Programowanie Proceduralne

Marek Grochowski

Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK

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

Semestr zimowy 2013/2014

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

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

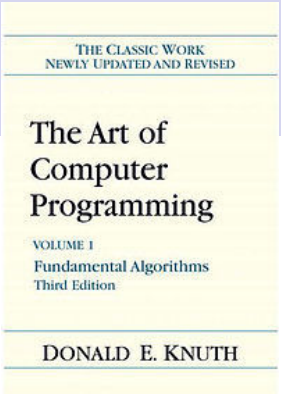
Celem programowania jest odnalezienie sekwencji instrukcji, które w sposób automatyczny wykonują pewne zadanie.

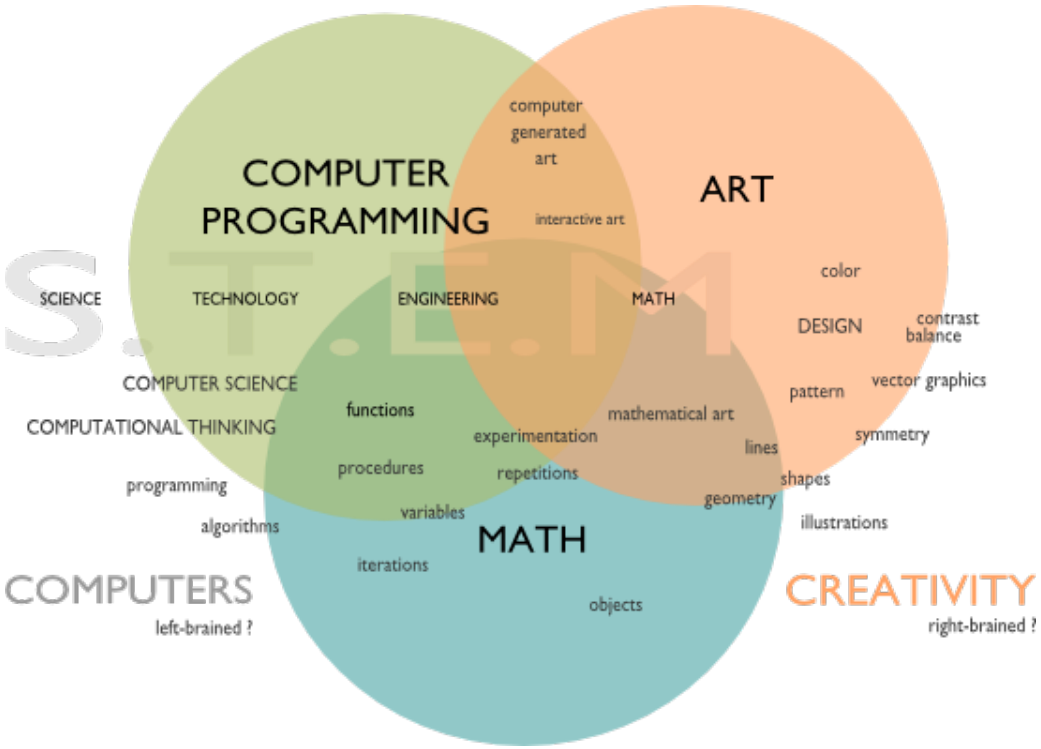
W inżynierii oprogramowania: programowanie = implementacja

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

pl.wikipedia.org

Sztuka, rzemiosło czy inżynieria?





http://www.computersforcreativity.com

Jak uczyć się programowania?

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

Ogólny przegląd zagadnień

- Problem $\rightarrow$ Algorytm $\rightarrow$ Program $\rightarrow$ Rozwiązanie
- *Case study* - przykłady programów, demonstracje
- Język C - tylko niezbędne podstawy

  Inne języki: Pascal, Fortran, C++ (obiektowe), ...
- Procedury, podprogramy, funkcje
- Reprezentacja danych w komputerze: typy proste, złożone, struktury dynamiczne, ...
- Elementy inżynierii oprogramowania: model, projekt, analiza, implementacja, wykrywanie błędów, testowanie
- Laboratorium: język C, środowisko MS Visual Studio

- N. Wirth, *Algorytmy + struktury danych = programy*, WNT, Warszawa, 1989.
- D. Harel, *Rzecz o istocie informatyki. Algorytmika.*, WNT, Warszawa, 1992.
- Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*, WNT, Warszawa, 2000.
- J. Bentley, *Perełki oprogramowania*, WNT, Warszawa, 2001.

Trochę historii

- p.n.e. liczydło (Abacus)
- 1500 Leaonrdo da Vinci, projekt maszyny arytmetycznej (150 lat przed Pascalem)
- 1614 John Napier, logarytm, kostka Naplera
- 1623 Wilhelm Schickard, pierwsza maszyna analityczna, zamówienie dla Jana Keplera, $(+ - \times /)$
- 1642 Blaise Pascal, „Pascalina", sumator Pascala $(+-)$
- 1671 Gottfried von Leibniz, rachmistrz krokowy $(+ - \times /\sqrt{})$


Maszyna Żakarda

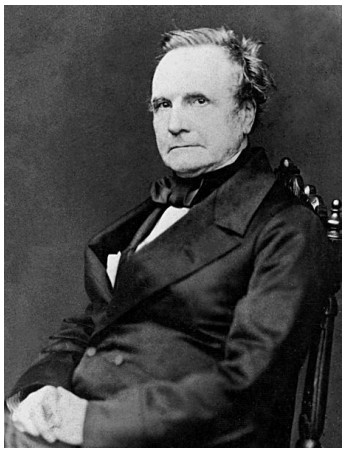
Joseph Marie Jacquard

1745 Jacques de Vaucanson, programowalne urządzenie włókiennicze

1805 Maszyna Żakarda, krosno programowanie kartami perforowanymi.

Pierwsze programowe sterowanie w dziejach techniki.

Charles Babbage (1791–1871)



1822 Projekt maszyny różnicowej.

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

Augusta Ada Lovelace (1815-1852)

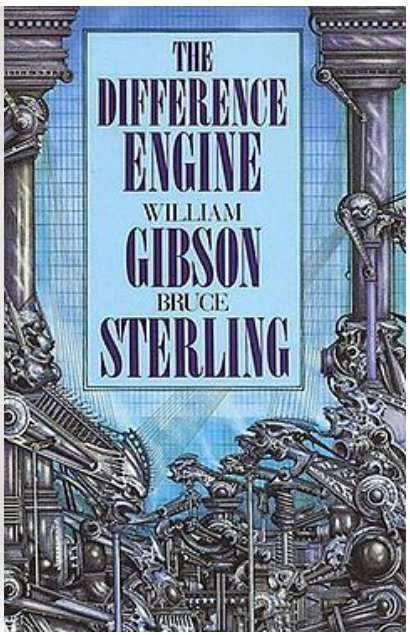
1842 „Note G", algorytm wyznaczania liczb Bernoulliego. **Pierwszy program komputerowy.**

1979 Język ADA.

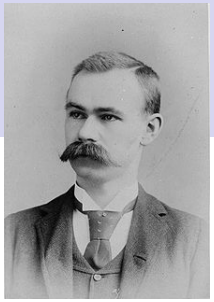
1849 Difference Engine No. 2, dokładność 31 cyfr, wydruk na wyjściu

2002 Realizacja projektu, ☞ Computer History Museum.

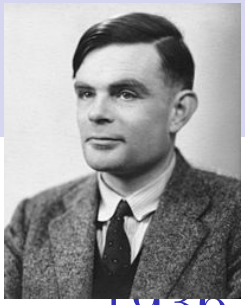
2011 Rozpoczęto 10 letni projekt rekonstrukcji maszyny analitycznej



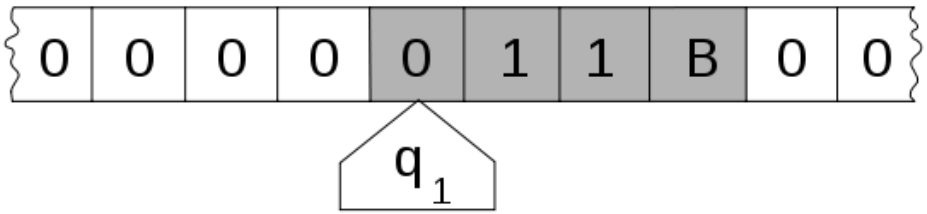
http://www.computerhistory.org/



1890 Herman Hollerith ze swoją maszyną tabulacyjną. Służyła ona do wprowadzania, sortowania i podliczania danych i wykorzystywała do tego celu dziurkowane karty. Hollerith wykorzystywał notatki Babbage'a.
W 1896 powstaje Tabulating Machine Company przekształcona w 1924 w International Business Machines (IBM).



1936 Maszyna Turinga (uniwersalny komputer) - matematyczny model komputera.



O maszynie mówimy, że jest ***kompletna w sensie Turinga****, kiedy można za jej pomocą zrealizować każdy algorytm. Istnieją problemy, które nie mogą byś rozwiązane przez proces sekwencyjny.*
Praktycznym przybliżeniem realizacji uniwersalnej Maszyny Turinga jest komputer, będący w stanie wykonać dowolny program na dowolnych danych.

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 ?

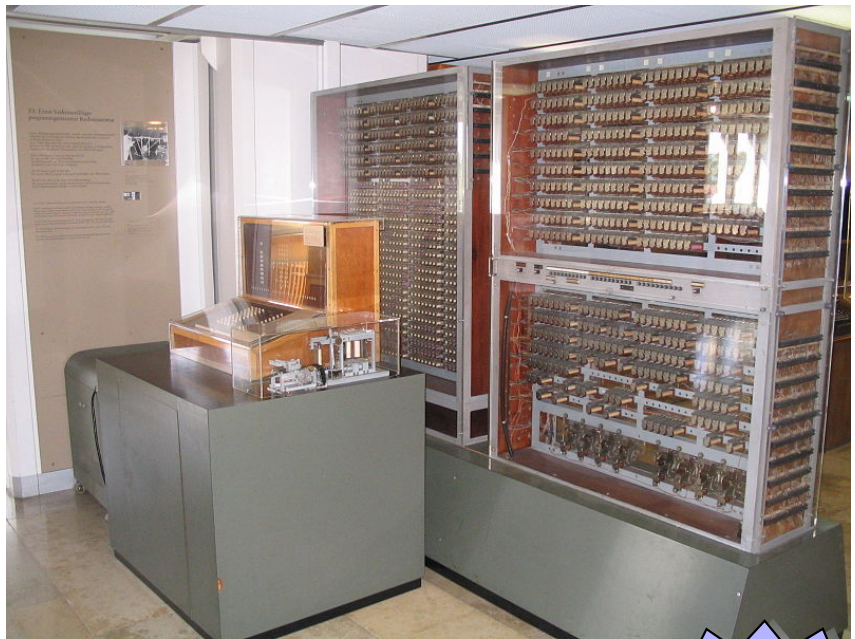
Komputery generacji 0

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)

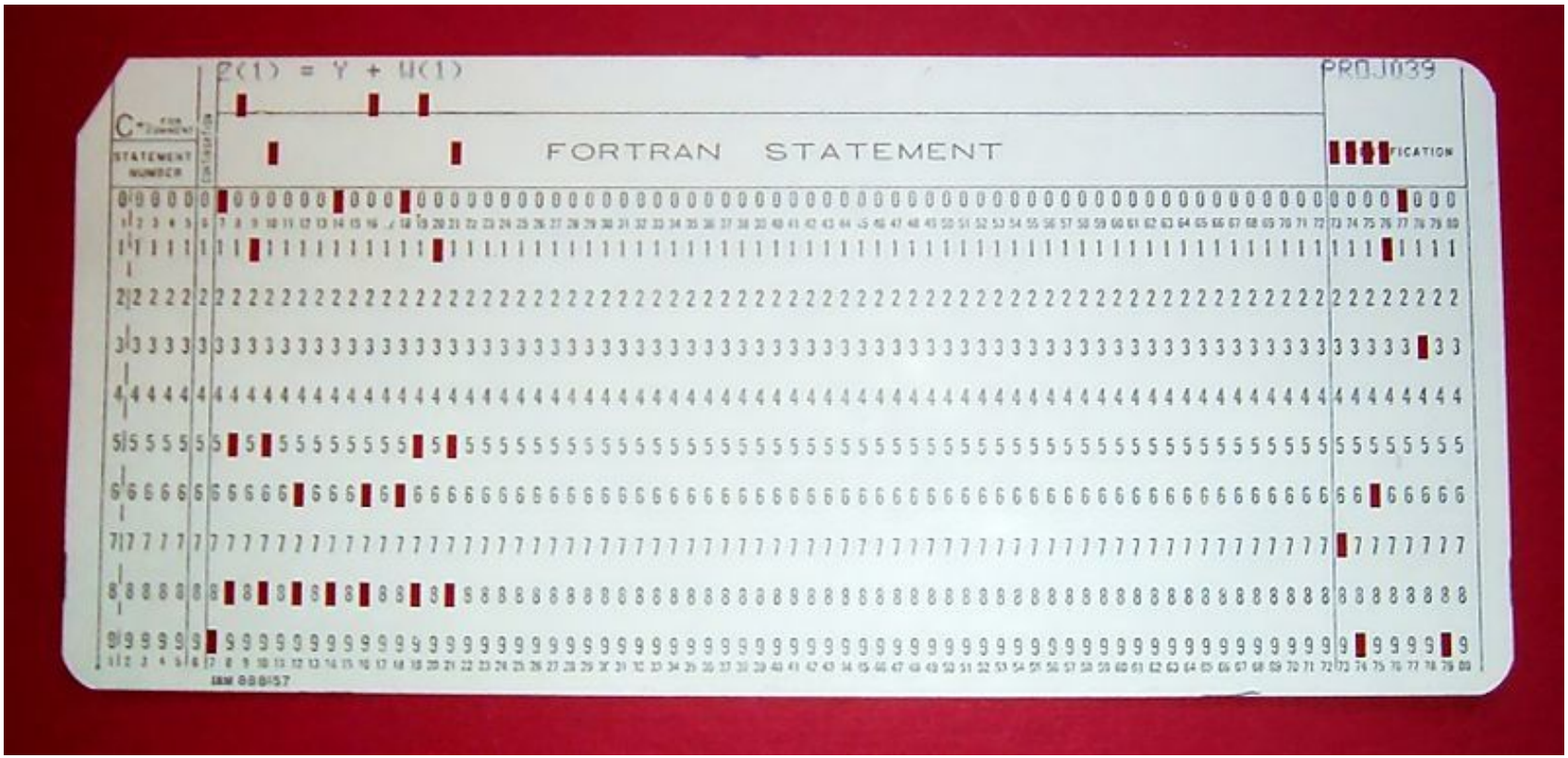
Lampy próżniowe

- 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

1949 (prawie) pierwszy assembler (EDSAC)
1952 Grace Hopper, pierwszy kompilator A-0 (UNIVAC I)
1954 Język Fortran

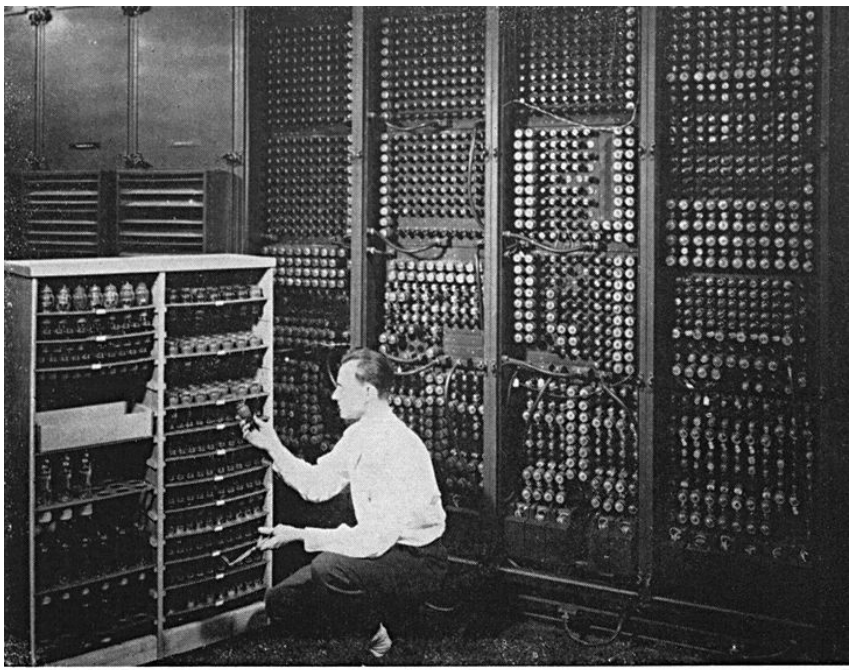
Karta perforowana

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



ENIAC (1946)

Elektroniczne urządzenie numeryczne całkujące i liczące



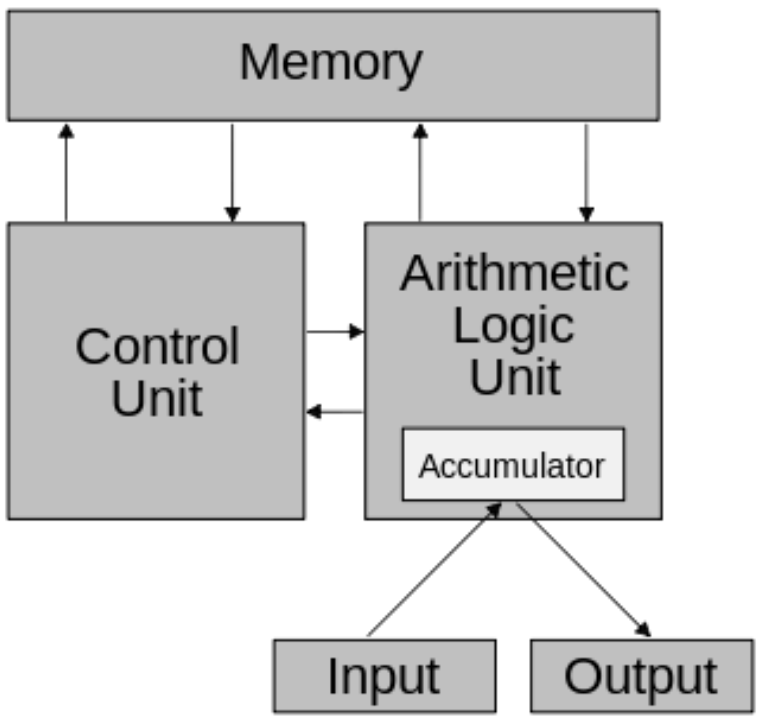
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

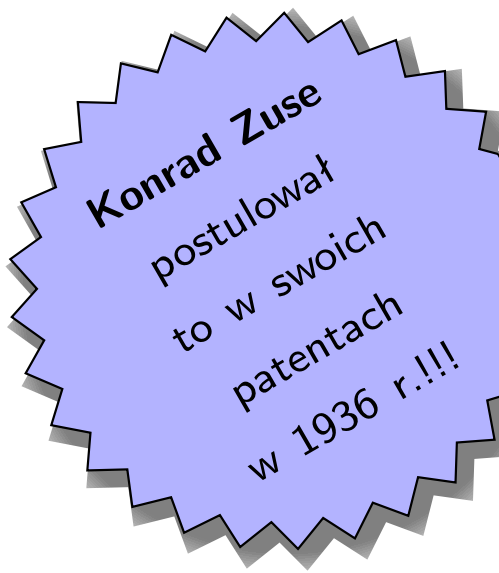
18 000 lamp,
30 ton, 170 m2, moc 160kW,
5000 operacji dodawania na sekundę
system dziesiętny,
ręczne programowanie przez ustawianie przełączników (6k) i wtykanie kabli

Współczesna koncepcja komputera

John von Neumann, 1945

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





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

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

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

Komputery 3 generacji (1965-70)

Układy scalone o małej skali integracji SSI

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

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

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

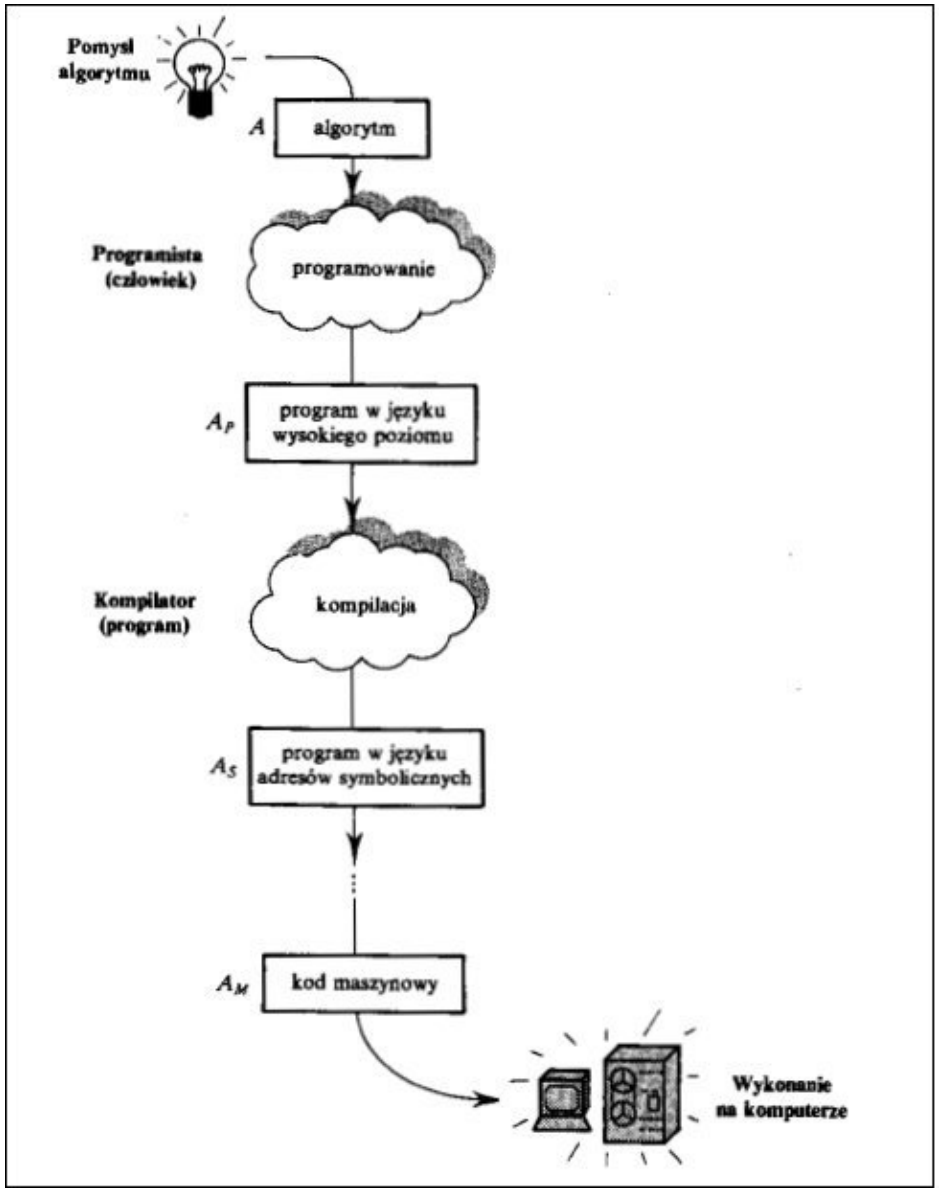
  Java (1955), Python (1991), PHP (1995), JS (1995), .NET (C#, 2001)

Źródło: ☞ History and Evolution of Programming Languages

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

Pieter Bruegel (starszy), 1563

Od pomysłu do programu

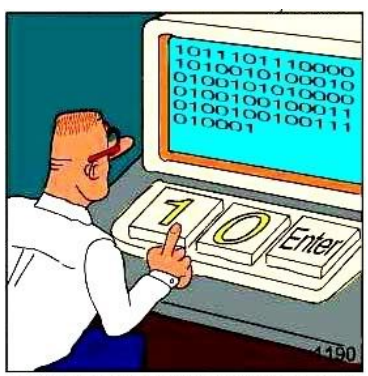


D. Harel, [2]

Kod maszynowy

ciąg instrukcji w postaci binarnej wykonywanych bezpośrednio przez procesor.

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



REAL Programmers code in BINARY.

Język assemblera

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

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

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

Języki wysokiego poziomu

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

Kompilator

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

- kod źródłowy $\longrightarrow$ kod maszynowy
- kod źródłowy $\longrightarrow$ *byte code*, kod pośredni rozumiany lub kompilowany przez maszynę wirtualną (Java, .Net)

Interpreter

odczytuje, analizuje i uruchamia instrukcje zawarte w kodzie źródłowym (brak procesu kompilacji).
Języki skryptowe: Bash, Perl, Python

- **Imperatywne** - sekwencja instrukcji wykonywanych przez program (prog. proceduralne, obiektowe)
  Przykłady: Fortran, Pascal, C/C++, Java, C#, Basic, Ada, ...
- **Deklaratywne** - opisują logikę obliczeń bez podawania sekwencji instrukcji. Interesuje nas co chcemy osiągnąć a nie jak to zrobić. Np.:
  - **Logiczne:** Prolog
  - **Funkcyjne:** Haskel, Scheme, Lisp

Środowiska programistyczne C/C++

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

Algorytmy

DOKŁADNA NAUKA

WARZENIA PIWA.

WEDŁUG METODY 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 *Sonnenthaubliithe*, 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**.

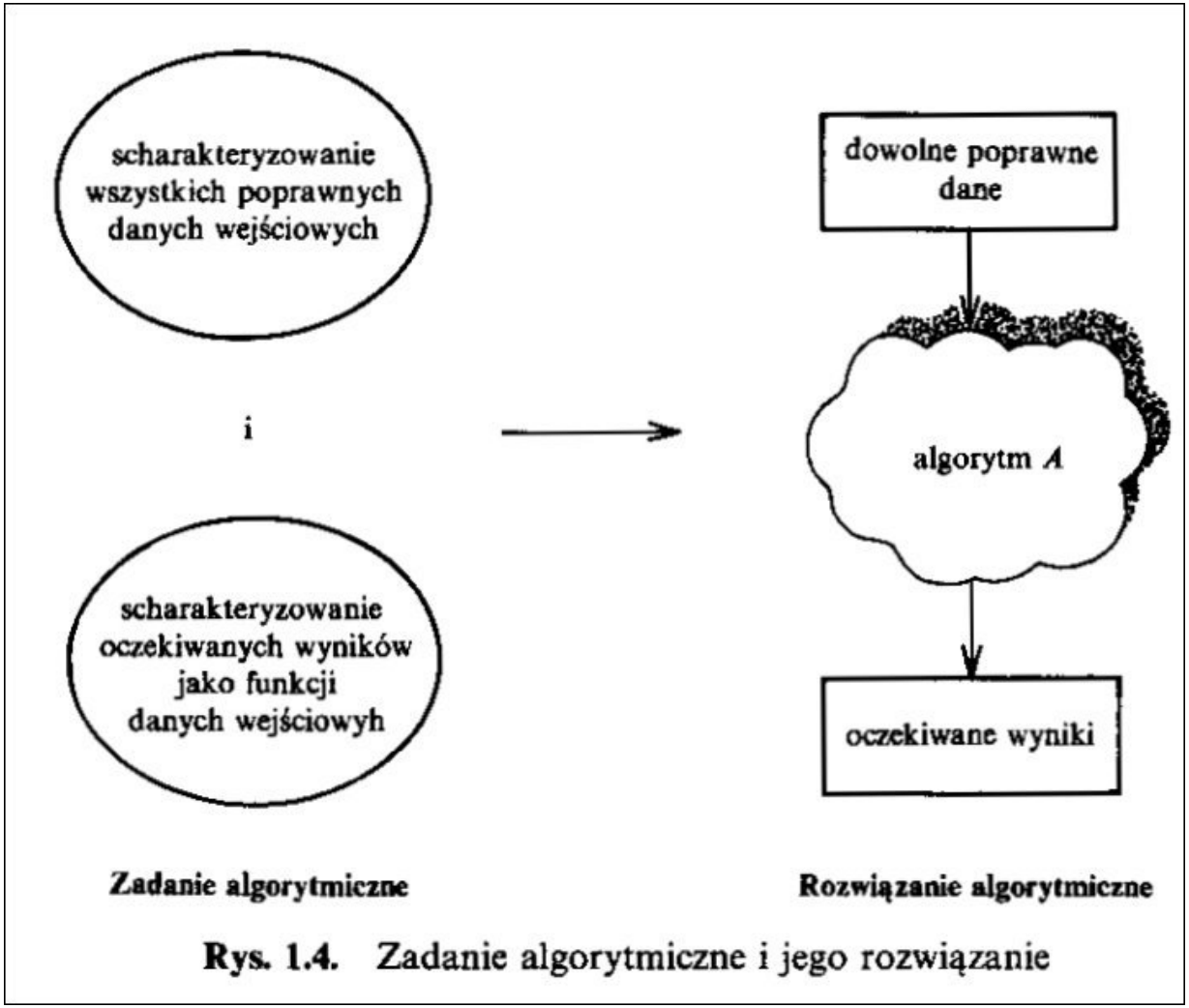
Specyfikacja zadania algorytmicznego

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

Dodatkowe ograniczenia algorytmu, np. dotyczące ilości operacji.

Przykład: Algorytm Euklidesa

Dane wejściowe: liczby całkowite $x, y > 0$
Wynik: $NWD(x, y)$



Rys. 1.4. Zadanie algorytmiczne i jego rozwiązanie

D. Harel, [2]

Cechy algorytmu

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

Ciąg struktur sterujących definiuje kolejność wykonywanych operacji.

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

Struktury sterujące można dowolnie składać: np. pętle zagnieżdżone. „Nie ma granic zawiłości algorytmów".

- Głównym przykazaniem **programowania strukturalnego** jest nieużywanie instrukcji skoku.
- 1960-70 gorąca debata na ten temat
- Trudności techniczne, niejednoznaczność, nieczytelność:
  skoki do wnętrza pętli, z pętli do pętli, z podprogramu (funkcji) do podprogramu, itp. ...
- Brak instrukcji skoku wymaga odpowiedniego używania pozostałych struktur sterujących (warunków i pętli)
- Przejrzystość - ważny aspekt programowania
- Używanie instrukcji skoku to zły styl programowania

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

Problem:

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

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 całkowite. 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.

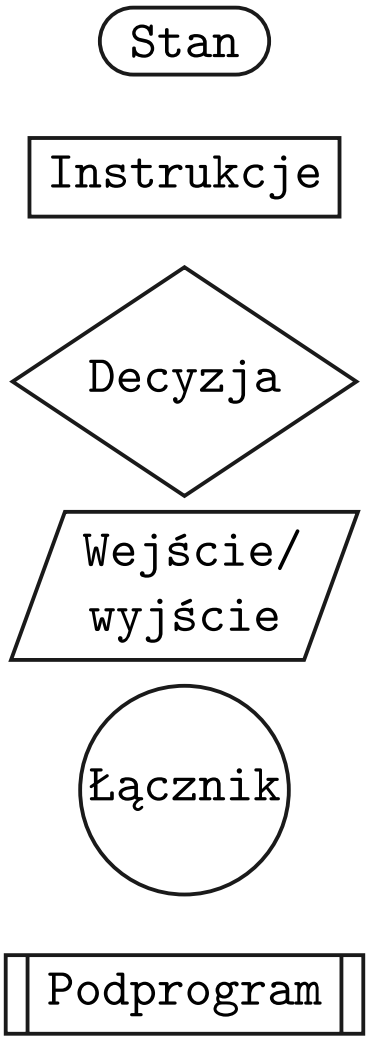
Specyfikacja

Dane wejściowe: liczby całkowite $a, b > 0$
Wynik: liczba całkowita a stanowiąca największy wspólny dzielnik

Lista kroków

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

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



Blok graniczny: start, stop, przerwanie, opóźnienie.

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

Blok decyzyjny, rozgałęzienie.

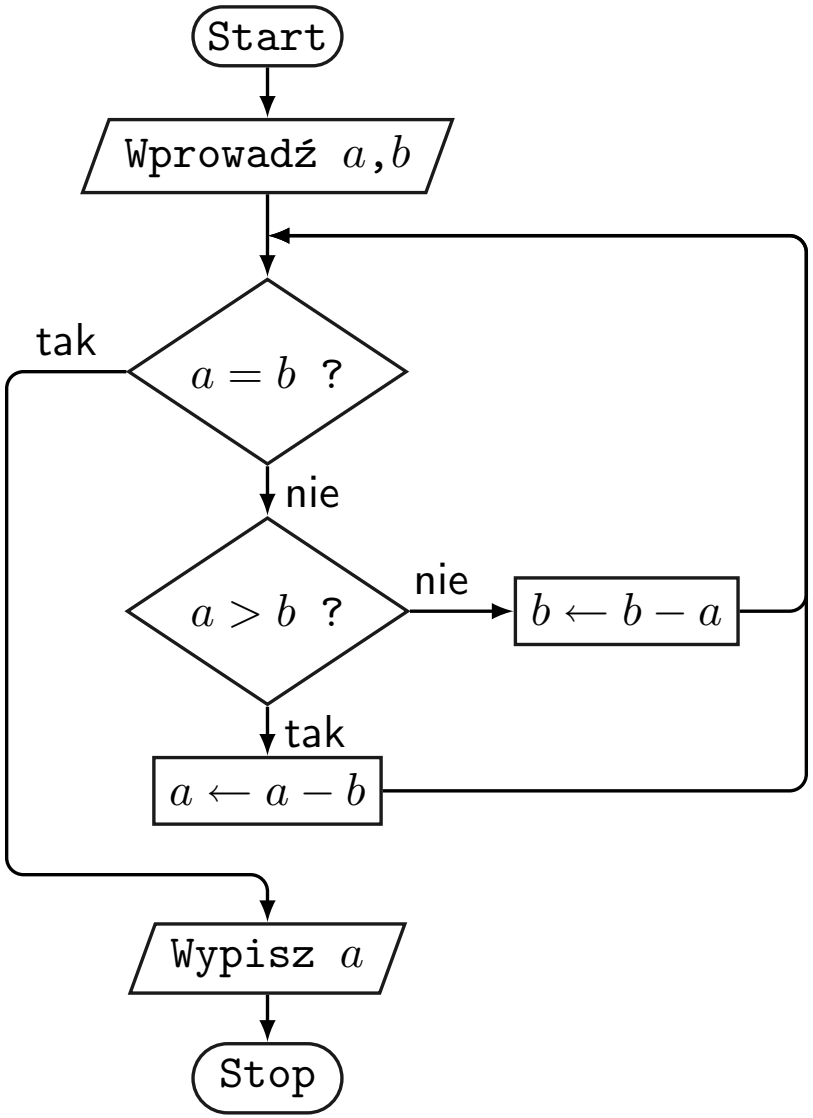
Wprowadzanie danych i wyprowadzenia wyników.

Połączenie z innym fragmentem diagramu.

Wywołanie podprogramu.

Przykład: Schemat blokowy

Algorytm Euklidesa z odejmowaniem



Algorytm 1 Algorytm Euklidesa

dopóki $a \neq b$ **wykonuj**
  jeżeli $a > b$ **wykonaj**
    $a \leftarrow a - b$
  w przeciwnym wypadku
    $b \leftarrow b - a$

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

```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

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

Algorytm 2 Algorytm Euklidesa

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

Algorytm 3 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: $\quad i \leftarrow i + 1$

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

Wizualizacja algorytmów sortowania: ☞ AlgoRythmics

Esencja programowania proceduralnego

- ekonomiczność - ujednolica powtarzające się bloki programu - mniej kodowania
- przejrzystość, nawet przy złożonych i obszernych algorytmach
- podprogram staje się nową instrukcją elementarną

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

Złożoność obliczeniowa

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

Ile operacji wymaga ...

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

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

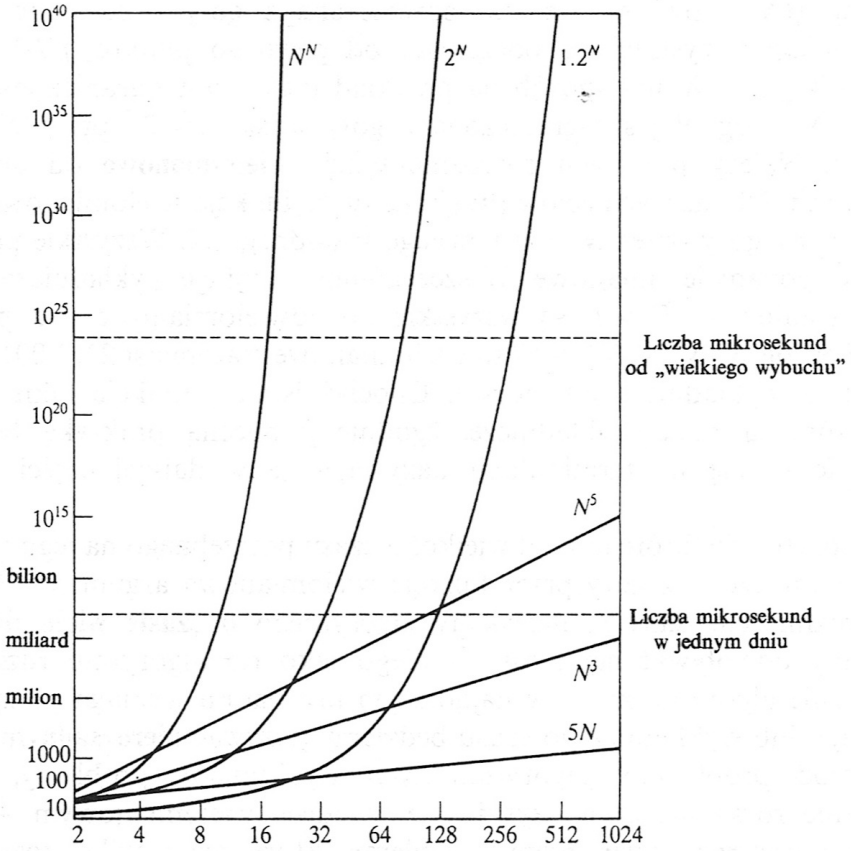
[2] D. Harel, Rzecz o istocie informatyki. Algorytmika.

- Rozsądne rozwiązania, wykonywalne w czasie wielomianowym

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

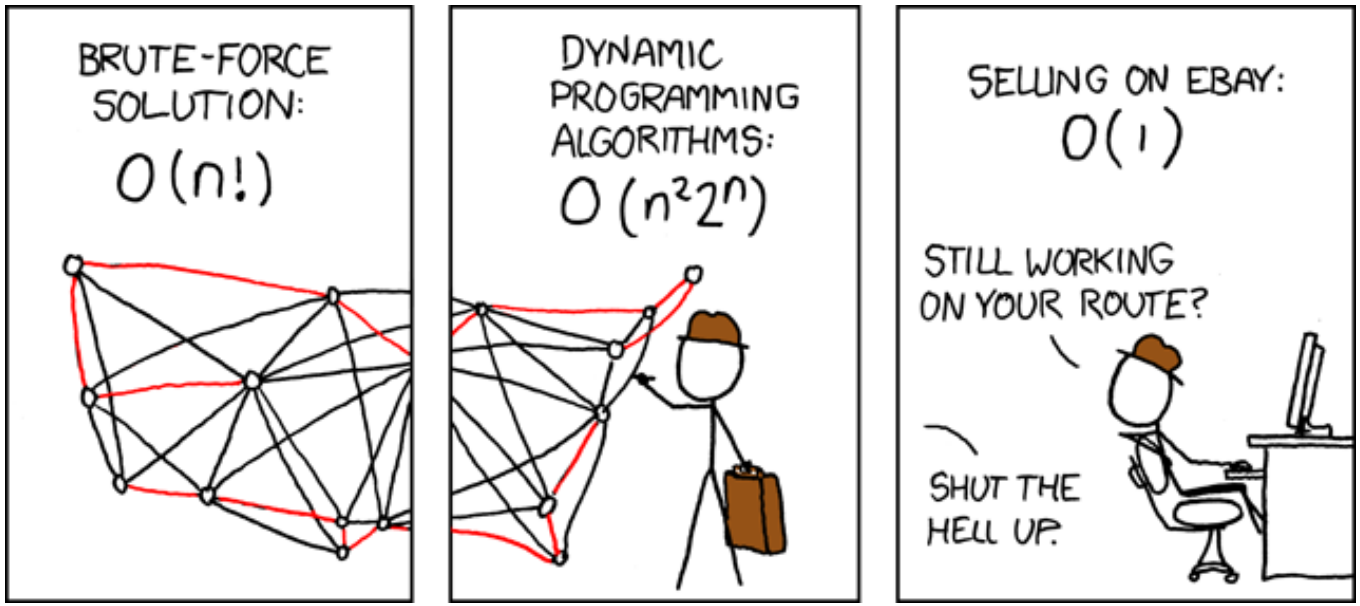
- Nierozsądne, niepraktyczne, czas ponad-wielomianowy

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



Rys. 7.4. Tempo wzrostu pewnych funkcji

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



http://xkcd.com

Algorytmy manipulują danymi

dane wejściowe ⇒ Algorytm ⇒ Dane wyjściowe

Reprezentacja danych kształtuje algorytm!

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

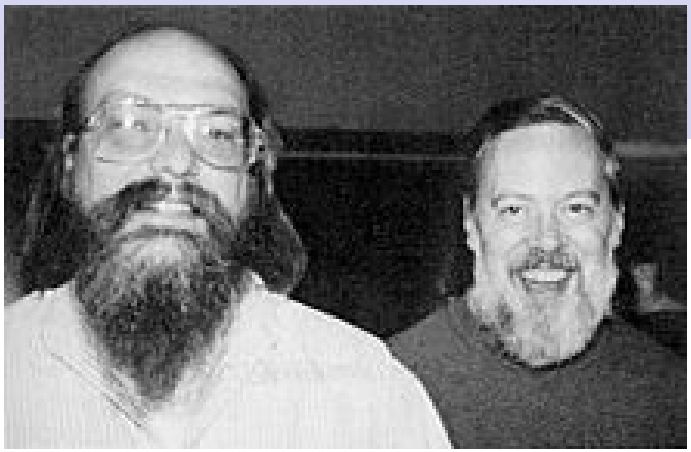
Od problemu do rozwiązania

- 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

```c
#include <stdio.h>          /* Dyrektywy preprocesora */
#define PI 3.1415

int main()                  /* Funkcja glowna */
{
    int i;                  /* Deklaracje */
    float x;

    i = 10 * PI;            /* Instrukcje */
    x = 1.0;

    return 0;
}
```

Najkrótszy program w C

```
main(){}
```

Witaj świecie

```
#include<stdio.h>

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

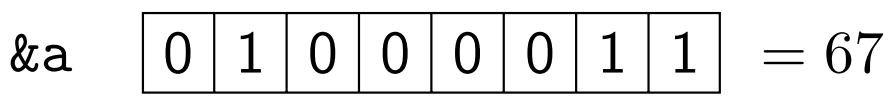
- mały język ale duże możliwości, ważna rola bibliotek
- pliki źródłowe `*.c`
- pliki nagłówkowe `*.h`, zawierają deklaracje typów i funkcji, nie zawierają instrukcji
- biblioteki: zbiory typów danych, stałych i funkcji
- modularność: programowanie proceduralne
- dostęp do pamięci i rejestrów
- zwięzły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

1. **preprocesor**
   pliki źródłowe (`*.c, *.h`) $\Rightarrow$ przetworzone pliki
   dyrektywy preprocesora, np.:
   `#include<stdio.h>`
   `#define PI 3.14`
2. **kompilacja**
   przetworzone pliki $\Rightarrow$ pliki obiektowe (`*.obj, *.o`)
3. **konsolidacja (linkowanie)**
   pliki obiektowe + biblioteki $\Rightarrow$ program (`*.exe , a.out`)

Binarna reprezentacja zmiennych

- zmienna zajmuje pamięć: 1B, 2B, 4B, 8B, ...
- adres zmiennej w pamięci:
- operacje na bitach
- typy całkowite ze znakiem i bez znaku, skończony zakres
- typy zmiennoprzecinkowe: rozdzielczość i skończoność

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



- `float` - zmiennoprzecinkowa (*floating point*),
  4 bajty
  zakres $[-3.4 \times 10^{38}, 3.4 \times 10^{38},]$,
  wartość minimalna większa od zera 1.17×10^{-38}
- `int` - liczby całkowite (*integer*)
  4 bajty
  zakres $[-2^{31}, 2^{31}]$
- `char` - znak (*character*)
  1 bajt
  zakres $[-127, 128]$, kod ASCII
- brak typu logicznego
  0 to fałsz, nie 0 to prawda
- `void` - typ pusty, brak typu
- `unsigned int` - całkowita bez znaku

Deklaracja

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

Identyfikatory

- litery `a-z`, `A-Z`, cyfry `0-9` (ale nie pierwsza), podkreślnik `_`
- małe i duże litery rozróżniane: `zmienna`, `Zmienna`, `ZMIENNA`

Przykłady deklaracji zmiennych:

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

Niepoprawne nazwy: `2abc`, `promień`, `wazna zmienna`, `pewna-zmienna`

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**

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`

Operatory arytmetyczne

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

Operatory relacji

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

Operatory logiczne

`!` `&&` `||`

Operator przypisania

`=`

```
zmienna = wyrażenie;
```

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

i = i + 3;                          /* brak inicjalizacji */
3 = x;                              /* zle */
x + y = x - y;                      /* zle */
znak = 65;
```

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

Operatory arytmetyczne

*	/	%	wyższy priorytet
+	-		niższy priorytet

Reszta z dzielenia (modulo, %) tylko dla zmiennych całkowitych
Dzielnie bez reszty (/) dla zmiennych całkowitych

Kolejność obliczeń

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

```
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);
```

Obliczenia od lewej do prawej, priorytet * / % przed + -

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

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

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

Funkcja printf()

formatowane wyjście (wyświetlanie w terminalu)

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

Funkcja scanf()

formatowane wejście (odczyt z klawiatury)

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

Format

%f	zmiennoprzecinkowa	`printf("%f",3.14);`	3.140000
%.2f	2 miejsca po przecinku	`printf("%.2f",3.14);`	3.14
%e	notacja naukowa	`printf("%e",0.01);`	1.000000e-02
%d	dziesiętne	`printf("%d",75);`	75
%c	znak	`printf("%c",75);`	K
%x	szesnastkowo	`printf("%x",75);`	4b
%s	łańcuch znakowy	`printf("%s","napis");`	napis

Symbole specjalne

\n	nowa linia	\\	\ *backslash*
\t	tabulator	\' \" \?	' " ?
\b	backspace	\r	powrót karetki
%%	%		

- dopasowanie formatu do typu zmiennej

```
int a;
scanf("%f",&a);                    /* problem !!! */
```

- argumentem jest adres zmiennej (pamiętaj o &)
- formaty liczbowe %f i %d zjadają poprzedzające białe znaki
- spacja w formacie pomija wszystkie białe znaki
- znak niepasujący do formatu przerywa wczytywanie i pozostaje w strumieniu wejściowym

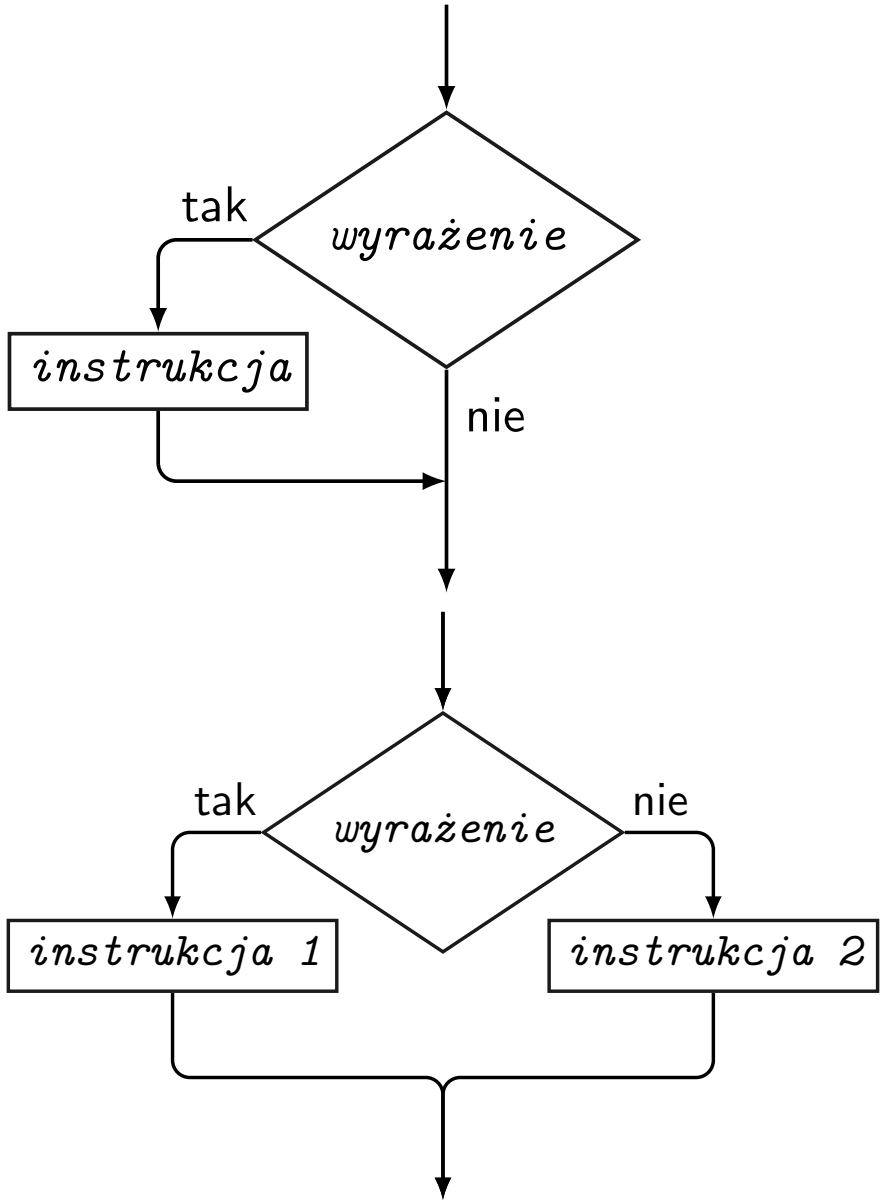
```
char a;
float x,y;
scanf("%f%f",&x,&y);          /* wczytanie 2 liczb */
scanf("%c",&a);               /* czyta znak */
scanf(" %c",&a);              /* pomija biale znaki */
```

Warunek if (jeżeli)

```
if ( wyrażenie )
    instrukcja
```

Warunek if else

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



Przykład: Równanie z jedną niewiadomą

Problem: znajdź miejsce zerowe funkcji liniowej

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

Algorytm 4 Równanie z jedną niewiadomą

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

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

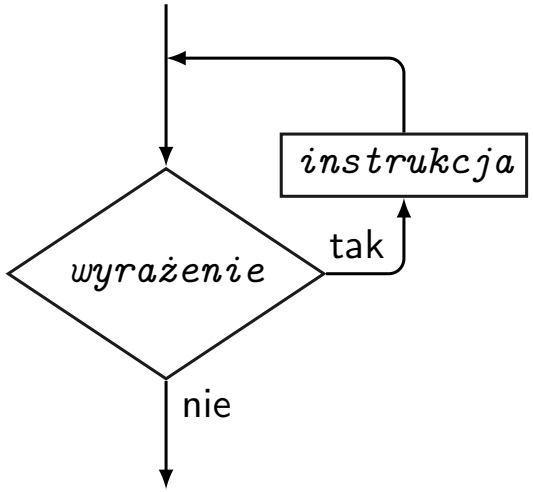
1: **jeżeli** $a \neq 0$ **wykonaj**
2: $\quad x_0 \leftarrow -\frac{b}{a}$
3: $\quad$ **wypisz:** x_0
4: **w przeciwnym wypadku**
5: $\quad$ **wypisz:** `Brak rozwiązania`

```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

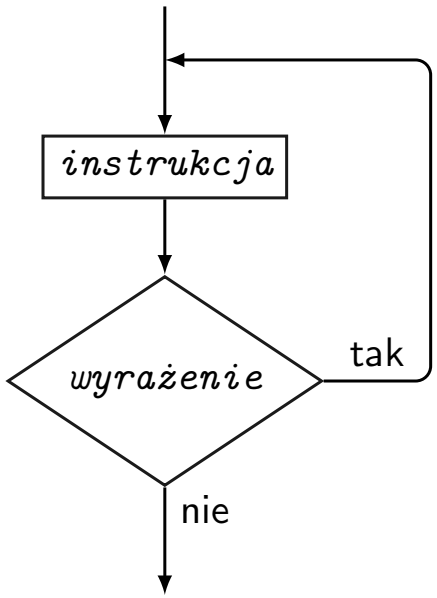
Pętla while (dopóki)

```
while ( wyrażenie )
    instrukcja
```



Pętla do while

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



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

Algorytm 5 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
    {
        i=2;  x=1;
        while( i <= n )
        {
            x = x * i;
            i = i + 1;
        }
        printf("%d! = %d\n",n,x);
    }
    return 0;
}
```

☞ silnia.c

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

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

Algorytm 6 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

```
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

Standard C99

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

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

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

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

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

Funkcje

czyli jak programować proceduralne.

```c
#include <stdio.h>                          /* Dyrektywy preprocesora */
#define PI 3.1415

float g = 2.5;                              /* Zmienne globalne */

float kwadrat(float x)                      /* Definicje funkcji */
{
   return x*x;
}

int main()                                  /* Funkcja glowna */
{
    float x, y;                             /* Zmienne lokalne */

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

    return 0;
}
```

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
- Nic za darmo - wywołanie funkcji to dodatkowy koszt czasu i pamięci
- Rekurencja(Rekurencja(Rekurencja(Rekurencja(...))))

Deklaracja funkcji (prototyp)

Zapowiedź użycia funkcji - określenie nazwy funkcji i typów

```
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);
```

```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;
}
```

☞ euclid3.c

Parametry formalne i aktualne funkcji

Parametry formalne

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

Parametry aktualne

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

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

W języku C argumenty są przekazywane wyłącznie przez wartość !

Funkcja bez wartości zwracanej - procedura

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

Funkcja z wartością zwracaną

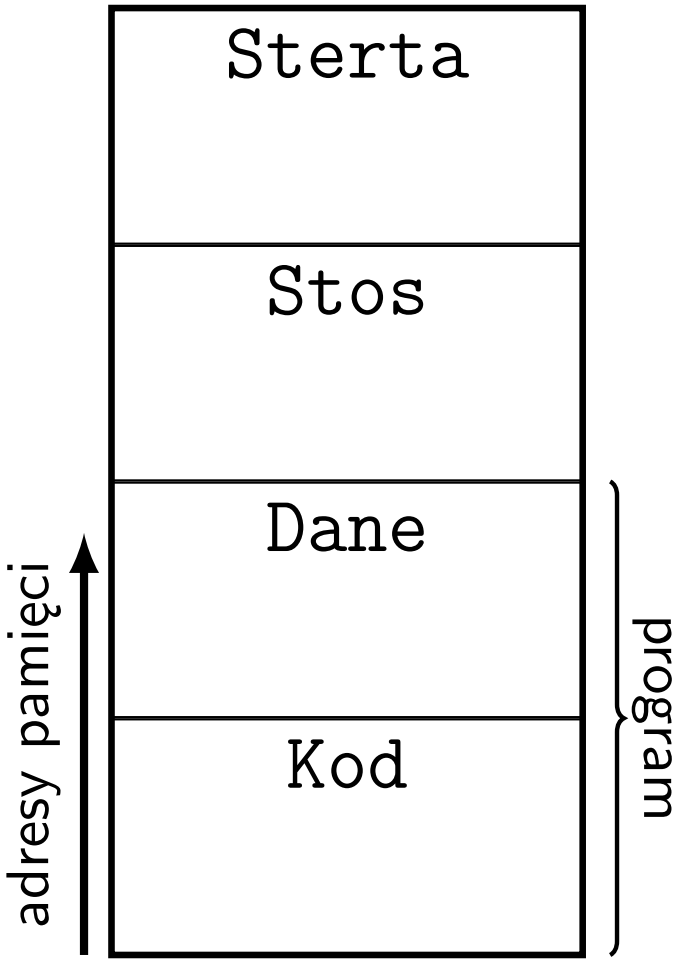
```
int silnia(int n)
{
   int i=2;  x=1;
   while( i <= n )
   {
      x = x * i;
      i = i + 1;
   }
   return x;
}
```

- Instrukcja `return` może pojawić się w dowolnym miejscu wewnątrz funkcji
- Wartość zwracana a typ deklarowany

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

Wartość zwracana jest podstawiona w miejsce wywołania

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



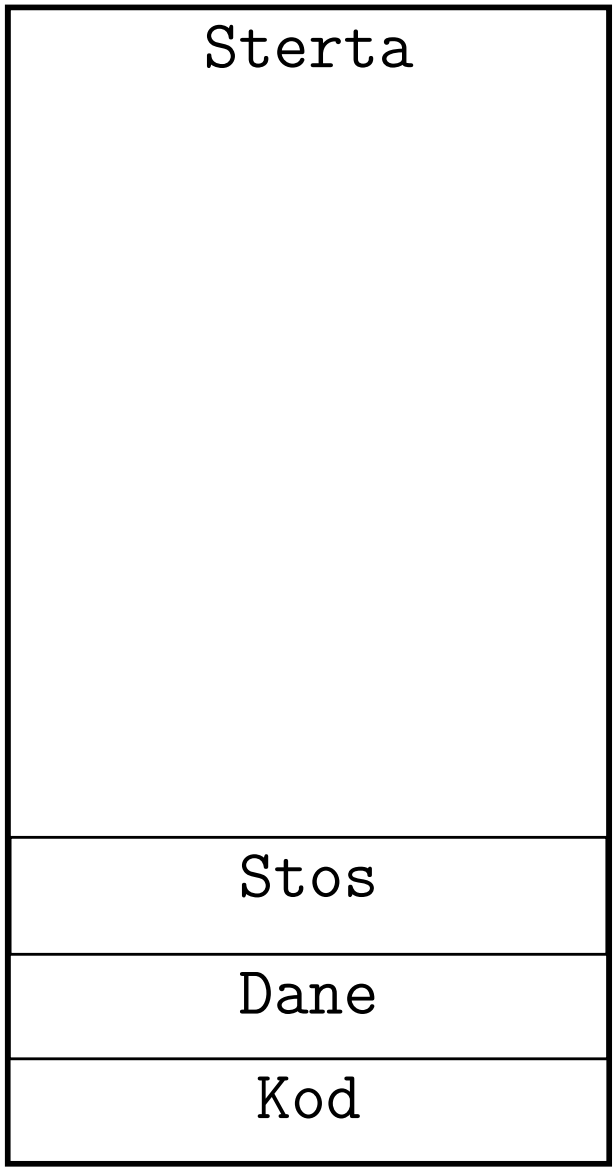
pamięć dostępna do alokacji

zmienne lokalne funkcji wkładane na stos przy wywołaniu funkcji

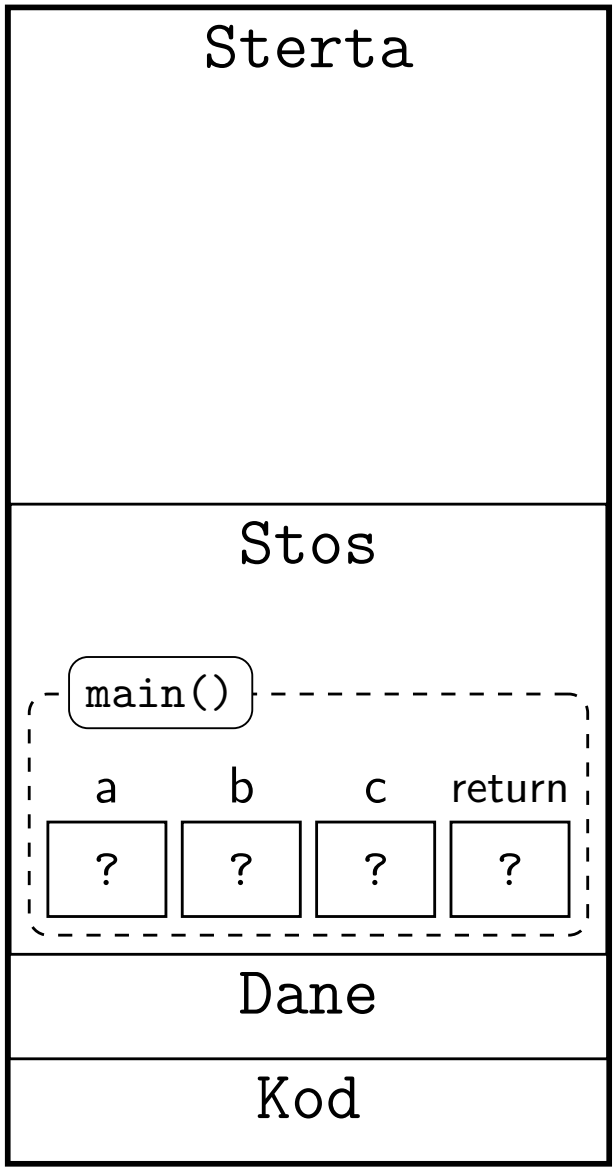
stałe oraz **zmienne globalne** i statyczne inicjowane przy uruchomieniu

tekst programu
tylko do odczytu

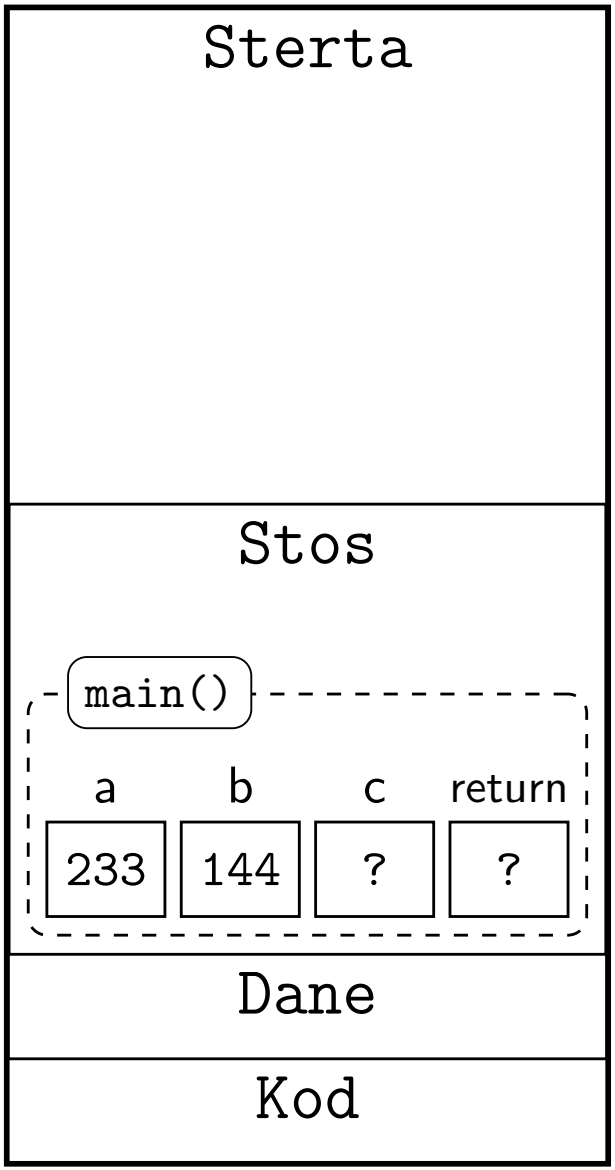
```
 1  int nwd(int a, int b)
 2  {
 3      int c;
 4      while (b != 0)
 5      {
 6          c = a % b;
 7          a = b;
 8          b = c;
 9      }
10      return a;
11  }
12
13  int main()
14  {
15      int a, b, c;
16      a = 233;
17      b = 144;
18      c = nwd(a, b);
19      return 0;
20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



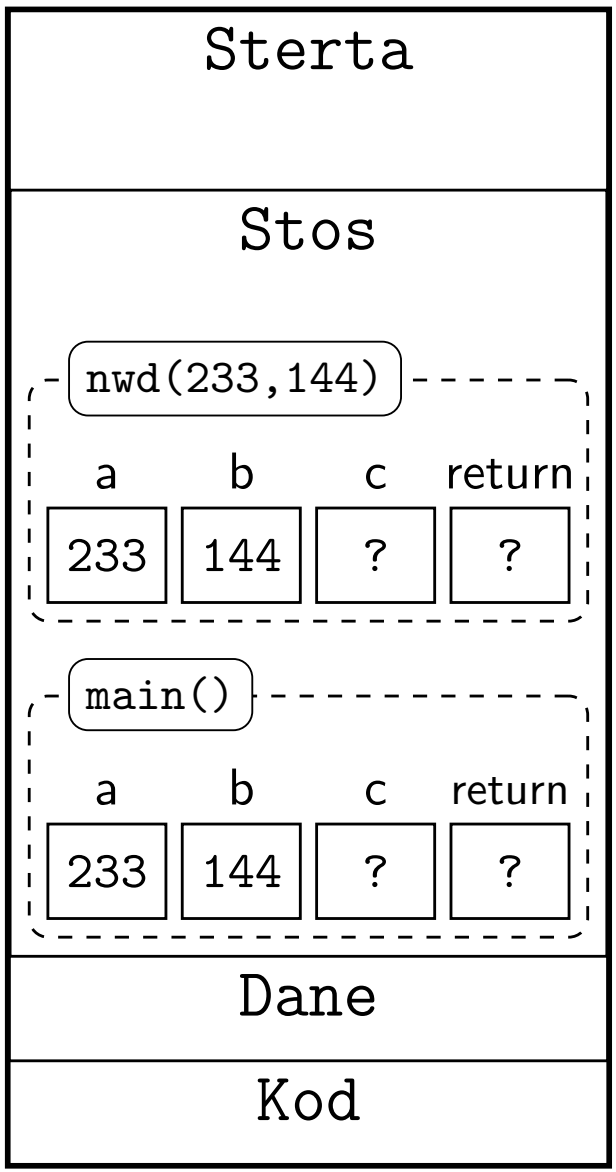
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



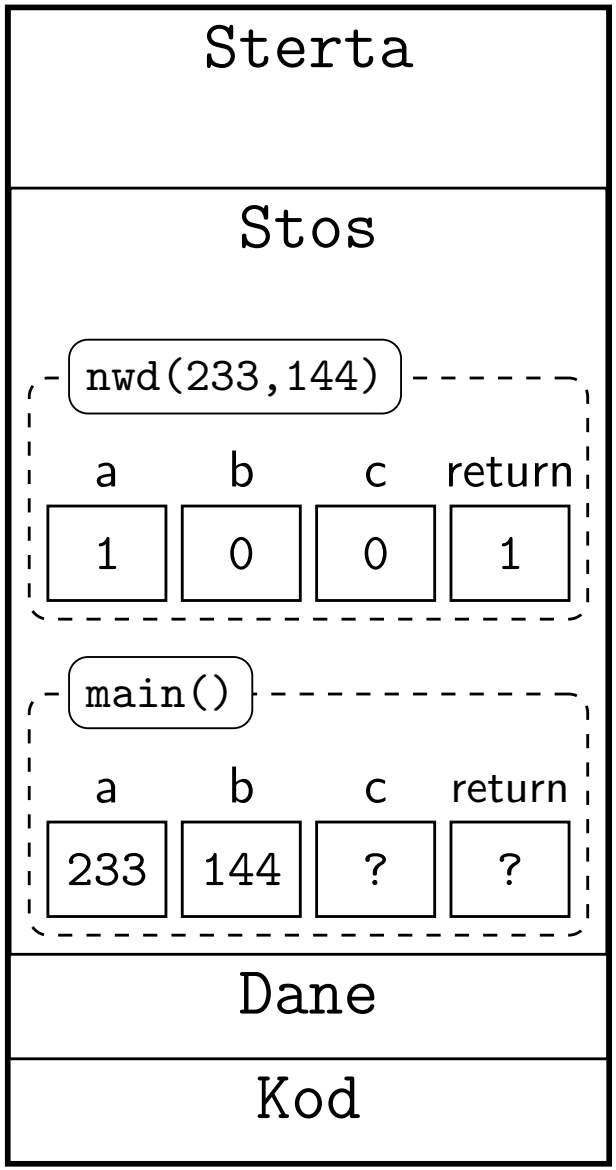
```
 1  int nwd(int a, int b)
 2  {
 3      int c;
 4      while (b != 0)
 5      {
 6          c = a % b;
 7          a = b;
 8          b = c;
 9      }
10      return a;
11  }
12
13  int main()
14  {
15      int a, b, c;
16      a = 233;
17      b = 144;
18      c = nwd(a, b);
19      return 0;
20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



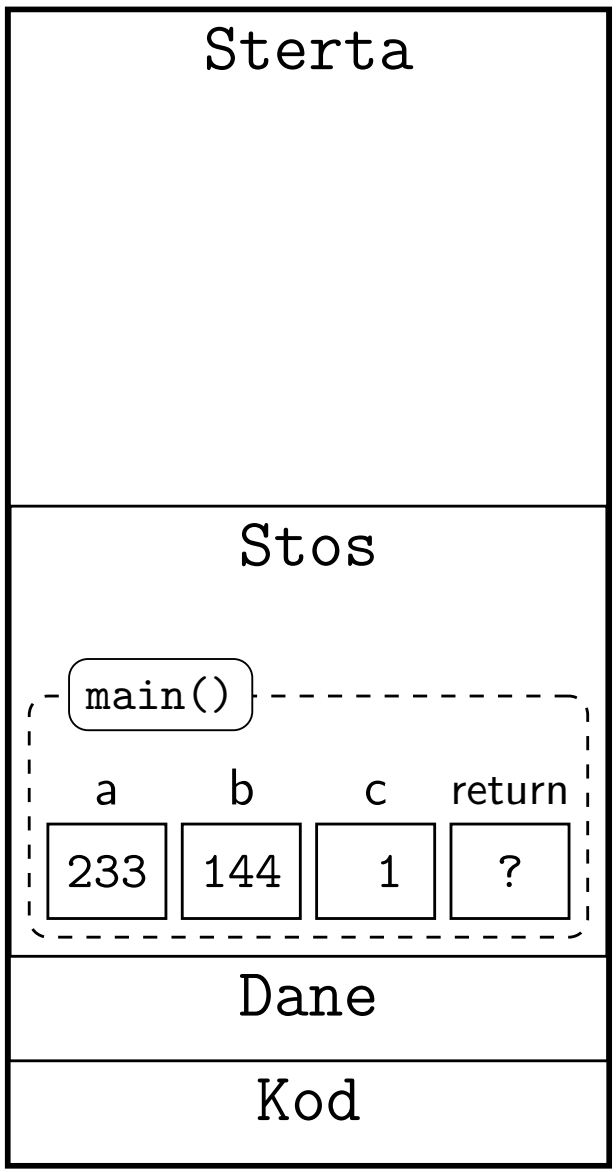
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



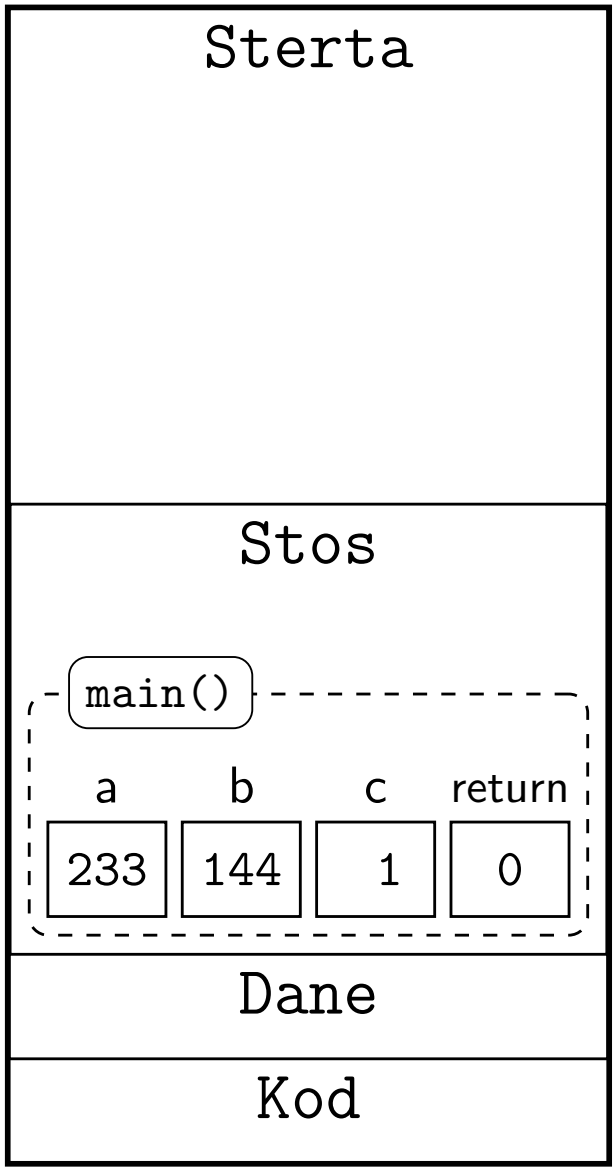
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



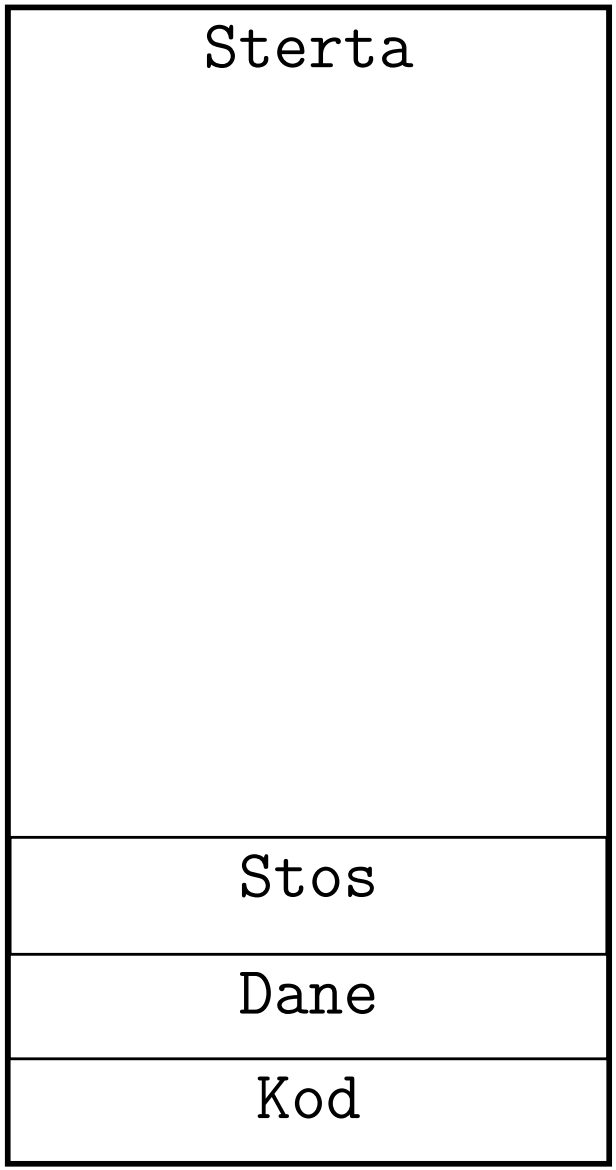
```
1  int nwd(int a, int b)
2  {
3      int c;
4      while (b != 0)
5      {
6          c = a % b;
7          a = b;
8          b = c;
9      }
10     return a;
11 }
12
13 int main()
14 {
15     int a, b, c;
16     a = 233;
17     b = 144;
18     c = nwd(a, b);
19     return 0;
20 }
```



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



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



Zmienna globalna

dostępna dla wszystkich funkcji programu. Zmienna istnieje przez cały czas działania programu.

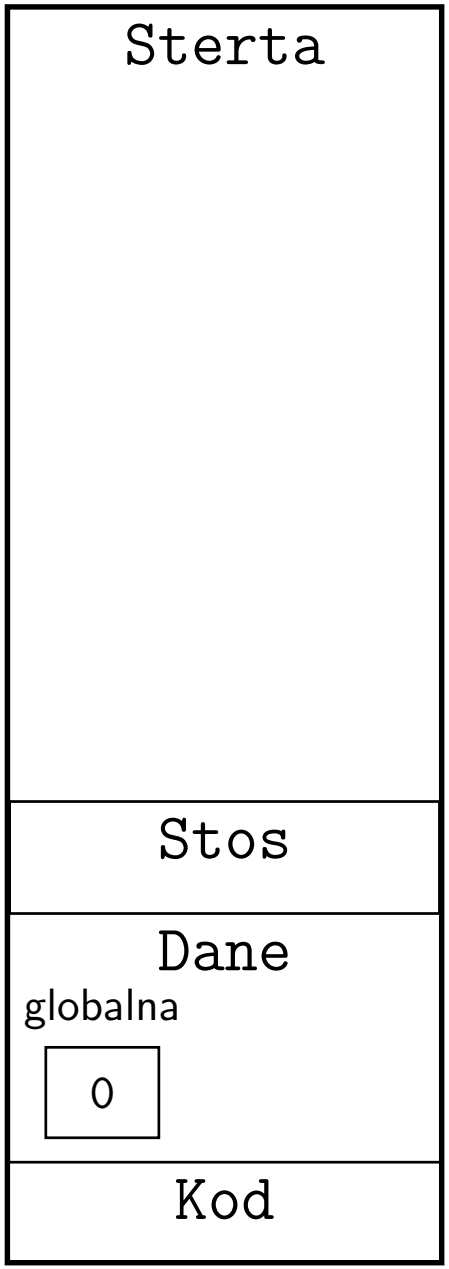
```
int a;
void funkcja()
{
    a = a + 1;
}
```

Zmienna lokalna (automatyczna)

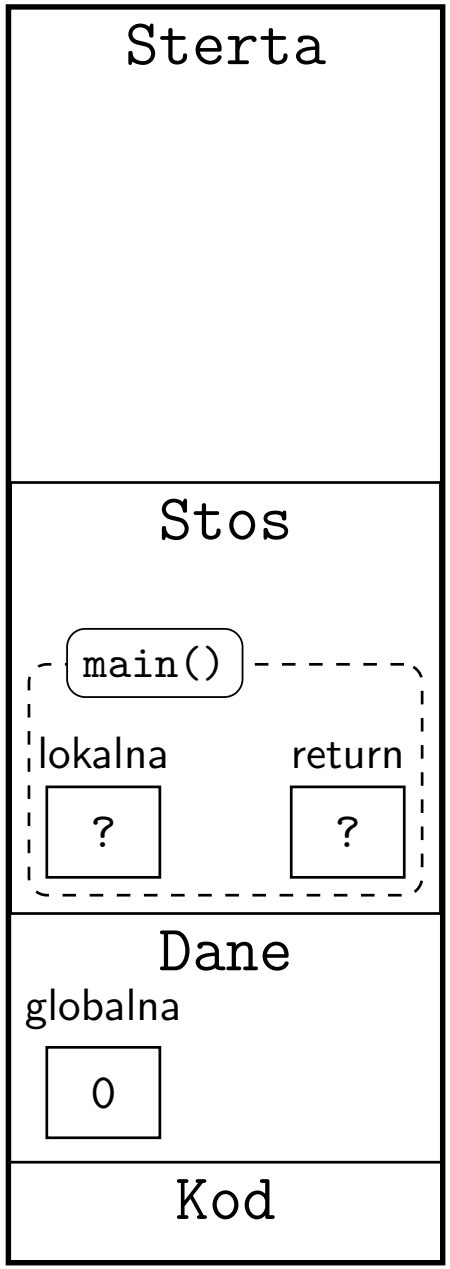
wewnętrzna zmienna funkcji. Czas życia ograniczony czasem działania funkcji.

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

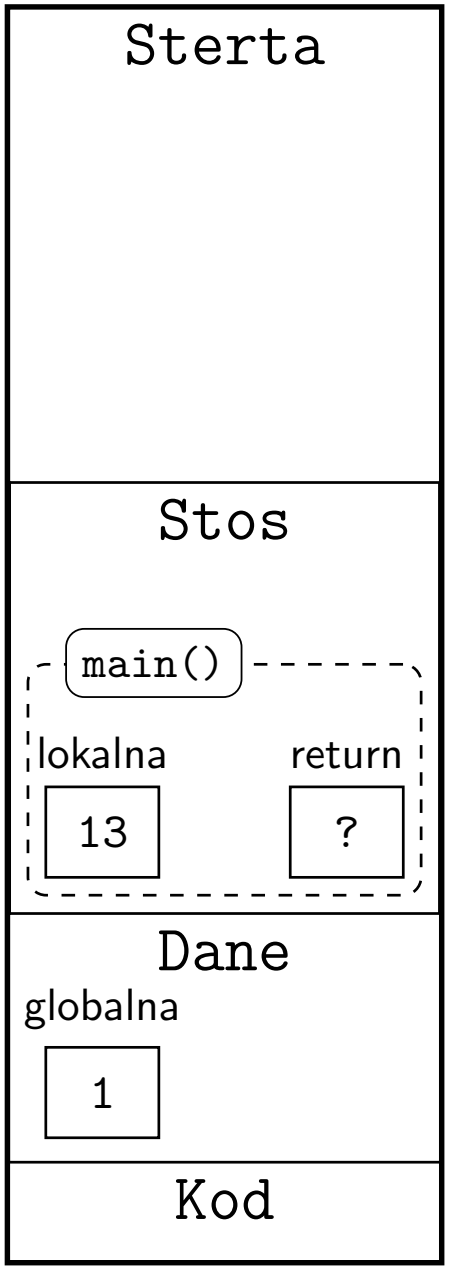
```c
#include<stdio.h>

int globalna = 0;

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

int main()
{
    int lokalna = 13;
    globalna = globalna + 1;
    f();

    return 0;
}
```



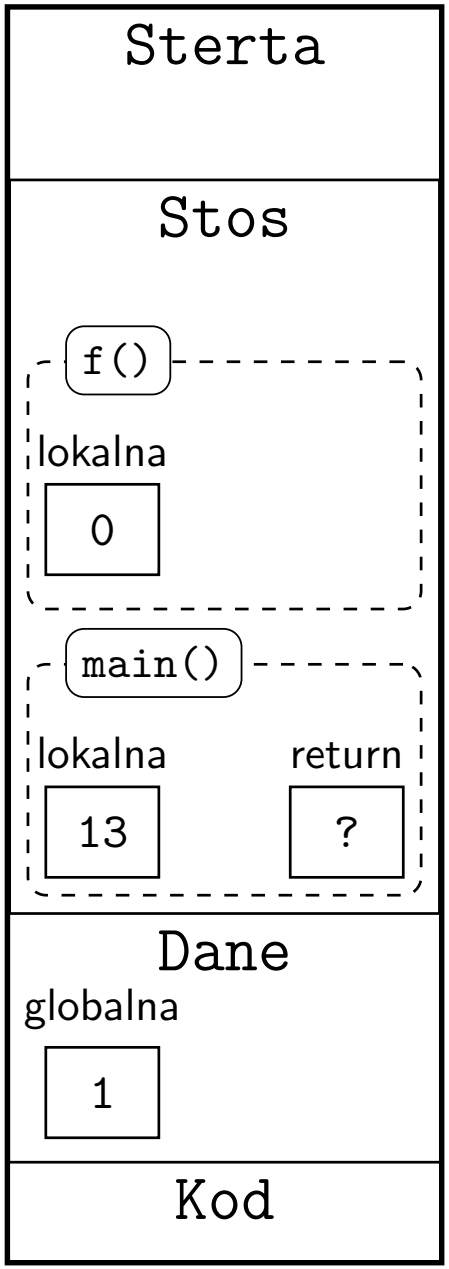
```c
#include<stdio.h>

int globalna = 0;

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

int main()
{
    int lokalna = 13;
    globalna = globalna + 1;
    f();

    return 0;
}
```



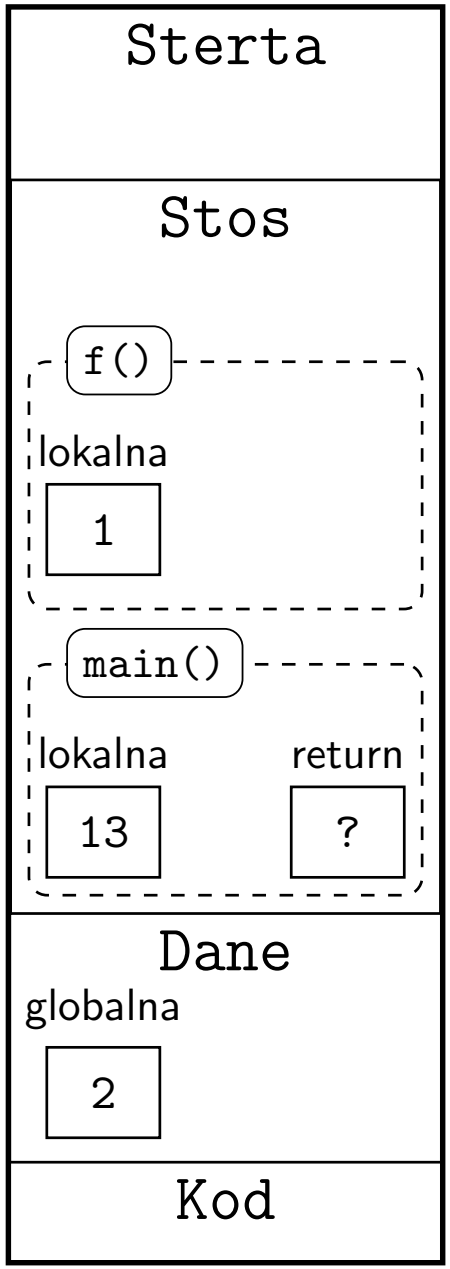
```c
#include<stdio.h>

int globalna = 0;

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

int main()
{
    int lokalna = 13;
    globalna = globalna + 1;
    f();

    return 0;
}
```



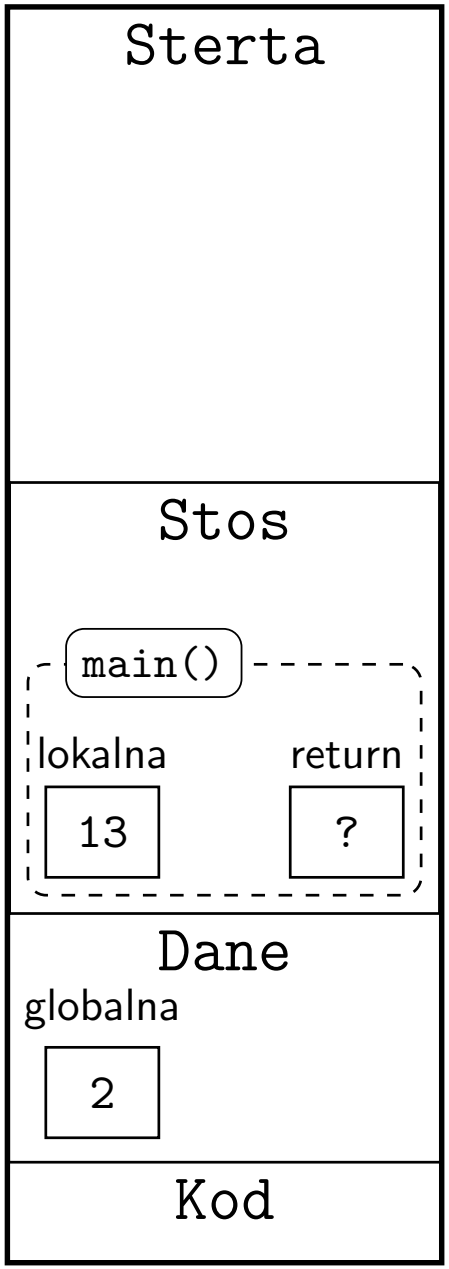
```c
#include<stdio.h>

int globalna = 0;

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

int main()
{
    int lokalna = 13;
    globalna = globalna + 1;
    f();

    return 0;
}
```



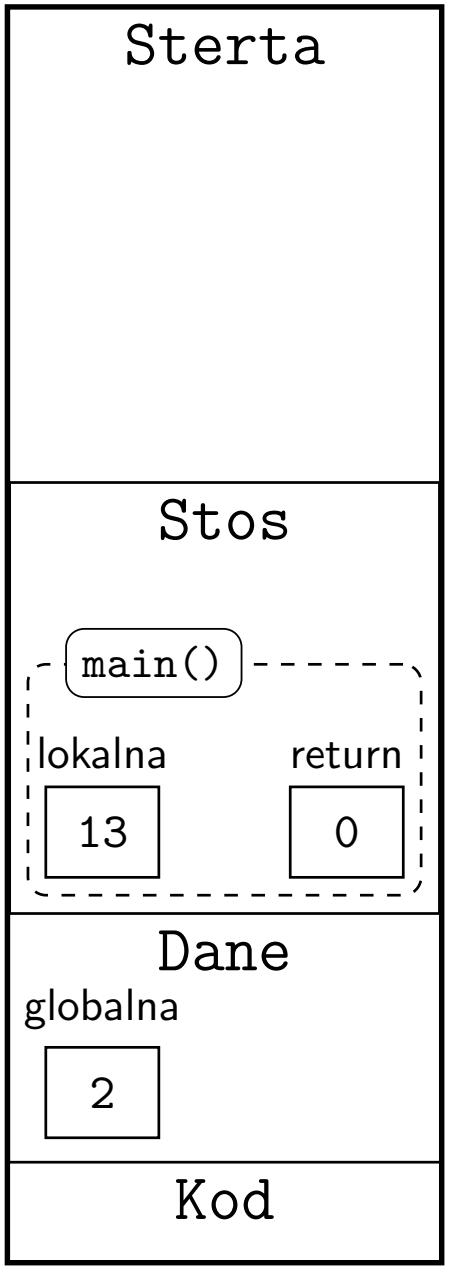
```c
#include<stdio.h>

int globalna = 0;

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

int main()
{
    int lokalna = 13;
    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 = 13;
    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 = 13;
    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 - diagnostyka kodu
`<ctype.h>`	Klasyfikacja znaków
`<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>`	Zestaw podstawowych narzędzi, np. `rand()`
`<string.h>`	Obsługa łańcuchów znakowych
`<time.h>`	Funkcje obsługi czasu

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

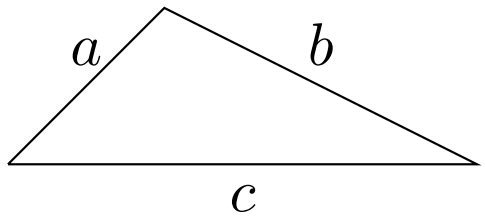
Biblioteka matematyczna

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

Uwaga: korzystając z kompilatora GCC należy dodać opcję `-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

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

☞ heron2.c

```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

- Zmienne globalne i lokalne
- Programowanie proceduralne - zmienne globalne z umiarem
- Deklaracja a definicja funkcji
- Funkcja przed użyciem musi być przynajmniej zadeklarowana
- `return` zwrócenie pojedynczej wartości
- Pytanie: czy można zwrócić z funkcji więcej wartości?
  Np. wyznaczanie miejsc zerowych paraboli.

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

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

 „C Reference”, `http://en.cppreference.com/w/c`

Tablice i struktury

czyli złożone typy danych.

Tablica

przechowuje elementy tego samego typu

struktura jednorodna, homogeniczna

Elementy identyfikowane liczbami (indeksem).

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

Struktura

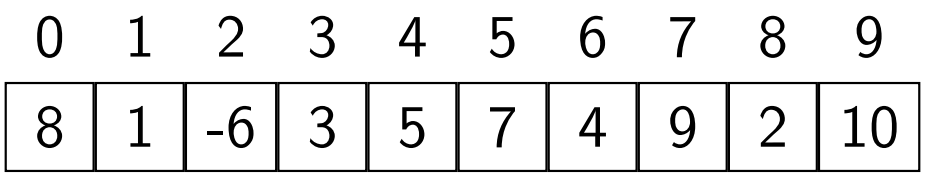
przechowuje elementy dowolnego typu

struktura niejednorodna, heterogeniczna

Elementy identyfikowane przez nazwy.

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

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



Deklaracja tablicy

```
typ identyfikator[rozmiar];
```

Przykład

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

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

- operator []
- indeksowanie wartościami całkowitymi
- brak kontroli zakresu tablicy

Przykłady

```
x[0] = 1.3;

napis[3] = 'x';

i = a[i];

a[i] = a[i] + 5;

x[i-1] = x[i];

x[maxind(x)] = x[0];
```

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

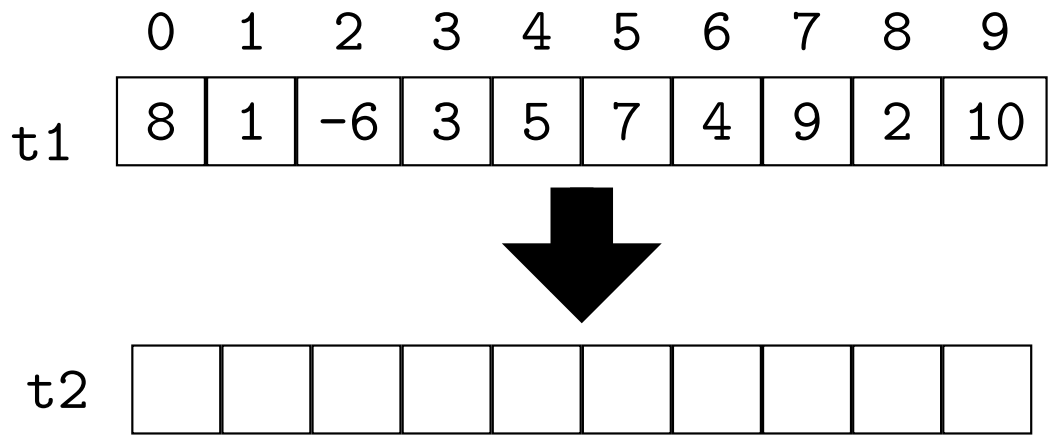
Wczytywanie wartości

```
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;

i = 0;
while(i<10)
{
    t[i] = 0;
    i = i + 1;
}
```

Kopiowanie tablic




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

t1 = t2;

i = 0;
while( i<10 )
{
    t2[i] = t1[i];
    i = i + 1;
}
```

Źle !

Deklaracja funkcji

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

Dobrze

Źle

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

Wywołanie funkcji

```
float t[10], x;
int n=10;

wczytaj(t,n);
x = max(t,n);
```

```
x = max(t[10],10);
x = max(t[],10);
```

Problem: w zbiorze zawierającym n elementów odnajdź element x.

Algorytm 7 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 ← i + 1
5: zwróć −1
```

Przykład w C: przeszukiwanie liniowe

```c
int szukaj(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?

Algorytm 8 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;
4      while( t[i] != x ) i = i + 1;
5      if( i != n ) return i;
6      return -1;
7  }
```

☞ przeszukiwanie2.c

- przechowuje zmienne dowolnego typu
- pole, czyli składowa struktury
- elementy (pola) identyfikowane nazwami
- operator dostępu bezpośredniego .
- ułatwiają organizację danych $\longrightarrow$ zalążek obiektowości
- jak każda zmienna struktura może być argumentem funkcji oraz wartością zwracaną z funkcji

"Bond"
"James"
007
3.2

Deklaracja struktury

```
struct nazwa
{
    typ identyfikator1;
    typ identyfikator2;
    ...
};
```

```
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;
```

```c
#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

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;
}
```

- 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 dla n punktów materialnych.

Punkt materialny

$$p_i = \{m, \vec{r}_i\}, \qquad \vec{r}_i = [x_i, y_i, z_i]$$

Środek masy dwóch punktów

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

Środek masy n punktów

$$\vec{r}_0 = \frac{\sum_{i=0}^{n} m_i\vec{r}_i}{\sum_{i=1}^{n} m_i}$$

Tablica 4 elementowa

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

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

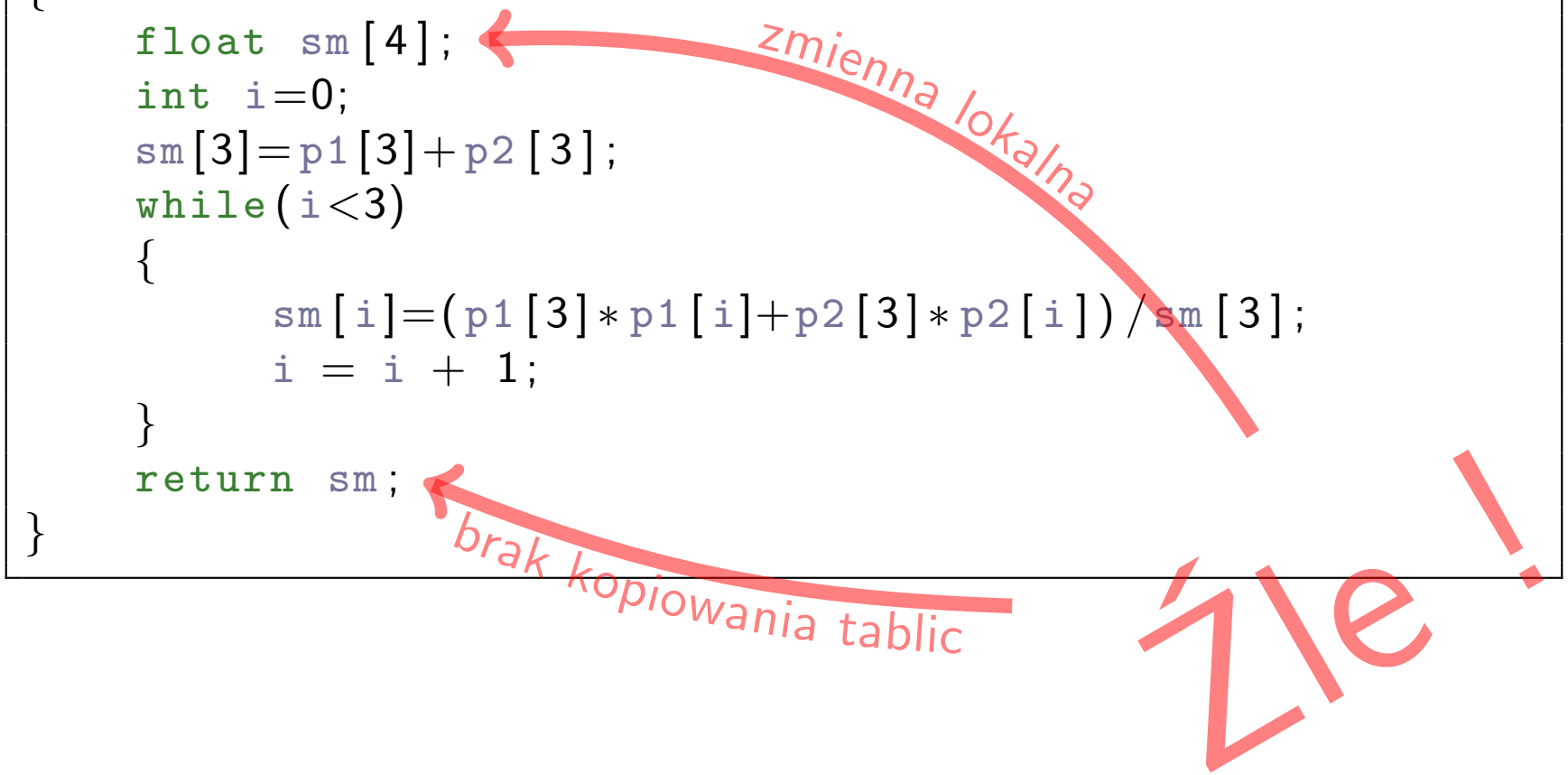
Struktura

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

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

```
float* srodek(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;
}
```

```
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;
}
```



```c
void srodek(float p1[], 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;
     }
}
```

☞ sm1.c

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

☞ 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 }
```

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

☞ sm2.c

Tablica tablic

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

Tablica struktur

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

Struktura z tablicami

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

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

Środek masy n punktów

Tablica struktur

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

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

☞ 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 < MAX );
13
14   printf("Aktualny zbior punktow:\n");
15   wypisz_chmure(c);
16   printf("Srodek masy:\n");
17   wypisz_punkt(srodek(c));
18
19   return 0;
20 }
```

☞ 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
- Reprezentacja danych a czytelność kodu i efektywność programu
- Dobra zasada: twórz funkcje do manipulowania złożonymi typami danych


Maciej M. Sysło, *„Algorytmy”*, WSiP, Warszawa, 2002.


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

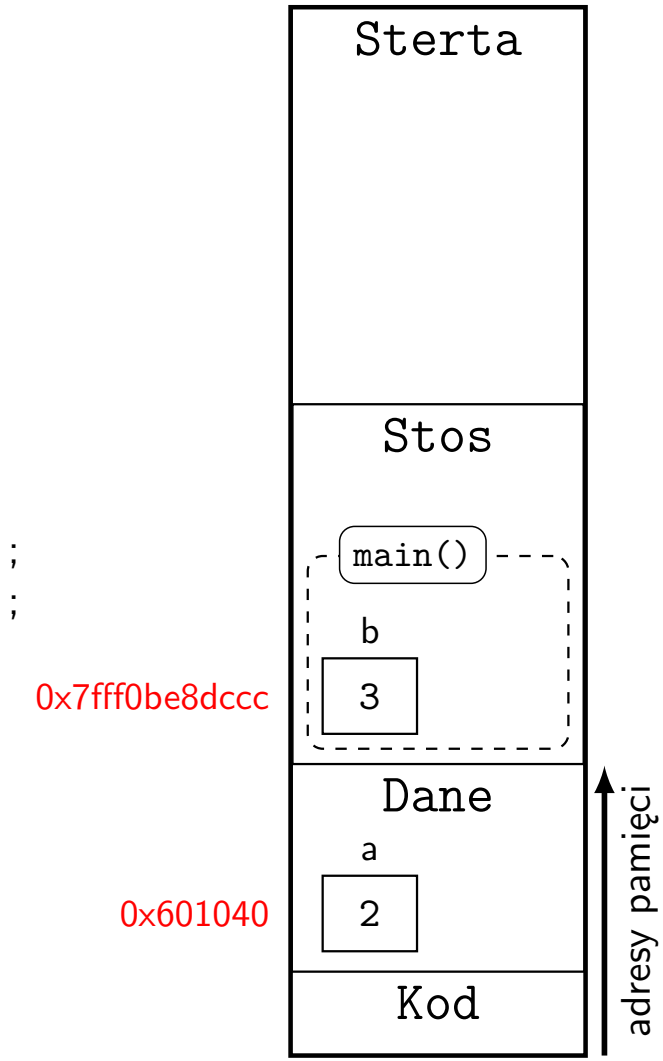

„Kurs programowania w C”, WikiBooks, `http://pl.wikibooks.org/wiki/C`

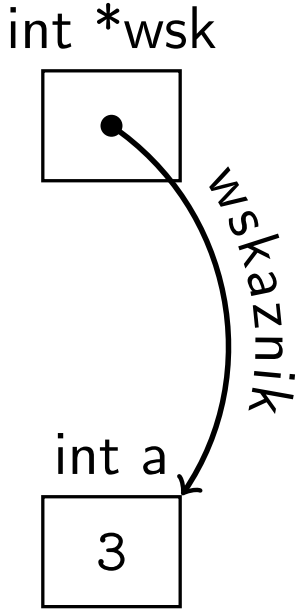
Wskaźniki

i zmienne przechowujące adresy.

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

```
adres zmiennej a 0x601040
adres zmiennej b 0x7fff0be8dccc
```





Wskaźnik (*pointer*)

adres zmiennej w pamięci (np. `&a`)

Zmienna wskaźnikowa

zmienna przechowująca adres

Typ wskaźnikowy

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

Deklaracja

```
typ *identyfikator;
```

Przykład

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

Operator referencji & zwraca adres zmiennej

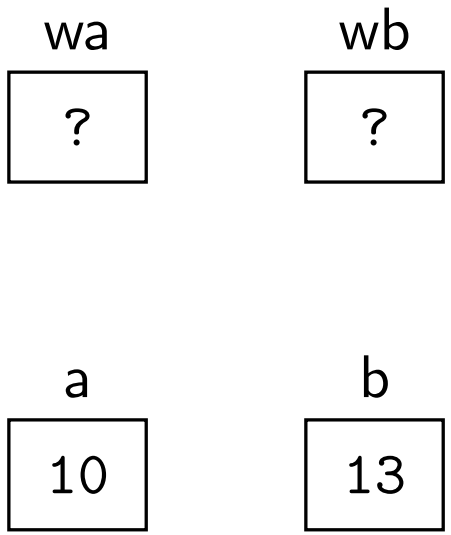
Przykład

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

Operator referencji & zwraca adres zmiennej

Przykład

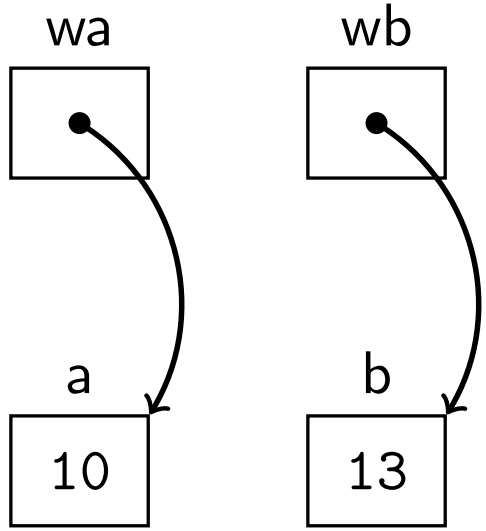
```
1  int a=10;
2  int b=13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```



Operator referencji & zwraca adres zmiennej

Przykład

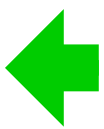
```
1  int a=10;
2  int b=13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```

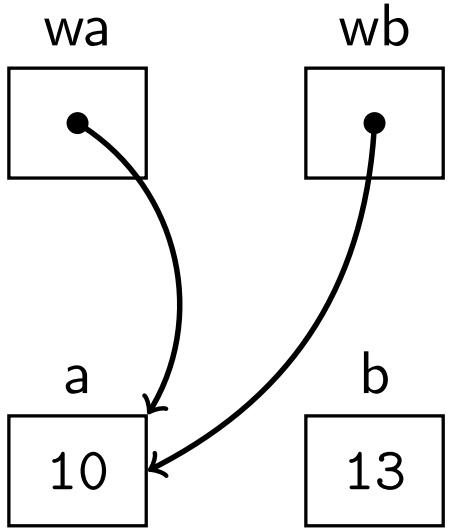


Operator referencji & zwraca adres zmiennej

Przykład

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





Operator dereferencji * daje dostęp do wskazanego adresu

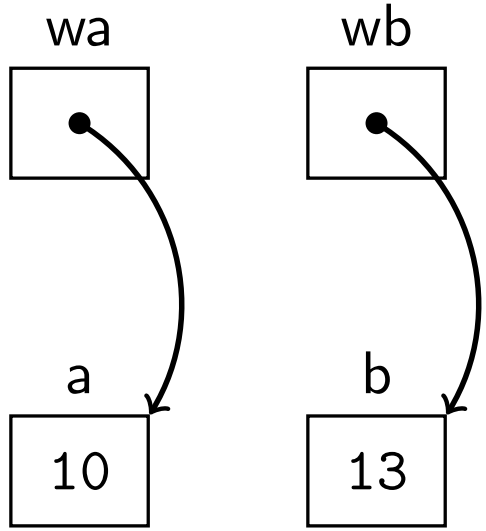
Przykład

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

Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

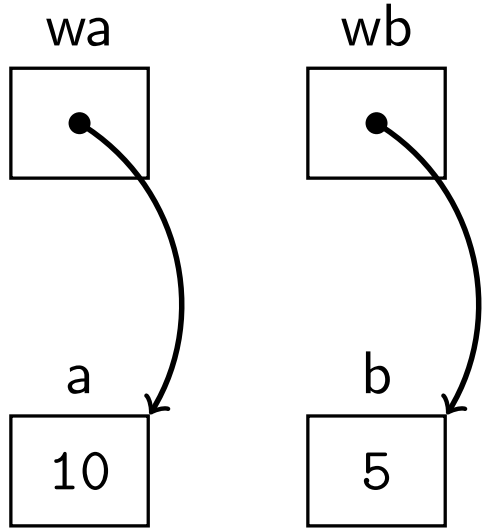
```
1   int a=10;
2   int b=13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wa = wb;
11  *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

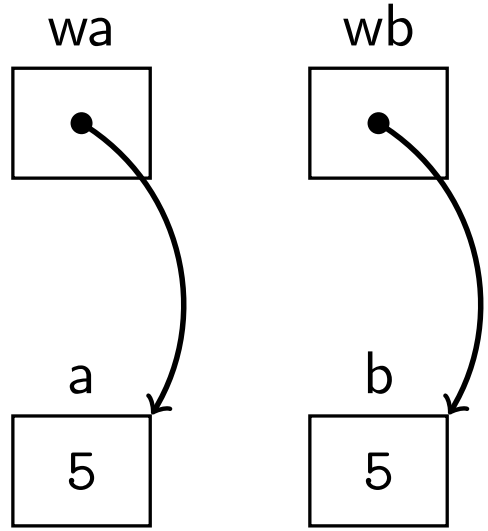
```
1   int a=10;
2   int b=13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wa = wb;
11  *wb = *wb + 1;
```



Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

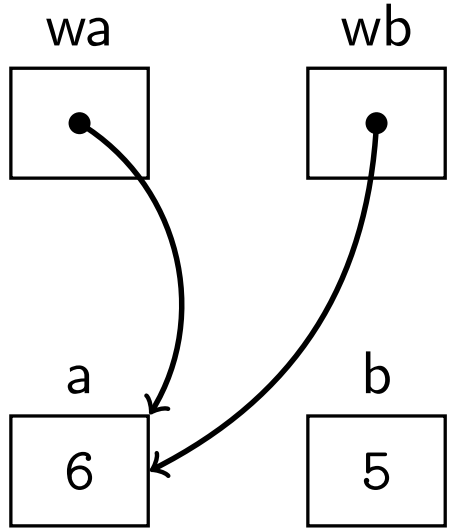
```
1   int a=10;
2   int b=13;
3   int *wa = &a;
4   int *wb = &b;
5
6   *wb = 5;
7
8   *wa = *wb;
9
10  wa = wb;
11  *wb = *wb + 1;
```



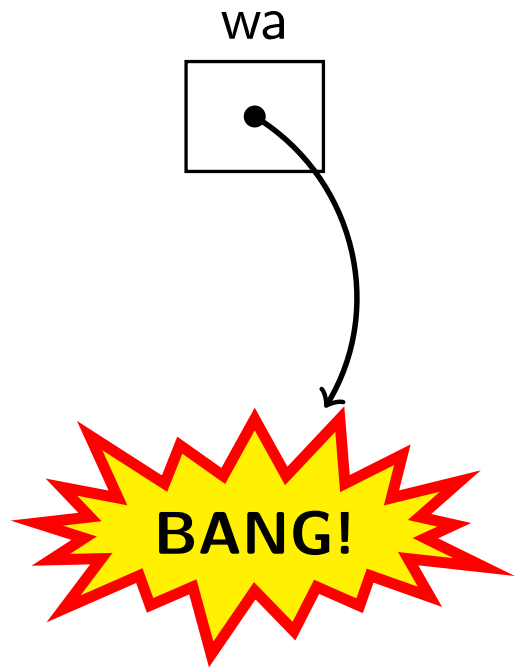
Operator dereferencji * daje dostęp do wskazanego adresu

Przykład

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



Uważaj na niezainicjowane zmienne wskaźnikowe !



Przykład

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

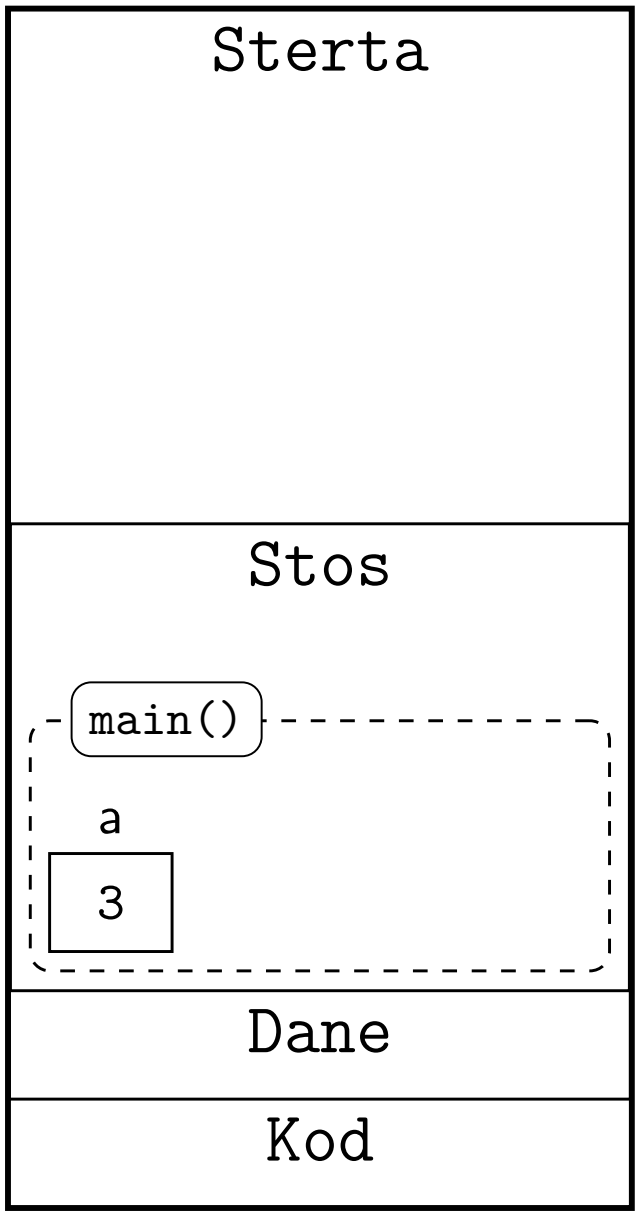
Adres 0 - NULL

```
int *wa;
wa = 0;
```

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

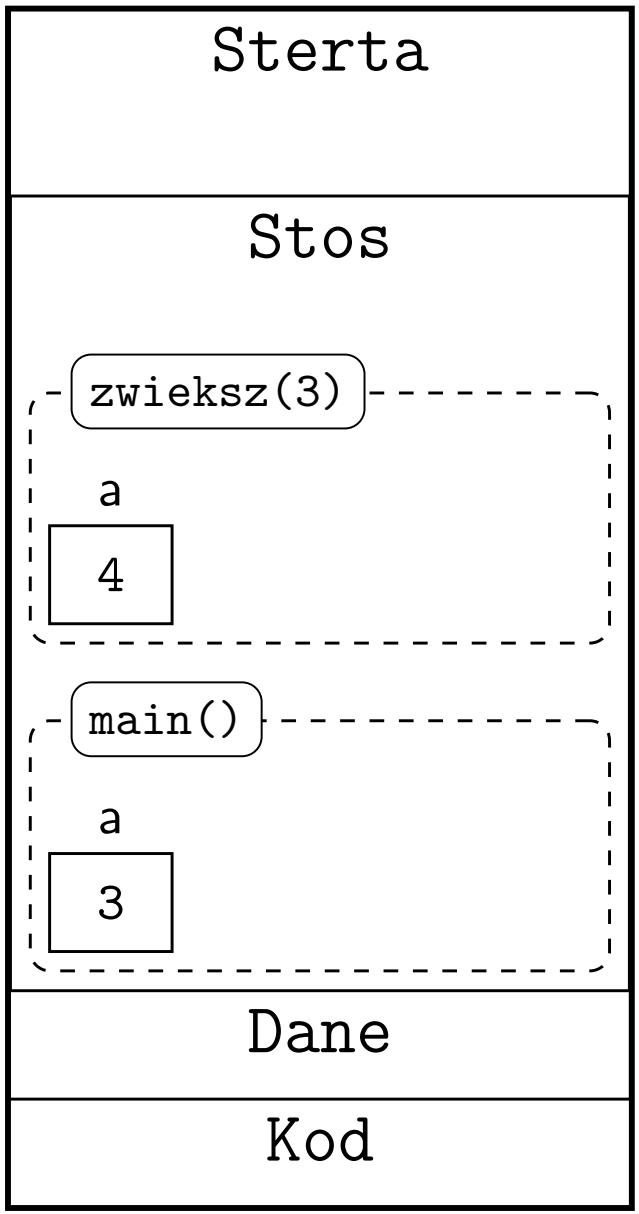
Wskaźniki jako argumenty funkcji

```
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 }
```



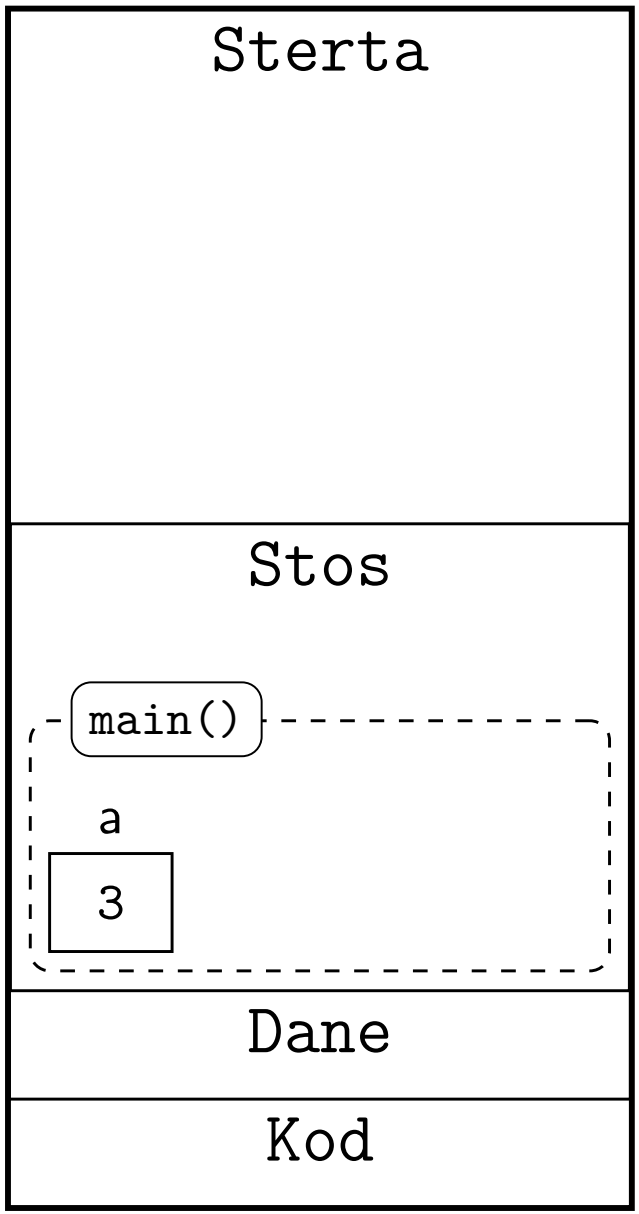
Wskaźniki jako argumenty funkcji

```
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 }
```



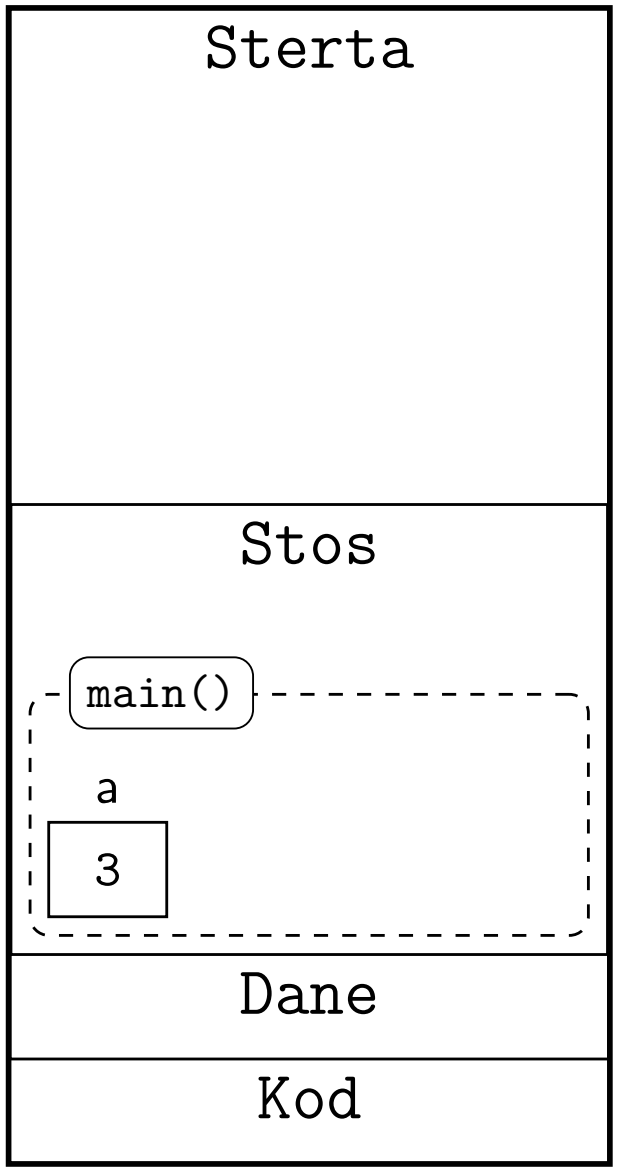
Wskaźniki jako argumenty funkcji

```
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 }
```



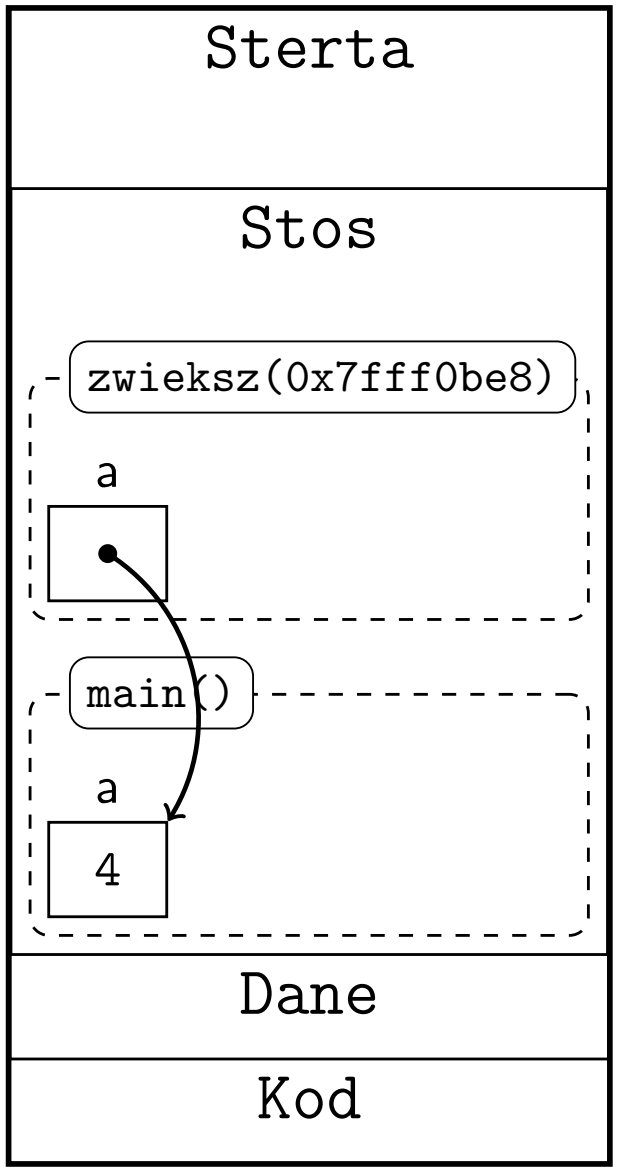
Wskaźniki jako argumenty funkcji

```
 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  }
```

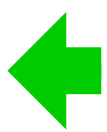


Wskaźniki jako argumenty funkcji

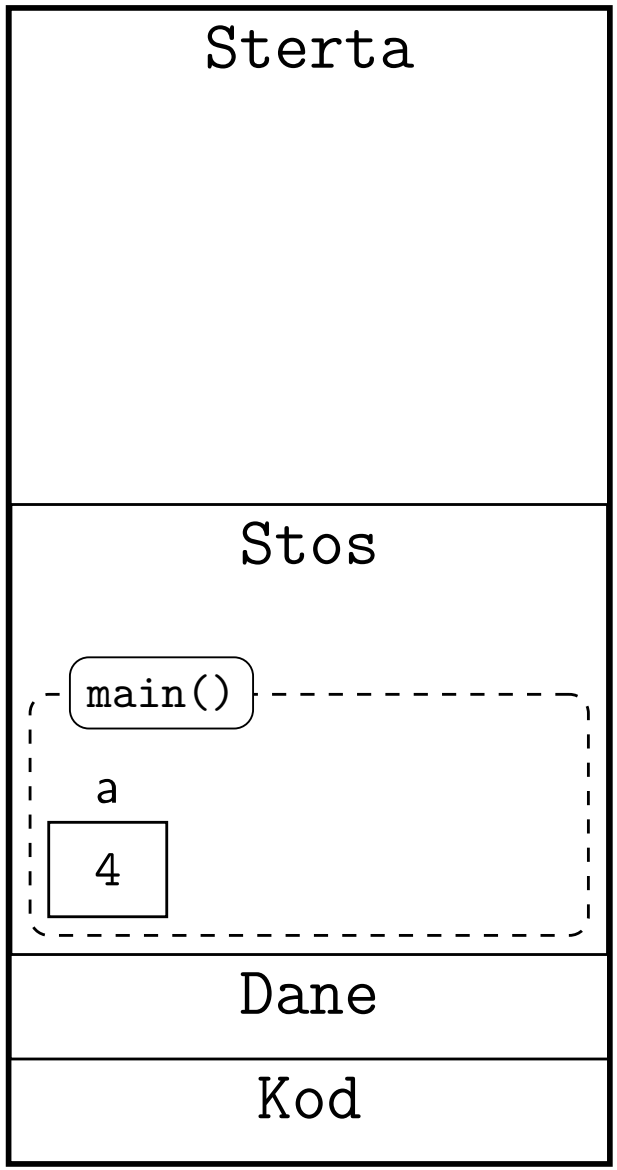
```
 1  #include<stdio.h>
 2
 3  void zwieksz(int *a)
 4  {
 5      *a = *a + 1;
 6  }
 7
 8
 9  int main()
10  {
11      int a = 3;
12
13      zwieksz(&a);
14      printf("%d\n",a);
15
16      return 0;
17  }
```



```c
#include<stdio.h>

void zwieksz(int *a)
{
    *a = *a + 1;
}


int main()
{
    int a = 3;

    zwieksz(&a);
    printf("%d\n",a);

    return 0;
}
```

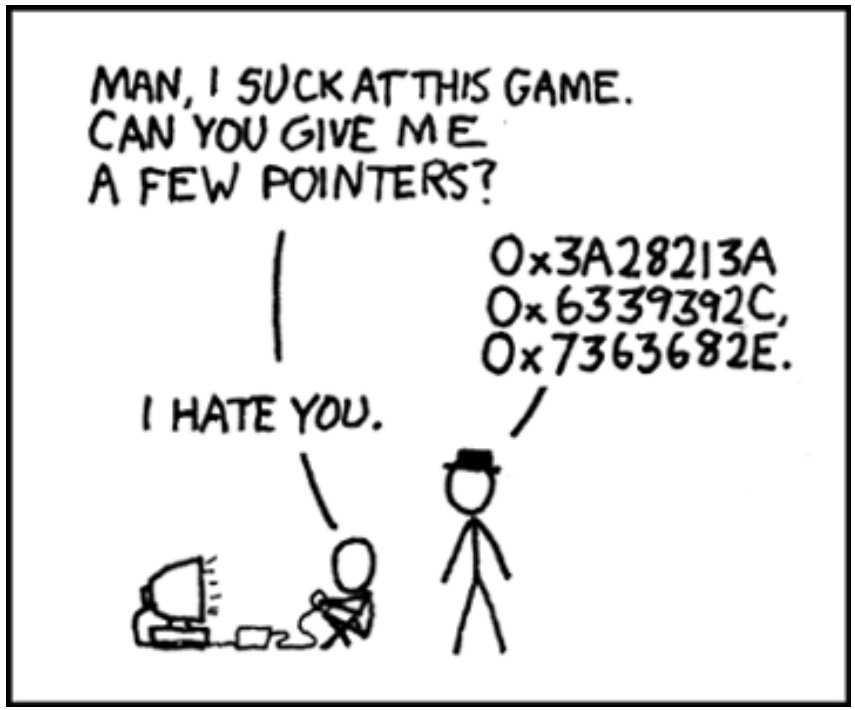


0x7fff0be8



Wskaźniki jako argumenty funkcji

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



http://xkcd.com

Problem: znajdź wartość minimalną i maksymalną w zbiorze liczb

Algorytm 10 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(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

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

Algorytm 11 Wyznaczanie maksimum i minimum

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

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

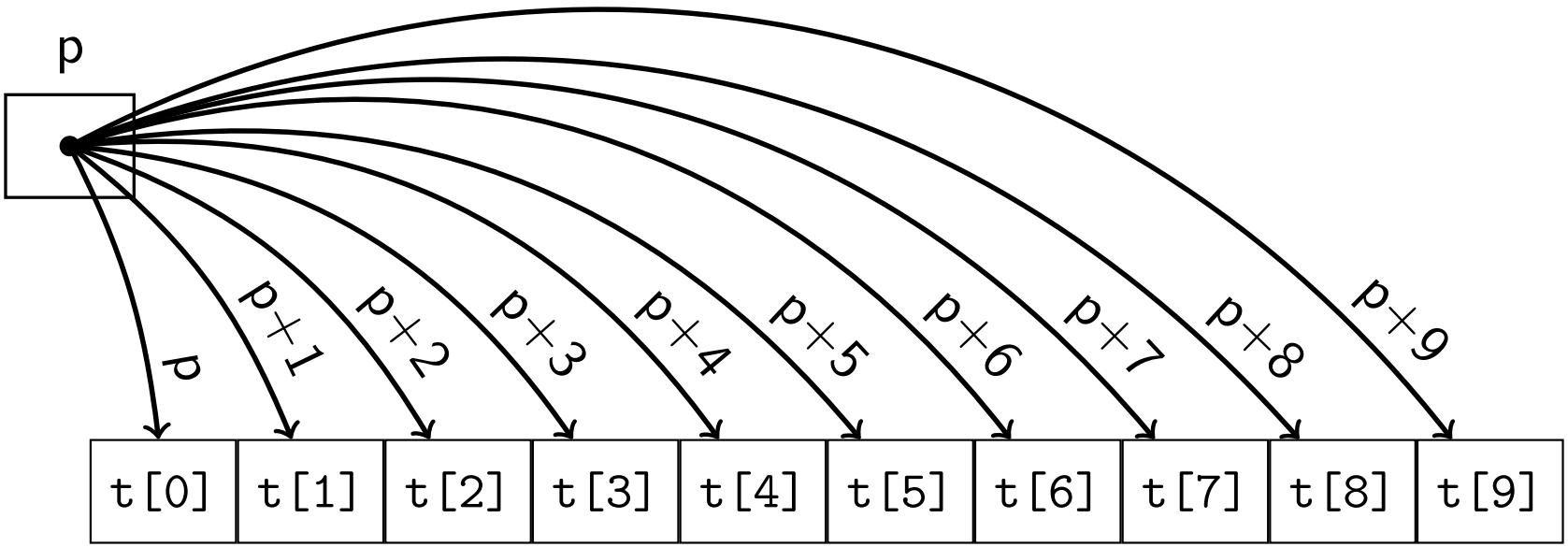
1: $\mathcal{A} \leftarrow \emptyset, \quad \mathcal{B} \leftarrow \emptyset$
2: **dla** $i = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$ **wykonuj**
3: **jeżeli** $x_{2i-1} < x_{2i}$ **wykonaj**
4: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i-1}\}, \qquad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i}\}$
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(float t[], int n, float *min, float *max)
2  {
3      int i = 2;
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     while( i < n-1 ){
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          i = i + 2;
33      }
34      if( i == n-1 ){
35          if( *max<t[i] ) *max=t[i];
36          if( *min>t[i] ) *min=t[i];
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:

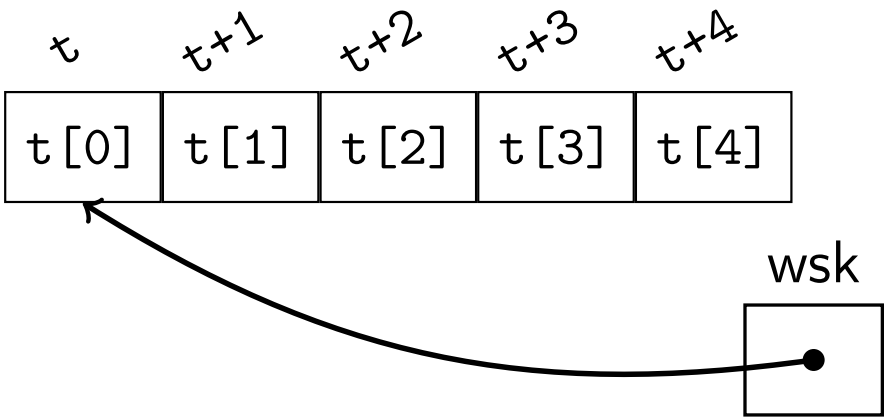
```
pa   = 0x7fff44cb239c 140734347551644
pa+1 = 0x7fff44cb23a0 140734347551648
pb   = 0x7fff44cb239b 140734347551643
pb+1 = 0x7fff44cb239c 140734347551644
```

Tablica jest wskaźnikiem !

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;
```

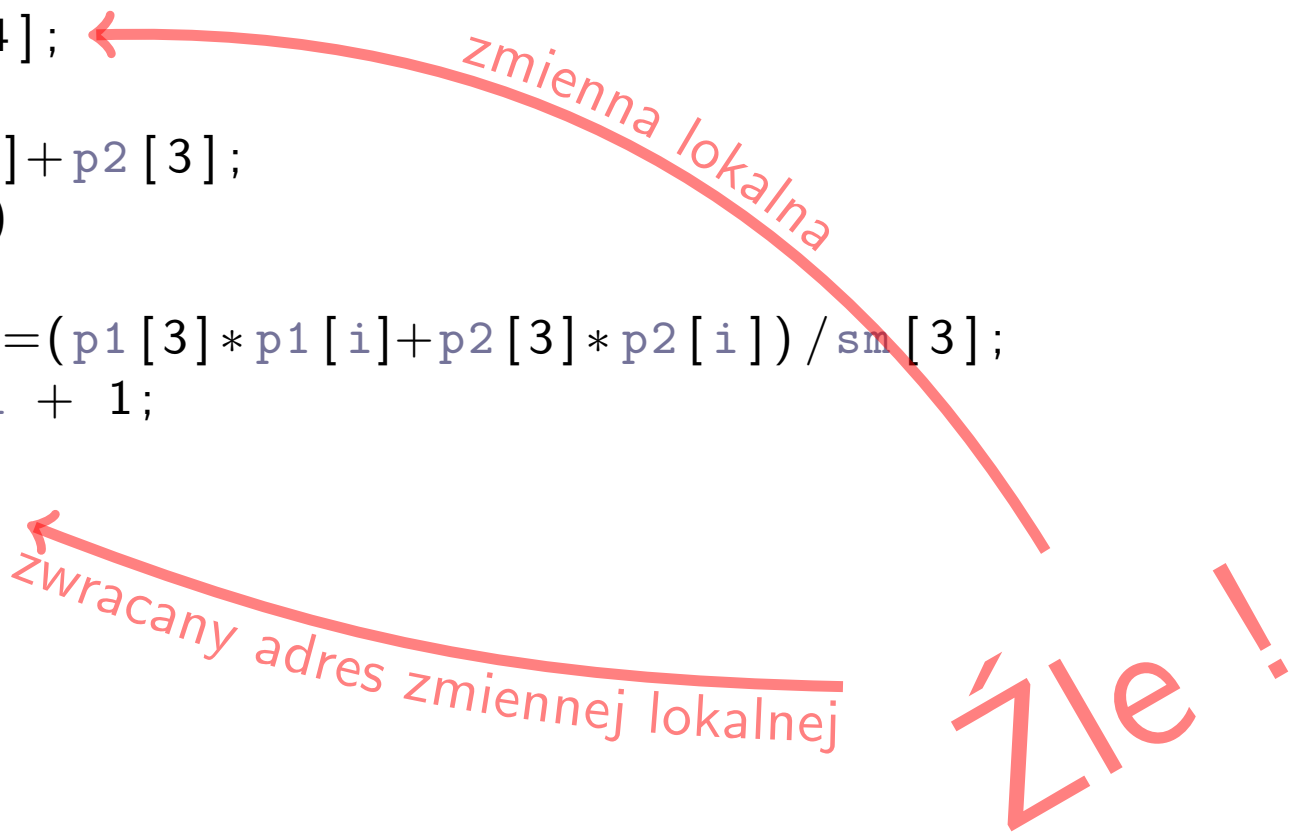


Adres tablicy jest staly

Źle !

Ponownie środek masy 2 punktów

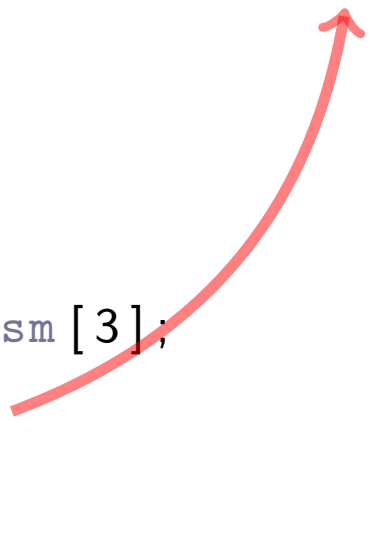
```
float* srodek(float p1[], float p2[])
{
    float sm[4];
    int i=0;
    sm[3]=p1[3]+p2[3];
    while(i<3)
    {
        sm[i]=(p1[3]*p1[i]+p2[3]*p2[i])/sm[3];
        i = i + 1;
    }
    return sm;
}
```



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

Algorytm 12 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 na pozycji $n - k$ w zbiorze

```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

- Operator referencji (adresowania) `&`
  `&a` to adres zmiennej `a`
- Operator dereferencji (wyłuskania) `*`
  `*b` daje dostęp do wartości wskazywanej przez zmienną `b`
- Deklaracja zmiennej wskaźnikowej również zawiera znak `*`

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

- Tablice to wskaźniki `t[i]` $\Leftrightarrow$ `*(t+i)`
- Przekazanie wskaźnika do funkcji pozwala na modyfikację zmiennej wskazywanej
- Uważaj na co wskazujesz !


Maciej M. Sysło, *„Algorytmy”*, WSiP, Warszawa, 2002.


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


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

Reprezentacja symboli w komputerze. Liczby całkowite i zmiennoprzecinkowe.

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

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

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

http://www.wikipedia.org

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

a - baza (podstawa) systemu

Dziesiętny

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

Dwójkowy (binarny)

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

Ósemkowy (oktalny)

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

Szesnastkowy (heksadecymalny)

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

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 odpowiadają 4 bity

1111	1101	0111	1110
F	D	7	E

Binarny na ósemkowy

1 cyfrze odpowiadają 3 bity

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

Liczby ósemkowe i szesnastkowe w C

```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 $x_{(10)}$ w dowolnym systemie pozycyjnym o podstawie $p \in \{2, \ldots, 10\}$.

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

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

139 % 16	11 B
8 % 16	8

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

Algorytm 13 Zapis liczby dziesiętnej w innej podstawie

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

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

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)

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

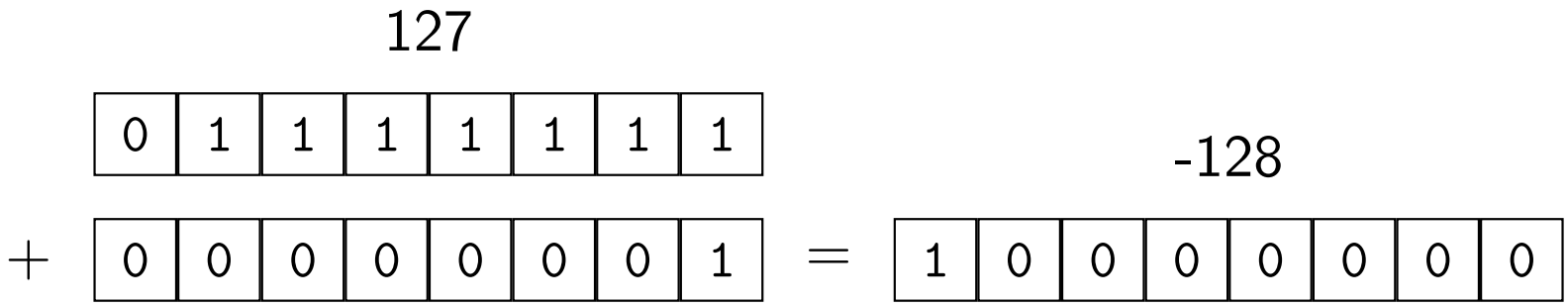
- liczby bez znaku
- zakres $[0, 2^n - 1]$
- `unsigned int`
- `unsigned char`
- operacje arytmetyczne przeprowadza się podobnie jak w systemie dziesiętnym
- John Napier, XVI w.

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 $\left[-2^{n-1}, 2^{n-1} - 1\right]$
- `int`, `char`
- operacje arytmetyczne przeprowadza się podobnie jak w NKB
- Pascal, uzupełnienie do 10, XV w.

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.
- Dodawanie dużych liczb dodatnich (lub odejmowanie liczb ujemnych). W U2 widoczna zmiana znaku.
- Próba zapisu liczby typu bardziej pojemnego do typu mniej pojemnego (np. `float` do `char`)
- Dodawanie liczby ujemnej i dodatniej nie powoduje nadmiaru
- 4 czerwiec 1996, pierwszy lot rakiety Ariane 5 zakończony zniszczeniem w skutek nadmiaru, 370 mln $

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

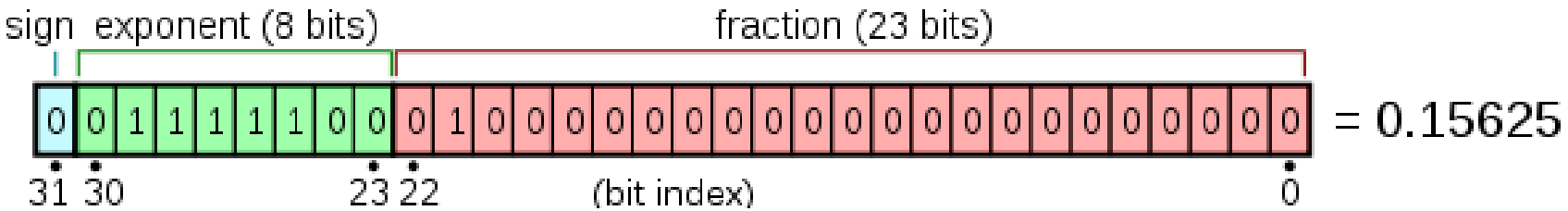
$$3,14 \cdot 10^0 = 0,0314 \cdot 10^2 = 314 \cdot 10^{-2}$$

Normalizacja

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

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

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



$s = 0$

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

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

Podobne rozwiązanie stosował Konrad Zuse, 1936 r.

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

Pojedyncza precyzja

- zakres $\pm 3.4 \cdot 10^{38}$
- precyzja 7 cyfr dziesiętnych

Podwójna precyzja (*double precision*)

- `double`
  cecha 11 bitów, mantysa 52 bity, bias 1024
- zakres $\pm 1.8 \cdot 10^{308}$
- precyzja 15 cyfr znaczących

Znaczenie specjalne bitów IEEE-754

cecha	mantysa	znaczenie
$\neq 0\ldots0$	dowolne	znormalizowana
$\neq 1\ldots1$	dowolne	znormalizowana
$0\ldots0$	$\neq 0\ldots0$	zdenormalizowana $(-1)^s \cdot 0.m \cdot 2^{-127}$
$0\ldots0$	$0\ldots0$	± 0
$1\ldots1$	$\neq 1\ldots1$	nieliczby `NaN`
$1\ldots1$	$0\ldots0$	nieskończoność `Inf`, `-Inf`

Wyniki operacji z wartościami specjalnymi

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 $|a - b| < \epsilon$

```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

- **Epsilon maszynowy** określa precyzję obliczeń numerycznych na liczbach zmiennopozycyjnych
- Najmniejsza liczba nieujemna, której dodanie do jedności daje wynik nierówny 1.
  Najmniejsze $\epsilon > 0$ takie, że $1 + \epsilon \neq 1$
- Im mniejsza wartość epsilona maszynowego, tym większa jest względna precyzja obliczeń.
- Nie mylić z najmniejszą liczbą większą od zera

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

int main()
{
    float epsilon = 1.0;
    float p_epsilon = 1.0;
    float x = 1.0;

    printf( "epsilon          1.0 + epsilon\n" );

    while (x + epsilon != 1.0)
    {
        printf( "%e\t%.20f\n", epsilon, (x + epsilon) );
        p_epsilon = epsilon;
        epsilon = epsilon / 2;
    }
    printf( "\nObliczony epsilon: %e\n", p_epsilon );
    printf( "FLT_EPSILON      : %e\n", FLT_EPSILON );
    return 0;
}
```

☞ epsilon.c

Limity dla obliczeń zmiennopozycyjnych w C

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

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

   return 0;
}
```

☞ float.c

```
Najwieksza wartosc   : 3.402823e+38
Najmniejsza dodatnia : 1.175494e-38
Epsilon maszynowy    : 1.192093e-07
```

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

Problem: wyznaczyć sumę pierwszych n elementów szeregu harmonicznego

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

```c
#include <stdio.h>

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

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

    while(i <= n)
    {
        x = x + 1.0/i;
        y = y + 1.0/(n-i+1);
        i = i + 1;
    }

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

```
n=150
x=5.591181
y=5.591181

n=600
x=6.974980
y=6.974978

n=2000
x=8.178369
y=8.178370

n=500000
x=13.690692
y=13.699607
```

☞ szereg.c

Problem: wyznacz miejsca zerowe wielomianu stopnia 2

$$y(x) = ax^2 + bx + c, \qquad a \neq 0$$

$$\Delta = b^2 - 4ac$$

Dla $\Delta > 0$ istnieją 2 miejsca zerowe

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \qquad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

Dla $\Delta = 0$ istnieje 1 miejsca zerowe

$$x_1 = -\frac{b}{2a}$$

Dla $\Delta < 0$ brak rzeczywistych miejsc zerowych

```c
int miejsca_zerowe(float a, float b, float c, float *x1, float *x2)
{
    float delta;

    delta = b*b - 4*a*c;

    if( fabs(delta) < EPS )
    {
        *x1=-b/a/2;
        *x2=*x1;
        return 1;
    }

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

☞ miejscazerowe.c

- możliwe błędy, gdy jeden z pierwiastków bliski 0, tzn $b \approx \sqrt{\Delta}$

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \qquad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

- np.: $a = 1, b = -100, c = 1 \implies \sqrt{\Delta} \approx 99.979997999$
- `x1=99.99000e+01, x2=1.000214e-02`
- `w(x1)=0.000000e+00, w(x2)=-1.136065e-04`

- możliwe błędy, gdy jeden z pierwiastków bliski 0, tzn $b \approx \sqrt{\Delta}$

$$x_1 = -\frac{b + \sqrt{\Delta}}{2a} \qquad x_2 = -\frac{b - \sqrt{\Delta}}{2a}$$

- Jeśli problem jest trudny - najlepiej zmienić problem
- wzory Viete'a

$$\begin{cases} x_1 + x_2 = -\frac{b}{a} \\ x_1 x_2 = \frac{c}{a} \end{cases}$$

- `x1=9.999000e+01, x2=1.000100e-02`
- `w(x1)=0.000000e+00, w(x2)=0.000000e+00`

```c
int miejsca_zerowe2(float a, float b, float c, float *x1, float *x2)
{
    float delta;

    delta = b*b - 4*a*c;

    if( fabs(delta) < EPS )
    {
        *x1=-b/a/2;
        *x2=*x1;
        return 1;
    }

    if( delta >= EPS )
    {
        delta=sqrt(delta);
        if(b < 0) *x1 =  (delta-b)/(2*a);
        else      *x1 = -(b+delta)/(2*a);
        *x2 = (c/a)/ *x1;
        return 2;
    }
    return 0;
}
```

☞ miejscazerowe.c

- nadmiar - przekroczenie zakresu typu
- niedomiar - zaokrąglenie do 0 w arytmetyce zmiennopozycyjnej
- `3/2` wynosi `1`, zaś `3/2.0` wynosi `1.5`
- przybliżenia arytmetyki zmiennopozycyjnej
- 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 alfabetu 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, ...

USASCII code chart

b_7 b_6 b_5 → / Bits	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 / Row ↓ Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

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

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

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

Manipulacje na znakach

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

Przykłady w C: zamiana wielkości znaków

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

int main()
{
    int a;

    do
    {
        a=getchar();
        putchar(toupper(a));
    }while( a != EOF );

    return 0;
}
```

☞ toupper2.c

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

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

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

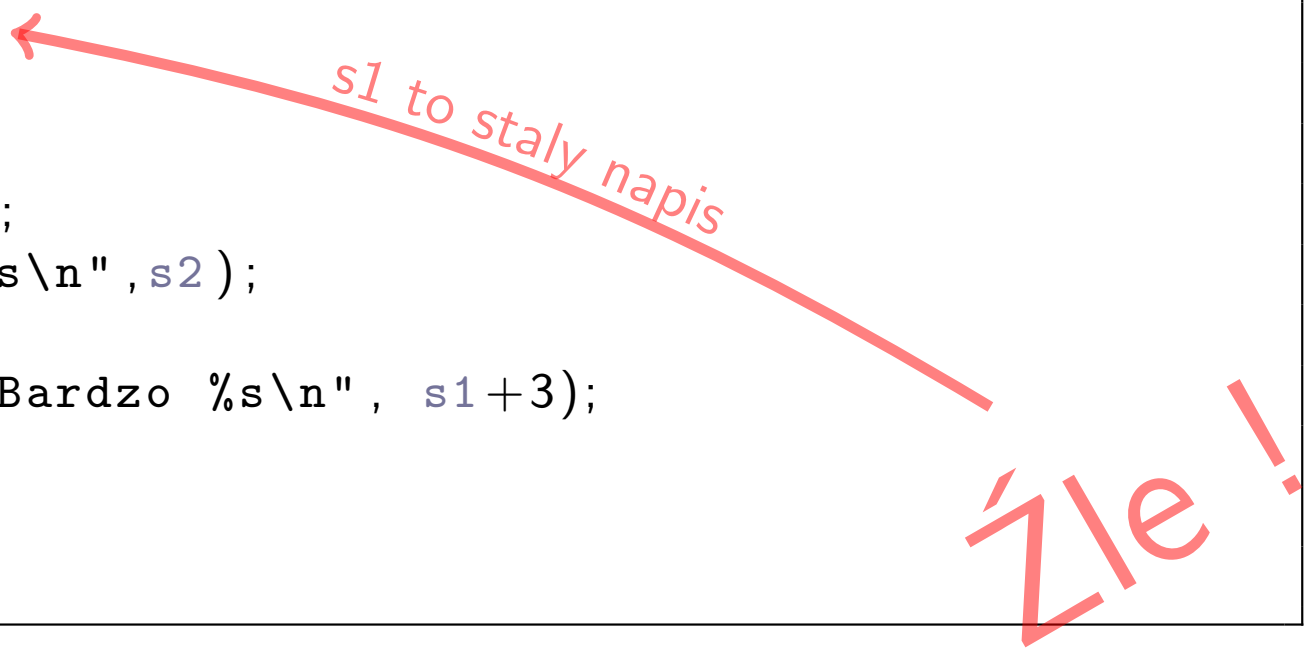
- **Łańcuch** (*string*) - skończony ciąg symboli alfabetu
- W języku C brak typu `string`, występuje w C++
- Łańcuch to tablica zawierająca ciąg znaków zakończony znakiem `\0`

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

```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);`
- wczyta tylko pierwszy wyraz, nie wczyta całego łańcucha z odstępami: `"Ala ma kota"`

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

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

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

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

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

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

Podstawowe operacje na łańcuchach

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

Problem: wyszukiwanie podciągu (*pattern matching*).
W ciągu $\mathcal{T}$ znajdź wystąpienie wzorca $\mathcal{W}$.

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

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

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

g	r	a	\0

Algorytm 14 Naiwne wyszukiwanie wzorca

Dane wejściowe: łańcuch znaków $\mathcal{T}$, wzorzec $\mathcal{W}$
Wynik: pozycja tekstu $\mathcal{W}$ w $\mathcal{T}$ lub wartość -1, gdy brak

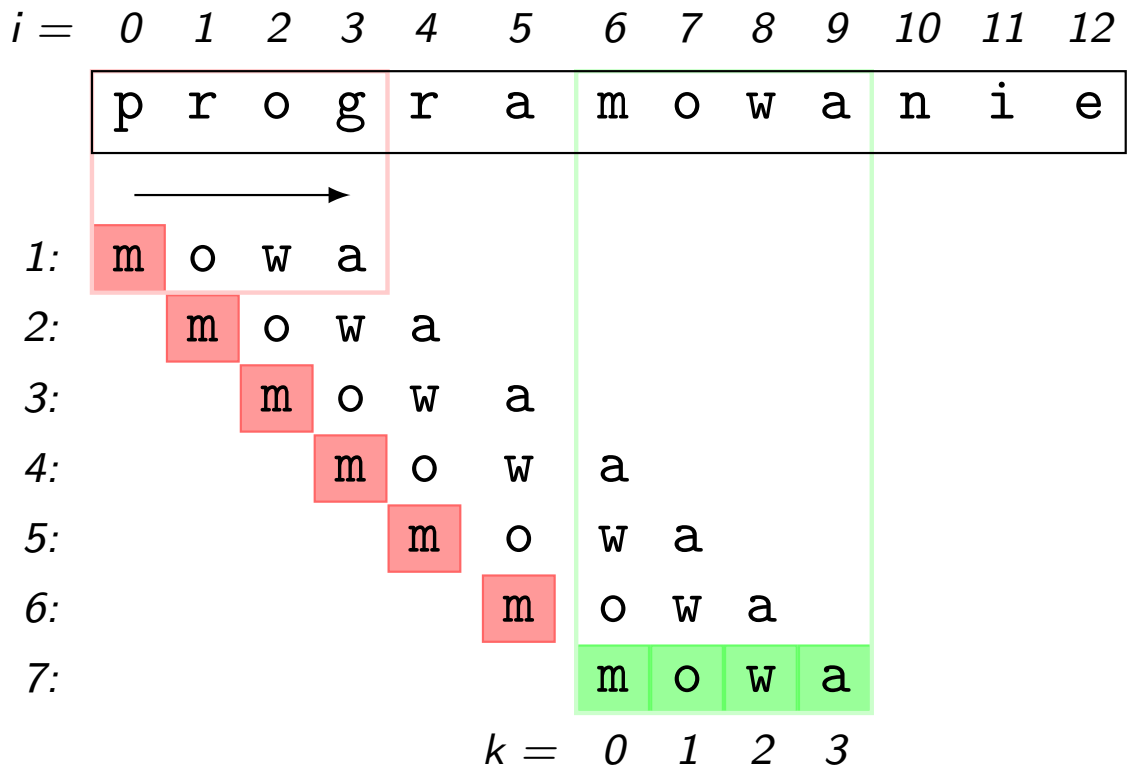
```
1: dla każdego i = 0, 1, 2, ..., |T| − |W| wykonuj
2:     k ← 0
3:     dopóki T[i + k] = W[k] i k < |W| wykonuj
4:         k ← k + 1
5:     jeżeli k = |W| wykonaj
6:         zwróć i
7: zwróć −1
```

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

```c
int strindex(char t[], 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;
}
```

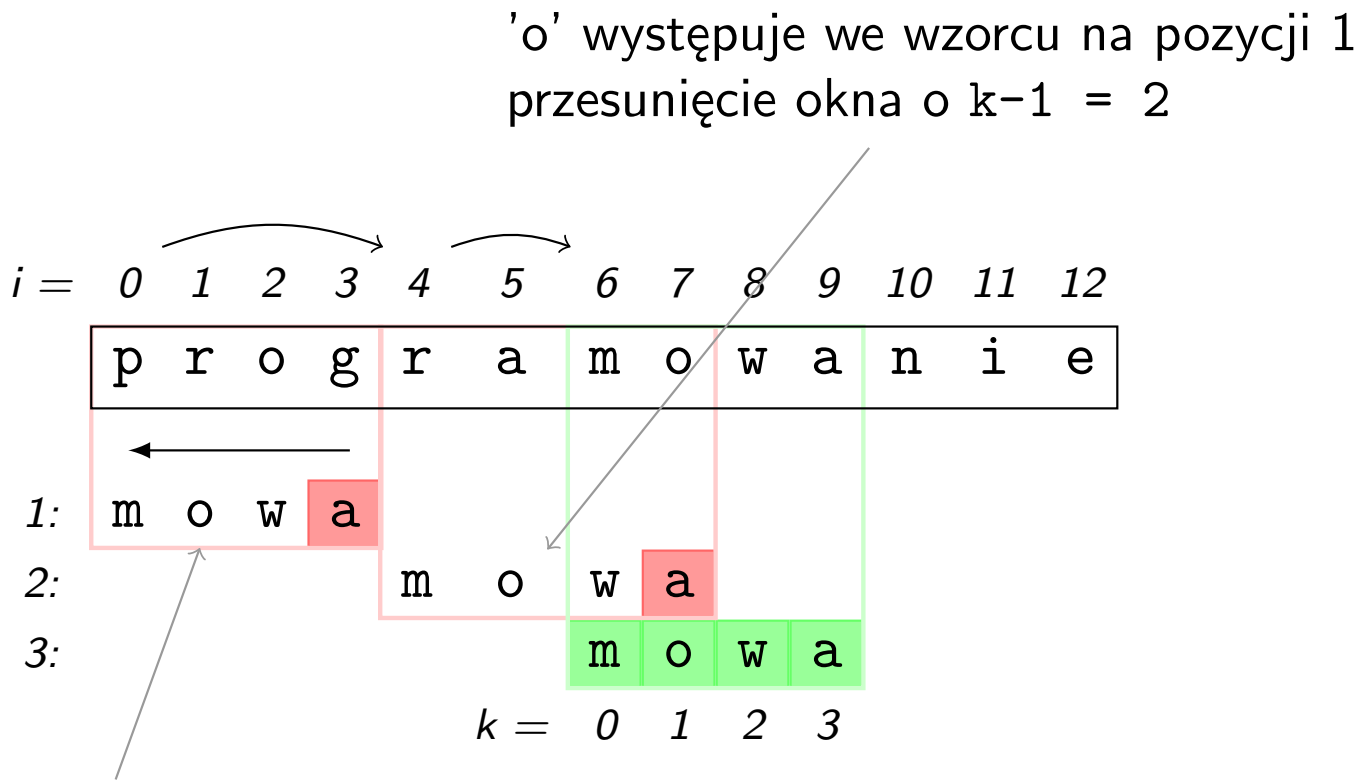
☞ strindex.c

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



Uproszczony algorytm Boyera-Moore'a

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



Tablica przesunięć

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

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

Algorytm 15 Uproszczony algorytm Boyera-Moore'a

Dane wejściowe: łańcuch znaków $\mathcal{T}$, wzorzec $\mathcal{W}$
Wynik: pozycja tekstu $\mathcal{W}$ w $\mathcal{T}$ lub wartość -1, gdy brak

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

gdzie $kod(x)$ oznacza kod liczbowy znaku x.

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

- Typ znakowy jest liczbą całkowitą
- Łańcuch to tablica znaków zakończona znakiem `'\0'`
- Stała znakowa w apostrofach `'A'`
- Stała napisowa w cudzysłowach `"A"` jest tablicą
- Dostęp do znaku `t[i]`
- `t == "napis"` tak nie wolno porównywać, trzeba znak po znaku

- 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`

Strumienie i pliki.

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

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

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

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

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

System plików: sposób przechowywania plików na nośniku.

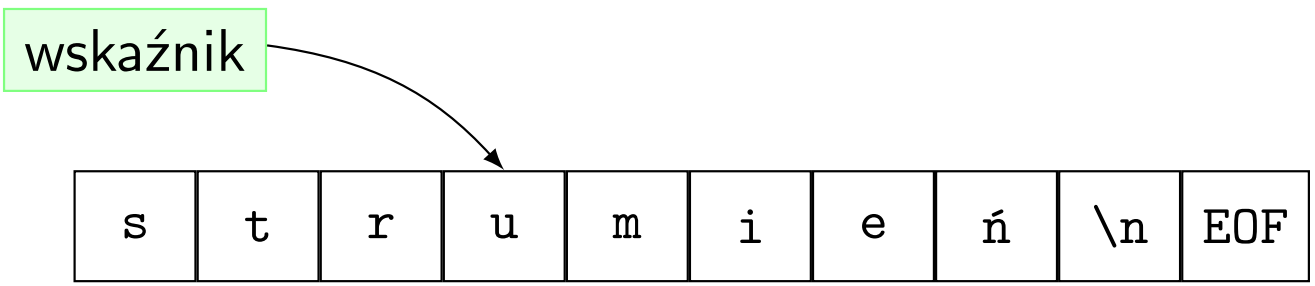
- fizyczny zapis danych, blokowa struktura danych (sektory, klastry)
- logiczna struktura widoczna dla użytkownika
- DOS/Windows: FAT, FAT32, NTFS
- Linux: ext2, ext3, ext4
- inne: HFS (Mac OS), NFS (sieć), ISO9660 (CD-ROM)
- Hierarchia systemów plików: katalogi, podkatalogi i pliki.
  Dostęp do pliku: ścieżka + nazwa pliku
  `C:\Documents and Settings\user\moje dokumenty\plik.txt`
  `/home/user/doc/plik.txt`

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

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

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

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



1. Deklaracja zmiennej typu `FILE*`

```
FILE* plik;
```

2. Uzyskanie dostępu do strumienia `fopen()`

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

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

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

4. Zamknięcie strumienia `fclose()`

```
fclose(plik);
```

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

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

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

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

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: 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 `EOF`

Inne błędy operacji wejścia-wyjścia

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

Wartość niezerowa gdy 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 | wc
```

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

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

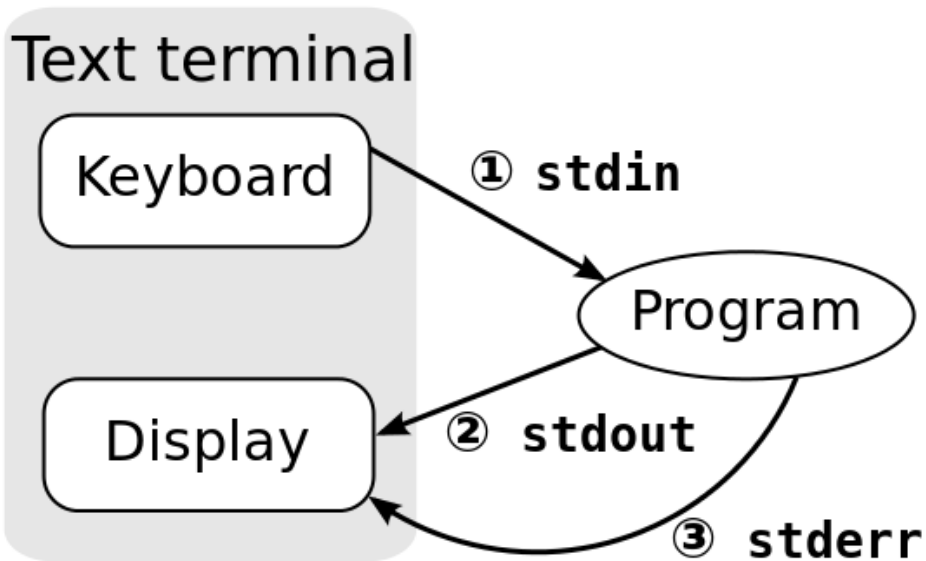
0 `stdin`
standardowe wejście, domyślnie klawiatura
`getchar(), scanf(), ...`

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

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

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

```c
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

Podsumowanie.

Elementy inżynierii oprogramowania.

- Zadanie algorytmiczne, specyfikacja
- Podstawy analizy algorytmów: poprawność, złożoność
- Podstawy języka C:
  - typy (`int`, `float`, `char`),
  - instrukcje sterujące (`while`, `do while`, `if`, `else`),
  - operatory arytmetyczne, logiczne, przypisania (`+ = - * / % && || < > != `)
  - funkcje, zakres zmiennych (lokalne, globalne)
  - tablice jednowymiarowe, łańcuchy, struktury
  - wskaźniki,
    strumienie
- Paradygmaty: programowanie proceduralne i strukturalne

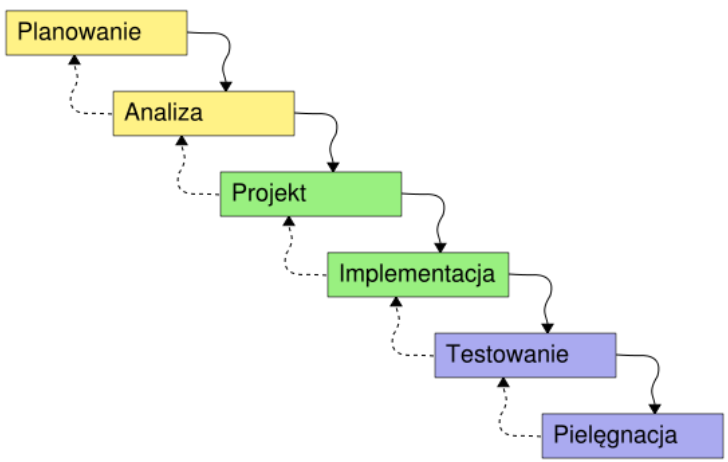
Czego nie było?

- Dynamiczny przydział pamięci
- Pliki nagłówkowe i źródłowe, biblioteki statyczne, dynamiczne
- Typy danych, m.in.: unie, pola bitowe, `typedef`
- Tablice wielowymiarowe (tablice tablic)
- Inne złożone struktury danych: listy, kolejki, stosy, drzewa, ...
- Rekurencja, wskaźniki do funkcji, funkcja argumentem funkcji
- Obsługa błędów systemowych
- Operatory bitowe, rzutowania i inne
  `! ++ -- =+ & | ^ >> << -> =/ ?: sizeof()`
- Operacje na wskaźnikach
- Instrukcje preprocesora i makra
- Pętla `for`,
- Masa przydatnych funkcji z biblioteki standardowej
- itd...

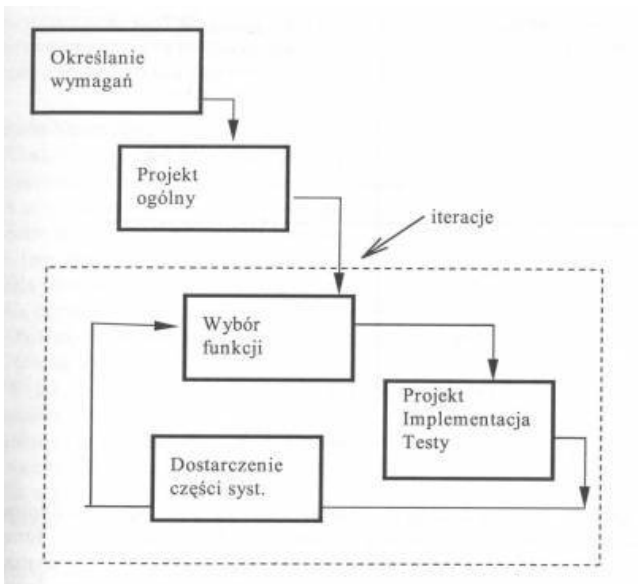
Inżynieria oprogramowania

zajmuje się wszelkimi aspektami produkcji oprogramowania we wszystkich fazach **cyklu życia oprogramowania**

Kaskadowy



Iteracyjny



Źródło: http://pl.wikipedia.org/wiki/Model_kaskadowy, Ilona Bluemke, Inżynieria oprogramowania

- **Analiza wymagań:** opis funkcjonalności i sposób ich realizacji
- **Modelowanie systemu:** organizacja systemu w kategoriach informatycznych. Modele danych, model architektury systemu, model przepływ danych, modele zjawisk.
- **Etap projektowy:** wybór narzędzi, środków technicznych, rozwiązań algorytmicznych.
- **Programowanie:** implementacja, konserwacja kodu, testowanie
- Błędy możliwe na każdym etapie, kumulacja błędów

Hierarchiczne podejście do rozwiązania problemu

- metodyka „zstępująca” (*top-down*), dedukcyjna. Dekompozycja problemu, od ogółu do szczegółu, podział większego programu na podprogramy
- metodyka „wstępująca” (*bottom-up*), syntetyczna. Modularne podejście do tworzenia programu. Składanie rozwiązania z istniejących małych narzędzi

- Właściwa definicja problemu - zasadnicza część programowania
- Dobór właściwych struktur danych może znacznie uprościć problem
- „Dobry programista jest trochę leniwy” - najpierw pomyśl potem programuj

- Język C bardzo elastyczny, nieostrożne używanie oznacza kłopoty
- Czytelność przede wszystkim
- Spójność stylistyczna - w całym kodzie jednakowo
- Zrozumiałe i jednolite nazwy zmiennych i funkcji.
  Nazwa nie powinna wprowadzać w błąd.
  `wypiszwiersz()`, `WypiszWiersz()`, `wypisz_wiersz()`
- Małe i duże litery są rozróżnialne
- Nazwy zmiennych zazwyczaj z małej litery
  np. indeksy `i, j, k`
- Stałe symboliczne i makra dużymi literami
  `#define MAX 500`

- Opis funkcji

```
/* Zwraca pozycje wzorca 'w' w lancuchu 't' */
int strindex(char *t, char *w) { ... }
```

- Nagłówek pliku: autor, data, licencja, wersja
- Nie komentuj oczywistych rzeczy, raczej opis sensu operacji

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

- W C lepiej nie używać komentarzy //
- *Jeśli dokumentacja nie powstaje równolegle z programem – nie powstanie nigdy!*
- Dokumentacja techniczna, generatory np. Doxygen

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

☞ nwd–balagan.c

- Czytelność przede wszystkim
- ☞ The International Obfuscated C Code Contest

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

Styl K&R (GNU)

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

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

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

- Python - wcięcia elementem składni języka

- Nie więcej niż jedna instrukcja w linii
- Nie deklaruj więcej niż jedną zmienną w linii
- Otwierając nawias od razu go zamknij `{([])}`
- Nawiasy `()` przyklej do nazwy funkcji ale oddziel od instrukcji `if` i `while`

```
void czy_parzysta(int i)
{
    if ( i%2 == 0 ) printf("Parzysta!\n");
}
```

- Zawsze inicjuj zmienne
- Nie używaj magicznych liczb

```
int main()
{
    /* ... */
    if ( x == 42 ) wystrzel_rakiete();
}
```

- Lepiej zdefiniuj stałą

```
#define KONIEC_ODLICZANIA 42
int main()
{
    /* ... */
    if ( x == KONIEC_ODLICZANIA ) wystrzel_rakiete();
}
```

Programowanie proceduralne i strukturalne

- **Programowanie proceduralne.**
  Unikaj zmiennych globalnych.
  Przekazuj dane do funkcji przez argumenty.
- Unikaj długich funkcji.
  Dobrze gdy w całości mieści się na ekranie.
- Funkcje powinny realizować jedną operację, jednak w sposób pełny i skończony
- Unikaj powtórzeń kodu - pisz funkcje.
  Don't repeat yourself (DRY), *once and only once* (OAOO)
- **Strukturalność w programowaniu.**
  Nie używaj instrukcji skoku

- *Less is more*
- *„Konstruktor wie, iż osiągnął doskonałość, nie wtedy, gdy już nic nie można dodać, lecz wtedy, gdy już nic nie da się ująć”.* Antoine de Saint-Exupéry
- Jednak nie upraszczaj na siłę

```
void strcpy(char s[],char t[])
{
    int i=0;
    while (t[i] != '\0')
    {
        s[i] = t[i];
        i = i + 1;
    }
    s[i] = '\0';
}
```

```
void strcpy(char *s, char *t)
{
    while (*s++=*t++);
}
```

Jeżeli coś może się nie udać, to się nie uda.

Prawo Murphy'ego

Sytuacje wyjątkowe w funkcjach

- Zanim użyjesz nieznanej funkcji przeczytaj dokumentację!
- Wartość zwracana: `NULL`, `EOF`, inna specjalna wartość

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

- Plik nagłówkowy `errno.h` i zmienna `errno` kodująca błąd
- ☞ Kody błędów różne na różnych platformach
- Funkcja `perror()` wypisuje komunikat błędu

```
x = sqrt(-1);
if( errno == EDOM ) perror("Blad ");
```

☞ errno.c

Zabezpieczanie sensowności danych wejściowych

- niepoprawne dane, zasada GIGO *Garbage In, Garbage Out*
- *idiotoodporność*, przewiduj nawet najgłupsze dane wejściowe
- przepełnienie bufora

```
scanf("%s", tablica);
gets(tablica);
```

Błędy w aplikacjach konsolowych

- Standardowy strumień błędów `stderr`

```
fprintf(stderr, "Don\'t panic!!!\n");
```

- Status zakończenia programu. 0 to sukces.

```
void exit(int status);                    /* stdlib.h */
```

- Poprawność algorytmów: niezmienniki, zbieżność, skończoność
- Testy poprawności implementacji: asercje
  `assert( wyrazenie )` `/* assert.h */`
  Przerywa program gdy predykat jest fałszywy.

```
1  #include <assert.h>
2  int main()
3  {
4      int i=0, t[10];
5
6      while( i <= 10 )
7      {
8          assert( i < 10 );
9          t[i] = i;
10         i = i + 1;
11     }
12     return 0;
13 }
```

☞ assert.c

- **Debugowanie** - kontrolowane wykonywanie kodu
- **Profilowanie** - pomiary czasu i zajętej pamięci
- Testy funkcjonalności: testy jednostkowe, testowanie pojedynczych funkcji
- Testy aplikacji - programy testujące programy
- Automatyzacja testów
- To, że nie znaleziono błędu, nie znaczy, że go tam nie ma

Najtańszymi, najszybszymi i najsolidniejszymi elementami systemu komputerowego są te, których w nim nie ma”

Gordon Bell, DEC

Literatura dodatkowa


Ilona Bluemke, *Inżynieria oprogramowania*


Dawid Czerner, Wykład: ☞ Inżynieria oprogramowania


Wikipedia: ☞ Indent style, ☞ Asercja


Code Smell, `http://c2.com/xp/CodeSmell.html`


C Coding Standards,
`http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html`


Recommended C Style and Coding Standards, Bell Labs
`http://www.doc.ic.ac.uk/lab/cplus/cstyle.html`


J. Bentley, *Perełki oprogramowania*, WNT, Warszawa, 2001.


☞ errno.h - system error numbers


Linux Programmer's Manual: `man errno assert`


Errno Codes by Platform,
`http://www.ioplex.com/~miallen/errcmp.html`