zalążek obiektywiści
- 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);

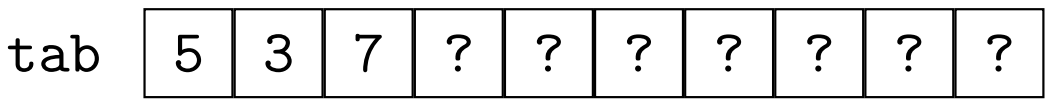
int main()
{
    struct zespolona z1, z2, z3;
    struct student s;

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

    return 0;
}
```

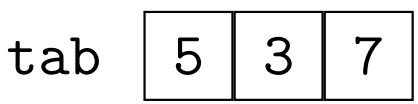
Inicjalizacja elementów tablicy

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



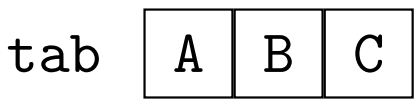
Gdy pominiemy rozmiar tablicy to jest on wyznaczany automatycznie

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



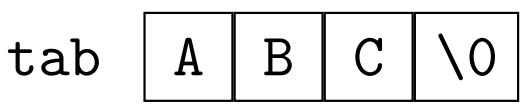
Tablica znaków

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



Napis (łańcuch znakowy)

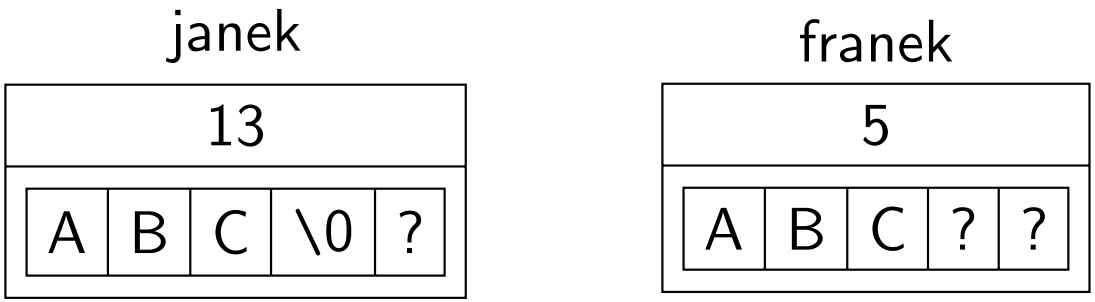
```
int tab[] = "ABC";
```



Inicjalizacja wartości struktur

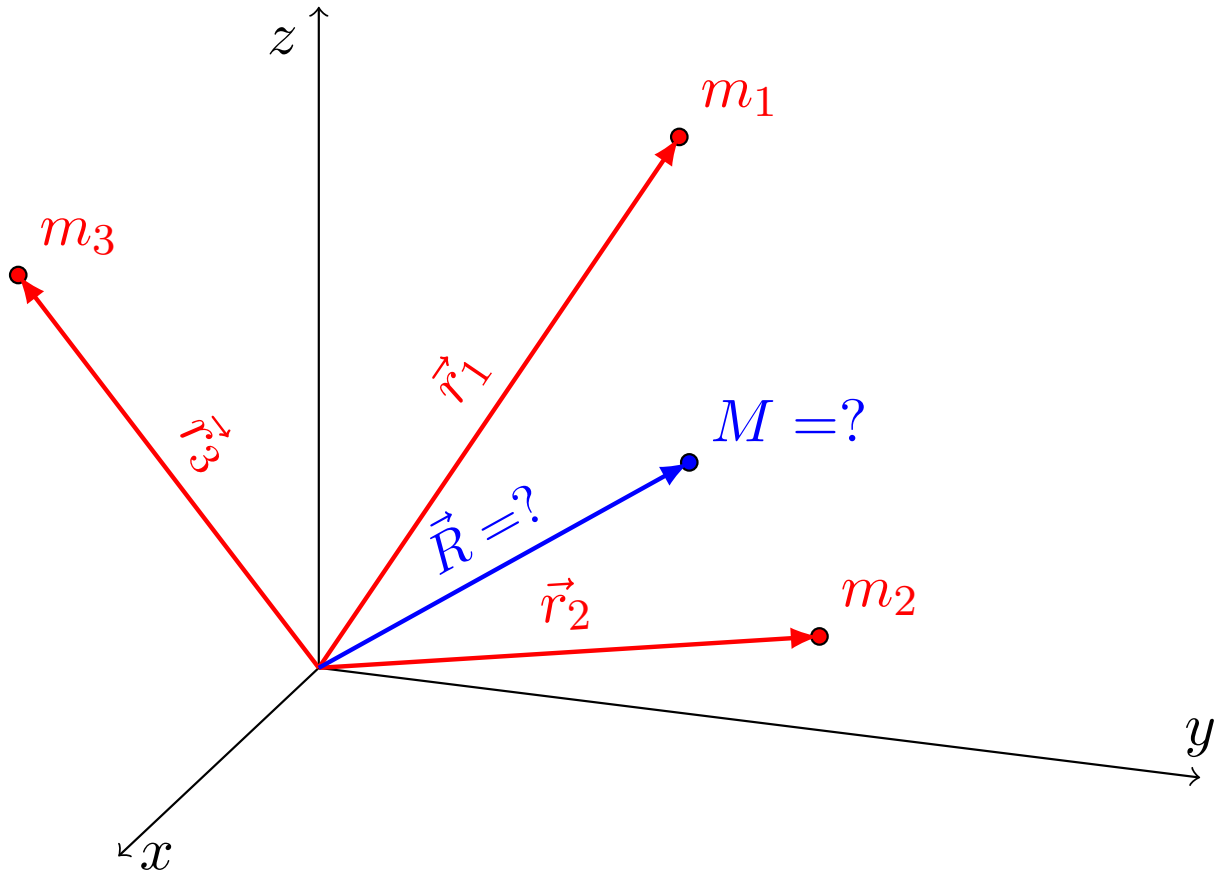
```
struct student
{
    int numer;
    char nazwisko[5];
};

struct student janek = { 13, "ABC"};
struct student 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: wyznacz położenie środka masy $\vec{R}$ oraz masę całkowitą M układu zawierającego n punktów materialnych.



Punkt materialny

$$p_i = \{m_i, \vec{r_i}\}, \qquad \vec{r_i} = [x_i, y_i, z_i]$$

Środek masy dwóch punktów

$$\vec{r}_{12} = \frac{m_1\vec{r}_1 + m_2\vec{r}_2}{m_1 + m2}, \qquad m_{12} = m_1 + m_2$$

Środek masy n punktów

$$\vec{R} = \frac{\sum_{i=1}^{n} m_i\vec{r_i}}{\sum_{i=1}^{n} m_i}, \qquad M = \sum_{i=1}^{n} m_i$$

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

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

struct punkt srodek(struct punkt p1, struct punkt p2)
{
     struct punkt sm;
     sm.m = p1.m + p2.m;
     sm.x = ( p1.m * p1.x + p2.m * p2.x ) / sm.m;
     sm.y = ( p1.m * p1.y + p2.m * p2.y ) / sm.m;
     sm.z = ( p1.m * p1.z + p2.m * p2.z ) / sm.m;
     return sm;
}
```

☞ sm2.c

Środek masy 2 punktów


```
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 }
```

Algorithm Środek masy n punktów materialnych

Dane wejściowe: zestaw punktów $\{p_1, p_2, \ldots, p_n\}$ 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: $p_0 \leftarrow srodek(p_i, p_0)$
6: $i \leftarrow i + 1$
7: **wypisz** p_0

Środek masy n punktów

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

☞ sm2.c

Model układu n punktów materialnych

Tablica tablic

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

Przykład: masa punktu nr 42

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

☞ sm4.c

```
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 < N);
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 }
```

☞ sm6.c

- **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
- Inne typy złożone: unie, pola bitowe, typ wyliczeniowy (zob. wykład 11)


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

Adresy zmiennych

```c
#include <stdio.h>

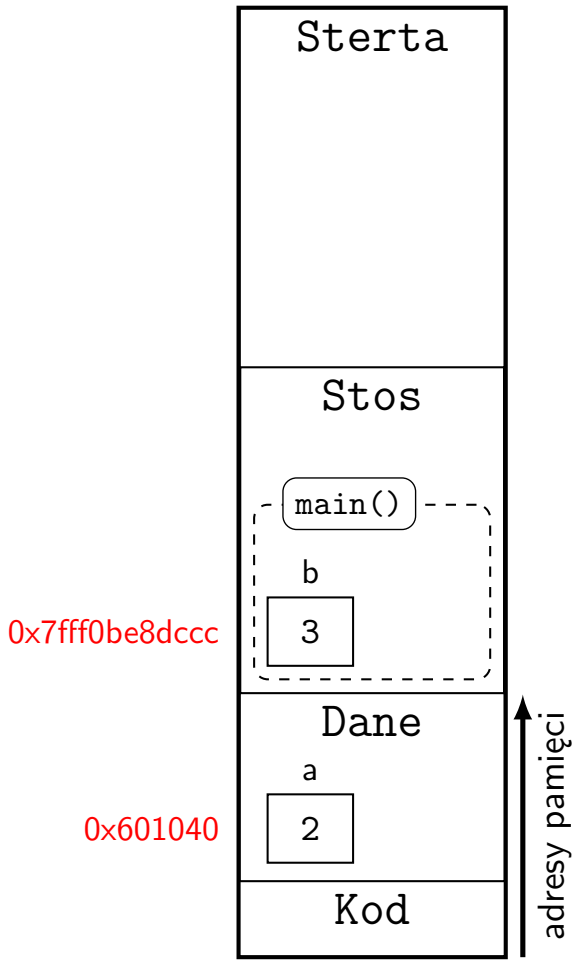
int a = 2;

int main()
{
    int b = 3;

    printf("adres zmiennej a %p\n", &a);
    printf("adres zmiennej b %p\n", &b);

    return 0;
}
```

☞ adresy.c



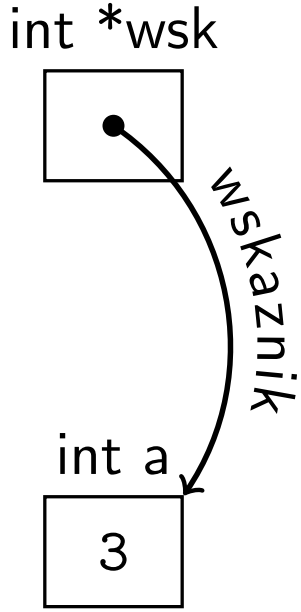
Przykładowy wynik:

```
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 wskaźnikowej określa typ wartości wskazywanej

np. `int*`, `void*`, `struct punkt*`

Deklaracja

```
typ *identyfikator;
```

lub

```
typ* identyfikator;
```

Przykład

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

Operator referencji & zwraca adres zmiennej

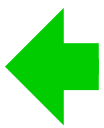
Przykład

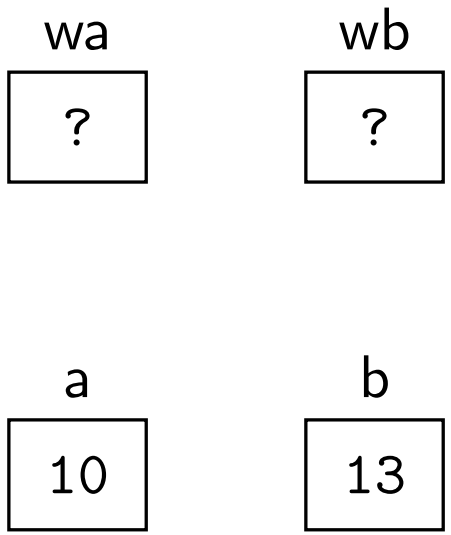
```
1  int a = 10;
2  int b = 13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```

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;
```



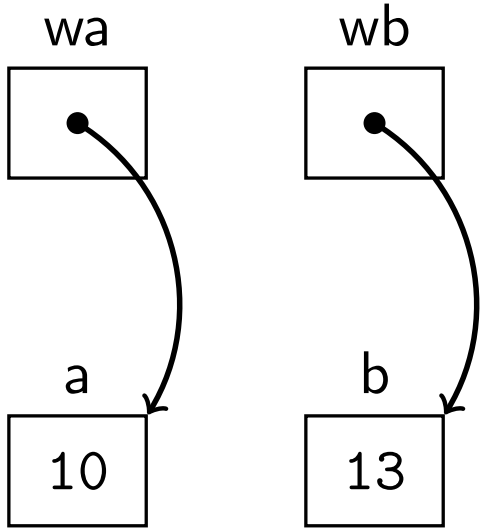


Operator referencji & zwraca adres zmiennej

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;
```

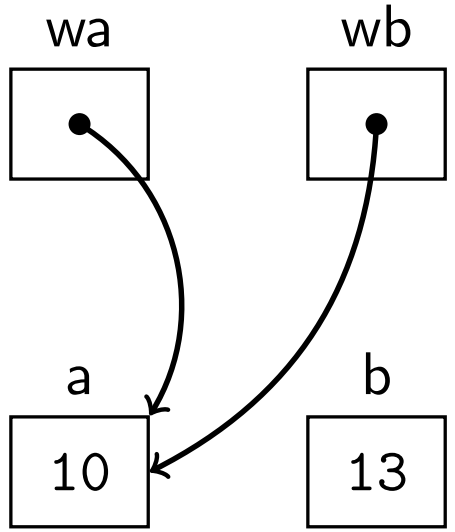




Operator referencji & zwraca adres zmiennej

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

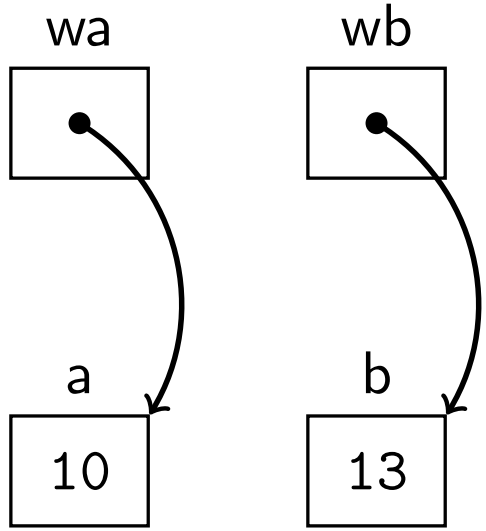
Przykład

```
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 wb = wa;
11 *wb = *wb + 1;
```

Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

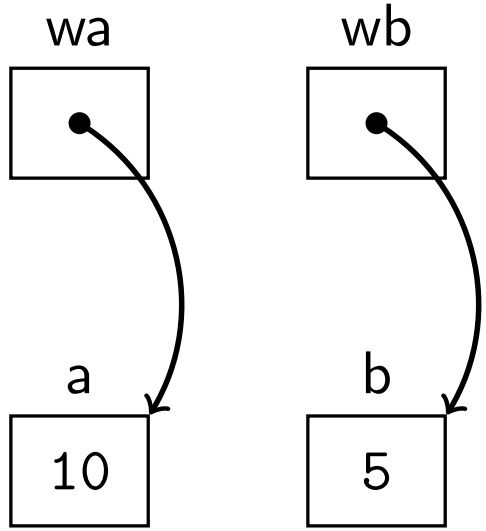
```
1   int a = 10;
2   int b = 13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wb = wa;
11  *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

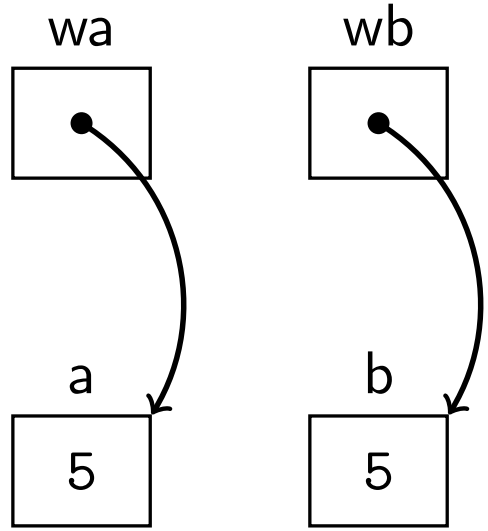
```
1   int a = 10;
2   int b = 13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wb = wa;
11  *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

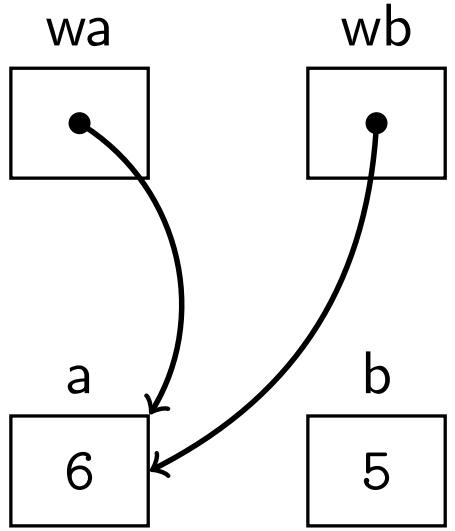
```
1   int a = 10;
2   int b = 13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wb = wa;
11  *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

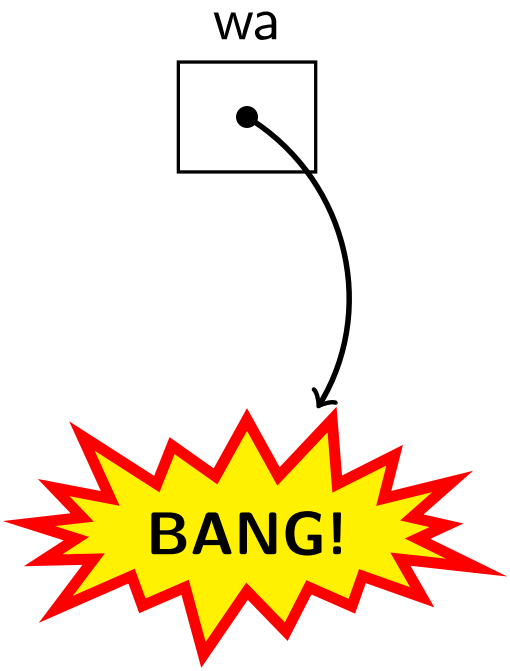
```
1   int a = 10;
2   int b = 13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wb = wa;
11  *wb = *wb + 1;
```



Uważaj na niezainicjowane zmienne wskaźnikowe.

Przykład

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



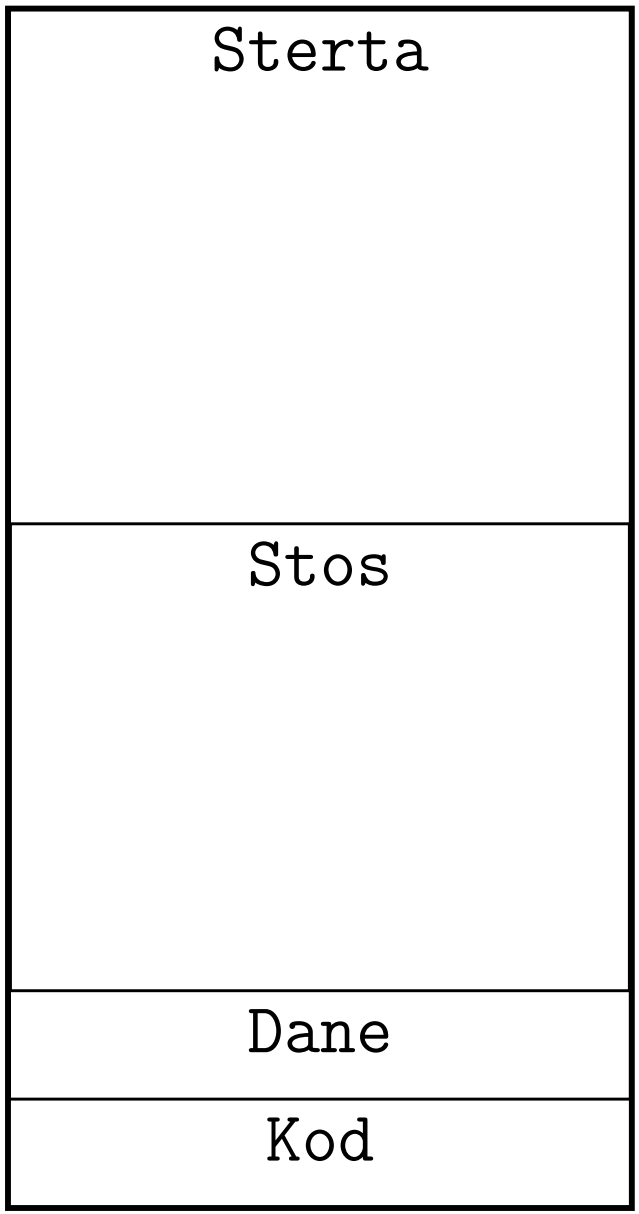
NULL to adres 0

```
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

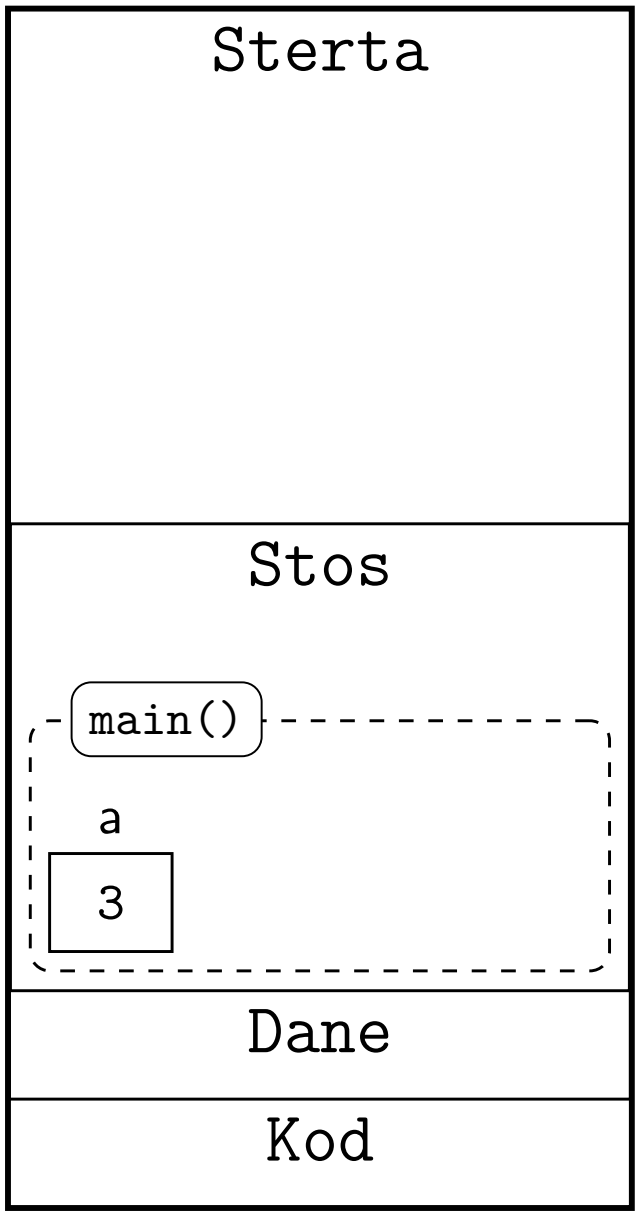
Wskaźniki jako argumenty funkcji

```
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 }
```



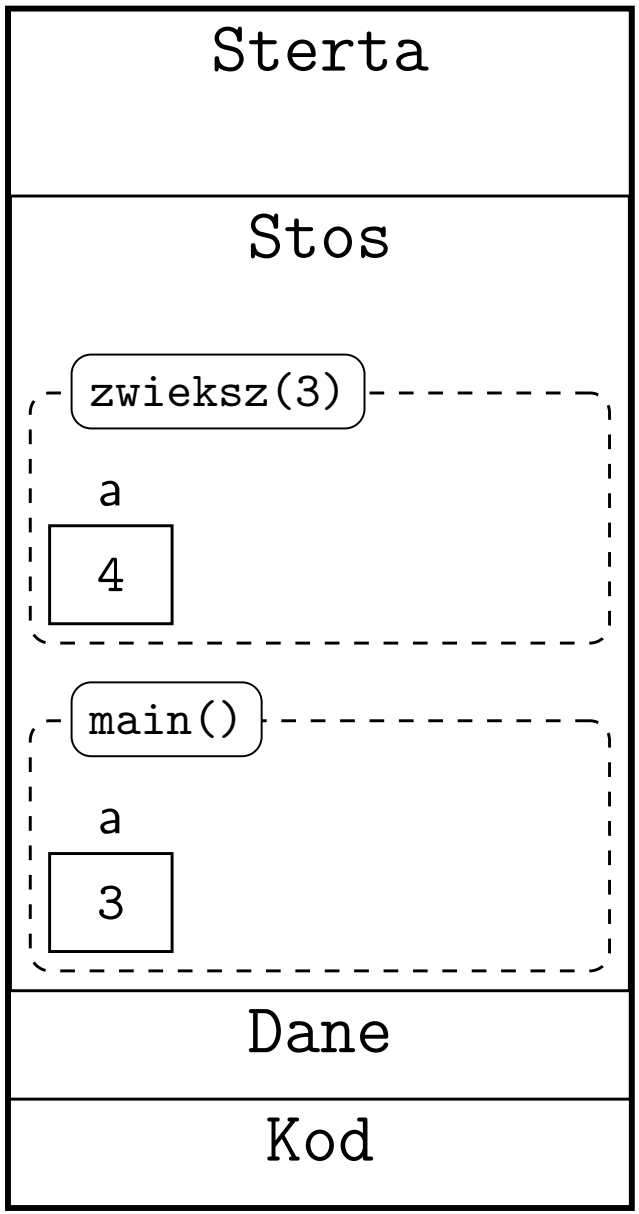
Wskaźniki jako argumenty funkcji

```
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 }
```



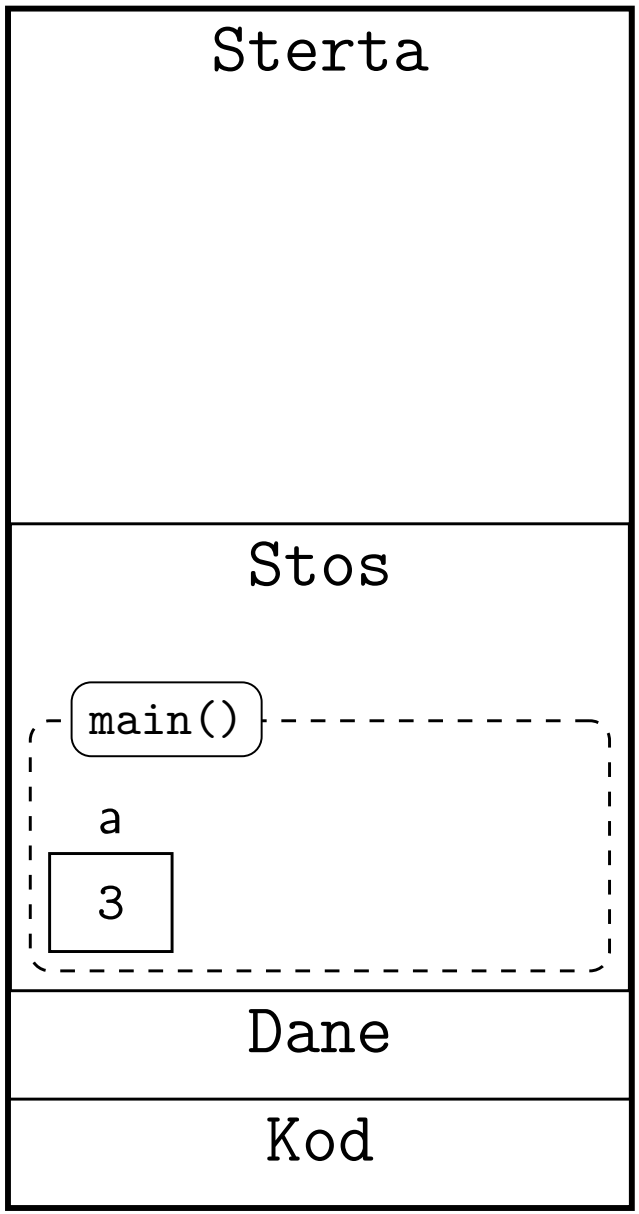
Wskaźniki jako argumenty funkcji

```
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 }
```



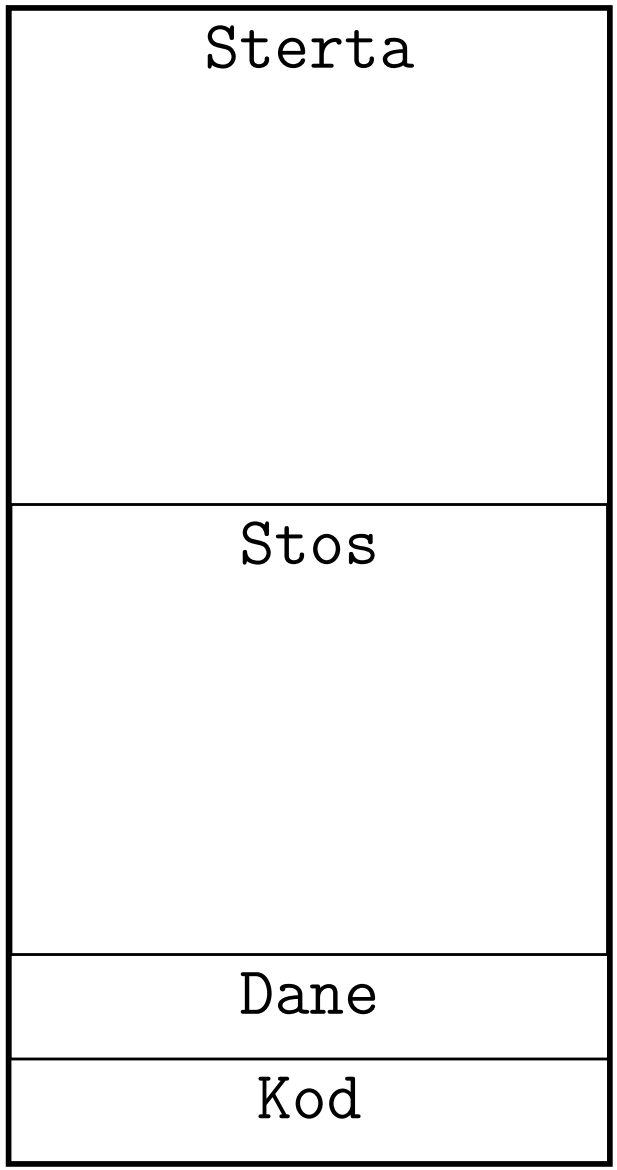
Wskaźniki jako argumenty funkcji

```
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 }
```



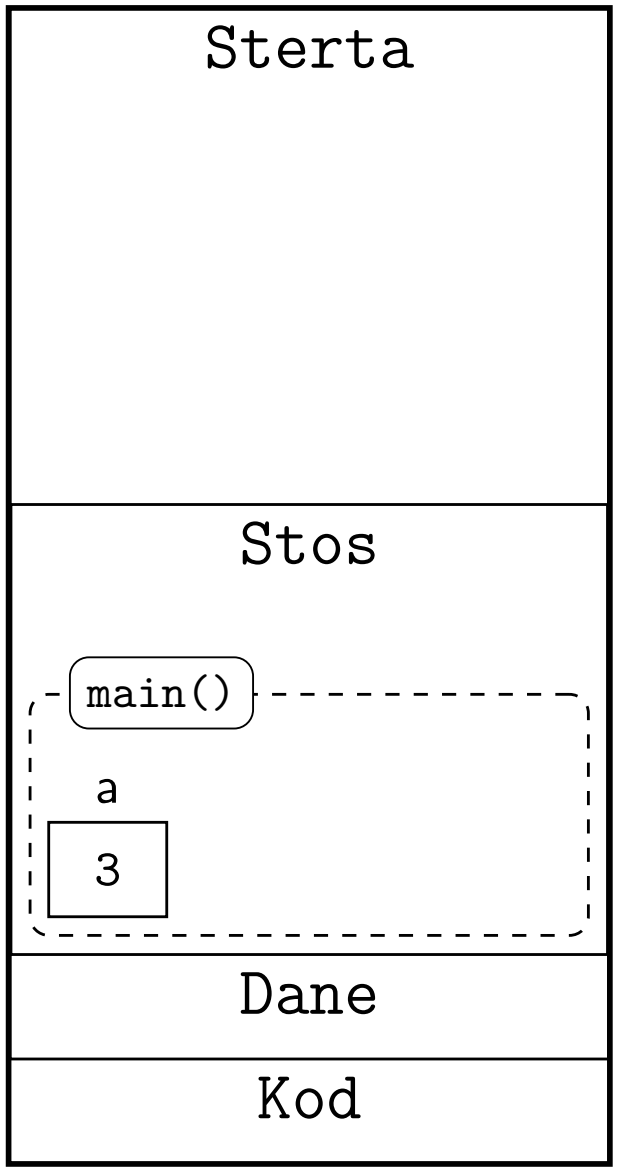
Wskaźniki jako argumenty funkcji

```
 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  }
```



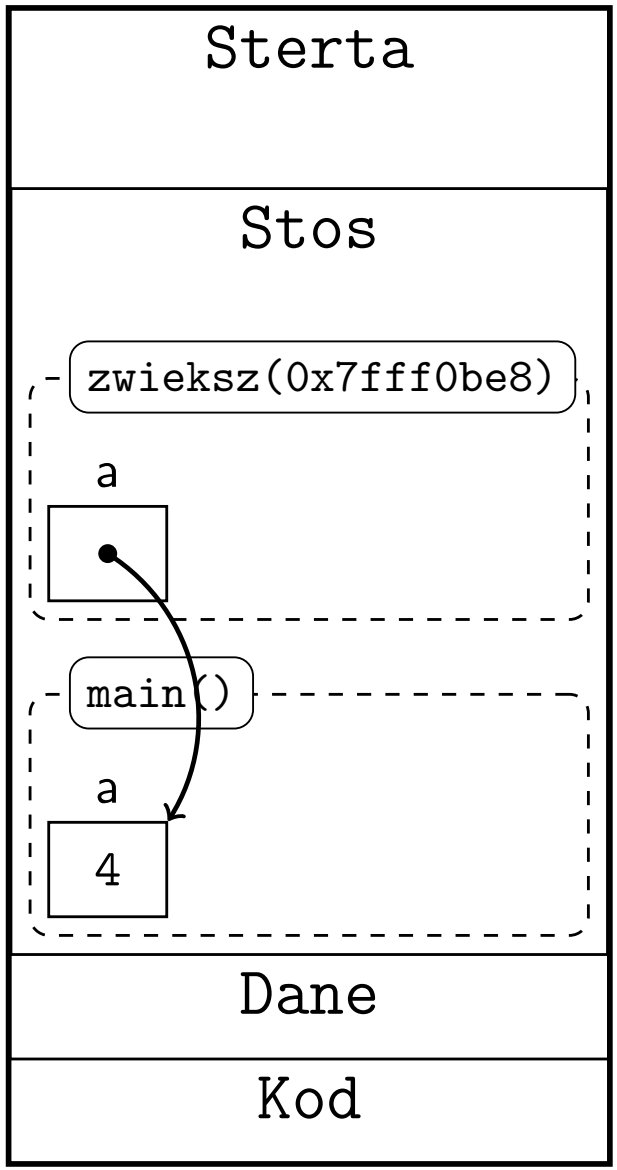
Wskaźniki jako argumenty funkcji

```
 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  }
```

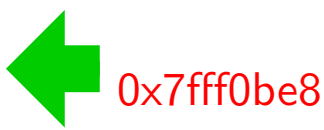


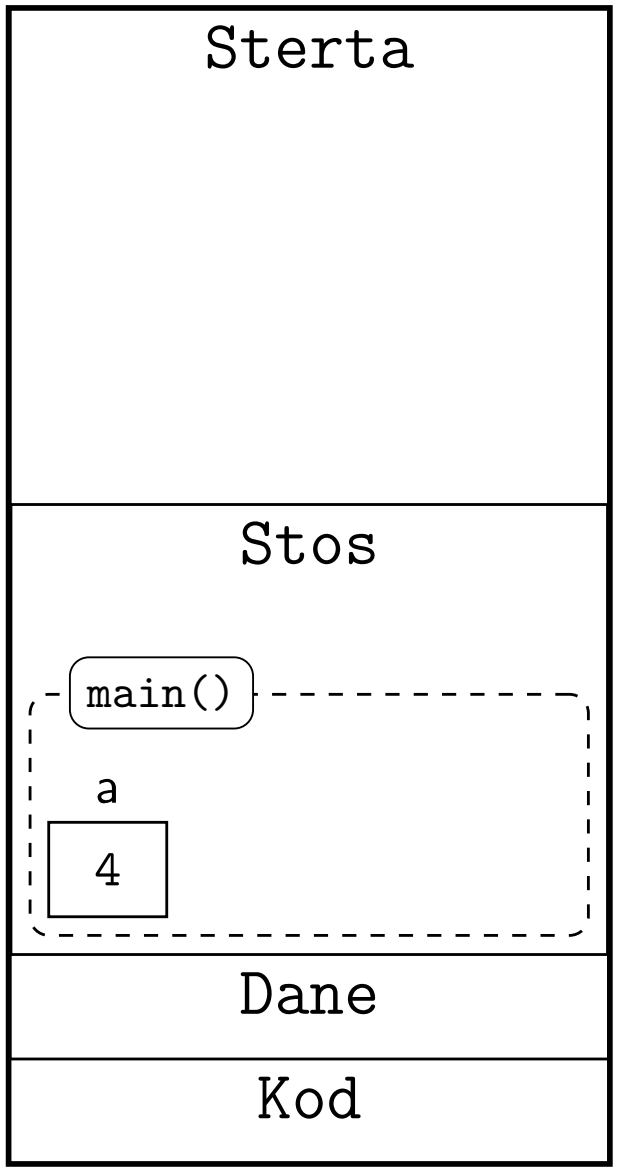
Wskaźniki jako argumenty funkcji

```
 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  }
```



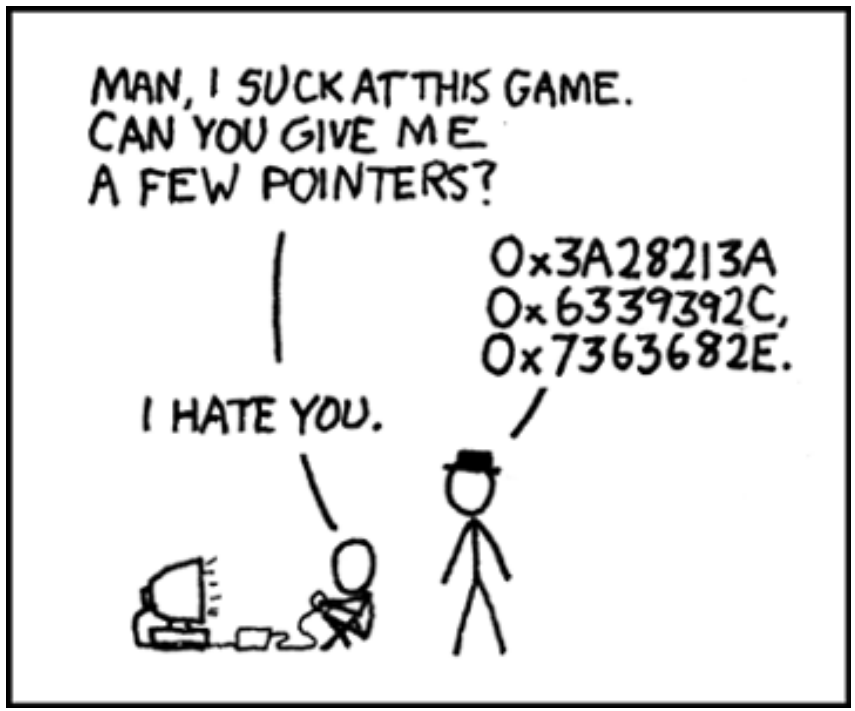
```
 1  #include<stdio.h>
 2
 3  void zwieksz(int *a)
 4  {
 5      *a = *a + 1;
 6  }
 7
 8
 9  int main()
10  {
11      int a = 3;
12
13      zwieksz(&a);
14      printf("%d\n", a);
15
16      return 0;
17  }
```





Wskaźniki jako argumenty funkcji

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



http://xkcd.com

Problem: znajdź wartość minimalną i maksymalną w zbiorze liczb

Algorithm Wyznaczanie maksimum i minimum

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

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** $x \in \{x_2, \ldots, x_n\}$ **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}

Element minimalny i maksymalny

Przykład w C

```
1  void minmax(const 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 !

Algorithm Wyznaczanie maksimum i minimum

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

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

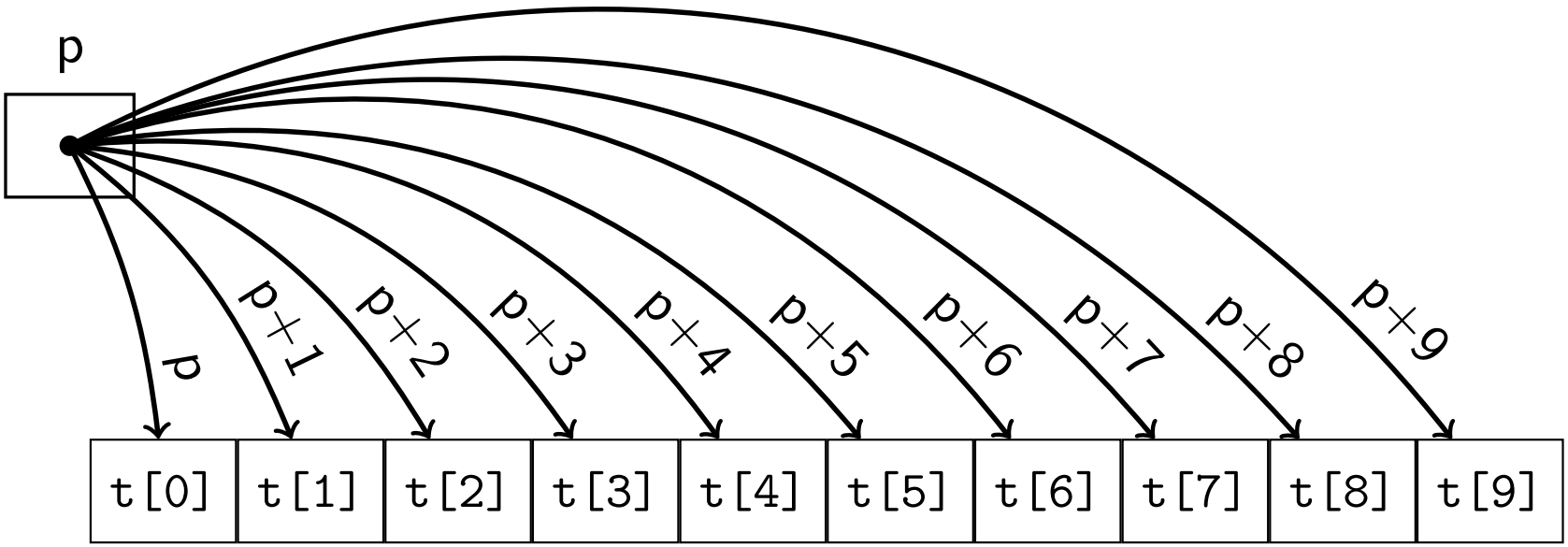
1: $\mathcal{A} \leftarrow \emptyset, \quad \mathcal{B} \leftarrow \emptyset$
2: **dla** $i = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$ **wykonuj**
3: **jeżeli** $x_{2i-1} < x_{2i}$ **wykonaj**
4: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i-1}\}, \qquad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i}\}$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

```
1  void minmax(const 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

```
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
```

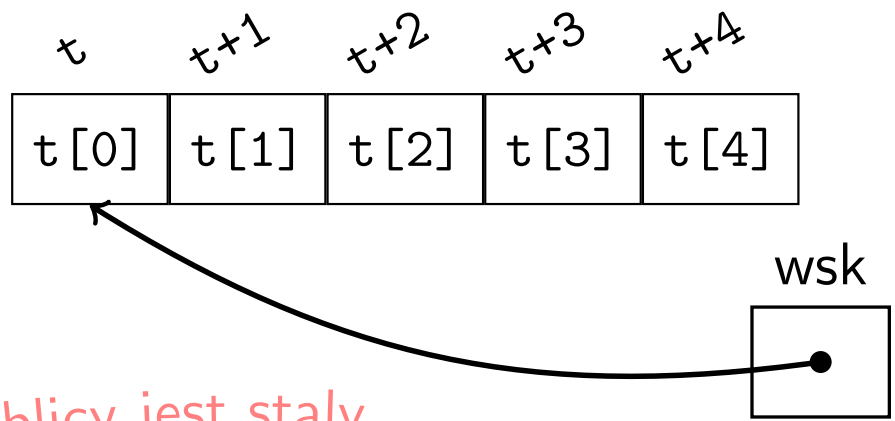
Tablica jest wskaźnikiem !

Zmienna tablicowa zawiera stały adres pierwszego elementu tablicy

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;
```

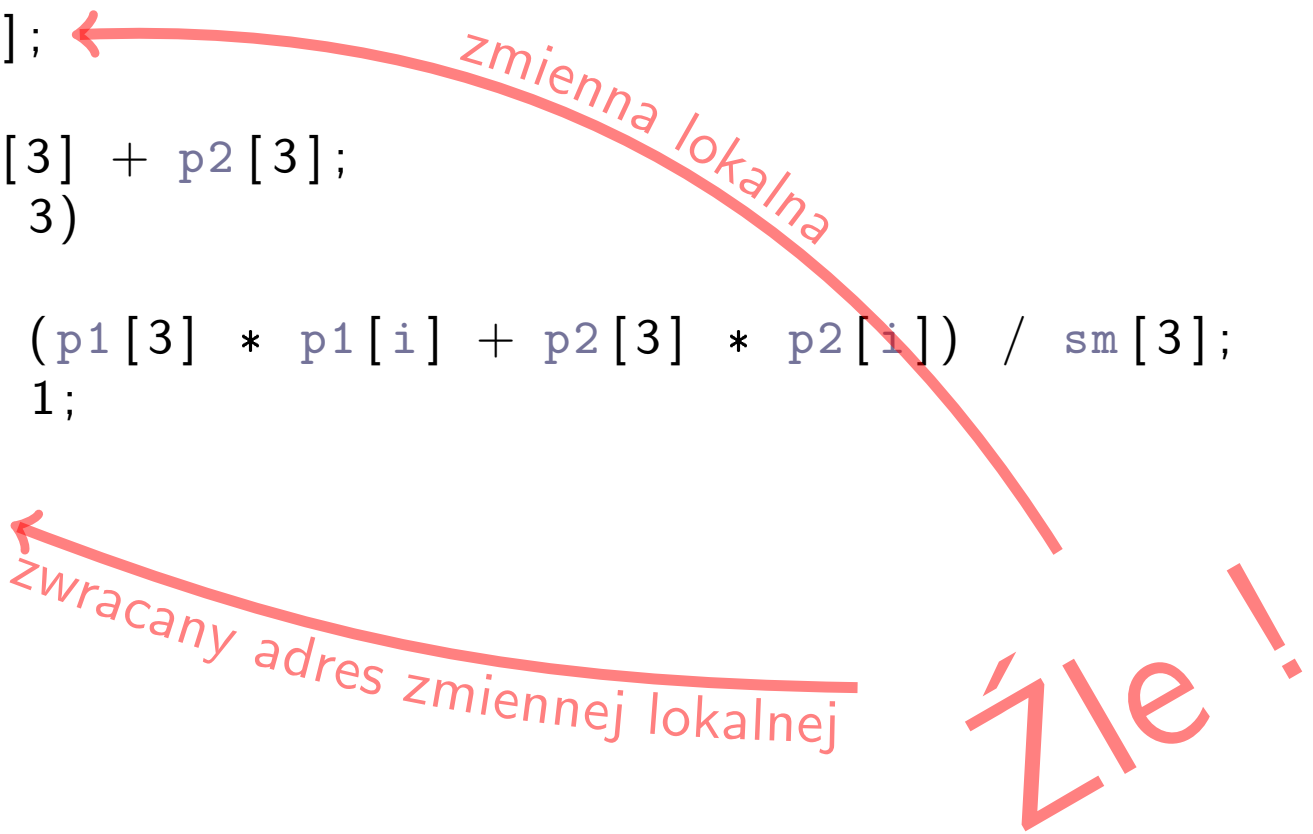


Adres tablicy jest staly

Źle !

Ponownie środek masy 2 punktów

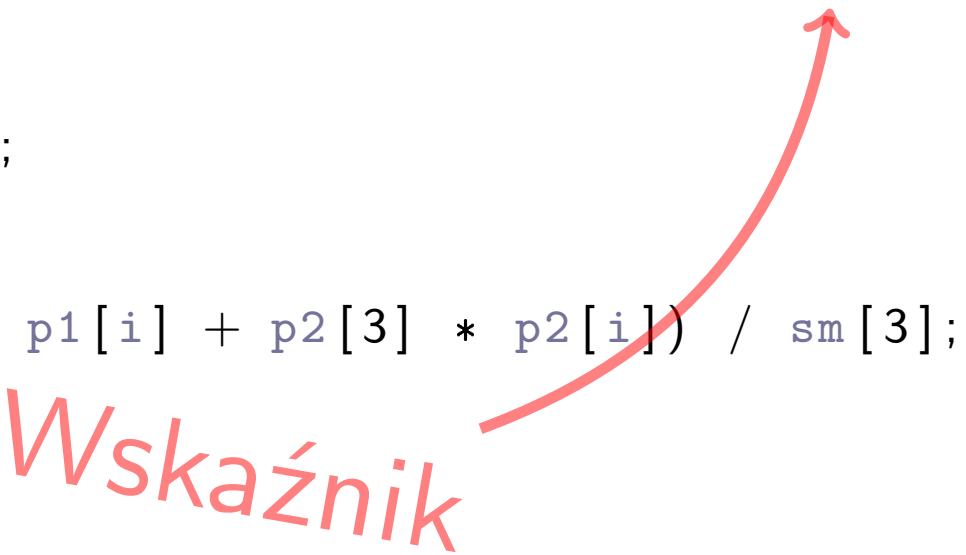
```
float* srodek(float p1[], float p2[])
{
    float sm[4];
    int i = 0;
    sm[3] = p1[3] + p2[3];
    while (i < 3)
    {
        sm[i] = (p1[3] * p1[i] + p2[3] * p2[i]) / sm[3];
        i = i + 1;
    }
    return sm;
}
```



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 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

Algorithm Wyznaczanie dominanty (mody) - algorytm naiwny

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

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

```c
int dominanta(const int *t, int n, int *c)
{
    int i, j, k, x;

    *c = 0;
    i = 0;
    while (i < n)
    {
        k = 0;
        j = 0;
        while (j < n)
        {
            if (t[i] == t[j]) k = k + 1;
            j = j + 1;
        }
        if (k > *c)
        {
            *c = k;
            x = t[i];
        }
        i = i + 1;
    }
    return x;
}
```

☞ dominanta1.c

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 wcześniej, na pozycji $n - k$

```c
int dominanta2(const int *t, int n, int *c)
{
    int i, j, k, x;

    *c = 0;
    x = t[0]-1;
    i = 0;
    while (i < n - *c)
    {
        if (t[i] != x)
        {
            k = 1;
            j = i + 1;
            while (j < n)
            {
                if (t[i] == t[j]) k = k + 1;
                j = j + 1;
            }
            if (k > *c)
            {
                *c = k;
                x = t[i];
            }
        }
        i = i + 1;
    }
    return x;
}
```

☞ dominanta2.c

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`.

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

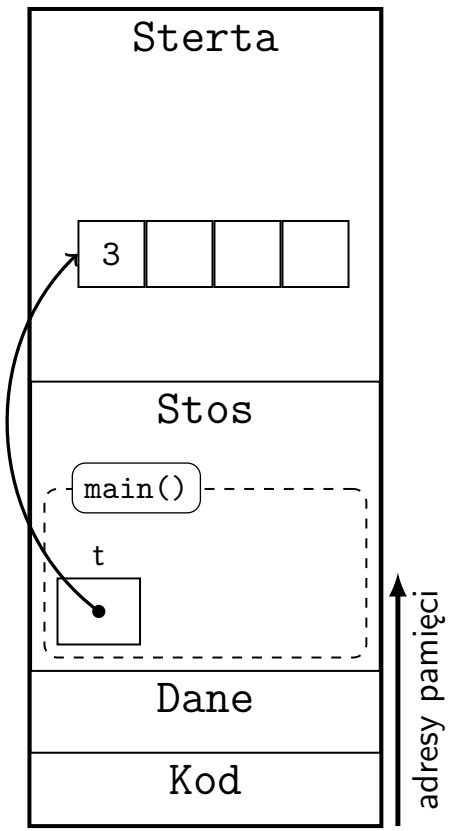
int main()
{
    float *t;

    t = malloc(4 * sizeof(float));
    if (t == NULL)
    {
        printf("Blad alokacji pamieci.\n");
        exit(1);
    }

    *t = 3;

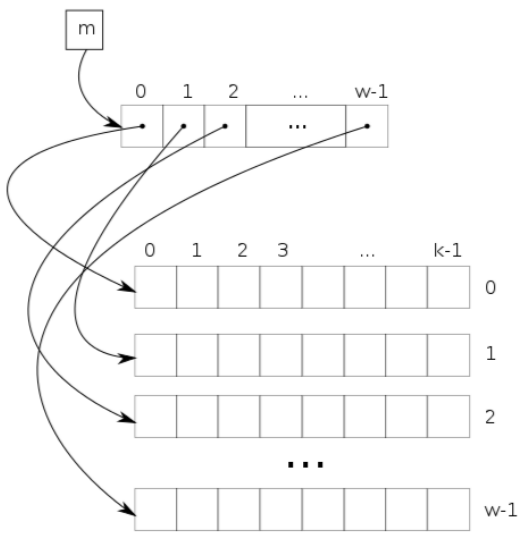
    free(t);

    return 0;
}
```



Tablice wskaźników (macierz przydzielona dynamicznie):

```
1  float **utworz_macierz(int w, int k)
2  {
3     float **m;
4     int i;
5     m = (float**) malloc(w * sizeof(float*));
6
7     for(i=0 ; i<w ; i++)
8        m[i] = (float*) malloc(k * sizeof(float));
9
10    return m;
11 }
```



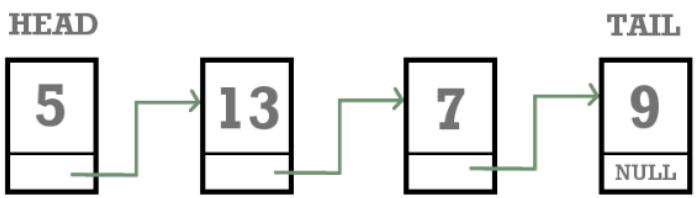
Lista jednokierunkowa:

```
struct element {
  int wartosc;
  struct element *nastepny;
};
```

Przykład tworzenia elementu listy:

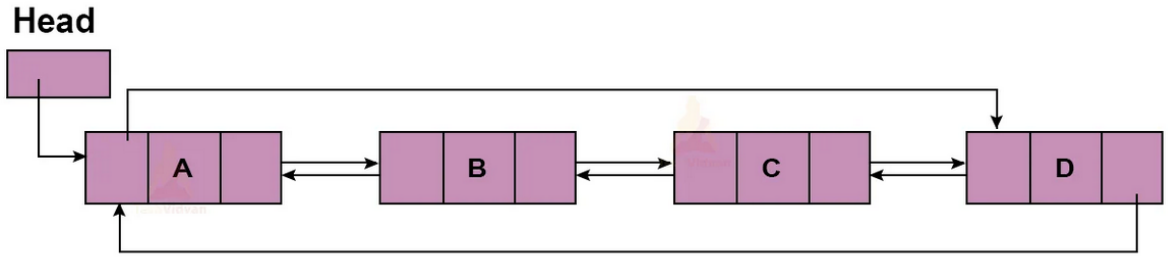
```
struct element *glowa = NULL;

struct element *nowy = malloc(sizeof(struct element));
nowy->wartosc = 10;
nowy->nastepny = glowa;
glowa = nowy;
```

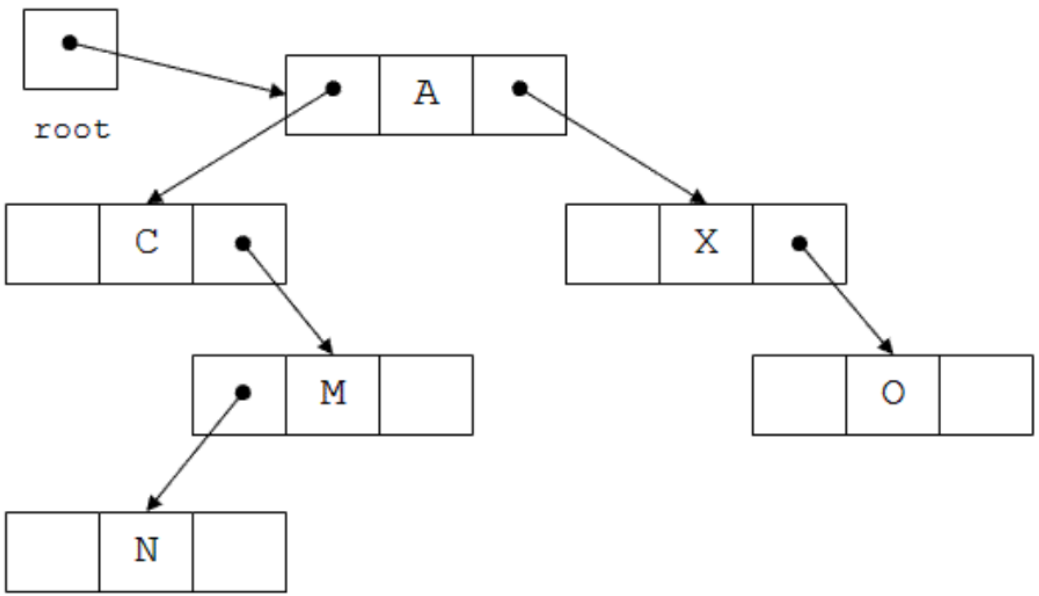


Rys. https://binarnie.pl/lista-jednokierunkowa/

Lista dwukierunkowa:



Drzewo binarne:



Źródło:https://techvidvan.com/tutorials/doubly-linked-list/

- **Wskaźnik** to adres zmiennej w pamięci a **zmienna wskaźnikowa** to zmienna przechowująca adres
- Operator **referencji** (adresowania) `&` zwraca adres zmiennej, `&x` to adres zmiennej `x`
- Operator **dereferencji** (wyłuskania) `*` udostępnia wartość ze wskazanego adresu,
  `*wsk` udostępnia wartość wskazaną przez zmienną `wsk`
- Deklaracja zmiennej wskaźnikowej również zawiera znak `*`

```
int *wsk;
float* wsk2;
```

- Tablice to wskaźniki `t[i]` $\Leftrightarrow$ `*(t+i)`
- Użycie zmiennej wskaźnikowej w argumencie funkcj umożliwia modyfikację zmiennej wywołującej funkcji
- Uważaj na co wskazujesz! Odwołanie pod nieprawidłowy adres może prowadzić do błędów w programie.


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** (***bi**nary **d**igit*) 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*) $1\text{B} = 8\text{b}$

- wg. standardu C: najmniejsza porcja adresowalnej pamięci
- stała `CHAR_BIT` z pliku `limits.h`
- *Najmniejsza porcja danych, którą może „ugryźć" komputer.*
- utożsamiany ze znakiem (zmienna typu `char` ma rozmiar 1B).

Ile pamięci można zaadresować zależnie od szerokości słowa bitowego adresu?

$16\text{b} \rightarrow 2^{16}\text{B} = 64\text{KB}$
$32\text{b} \rightarrow 2^{32}\text{B} = 4294967296\text{B} \sim 4\text{GB}$
$64\text{b} \rightarrow 2^{64}\text{B} \approx 16\text{EB}$ (eksa bajty)

- układ SI: $1\text{k} = 10^3 = 1000 \neq 1024 = 2^{10} = 1K$
- przedrostki dwójkowe zgodnie z normą Międzynarodowej Komisji Elektrotechnicznej (IEC, 1998)

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

https://pl.wikipedia.org/wiki/Przedrostek_dwójkowy

Operator `sizeof` zwraca rozmiar typu lub zmiennej w bajtach

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);
```

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

struct s {
    int a;
    char b[100];
};

int main()
{
    int tab[5];

    printf("CHAR_BIT          = %d\n", CHAR_BIT);
    printf("sizeof (char)     = %d\n", sizeof (char));
    printf("sizeof (int)      = %d\n", sizeof (int));
    printf("sizeof (long)     = %d\n", sizeof (long));
    printf("sizeof (float)    = %d\n", sizeof (float));
    printf("sizeof (double)   = %d\n", sizeof (double));
    printf("sizeof (int *)    = %d\n", sizeof (int*));
    printf("sizeof (char *)   = %d\n", sizeof (char*));
    printf("sizeof (struct s) = %d\n", sizeof (struct s));
    printf("sizeof tab        = %d\n", sizeof tab);

    return 0;
}
```

☞ sizeof.c

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

struct s {
    int a;
    char b[100];
};

int main()
{
    int tab[5];

    printf("CHAR_BIT          = %d\n", CHA
    printf("sizeof (char)     = %d\n", siz
    printf("sizeof (int)      = %d\n", siz
    printf("sizeof (long)     = %d\n", sizeof (long));
    printf("sizeof (float)    = %d\n", sizeof (float));
    printf("sizeof (double)   = %d\n", sizeof (double));
    printf("sizeof (int *)    = %d\n", sizeof (int*));
    printf("sizeof (char *)   = %d\n", sizeof (char*));
    printf("sizeof (struct s) = %d\n", sizeof (struct s));
    printf("sizeof tab        = %d\n", sizeof tab);

    return 0;
}
```

Przykładowy wynik (może się różnić zależnie od architektury komputera)

```
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
```

☞ sizeof.c

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

a - baza (podstawa) systemu liczbowego

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)

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

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

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

Binarny na szesnastkowy

1 cyfrze szesnastkowej odpowiadają 4 bity

1111	1101	0111	1110
F	D	7	E

Binarny na ósemkowy

1 cyfrze ósemkowej odpowiadają 3 bity

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

Liczby ósemkowe i szesnastkowe w C

```c
#include <stdio.h>

int main()
{
    int a = 10;         /* dec */
    int b = 010;        /* oct */
    int c = 0x10;       /* hex */

    printf("dec: %d %d %d\n", a, b, c);
    printf("oct: %o %o %o\n", a, b, c);
    printf("hex: %x %x %x\n", a, b, c);

    return 0;
}
```

☞ dox.c

Problem: wyrazić liczbę dziesiętną $x_{(10)}$ w innym systemie pozycyjnym o podstawie p

binarny $p = 2$

139 % 2	1
69 % 2	1
34 % 2	0
17 % 2	1
8 % 2	0
4 % 2	0
2 % 2	0
1 % 2	1

ósemkowy $p = 8$

139 % 8	3
17 % 8	1
2 % 8	2

szesnastkowy $p = 16$

139 % 16	11 B
8 % 16	8

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

Algorithm Zapis liczby dziesiętnej w innej podstawie

Dane wejściowe: liczba całkowita $x \geqslant 0$ oraz podstawa systemu liczbowego p

Wynik: ciąg cyfr $\{t_{n-1}, \ldots, t_1, t_0\}$ reprezentujący liczbę x w systemie o podstawie p

1: $i \leftarrow 0$
2: **dopóki** $x \neq 0$ **wykonuj**
3: $\quad t_i \leftarrow x \bmod p$
4: $\quad x \leftarrow \lfloor \frac{x}{p} \rfloor$
5: $\quad i \leftarrow i + 1$
6: **zwróć** $\{t_{i-1}, \ldots, t_1, t_0\}$ ▷ w odwrotnej kolejności

```c
int zmien_podstawe(int x, int *t, int p)
{
    int i = 0;

    while (x != 0)
    {
        t[i] = x % p;
        x = x / p;
        i = i + 1;
    }
    return i;
}
```

☞ dec2all.c

Operator / dla liczb całkowitych dzieli bez reszty!

Naturalny kod binarny (NKB)

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

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

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

- 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, standard C gwarantuje minimalne rozmiary (np. `int` co najmniej 2B)
- Zakresy typów całkowitych są określone w `limits.h` (np. stała `UINT_MAX` określa maksymalną wartość typu `unsigned int`)

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

`unsigned char` i `unsigned short` promowane do `unsigned 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
37777777777
ffffffff
0
4294967296
```

☞ uint2.c

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

int main()
{
    printf("sizeof\n");
    printf("unsigned char          = %lu\n", sizeof (unsigned char));
    printf("unsigned short         = %lu\n", sizeof (unsigned short));
    printf("unsigned int           = %lu\n", sizeof (unsigned int));
    printf("unsigned long int      = %lu\n", sizeof (unsigned long int));
    printf("unsigned long long int = %lu\n", sizeof (unsigned long long int));
    printf("Zakres:\n");
    printf("UCHAR_MAX              = %u\n"  , UCHAR_MAX);
    printf("USHRT_MAX              = %u\n"  , USHRT_MAX);
    printf("UINT_MAX               = %u\n"  , UINT_MAX);
    printf("ULONG_MAX              = %lu\n" , ULONG_MAX);
    printf("ULLONG_MAX             = %llu\n", ULLONG_MAX);

    return 0;
}
```

☞ uint1.c

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

int main()
{
    printf("sizeof\n");
    printf("unsigned char
    printf("unsigned short
    printf("unsigned int
    printf("unsigned long in
    printf("unsigned long lo
    printf("Zakres:\n");
    printf("UCHAR_MAX
    printf("USHRT_MAX                = %u\n"   , USHRT_MAX);
    printf("UINT_MAX                 = %u\n"   , UINT_MAX);
    printf("ULONG_MAX                = %lu\n"  , ULONG_MAX);
    printf("ULLONG_MAX               = %llu\n", ULLONG_MAX);

    return 0;
}
```

Wartości mogą się róznić 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
```

☞ uint1.c

U2, kod uzupełnień do dwóch

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

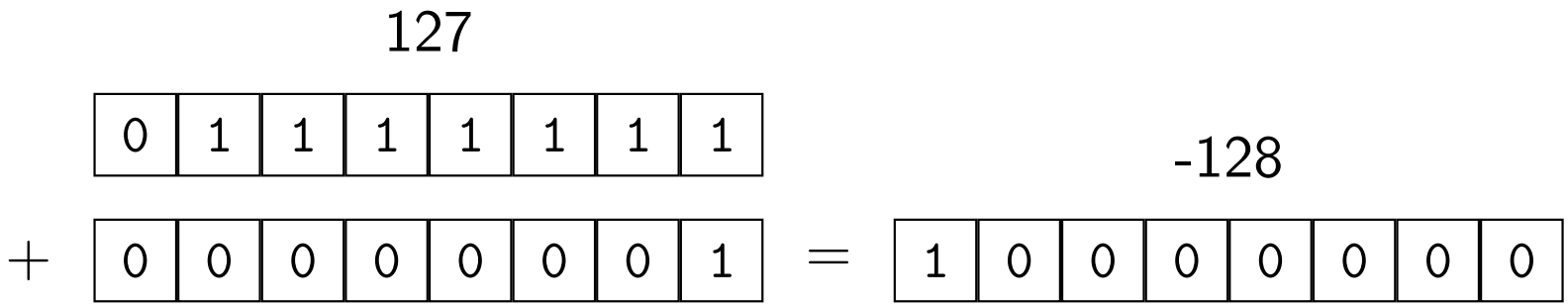
- liczby ze znakiem, najstarszy bit b_{n-1} odpowiada za znak
- zakres $\left[-2^{n-1}, 2^{n-1} - 1\right]$
- typy całkowite ze znakiem w C:

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

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

Porównanie reprezentacji NKB i U2

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



- **Nadmiar** (*overflow*) - przekroczenie zakresu wynikające ze skończonej liczby bitów reprezentujących liczbę.
- Zazwyczaj błąd nie jest sygnalizowany.
- Spowodowany przez dodawanie dużych liczb dodatnich (lub odejmowanie liczb ujemnych). W U2 widoczna zmiana znaku.
- Zapis liczby typu bardziej pojemnego do zmiennej typu mniej pojemnego (np. `float` do `char`) może spowodować nadmiar.
- Dodawanie liczby ujemnej i dodatniej nie powoduje nadmiaru.
- 4 czerwiec 1996, pierwszy lot rakiety Ariane 5 zakończony zniszczeniem w skutek nadmiaru, straty 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)	$[-9223372036854775808,$
long long int	8	$9223372036854775807]$

- Zakresy typów całkowitych są określone w `limits.h`
- Specyfikacja formatu `printf` i `scanf`:

specyfikator	typ
`d, i`	`int, char, short`
`ld, li`	`long int`
`lld, lli`	`long long int`

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

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

int main()
{
    printf("sizeof:\n");
    printf("char           = %lu\n", sizeof (char));
    printf("short          = %lu\n", sizeof (short));
    printf("int            = %lu\n", sizeof (int));
    printf("long int       = %lu\n", sizeof (long int));
    printf("long long int = %lu\n", sizeof (long long int));
    printf("Zakresy:\n");
    printf("CHAR_MIN       = %d\n"  , CHAR_MIN);
    printf("CHAR_MAX       = %d\n"  , CHAR_MAX);
    printf("SHRT_MIN       = %d\n"  , SHRT_MIN);
    printf("SHRT_MAX       = %d\n"  , SHRT_MAX);
    printf("INT_MIN        = %d\n"  , INT_MIN);
    printf("INT_MAX        = %d\n"  , INT_MAX);
    printf("LONG_MIN       = %ld\n" , LONG_MIN);
    printf("LONG_MAX       = %ld\n" , LONG_MAX);
    printf("LLONG_MIN      = %lld\n", LLONG_MIN);
    printf("LLONG_MAX      = %lld\n", LLONG_MAX);

    return 0;
}
```

☞ int1.c

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

int main()
{
    printf("sizeof:\n");
    printf("char
    printf("short
    printf("int
    printf("long int
    printf("long long in
    printf("Zakresy:\n")
    printf("CHAR_MIN
    printf("CHAR_MAX
    printf("SHRT_MIN
    printf("SHRT_MAX
    printf("INT_MIN
    printf("INT_MAX
    printf("LONG_MIN
    printf("LONG_MAX
    printf("LLONG_MIN
    printf("LLONG_MAX

    return 0;
}
```

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

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
```

☞ int1.c

Kod stałoprzecinkowy

$$3,14_{(10)} = 3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$$
$$11,001_{(2)} = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 3,125_{(10)}$$

Zapis zmiennoprzecinkowy

$$L = m \cdot p^c$$

c - cecha
p - podstawa (baza)
m - mantysa

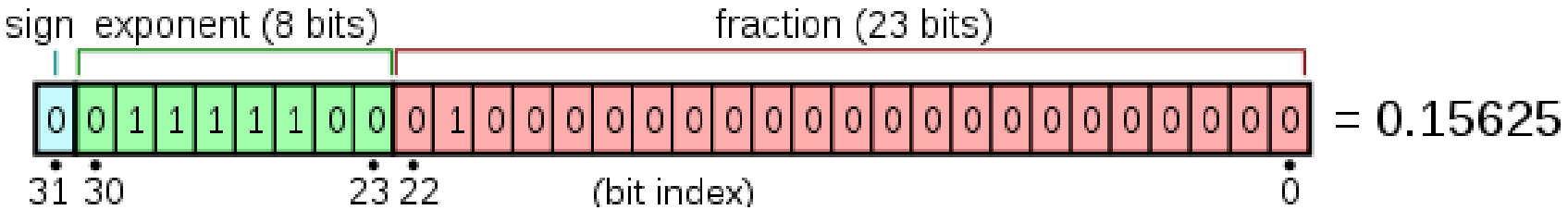
$$3,14 \cdot 10^0 = 0,0314 \cdot 10^2 = 314 \cdot 10^{-2}$$

Normalizacja: w celu uniknięcia wieloznaczności zapisu wymaga się, aby mantysa spełniała warunek:

$$1 \leqslant |m| < p$$

pojedyncza precyzja (*single precision*), `float`

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



$s = 0$

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

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

$x = (-1)^s \cdot 1.m \cdot 2^{(c-127)} = 1 \cdot 1.25 \cdot 0.125 = 0.15625$

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`
- Zakres `long double` zalezy od implementacji
  (w kompilatorze MS VC++ jest tożsamy z `double`)

Formatowanie liczb zmiennopozycyjnych za pomocą `printf`

specyfikator	znaczenie	przykład
`f`	dziesiętne	`123.45678`
`e`, `E`	notacja naukowa	`1.2345678e+2`
`g`	krótki zapis (dziesiętny lub naukowy)	`123.4657`
`Lf`, `Le`, `Lg`	dla typu `long double`	1.2345e+400
`.2f`	dokładność 2 miejsc po przecinku	`123.46`

typ `float` jest promowany do `double`

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
```

Typ zmiennopozycyjny pozwala reprezentować specjalne wartości:

- `NaN` to wartość nieokreślona (*not a number*), np.: $\frac{0}{0}$, $\sqrt{-1}$
- `Inf` to nieskończoność, np. wynik dzielenia przez 0
- Wartości `NaN` i `Inf` są propagowane w dalszych obliczeniach i nie powodują błędu wykonania programu

Operacja	Wynik
$\pm x / \pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\pm x / 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 / \pm 0$	`NaN`
$\infty - \infty$	`NaN`
$\pm\infty / \pm\infty$	`NaN`
$\pm\infty \cdot 0$	`NaN`

```c
#include <stdio.h>

int main()
{
    float x = 0.1;

    if (x == 0.1) printf ("OK, %f jest rowne 0.1\n", x);
    else printf("Nie OK, %f nie jest rowne 0.1\n", x);

    return 0;
}
```

☞ prec.c

```c
#include <stdio.h>

int main()
{
    float x = 0.1;

    if (x == 0.1) printf ("OK, %f jest rowne 0.1\n", x);
    else printf("Nie OK, %f nie jest rowne 0.1\n", x);

    return 0;
}
```

☞ prec.c

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

```c
#include <stdio.h>
#include <math.h>
#define EPS 0.000001

int main()
{
    float x = 0.1;

    if (fabs(x-0.1) < EPS)
        printf("OK, %f jest rowne 0.1\n", x);
    else
        printf("Nie OK, %f nie jest rowne 0.1\n", x);

    return 0;
}
```

☞ prec2.c

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

int main()
{
   printf("Limity typu float:\n");
   printf("sizeof(float)         : %lu\n", sizeof(float));
   printf("Najwieksza wartosc    : %e\n", FLT_MAX );
   printf("Najmniejsza dodatnia  : %e\n", FLT_MIN );
   printf("Epsilon maszynowy     : %e\n", FLT_EPSILON );
   printf("Cyfry znaczace        : %d\n", FLT_DIG );

   printf("\nLimity typu double:\n");
   printf("sizeof(double)        : %lu\n", sizeof(double));
   printf("Najwieksza wartosc    : %e\n", DBL_MAX );
   printf("Najmniejsza dodatnia  : %e\n", DBL_MIN );
   printf("Epsilon maszynowy     : %e\n", DBL_EPSILON );
   printf("Cyfry znaczace        : %d\n", DBL_DIG );

   printf("\nLimity typu long double:\n");
   printf("sizeof(long double)   : %lu\n", sizeof(long double));
   printf("Najwieksza wartosc    : %Le\n", LDBL_MAX );
   printf("Najmniejsza dodatnia  : %Le\n", LDBL_MIN );
   printf("Epsilon maszynowy     : %Le\n", LDBL_EPSILON );
   printf("Cyfry znaczace        : %d\n" , LDBL_DIG );

   return 0;
}
```

☞ float.c

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

int main()
{
    printf("Limity typu float:\n");
    printf("sizeof(float)         : %lu\n", size
    printf("Najwieksza wartosc    : %e\n", FLT_M
    printf("Najmniejsza dodatnia  : %e\n", FLT_M
    printf("Epsilon maszynowy     : %e\n", FLT_E
    printf("Cyfry znaczace        : %d\n", FLT_D

    printf("\nLimity typu double:\n");
    printf("sizeof(double)        : %lu\n", size
    printf("Najwieksza wartosc    : %e\n", DBL_M
    printf("Najmniejsza dodatnia  : %e\n", DBL_M
    printf("Epsilon maszynowy     : %e\n", DBL_E
    printf("Cyfry znaczace        : %d\n", DBL_D

    printf("\nLimity typu long double:\n");
    printf("sizeof(long double)   : %lu\n", size
    printf("Najwieksza wartosc    : %Le\n", LDBL
    printf("Najmniejsza dodatnia  : %Le\n", LDBL
    printf("Epsilon maszynowy     : %Le\n", LDBL_EPSILON );
    printf("Cyfry znaczace        : %d\n" , LDBL_DIG );

    return 0;
}
```

Wartości mogą się różnić zależnie od implementa

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

Limity typu double:
sizeof(double)        : 8
Najwieksza wartosc    : 1.797693e+308
Najmniejsza dodatnia  : 2.225074e-308
Epsilon maszynowy     : 2.220446e-16
Cyfry znaczace        : 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 znaczace        : 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) bardzo dużej i bardzo małej liczby powoduje utratę precyzji
- kolejność operacji może mieć wpływ na wynik
  $(a + b) + c \neq a + (b + c)$
- zaokrąglenia wynikające z reprezentacji binarnej, np. liczby 0.1 nie można dokładnie reprezentować w systemie binarnym
- 25 luty 1991 r., wojna w zatoce perskiej, awaria systemu antyrakietowego Patriot wywołana utratą precyzji (zegar rakiety tykał co 0.1 s.), zginęło 28 amerykańskich żołnierzy a 100 zostało rannych.

Problem: wyznaczyć sumę pierwszych n elementów szeregu harmonicznego

$$\sum_{i=1}^{n} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots \xrightarrow[n\to\infty]{} \infty$$

```c
#include <stdio.h>

int main()
{
    float x=0.0, y=0.0;
    int i=1, n;

    printf("n="); scanf("%d", &n);

    for (i=1; i <= n; i++)
        x = x + 1.0 / i;

    for (i=n; i >= 1; i--)
        y = y + 1.0 / i;

    printf("x=%f\ny=%f\n", x, y);
    return 0;
}
```

☞ szereg.c

```
n=66
x=4.774428
y=4.774427

n=600
x=6.974980
y=6.974978

n=2000
x=8.178369
y=8.178370

n=500000
x=13.690692
y=13.699607

n=1000000
x=14.357358
y=14.392652
```

- 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, WMIiM, 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 i łańcuchy znakowe

ASCII

The American Standard Code for Information Interchange, 1968 r.

- 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 systemie MS Windows koniec linii to dwa bajty `\r\n`
  - znaki o specjalnym zapisie:
    ukośnik w lewo `'\\'`,
    apostrof `'\''`,
    cudzysłów `'\"'`,
    znak zapytania `'\?'`
- Uwaga: `"A"` nie jest pojedynczym znakiem lecz tablicą znaków!

```c
#include<stdio.h>

int main()
{
    char a;

    printf("Podaj znak: ");
    scanf("%c",&a);

    printf("znak %c, dec %d, oct %o, hex %x\n",a,a,a,a);
    return 0;
}
```

☞ 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 - funkcja zwraca 0 lub 1	
`int isalnum(int c)`	litery alfabetu lub cyfry `[A-Za-z0-9]`
`int isalpha(int c)`	litery alfabetu `[A-Za-z]`
`int islower(int c)`	małe litery alfabetu `[a-z]`
`int isupper(int c)`	wielkie litery alfabetu `[A-Z]`
`int isdigit(int c)`	cyfry `[0-9]`
`int isgraph(int c)`	znaki drukowalne (ze spacją)
`int isspace(int c)`	znaki białe
`int isprint(int c)`	znaki drukowalne (bez spacji)
`int ispunct(int c)`	znaki przestankowe (drukowalne bez liter i cyfr)

Zamiana znaków	
`int tolower(int c)`	z wielkiej litery na małą
`int toupper(int c)`	z małej litery na wielką

Przykłady w C: zamiana wielkości znaków

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

int main()
{
    int a;

    while( (a = getchar()) != EOF )
        putchar(toupper(a));

    return 0;
}
```

☞ toupper2.c

`int getchar()`
zwraca pojedynczy znak ze standardowego wejścia lub `EOF`

`void putchar(int c)`
wypisuje znak

`EOF` (*end of file*)
bajt końca pliku

```
toupper2 < plik.txt
```

`EOF` w terminalu:
`Ctrl+Z` (Windows)
`Ctrl+D` (Unix/Linux)

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

A	l	a		m	a		k	o	t	a	\0

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

A	\0

Zmienna wskaźnikowa wskazująca na stały napis (nie można go zmodyfikować)

```
char *s1 = "nieciekawy fragment tekstu.";
```

Tablica zawierająca napis (może być modyfikowana)

```
char s2[]="Ala ma kota";
```

Tablica zawierająca napis "Ala"

```
char s3[] = {'A', 'l', 'a', '\0'};
```

```c
#include<stdio.h>

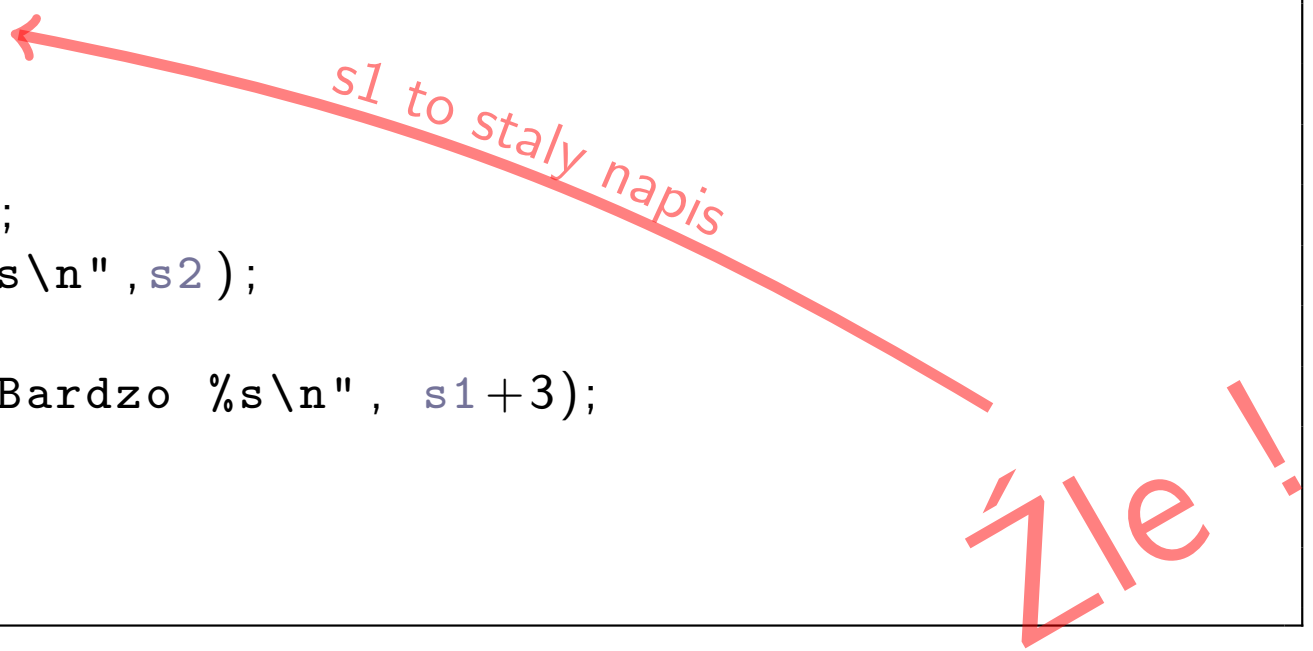
int main()
{
    char *s1="nieciekawy fragment tekstu.";
    char s2[]="Ala ma kota";

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

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

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

    return 0;
}
```



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

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

- niebezpieczeństwo przepełnienia tablicy
- `scanf("%10s", text);` poprawnie, wczyta max. 10 znaków
- odczytuje tylko pierwszy wyraz napisu, do wystąpienia białego znaku

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

- funkcja `gets()` czyta linię tekstu ze standardowego wejścia i umieszcza ją w tablicy `s`
- Uwaga: brak zabezpieczenia przed przepełnieniem tablicy, nie należy stosować tej funkcji

```
#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` - użycie funkcji jest bezpieczne
- standardowy strumień wejściowy to `stdin`
- znak `'\n'` również umieszczony jest na końcu napisu

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

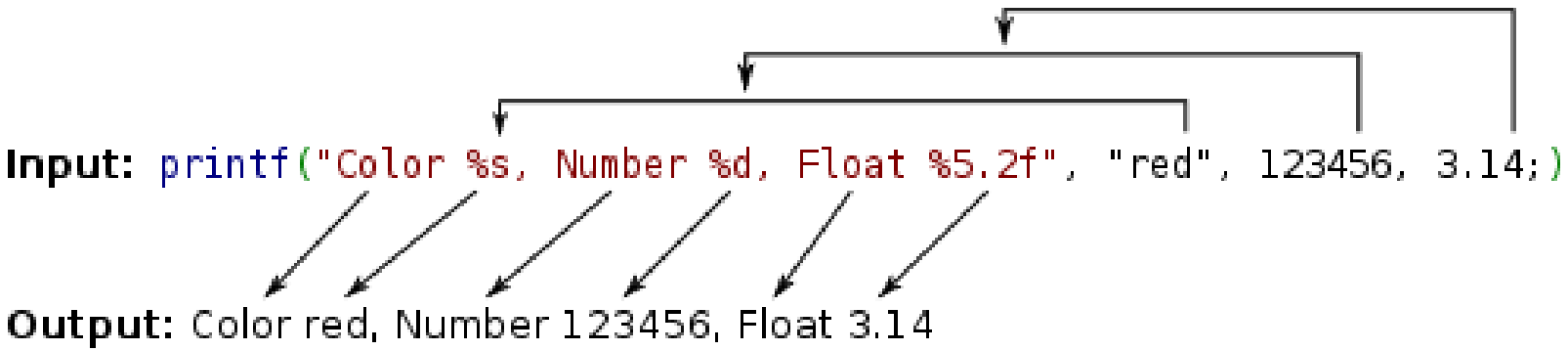
Wypisywanie łańcucha znakowego

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

- wypisuje łańcuch i dodaje znak nowej lini

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

- wypisuje napis zgodnie z zadanym formatem



Specyfikacja formatu

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

http://wikipedia.org

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

specyfikator	znaczenie	przykład	wynik
d lub i	liczba całkowita ze znakiem	`printf("%d", -123)`	`-123`
u	liczba całkowita bez znaku	`printf("%u", 123)`	`123`
o	liczba całkowita bez znaku ósemkowa	`printf("%o", 123)`	`173`
x lub X	liczba całkowita bez znaku szesnastkowo	`printf("%o", 123)`	`7b`
f lub F	zmiennopozycyjna w zapisie dziesiętnym	`printf("%f", 12.6)`	`12.600000`
e lub E	notacja naukowa	`printf("%e", 12.6)`	`1.260000e+01`
g lub G	krótszy zapis %f lub %e	`printf("%g", 12.6)`	`12.6`
c	pojedynczy znak (ASCII)	`printf("%c", 'a')`	`a`
s	łańcuch znakowy	`printf("%s", "Ala ma kota")`	`Ala ma kota`
p	adres (wskaźnik), szesnastkowo	`printf("%p", &a)`	`ff01ffab`
%	wypisuje znak %	`printf("%%")`	`%`

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

szerokość	znaczenie	przykład	wynik
liczba	minimalna ilość znaków (szerokość pola), brakujące miejsca są dopełniane spacjami. Jeżeli szerokość pola jest za mała to wynik nie jest obcinany.	`printf("_%5d_", 42)`	`_ 42_`
*	szerokość zależna od argumentu funkcji `printf`	`printf("_%*d_", 5, 42)`	`_ 42_`

flaga	znaczenie	przykład	
-	wyrównanie do lewej (względem podanej szerokości)	`printf("_%-5d_", 42)`	`_42 _`
+	liczby dodatnie poprzedzone znakiem +	`printf("_%-+5d_", 42)`	`_+42 _`
spacja	wstawia spację zamiast znaku +	`printf("_%- 5d_", 42)`	`_ 42 _`
0	wypełnia podaną szerokość znakiem 0	`printf("_%05d_", 42)`	`_00042_`
#	liczby ósemkowe i szesnastkowe poprzedza 0, 0x, 0X	`printf("%#x, 42)`	`0x2a`

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

precyzja	znaczenie	przykład	wynik
.liczba	ilość cyfr wypisywanych po przecinku. Dla napisów oznacza maksymalną liczbę wypisanych znaków (łańcuch jest obcinany)	`printf("%.2f", 1.66666)`	`1.67`
`.*`	precyzja liczby zmiennopozycyjnej nie jest podawana w formacie lecz poprzez argument funkcji `printf`.	`printf("%.*f", 2, 1.66666)`	`1.67`

długość	znaczenie	przykład
brak	typy `char`, `short`, `int`, `double`, `float`	`printf("%f", x)`
l	typ `long int`	`printf("%ld", x)`
ll	typ `long long int`	`printf("%lld", x)`
L	typ `long double`	`printf("%Lf", x)`

```c
#include <stdio.h>

int main()
{
    printf ("Znaki                    : %c %c \n", 'A', 65);
    printf ("Liczby calkowite         : %d \n", 123);
    printf ("  ze znakiem             : %+d \n", 123);
    printf ("  szesnastkowo           : %x %#x \n", 123, 123);
    printf ("  dopelnienie spacjami   : %20d \n", 123);
    printf ("  dopelnienie zerami     : %020d \n", 123);
    printf ("Zmienopozycyjne          : %f \n", 3.1416);
    printf ("  notacja naukowa        : %e \n", 3.1416);
    printf ("  precyzja               : %.3f \n", 3.1416);
    printf ("  dopelnienie            : %20.3f \n", 3.1416);
    printf ("Szerokosc pola  *        : %*d \n", 20, 123);
    printf ("Precyzja  .*             : %20.*f \n", 3, 3.1416);
    printf ("Napis                    : %s \n", "Ala ma kota");
    printf ("  dopelnienie            : %20s \n", "Ala ma kota");
    printf ("  obciecie               : %20.5s \n", "Ala ma kota");

    return 0;
}
```

☞ printf.c

```c
#include <stdio.h>

int main()
{
    printf ("Znaki                    : %c %c \n", 'A', 65);
    printf ("Liczby calkowite         : %d \n", 123);
    printf ("  ze znakiem             : %+d \n", 123);
    printf ("  szesnastkowo           : %x %#x \n", 123, 123);
    printf ("  dopelnienie spacjami   : %20d \n", 123);
    printf ("  dopelnienie zerami     : %020d \n", 123);
    printf ("Zmienopozycyjne          : %f \n", 3.1416);
    printf ("  notacja naukowa        : %e \n", 3.1416);
    printf ("  precyzja               : %.3f \n", 3.1416);
    printf ("  dopelnienie            : %20.3f \n", 3.1416);
    printf ("Szerokosc pola  *        : %*d \n", 20, 123);
    printf ("Precyzja  .
    printf ("Napis
    printf ("  dopelnier
    printf ("  obciecie

    return 0;
}
```

```
Znaki                    : A A
Liczby calkowite         : 123
  ze znakiem             : +123
  szesnastkowo           : 7b 0x7b
  dopelnienie spacjami   :                  123
  dopelnienie zerami     : 00000000000000000123
Zmienopozycyjne          : 3.141600
  notacja naukowa        : 3.141600e+00
  precyzja               : 3.142
...
```

```c
#include <stdio.h>

int main()
{
    printf ("Znaki
    printf ("Liczby calk
    printf ("  ze znakie
    printf ("  szesnastk
    printf ("  dopelnier
    printf ("  dopelnienie zerami  : %020d \n", 123);
    printf ("Zmienopozycyjne       : %f \n", 3.1416);
    printf ("  notacja naukowa     : %e \n", 3.1416);
    printf ("  precyzja            : %.3f \n", 3.1416);
    printf ("  dopelnienie         : %20.3f \n", 3.1416);
    printf ("Szerokosc pola  *     : %*d \n", 20, 123);
    printf ("Precyzja  .*          : %20.*f \n", 3, 3.1416);
    printf ("Napis                 : %s \n", "Ala ma kota");
    printf ("  dopelnienie         : %20s \n", "Ala ma kota");
    printf ("  obciecie            : %20.5s \n", "Ala ma kota");

    return 0;
}
```

```
...
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
```

☞ printf.c

Podstawowe operacje na łańcuchach

Plik nagłówkowy `string.h`

- długość łańcucha `s`

```
int strlen(const char *s);
```

- konkatenacja (łączenie łańcuchów), na końcu `dest` doklej `src`

```
char *strcat(char *dest, const char *src);
```

- porównywanie łańcuchów

```
int strcmp(const char *s1, const char *s2);
```

  wynik 0 gdy `s1` i `s2` są takie same, wynik -1 gdy `s1 < s2`, lub 1 gdy `s1 > s2`

- kopiowanie łańcuchów, umieszcza `src` w tablicy `dest`

```
char *strcpy(char *dest, const char *src);
```

- wyszukiwanie wzorca `wzor` w tekscie `napis`

```
char *strstr(const char *napis, const char *wzor);
```

  wynikiem jest adres pierwszego wsytąpienia wzorca lub `NULL` gdy wzorzec nie występuje w napisie

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

Algorithm Naiwne wyszukiwanie wzorca

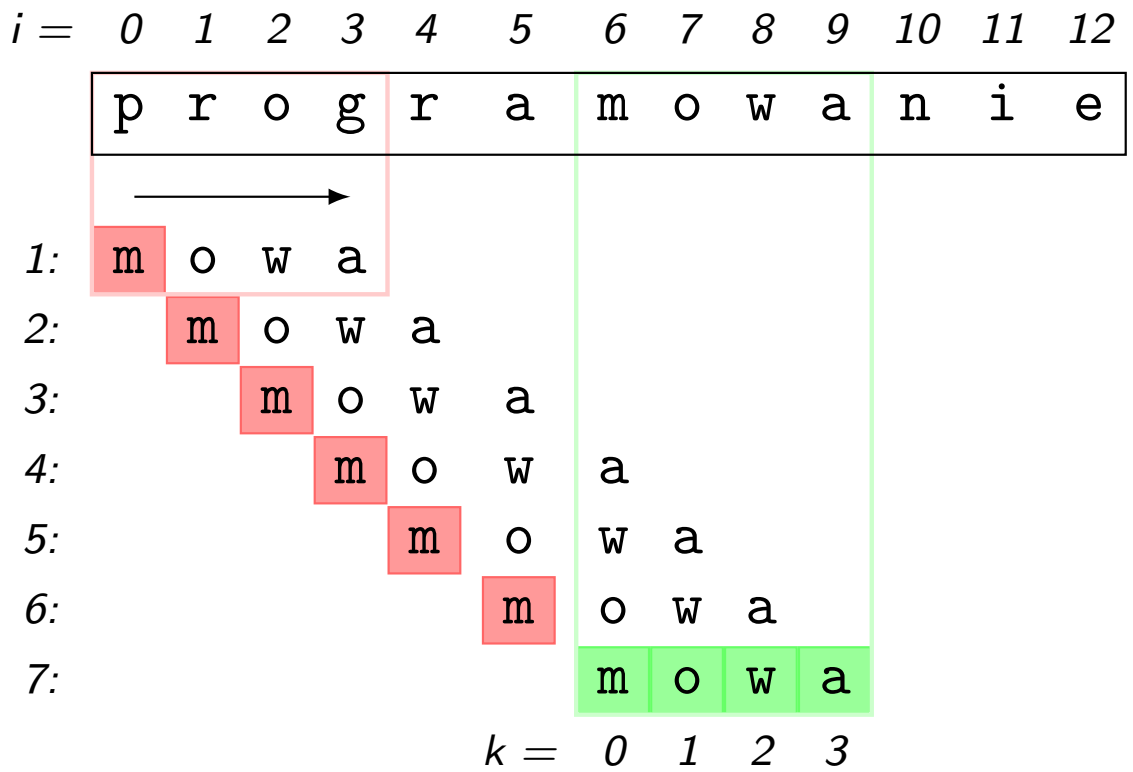
Dane wejściowe: łańcuch znaków $\mathcal{T}$, wzorzec $\mathcal{W}$

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

```
1: dla każdego  i = 0, 1, 2, ..., |T| − |W|  wykonuj
2:     k ← 0
3:     dopóki  T[i + k] = W[k] i k < |W|  wykonuj
4:         k ← k + 1
5:     jeżeli  k = |W|  wykonaj
6:         zwróć i
7: zwróć −1
```

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

Tekst $\mathcal{T}$ = "programowanie"
Wzorzec $\mathcal{W}$ = "mowa"



```c
int strindex(const char t[], const char w[])
{
    int i, k;

    i=0;
    while(t[i] != '\0')
    {
        k = 0;
        while (t[i+k] == w[k] && w[k] != '\0')
        {
            k = k + 1;
        }
        if (w[k] == '\0') return i;
        i = i + 1;
    }
    return -1;
}
```

☞ strindex.c

To samo z użyciem wskaźników

```
1  int strindex(const char *t, const char *w)
2  {
3      char *pt, *pw, *pt2;
4
5      for (pt=t; *pt != '\0'; pt++)
6      {
7          pw = w;
8          pt2 = pt;
9          while (*pt2 == *pw && *pw != '\0')
10         {
11             pw++;
12             pt2++;
13         }
14         if (*pw == '\0') return pt - t;
15     }
16     return -1;
17 }
```

☞ strindex3.c

- 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 `srtcpy()`)



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`	$x \mod 2$
++	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 <= y`	$x \leqslant y$
>=	większy lub równy	`x >= y`	$x \geqslant y$
==	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 liczby w jej reprezentacji binarnej
- są zdefiniowane dla liczb całkowitych
- w praktyce używane najczęściej dla liczb całkowitych bez znaku (np. `unsigned int`)

Negacja

~	0	1
	1	0

Koniunkcja (AND)

&	0	1
0	0	0
1	0	1

Alternatywa (OR)

|	0	1
0	0	1
1	1	1

Alternatywa rozłączna (XOR)

^	0	1
0	0	1
1	1	0

```c
#include <stdio.h>

int main()
{
    unsigned int a = 9;
    unsigned int b = 12;

    printf("a      = %u\nb      = %u\n", a, b);
    printf("~a     = %u\n~b     = %u\n", ~a, ~b);
    printf("a & b = %u\n", a & b);
    printf("a | b = %u\n", a | b);
    printf("a ^ b = %u\n", a ^ b);

    return 0;
}
```

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

☞ bitowe1.c

```
a                9     00000000000000000000000000001001
~a      4294967286     11111111111111111111111111110110

a                9     00000000000000000000000000001001
b               12     00000000000000000000000000001100
a & b            8     00000000000000000000000000001000

a                9     00000000000000000000000000001001
b               12     00000000000000000000000000001100
a | b           13     00000000000000000000000000001101

a                9     00000000000000000000000000001001
b               12     00000000000000000000000000001100
a ^ b            5     00000000000000000000000000000101
```

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

$$\texttt{A ^ B ^ B} \longrightarrow A$$

```c
#include<stdio.h>

int main()
{
    unsigned char klucz = 13;
    char znak;

    while ((znak = getchar()) != EOF)
        putchar(znak ^ klucz);

    return 0;
}
```

☞ szyfr–xor.c

operator	znaczenie	przykład
<<	przesunięcie bitowe w lewo	`x << y`
>>	przesunięcie bitowe w prawo	`x >> y`

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

```c
#include <stdio.h>

int main()
{
    unsigned int a = 5;

    printf("a       = %u\n", a);
    printf("a << 1 = %u\n", a << 1);
    printf("a << 2 = %u\n", a << 2);
    printf("a >> 1 = %u\n", a >> 1);
    printf("a >> 2 = %u\n", a >> 2);

    return 0;
}
```

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

☞ bitowe2.c

```
a       = 5     00000000000000000000000000000101
a << 1  = 10    00000000000000000000000000001010
a << 2  = 20    00000000000000000000000000010100

a       = 5     00000000000000000000000000000101
a >> 1  = 2     00000000000000000000000000000010
a >> 2  = 1     00000000000000000000000000000001
```

Maska bitowa: słowo bitowe (liczba) 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 | 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
  ^ 00001000    00001000
  = 10010101    10011101
```

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

Zamiana wielu bitów zadaną maską

```
x = x ^ 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` $\Leftrightarrow$ `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 zapisu, zamiana nie zawsze prowadzi do błędu składniowego

```
a -=b;        /*  OK  */
a =-b;        /*  czy poprawnie? */
```

- Jednoargumentowy operator inkrementacji ++ i operator dekrementacji --

`++a`	pre-inkrementacja, to samo co `a=a+1`
`a++`	post-inkrementacja
`--a`	pre-dekrementacji, to samo co `a=a-1`
`a--`	post-dekremenatacji

- Operator post-inkrementacji i post-dekrementacji **zwracają wartość poprzednią** (sprzed zwiększenia/zmniejszenia). Zmiana wartości argumentu odbywa się później (po wyliczeniu całego wyrażenia).

```c
#include <stdio.h>

int main()
{
    int a = 42;
    int b;

    b = ++a;
    printf("a=%d, b=%d\n", a, b);

    b = a++;
    printf("a=%d, b=%d\n", a, b);

    b = --a;
    printf("a=%d, b=%d\n", a, b);

    b = a--;
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    int a = 42;
    int b;

    b = ++a;
    printf("a=%d, b=%d\n", a, b);

    b = a++;
    printf("a=%d, b=%d\n", a, b);

    b = --a;
    printf("a=%d, b=%d\n", a, b);

    b = a--;
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```
a=43, b=43
a=44, b=43
a=43, b=43
a=42, b=43
```

Kopiowanie napisu z tablicy a do b

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

Dokładnie to samo

```
void strcpy(const 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 z 3 argumentami

Przykład

Wyznaczanie większej wartości z 2 liczb:

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

Wartość absolutna:

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

- przecinek pozwala na wykonanie kilku wyrażeń (od lewej do prawej), w miejscu gdzie oczekiwane jest jedno wyrażenie. Wynikiem jest wynik ostatniej instrukcji.

```
a = 1, b = 2, c = 3;
int a = (x = 5, y = 10, x + y);
for (i = 0, j = 10; i < j; i++, j--)
```

- dostęp do elementów tablicy []

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

- wywołanie funkcji ()
- grupowanie wyrażeń ()

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

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

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

Priorytety i kolejność operatorów

- **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;
```

priorytet

Operator	Łączność
`()` wywołanie funkcji, przyrostkowe	
`[] . -> () ++ --`	lewostronna
jednoargumentowe, przedrostkowe, rzutowanie	
`! ~ + - * & 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

```
int a, b=1, c=2;
a = -b++ + ++c << 3 / 2 * (int)2.5 ;
```

1. b++ zwiększa wartość b na 2 ale zwraca 1
   ```
   a = -1 + ++c << 3 / 2 * (int)2.5 ;
   ```
2. ++c zwiększa wartość c na 3, rzutowanie `(int)2.5` daje 2
   ```
   a = -1 + 3 << 3 / 2 * 2 ;
   ```
3. dzielenie 3/2 zwraca 1, które następnie jest mnożone przez 2 (łączność lewostronna)
   ```
   a = -1 + 3 << 2 ;
   ```
4. ```
   a = 2 << 2 ;
   ```
5. ```
   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

Typ wyliczeniowy, pola bitowe i unie.

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

- Tablice
- Struktury `struct`
- Unie `union`
- Pola bitowe
- Typ wyliczeniowy `enum`
- 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;
```

Przykład:

```c
#include <stdio.h>

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

int main()
{
    enum kolory n;

    for(n=zielony; n<=10; n++)
        printf("%d\n", n);

    return 0;
}
```

```
0
1
2
3
```

☞ enum1.c

Przykład:

```c
#include <stdio.h>

enum kolory { zielony=3, niebieski=7, czerwony, czarny };

int main()
{
    enum kolory n;

    printf("sizeof = %d\n", sizeof(n));

    for(n=zielony; n<=czarny; n++)
        printf("%d\n", n);

    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.
`wskaznik->pole` jest równoważne z `(*wskaznik).pole`

```c
#include <stdio.h>

typedef enum { M, K } plec;

typedef struct
{
    char nazwisko[100];
    int indeks;
    plec p;
} student;

void wypisz_studenta(const student *s)
{
    printf("Nazwisko: %s\n", s->nazwisko);
    printf("Indeks  : %d\n", (*s).indeks);
    printf("Plec    : %s\n", s->p == M ? "M" : "K");
}

int main()
{
    student s = {"Zenon Adamski", 123456, M};
    wypisz_studenta(&s);
    return 0;
}
```

☞ struct2.c

- 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 wartośc pozostałych.

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

☞ union1.c

```c
#include <stdio.h>

union liczba
{
   float f;
   unsigned int i;
   unsigned char c;
};

int main()
{
  union liczba x;

  x.f = 3.14;

  printf("sizeof = %d\n", sizeof(union liczba));

  printf("adres pola f = %p\n", &x.f);
  printf("adres pola i = %p\n", &x.i);
  printf("adres pola c = %p\n", &x.c);

  printf("wartosc pola f = %f\n", x.f);
  printf("wartosc pola i = %x\n", x.i);
  printf("wartosc pola c = %x\n", x.c);

}
```

☞ union2.c

- 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 ;
};
```

```c
#include<stdio.h>

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

int main()
{
   data dzis = { 18, 16, 2022 };

   printf("Data: %u-%u-%u\n",
          dzis.dzien, dzis.miesiac, dzis.rok);

   printf("sizeof(data)=%ld\n", sizeof(data));

   return 0;
```

☞ bitfield.c

Pola bitowe pozwalają poprawić czytelnośc 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 zawartości, jest elementem systemu plików
- Lokalizacja pliku jest określona przez **ścieżkę dostępu**
- Pliki są opatrzone **atrybutami**: uprawnienia, własności pliku, np. plik ukryty, itp.
- Plik tekstowy a plik binarny - w obu wypadkach to jest ciąg bajtów

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: określa sposób przechowywania plików na nośniku fizycznym i organizację dostępu do nich z systemu operacyjnego

- fizyczny zapis danych, np. blokowa struktura danych na dysku (sektory, klastry)
- logiczna struktura widoczna dla użytkownika udostępniana przez system operacyjny
- przykłady systemów plików:
  - DOS/Windows: FAT, FAT32, NTFS
  - Linux: ext2, ext3, ext4
  - inne: HFS (Mac OS), NFS (sieć), ISO9660 (CD-ROM)
- hierarchia plików: katalogi (foldery), podkatalogi i pliki.
  Ścieżka dostępu od korzenia do pliku
  `C:\Documents and Settings\user\moje dokumenty\plik.txt`
  `/home/user/doc/plik.txt`

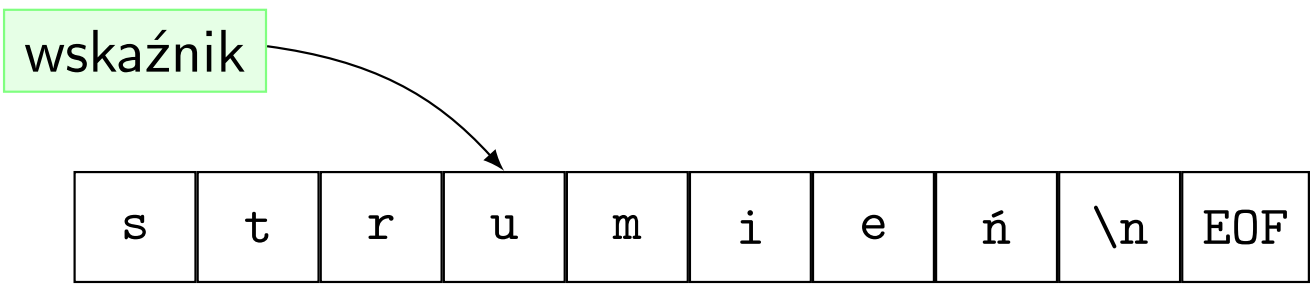
http://en.wikipedia.org/wiki/List_of_file_systems

Atrybuty plików

- 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

- **Deskryptor pliku**: niskopoziomowa reprezentacja (liczba całkowita) wykorzystywana przez funkcje systemowe (np. biblioteka `unistd.h` w systemach Unix/Linux)
- **Strumień:** ogólny sposób komunikacji między procesami i urządzeniami (obsługa wejścia/wyjścia)
- Wykorzystanie strumieni: dostęp do plików (odczyt/zapis), komunikacja między procesami (np. potoki), komunikacja sieciowa, komunikacja z urządzeniami, łańcuchy znakowe jako strumienie.
- Plik nagłówkowy `stdio.h` zawiera deklaracje funkcji obsługi wejścia i wyjścia (***St**andar**d** **i**nput **o**utput*)
- Strumień do/z pliku dostępny za pomocą zmiennej typu `FILE*` (wskaźnik do pliku, „uchwyt" do pliku)

- Typ `FILE*` z biblioteki `stdio.h`
- 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"`.

  Ścieżki bezwzględne w kodzie to nie jest najlepszy pomysł - brak przenośności!
- 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"`

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

Błędy przy otwieraniu strumienia

- 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 niepowodzenia: 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

```c
#include <stdio.h>

int main()
{
    FILE *plik = NULL;
    float pi = 3.1415;

    plik = fopen("plik.txt", "w");
    if (plik != NULL)
    {
        /* Tutaj operacje na pliku */
        fprintf(plik, "Witaj swiecie!\nPI=%f\n", pi);
        fclose(plik);
    }
    else
    {
        /* Obsluga bledu otwarcia pliku */
        printf("Blad otwarcia pliku %s\n", "plik.txt");
    }
    return 0;
}
```

☞ zapis.c

Koniec strumienia

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

Wartość niezerowa gdy wystąpił koniec pliku.

Funkcje odczytu: `fscanf()`, `fgetc()`, `fgets()` zwracają w takiej sytuacji wartość `EOF` (*end of file*)

Inne błędy operacji wejścia-wyjścia

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

Zwraca wartość niezerową, gdy w strumieniu wystąpił błąd

Typowy schemat: odczyt znaków z pliku

```c
#include<stdio.h>

int main()
{
    FILE *plik = NULL;
    int znak;

    plik = fopen( "plik.txt", "r");
    if (plik == NULL)
    {
        perror("Wystapil blad");
        return 1;
    }

    while (feof(plik) == 0)
    {
        znak = fgetc(plik);
        if (znak != EOF) printf("%c\n", znak);
    }
    fclose(plik);

    return 0;
}
```

☞ odczyt.c

Standardowe wejście i wyjście programu

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

< przekierowanie strumienia wejściowego

> przekierowanie strumienia wyjściowego

Potoki

```
ls | sort | head | wc
```

Strumienie to podstawowy sposób komunikacji miedzy procesami.

Strumienie `FILE*` dostępne w `stdio.h`

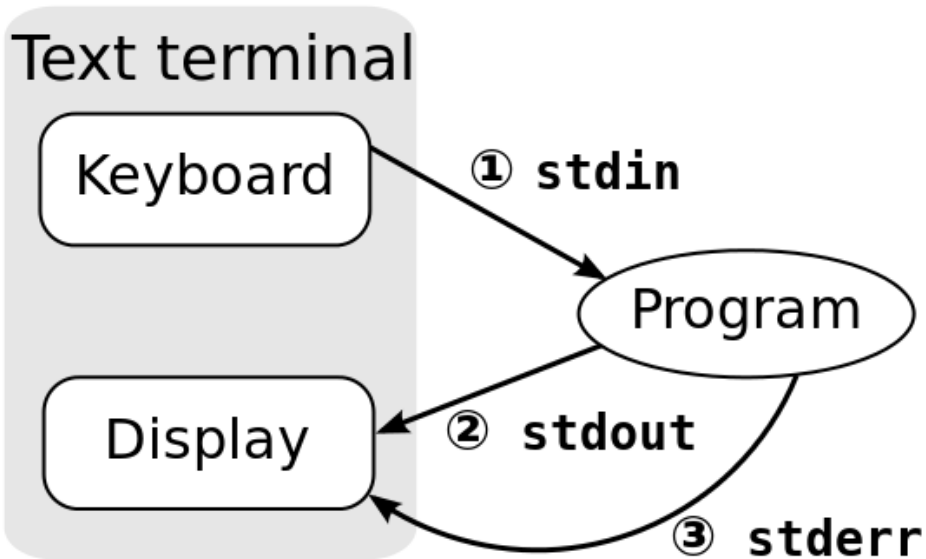
0 `stdin`
standardowe wejście, domyślnie klawiatura
`getchar(), scanf(), ...`

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);
```



Źródło: http://upload.wikimedia.org/

- DOS/Windows: konwersja plików tekstowych
  Nowy wiersz: `\r\n` $\Longleftrightarrow$ `\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 zostawil 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