Programowanie strukturalne w językach Pascal i C

Krzysztof Grąbczewski

Tematy zajęć

- Wprowadzenie do programowania
- Język Pascal
- Język C
- Podstawowe algorytmy
- Dynamiczne struktury danych
- Programowanie dla Windows

Plan zajęć

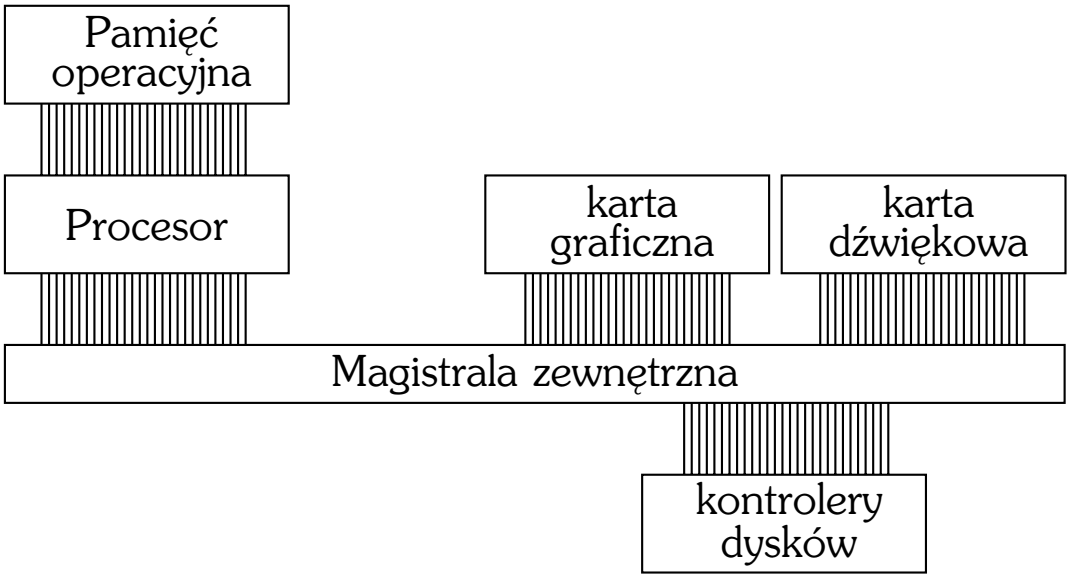
- Spotkanie 1: Podstawy, schematy blokowe
- Spotkania 2, 3, 4: Język Pascal
- Spotkania 5, 6, 7: Język C
- Spotkanie 8: Struktury dynamiczne
- Spotkanie 9: Programowanie dla Windows
- Spotkanie 10: Różne uwagi i drobne problemy

Bibliografia

- Niklaus Wirth, *Algorytmy + struktury danych = programy*, WNT 1989
- Andrzej Marciniak, *Turbo Pascal x.x*
- B. Kernighan, D. Ritchie, *Język C*, WNT 1988

Komputer i jego wykorzystanie

- Sprzęt to ciało - oprogramowanie to dusza
- Komputer - jak pracują krasnoludki



- Rodzaje oprogramowania - poziomy od niskiego do wysokiego
  - systemy operacyjne i podstawowe biblioteki obsługi urządzeń
  - oprogramowanie użytkowe i biblioteki użytkowe
  - wysoko wyspecjalizowane biblioteki - programy wysokiego poziomu
- Obsługa zasobów dostępnych w komputerze: pamięć operacyjna, dyski, porty komunikacyjne, wyświetlacze, klawiatury itp.
- Mechanizm przerwań - zgłaszanie przerwań przez urządzenia
- DMA (direct memory access) - bezpośredni dostęp do pamięci

Systemy liczenia: dwójkowy (binarny), ósemkowy, dziesiętny, szesnastkowy

system	cyfry	przykład	dziesiętnie
dwójkowy	0 1	101	$1 * 2^2 + 1 * 2^0 = 1 * 4 + 1 = 5$
ósemkowy	0 1 2 3 4 5 6 7	270	$2 * 8^2 + 7 * 8^1 = 2 * 64 + 7 * 8 = 184$
dziesiętny	0 1 2 3 4 5 6 7 8 9	309	$3 * 10^2 + 9 * 10^0 = 3 * 100 + 9 = 309$
szesnastkowy	0 1 2 3 4 5 6 7 8 9 A B C D E F	10A	$1 * 16^2 + 10 * 16^0 = 1 * 256 + 10 = 266$

Komputer „myśli" dwójkowo - organizacja pamięci

- 1 **bit** - miejsce na zapamiętanie znaku 0 bądź 1
- 1 **bajt** = 8 bitów - pozwala zapamiętać 2^8 różnych wartości, a więc np. liczby 0-255
- 1 **kB** (kilobajt) = 2^{10} B = 1024 B
- 1 **MB** (megabajt) = 2^{20} B = 1048576 B
- 1 **GB** (gigabajt) = 2^{30} B = 1073741824 B
- 1 **słowo** to ilość informacji przetwarzanej przez procesor w jednym cyklu (procesor 32 bitowy = procesor o słowie 32 bitowym)
- bity i bajty bardziej i mniej znaczące

- operacje bitowe
  - przesunięcia:

```
00000001 << 3 = 00001000 = 8
00001101 >> 2 = 00000011 = 3
00001101 << 5 = 10100000 = 160
```

  - operacje logiczne: koniunkcja `&`, alternatywa `|`, dysjunkcja `^`, dopełnienie `~`

```
01010101 & 11001100 = 01000100
01010101 | 11001100 = 11011101
00001101 ^ 11001100 = 11000001
~ 01010101 = 10101010
~ 11001100 = 00110011
```

- typowa liczba całkowita zajmuje jedno słowo
- typy zmiennoprzecinkowe (zmiennopozycyjne) - rzeczywiste
  - rozdzielczość i skończoność komputerowych liczb rzeczywistych
  - mantysa i cecha, liczba $=$ mantysa $* 2^{\text{cecha}}$
  - typy 4-bajtowe (3 bajty mantysy i 1 bajt cechy), 6-bajtowe, 8-bajtowe (podwójnej precyzji)
  - należy być ostrożnym przy dodawaniu małej liczby zmiennoprzecinkowej do dużej (mała może zniknąć)
  - wartości, które nie są poprawnymi liczbami (NAN - not a number)

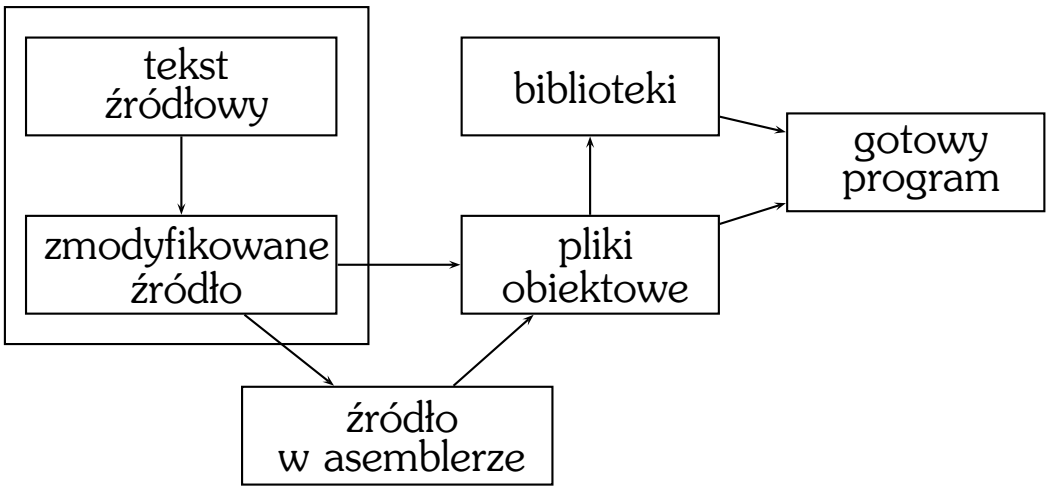
Języki programowania

- Imperatywne
  - liniowe (np. BASIC)
  - strukturalne (np. Pascal, C)
  - obiektowe (np. C++, SmallTalk, Java)
- Deklaratywne
  - logiczne (np. Prolog)
  - funkcyjne (np. ML, Lisp)
- Języki niskiego poziomu (asemblery), wysokiego poziomu (Pascal, C, C++, . . .), języki czwartej generacji (wizualne)
- Jak wybierać język programowania - analiza problemu

Umiejętność programowania

- Znajomość syntaktyki to nie wszystko
- Formatowanie tekstów programów
- Najpierw pomyśl (przynajmniej dwa razy), potem pisz
- Schematy blokowe
- Metody programowania
  - top-down (z góry na dół)
  - bottom-up (z dołu do góry)
- Uwaga na zależności od platformy sprzętowej i systemowej
  - programowanie dla UNIXa
  - programowanie dla DOSa (16 i 32 bitowe, modele pamięci)
  - programowane dla Windows (16 i 32 bitowe)

Cykl życia programu



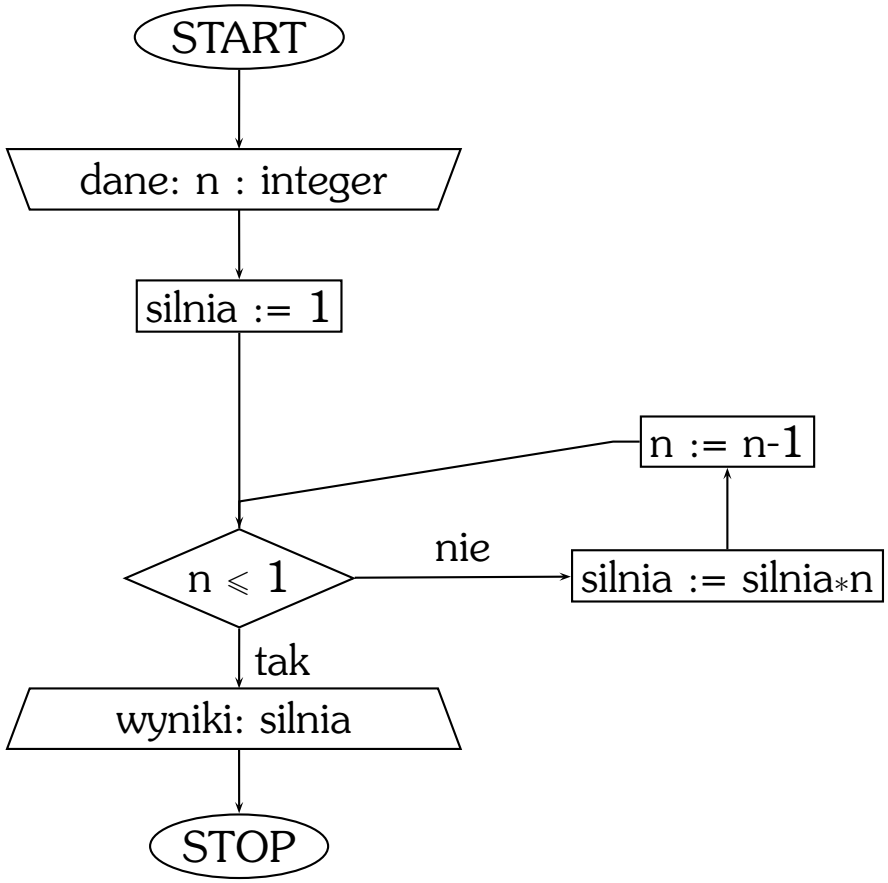
Biblioteki

- zbiory typów danych i funkcji dla wykonywania pewnej grupy zadań
- korzystający z biblioteki nie musi wiedzieć w jaki sposób ona działa
- proces tworzenia aplikacji staje się prostszy i przez swoją modularność bardziej czytelny i efektywny
- **Uwaga:** Nie przesadzać z uogólnieniami

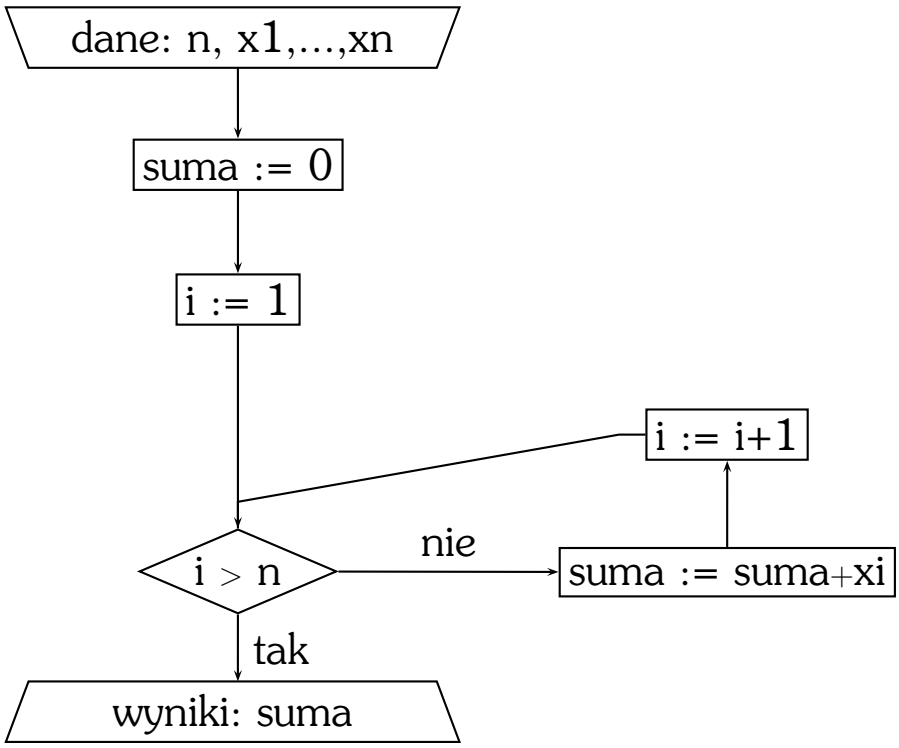
Pascal i C

- historia Pascala
  - 1971 - Niklaus Wirth - pierwszy raport języka
  - 1974 - N.Wirth & Kathleen Jensen - pierwszy podręcznik
  - Pascal językiem dydaktycznym
  - 1983 - Turbo Pascal firmy Borland Inc.
- historia C
  - 1972 - Dennis Ritchie (Bell Laboratories, New Jersey)
  - rozwój języka - Brian Kernighan & Dennis Ritchie
  - C następcą B (Ken Thompson), B następcą BCPL (1969 - Martin Richards)
- podobieństwa i różnice na podstawowym poziomie
  - Języki wysokiego poziomu (C również niskiego)
  - Czytelność Pascala i zwięzłość C
  - C daje łatwy dostęp do wszystkich wnętrzności komputera
  - modularność (moduły a biblioteki)
  - C „zrośnięte” z unixem
  - Pascal utożsamia wielkie i małe literki, a C je rozróżnia

Schemat blokowy - silnia



Schemat blokowy - suma n zadanych liczb



Zadania:

- suma n pierwszych liczb naturalnych
- n pierwszych wartości ciągu Fibonacciego

$$x_n = \begin{cases} 1 & \text{o ile } n < 3 \\ x_{n-1} + x_{n-2} & \text{w przeciwnym wypadku} \end{cases}$$

- największy wspólny dzielnik dwóch liczb naturalnych

Początki Pascala

- formatowanie tekstów programu
- komentarze `{ to jest komentarz w Pascalu }`
- najprostszy program (pusty)

```
begin
end.
```

- struktura programu

```
program etykieta;  { nazwa programu (opcjonalna) }
uses ...           { lista używanych modułów }
const ...          { deklaracje stałych }
type ...           { deklaracje typów }
var ...            { deklaracje zmiennych }
begin
  ...              { blok główny programu }
end.
```

- **identyfikator** - każdy ciąg znaków złożony z symboli będących literą, cyfrą lub znakiem podkreślenia rozpoczynający się literą bądź podkreśleniem
- Jak obsługiwać środowiska programistyczne Borlanda (IDE - Integrated Development Environment)
  - wczytywanie plików (F3) i zapisywanie (F2)
  - kompilacja (Alt-F9) i szukanie błędów (Alt-F7, Alt-F8)
  - odpluskwianie programów
  - korzystanie z pomocy kontekstowej (F1 i kombinacje)

Typy danych

- proste
  - porządkowe
    - wyliczeniowy

      `type identyfikator = (lista_identyfikatorów);`

      Przykład:

      ```
      type dni_tygodnia = (poniedzialek, wtorek,
        sroda, czwartek, piatek, sobota, niedziela);
      ```

    - całkowite

Nazwa	Zakres
ShortInt	[-128, 127]
Byte	[0, 255]
Integer	[-32768, 32767]
Word	[0, 65535]
LongInt	[-2147483648, 2147483647]

    - logiczny

      `Boolean = (False, True)`

    - znakowy `char` - 1 bajt
    - okrojony

      ```
      type dwucyfrowe = 10 .. 99;
      type litera = 'A' .. 'Z';
      type dni_robocze = poniedzialek .. piatek;
      ```

  - rzeczywiste

Nazwa	Rozmiar
Real	6 bajtów
Single	4 bajty
Double	8 bajtów
Extended	10 bajtów

- łańcuchowy

```
type nazwisko = string[20];
```

- strukturalne
  - tablicowy

```
type wektor = array [0..50] of Integer;
type macierz1 = array [1..20] of
                    array[1..20] of Real;
type macierz2 = array [1..20,1..20] of Real;
```

  - rekordowy

```
type data = record
              rok     : Integer;
              miesiac : 1 .. 12;
              dzien   : 1 .. 31;
            end;
     personalia = record
              nazwisko       : string[20];
              imie1, imie2   : string[15];
              data_urodzenia : data;
            end;
```

  - zbiorowy

```
type dni_pracy = set of dni_robocze;
```

  - plikowy

- definicje własne

```
type liczba = Integer;
```

- zgodność typów (konwersje)

Zmienne i stałe

- deklaracje zmiennych

```
var x : Integer;
    y : Real;
    b1, b2 : Boolean;
```

- deklaracje stałych

```
const pi   : Real = 3.14;
      znak : Char = 'a';
      tab  : array [1..2,1..3] of Integer
                   = ((1,2,3),(4,5,6));
```

- zmienne indeksowane (typu tablicowego)

```
type wektor = array [1..20] of Real;
     macierz = array [1..20] of wektor;
var a : wektor;
    b : macierz;
    c : array [1..20] of wektor;
    d : array [1..20] of macierz;
begin
  a[2] := 13.5;
  b[3][4] := 12.3;
  b[2,3] := 11.5;
  c[3,13] := c[3][12];
  d[1,12,4] := d[1][12,4];
  d[3,4][1] := 1;
  d[2][3][1] := 2;
end.
```

- zmienne rekordowe

```
type data = record
              rok : Integer;
              miesiac : 1..12;
              dzien : 1..31
            end;
var data_ur : data;
    osoba : record
              nazwisko : string[20];
              imie1, imie2 : string[15];
              data_ur : data
            end;
begin
  data_ur.rok := 1998;
  data_ur.miesiac := 1;
  data_ur.dzien := 12;
  osoba.nazwisko := 'Kowalski';
  osoba.imie1 := 'Jan';
  osoba.data_ur.rok := 1998;
  osoba.data_ur := data_ur
end.
```

Operatory i ich priorytety

Operatory	Priorytet	Kategoria
`+, -, @, not`	1	jednoargumentowe
`*, /, div, mod, and, shl, shr`	2	multiplikatywne
`+, -, or, xor`	3	addytywne
`=, <>, <, >, <=, >=, in`	4	relacyjne

Wyrażeniem jest:

- stała bez znaku
- nazwa zmiennej
- wywołanie funkcji
- wyrażenie w nawiasach
- wyrażenie poprzedzone operatorem jednoargumentowym
- dwa wyrażenia połączone operatorem dwuargumentowym

Relacje

- dają w wyniku wartości typu `Boolean`
- działają także dla napisów (leksykograficznie wg kodów ASCII)

Operacje teoriomnogościowe

- `+, -, *` - suma, różnica i iloczyn zbiorów o zgodnych typach bazowych
- Przykłady:

wyrażenie	wartość
`[2,3,4]+[4,5,6]`	`[2,3,4,5,6]`
`[2,3,4]-[4,5,6]`	`[2,3]`
`[2,3,4]*[4,5,6]`	`[4]`

Konkatenacja

wyrażenie	wartość
`'Turbo'+'Pascal'`	`'TurboPascal'`
`'Oddzielone'+' '+'wyrazy'`	`'Oddzielone wyrazy'`
`'25'+''+'kWh'`	`'25kWh'`

Instrukcja przypisania

```
zmienna := wyrażenie
```

Instrukcja złożona

```
begin
  Instrukcja_1;
  Instrukcja_2;
  ...
  Instrukcja_n
end
```

Instrukcja if-then-else

```
if wyrażenie then instrukcja
```

lub

```
if wyrażenie then instrukcja
             else inna_instrukcja
```

Przykład:

```
{ Liczymy maksimum z x i y }
if x>y then maksimum := x
       else maksimum := y
```

Zagnieżdżone if-then-else

```
if wyrażenie_1 then instrukcja_1
else if wyrażenie_2 then instrukcja_2
else if wyrażenie_3 then instrukcja_3
else instrukcja_4
```

Instrukcja case

```
case wyrażenie of
  sekwencja_instrukcji_wyboru
end
```

lub

```
case wyrażenie of
  sekwencja_instrukcji_wyboru
  else instrukcja
end
```

Przykłady:

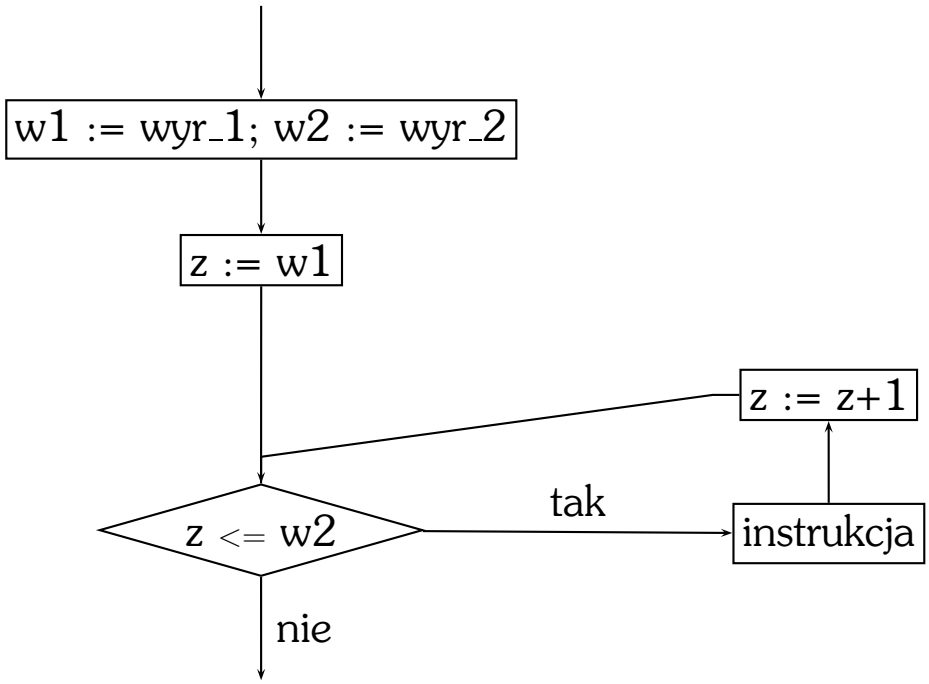
1. ```
   case miesiac of
     1,3,5,7,8,10,12 : dni := 31;
     4,6,9,11        : dni := 30;
     2               : dni := 28
   end
   ```
2. ```
   case miesiac of
     4,6,9,11        : dni := 30;
     2               : dni := 28;
     else              dni := 31
   end
   ```

Instrukcja for

```
for zmienna := wyr_1 to wyr_2 do instrukcja
```

lub

```
for zmienna := wyr_1 downto wyr_2 do instrukcja
```

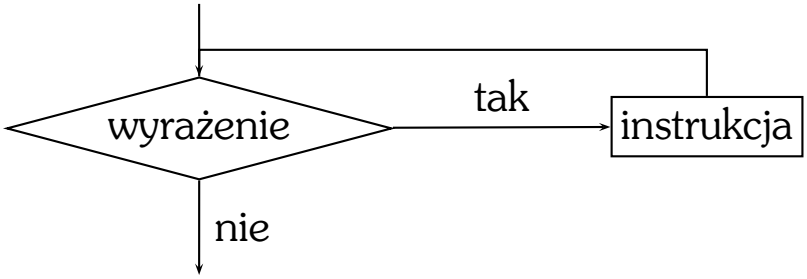


Przykład:

```
{ Liczymy silnię liczby n }
silnia := 1;
for i:=2 to n do
  silnia := silnia*i;
```

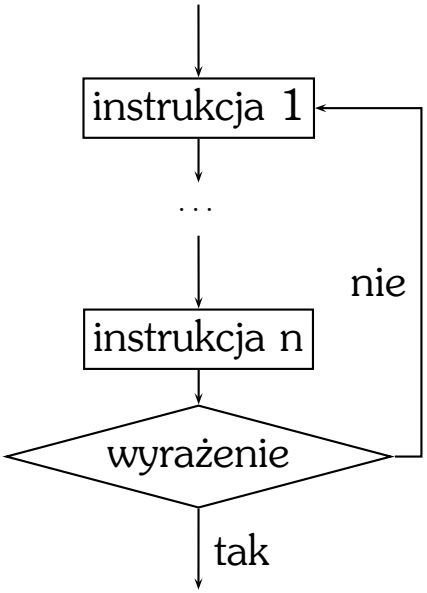
Instrukcja while

```
while wyrażenie do instrukcja
```



Instrukcja repeat

```
repeat instrukcja_1; ...; instrukcja_n
until wyrażenie
```



Funkcje obsługi standardowego wejścia-wyjścia

```
Write(lista_argumentów)
Writeln(lista_argumentów)
Read(lista_argumentów)
Readln(lista_argumentów)
```

- `lista_argumentów` może mieć dowolną długość
- wersje z końcówką `ln` dodatkowo zapisują/czytają znaki końca linii (CR+LF)
- każdy z argumentów powinien mieć jedną z postaci:
  - `wyrażenie`
  - `wyrażenie:długość`
  - `wyrażenie:długość:miejsca_dziesiętne`

Przykłady:

```
const n : Integer = 5;
      x : Real = 2.567;
      s : String = 'jakiś napis';

begin                          { wypisany tekst }
  Write('Dzień dobry');        { 'Dzień dobry' }
  Write(n:3);                  { '  5' }
  Write(x:6:2);                { '  2.57' }
  Write('2 * 2 = ', 2*2);      { '2 * 2 = 4' }
  Write('Oto ', s);            { 'Oto jakiś napis' }
end.
```

Pierwsze proste programy

- zamiana wartości zmiennych
- silnia

```
var n, i : Integer;
    silnia : LongInt;
begin
  Write('Proszę o liczbę : ');
  ReadLn(n);
  silnia := 1;
  for i:=2 to n do
    silnia := silnia * i;
  WriteLn(n, '! = ', silnia);
end.
```

- suma n pierwszych liczb naturalnych
- suma n liczb
- wypisywanie kolejnych liczb pierwszych
- wypisywanie kolejnych elementów ciągu Fibonacciego

Funkcje i procedury

- Definicja procedury

```
procedure nazwa_procedury(lista_parametrów);
część_opisowa
begin
  ciąg instrukcji
end;
```

- Definicja funkcji

```
function nazwa_funkcji(lista_parametrów) : typ_wyniku;
część_opisowa
begin
  ciąg instrukcji
end;
```

- Przykłady:

```
procedure WycentrujNapis(napis : string, n : Integer);
var ile_spacji, i : Integer;
begin
  ile_spacji := n - Length(napis);
  for i:=1 to ile_spacji div 2 do
    Write(' ');
  Write(napis);
  for i:=1 to (ile_spacji+1) div 2 do
    Write(' ')
end;

function Silnia(n : Integer) : LongInt;
var wynik, i : LongInt;
begin
  wynik := 1;
  for i:=2 to n do
    wynik := wynik * i;
  Silnia := wynik
end;
```

Argumenty przekazywane przez wartość i przez zmienną

```
var a, b : Integer;

procedure Zamien1(x, y : Integer);
var z : Integer;
begin
  z := x;
  x := y;
  y := z;
end;

procedure Zamien2(var x, y : Integer);
var z : Integer;
begin
  z := x;
  x := y;
  y := z;
end;

begin
  a := 5;
  b := 7;
  WriteLn('a = ', a, ', b = ', b); { 'a = 5, b = 7' }
  Zamien1(a, b);
  WriteLn('a = ', a, ', b = ', b); { 'a = 5, b = 7' }
  Zamien2(a, b);
  WriteLn('a = ', a, ', b = ', b); { 'a = 7, b = 5' }
end.
```

Stos wywołań

- Adresy funkcji i argumentów
- Rozmiar stosu jest skończony

Funkcje bezpośrednio rekurencyjne (wywołujące same siebie)

```
function Silnia(n : Integer) : LongInt;
begin
  if n <= 1 then Silnia := 1
            else Silnia := n*Silnia(n-1)
end;

function Potega(n, p:Integer) : LongInt;
var pomoc : LongInt;
begin
  if p = 0
  then potega := 1
  else
    begin
      pomoc := Potega(n, p div 2);
      if p mod 2 = 0
      then Potega := pomoc*pomoc
      else Potega := pomoc*pomoc*n
    end
end;
```

Typy rekurencyjne

- Liczby naturalne: 0 i operator następnika
- Drzewa: drzewo puste i drzewo łączące dwa drzewa

Funkcje pośrednio rekurencyjne (wywołujące siebie nawzajem)

```
type ListaDrzew = ...
     Drzewo = rekord
                wezel : String [10];
                poddrzewa : ListaDrzew;
              end;

function SzukajWDrzewie(d : Drzewo; var wynik : Drzewo)
    : Boolean;
begin
  if wezel = 'klucz' then
  begin
    wynik := d;
    SzukajWDrzewie := true
  end
  else
    SzukajWDrzewie := SzukajWLiscie(d.poddrzewa, wynik);
end;

function SzukajWLiscie(l : ListaDrzew; var wynik : Drzewo)
    : Boolean;
var i: Integer;
    znalezione : Boolean;
begin
  i:=1;
  znalezione := false;
  while i<=DlugoscListy(l) and not znalezione do
  begin
    znalezione := SzukajWDrzewie(ElementListy(l, i), wynik);
    i := i+1;
  end;
  SzukajWLiscie := znalezione;
end;
```

Typy proceduralne - przykład prostego kodowania

```
type Tablica = array [1..10] of Integer;
     Koder = function(n : Integer) : Integer;
var x : Tablica;
    i : Integer;

function Koduj1(n : Integer) : Integer; far;
begin
  Koduj1 := n xor 1998
end;
function Koduj2(n : Integer) : Integer; far;
begin
  Koduj2 := n + 27 xor 1998
end;
function Dekoduj2(n : Integer) : Integer; far;
begin
  Dekoduj2 := n xor 1998 - 27
end;
procedure Mapuj(var x : Tablica; koduj : Koder); far;
begin
  for i:=1 to 10 do x[i] := koduj(x[i]);
end;
procedure Pokaz(x : Tablica);
begin
  for i:=1 to 10 do Write(x[i], '  ');
  WriteLn;
end;

begin
  for i:=1 to 10 do x[i] := i;
  Pokaz(x);
  Mapuj(x, Koduj1); Pokaz(x);
  Mapuj(x, Koduj1); Pokaz(x);
  Mapuj(x, Koduj2); Pokaz(x);
  Mapuj(x, DeKoduj2); Pokaz(x);
end.
```

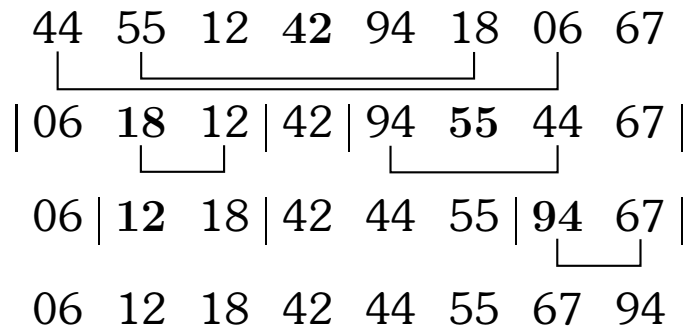
Sortowanie przez proste wybieranie

dane	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4	06	12	18	42	94	55	44	67
5	06	12	18	42	44	55	94	67
6	06	12	18	42	44	55	94	67
7	06	12	18	42	44	55	67	94

Sortowanie przez prostą zamianę (bąbelkowe)

dane	1	2	3	4	5	6	7
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

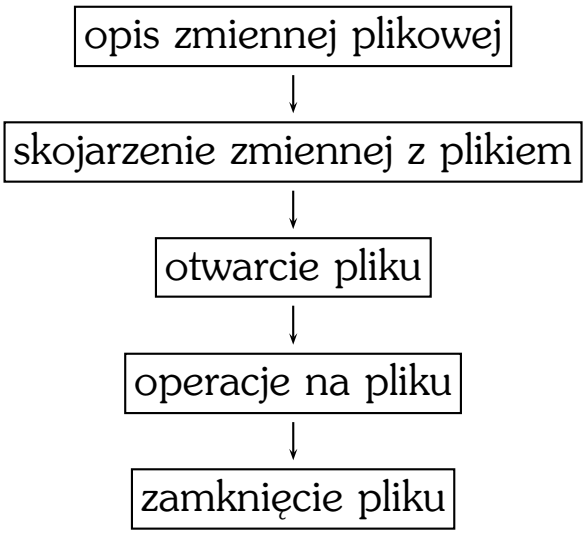
Sortowanie przez podział (szybkie)




```
type Tablica = array [1..100] of Real;
procedure SortowanieSzybkie(var t : Tablica; n : Integer);
  procedure Sortuj(var t : Tablica; l, p : Integer);
  var i, j : Integer; x, w : Real;
  begin
    i := l; j := p;
    x := t[(l+p) div 2];
    repeat
      while t[i] < x do i := i+1;
      while x < t[j] do j := j-1;
      if i <= j then
      begin
        w := t[i]; t[i] := t[j]; t[j] := w;
        i := i+1; j := j-1;
      end
    until i > j;
    if l < j then sortuj(t, l, j);
    if i < p then sortuj(t, i, p);
  end;
begin
  sortuj(t, 1, n)
end;
```

Obsługa wejścia-wyjścia

- Przetwarzanie plików



- Opis zmiennej plikowej

```
type Osoba = record
               nazwisko : string[20];
               imie     : string[15];
             end;
     TypPlikowy1 = file of Osoba;
     TypPlikowy2 = file;
     TypPlikowy3 = Text;
```

- Skojarzenie zmiennej z plikiem

```
var plik1 : Text;
    plik2 : TypPlikowy1;
...
  Assign(plik1, 'LPT1');
  Assign(plik2, 'D:\DANE\OSOBY.DAT');
```

- Otwieranie i zamykanie pliku

```
{ Rewrite - aby utworzyć nowy plik }
Rewrite(zmienna_plikowa)
Rewrite(zmienna_plikowa, rozmiar_zapisu)
{ Reset - aby otworzyć istniejący plik }
Reset(zmienna_plikowa)
Reset(zmienna_plikowa, rozmiar_zapisu)
{ Append - aby dopisywać do pliku tekstowego }
Append(zmienna_plikowa)
{ Close - aby zamknąć plik }
Close(zmienna_plikowa)
```

- Operacje na pliku

```
{ Dla plików tekstowych }
Write(zmienna_plikowa, lista_argumentów)
WriteLn(zmienna_plikowa, lista_argumentów)
Read(zmienna_plikowa, lista_zmiennych)
ReadLn(zmienna_plikowa, lista_zmiennych)
{ Dla plików zdefiniowanych }
Write(zmienna_plikowa, lista_zmiennych_wyjściowych)
Read(zmienna_plikowa, lista_zmiennych_wejściowych)
{ Dla plików niezdefiniowanych }
BlockWrite(zmienna_plikowa, bufor, licznik)
BlockWrite(zmienna_plikowa, bufor, licznik, wynik)
BlockRead(zmienna_plikowa, bufor, licznik)
BlockRead(zmienna_plikowa, bufor, licznik, wynik)
```

- Inne operacje na plikach

```
{ Czy to koniec pliku }
 Eof(zmienna_plikowa)
{ Czy to koniec pliku + ignoruj odstępy }
 SeekEof(zmienna_plikowa)
{ Czy to koniec linii }
 Eoln(zmienna_plikowa)
{ Czy to koniec linii + ignoruj odstępy }
 SeekEoln(zmienna_plikowa)
{ Aktualna pozycja w pliku }
 FilePos(zmienna_plikowa)
{ Rozmiar pliku }
 FileSize(zmienna_plikowa)
{ Skocz do pozycji }
 Seek(zmienna_plikowa, pozycja)
{ Usuń aż do końca pliku }
 Truncate(zmienna_plikowa)
{ Zmień bieżący katalog }
 ChDir(wyrażenie_łańcuchowe)
{ Pobierz bieżący katalog dysku określonego przez
  liczbę (0 - dysk bieżący, 1 - A:, 2 - B: ...) }
 GetDir(liczba, łańcuch)
{ Stwórz katalog }
 MkDir(wyrażenie_łańcuchowe)
{ Usuń pusty katalog }
 RmDir(wyrażenie_łańcuchowe)
{ Usuń plik (zamknięty) }
 Erase(zmienna_plikowa)
{ Zmień nazwę pliku dyskowego }
 Rename(zmienna_plikowa, wyrażenie_łańcuchowe)
```

- Przykład obsługi plików (drukowanie pliku tekstowego)

```
var nazwa, linia : string;
    plik, druk   : Text;
begin
  Write('Podaj nazwę pliku do wydrukowania : ');
  ReadLn(nazwa);
  Assign(plik, nazwa);
  Assign(druk, 'LPT1');
  Reset(plik);
  Rewrite(druk);
  while not SeekEof(plik) do
  begin
    ReadLn(plik, linia);
    WriteLn(druk, linia);
  end;
  Close(plik);
  Close(druk);
  WriteLn('Plik ', nazwa, ' został wydrukowany.')
end.
```

Moduły

- Ogólna postać modułu

```
unit nazwa_modułu;
część opisowa
część implementacyjna
część inicjująca
```

- Część opisowa

```
interface
deklaracje modułów
deklaracje stałych
deklaracje typów
deklaracje zmiennych
lista nagłówków procedur i funkcji
```

- Część implementacyjna

```
implementation
deklaracje wewnętrznych stałych, typów etc.
definicje procedur i funkcji
```

- Część inicjująca - instrukcja (prosta albo złożona), która będzie wykonana w celu zainicjowania modułu lub `end`.

Moduły standardowe i ich ciekawe procedury i funkcje

- Moduł **System**
  - Exit
  - Round, Trunc, Abs, Sqrt
  - Random
  - SizeOf
  - FillChar, Move
  - IOResult
  - UpCase
- Moduł **Crt**
  - TextMode
  - TextColor, TextBackground
  - Window, GoToXY
  - ClrScr, ClrEol, DelLine, InsLine
  - KeyPressed, ReadKey
  - Delay, Sound, NoSound
- Moduł **Dos**
  - GetDate, GetTime, SetDate, SetTime
  - GetFTime, SetFTime
  - DiskFree, DiskSize
  - GetFAttr, SetFAttr
  - FindFirst, FindNext
  - Exec

Grafika BGI (Borland Graphics Interface) - moduł **Graph**

- Sprawdzanie sprzętu
  - DetectGraph
- Inicjowanie i kończenie trybu graficznego
  - InitGraph, CloseGraph
  - RestoreCrtMode
  - SetGraphMode
- Rysowanie na ekranie
  - Line, LineRel, LineTo
  - PutPixel, GetPixel
  - Rectangle, Circle, DrawPoly
  - Arc, Ellipse
  - Bar, Bar3d, FillPoly
  - PieSlice
  - FloodFill
- Pisanie na ekranie
  - SetTextStyle
  - SetTextJustify
  - OutText, OutTextXY
  - TextHeight, TextWidth
  - GetTextSettings
- Obszary
  - ImageSize, GetImage, PutImage
- Inne procedury i funkcje
  - SetViewPort
  - MoveTo, MoveRel
  - SetColor, SetBkColor
  - SetLineStyle, SetFillStyle

Zalety i wady języka C

- jest językiem wysokiego i niskiego poziomu zarazem
- w C można zrobić wszystko
- dostęp do pamięci, rejestrów i innych wnętrzności
- nie ma ochrony programisty przed nim samym
- "mały język" - istotna rola bibliotek
- optymalizatory

Pliki nagłówkowe

- przeznaczone na definicje stałych, typów i deklaracje funkcji
- nie powinny zawierać kodu (obszary danych i kodu są rozłączne)

Najprostszy program

```
main()
{
}
```

Typy, operatory, wyrażenia

- identyfikatory

```
x  i  x12  _123  ilosc_iteracji
```

- typy i rozmiary danych

```
int  short int  long int  float  double
char *  int *  void *
```

- stałe

```
0  1  2.5  123.456e-7  0.123E4  123L  0x1F  \027
"tekst"  ""
```

- deklaracje

```
int i, j, ilosc;
int tablica[10];
char c, linia[1000];
```

- operatory
  - arytmetyczne: `+ - * / %`
  - logiczne
    - relacje: `> < >= <= == !=`
    - łączniki: `&&, ||`
  - bitowe: `& | ^ ~ << >>`
  - przypisania: `= += -= *= /= &= |= ^= ~=`
  - zwiększania i zmniejszania: `++ --`
  - inne: `() [] . -> , :: ?:`

Priorytety operatorów

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

Instrukcja if-else

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

- część `else` jest opcjonalna
- `if (wyrażenie != 0)` najczęściej zastępuje się przez `if (wyrażenie)`
- dwuznaczności

```
if (n>0)
  if (a>b)
    z = a;
  else
    z=b
```

```
if (n>0)
{
  if (a>b)
    z = a;
}
else
  z=b
```

- dwuznaczność + niepoprawne formatowanie

```
if (n>0)
  for (i=0; i<n, i++)
    if (jest_dobrze(i))
    {
      printf("Hurra! Znalazłem!\n");
      return i;
    }
else
  printf("Błąd - n jest zerem.\n");
```

- Konstrukcje złożone

```
if (wyrażenie)
  instrukcja-1
else
  if (wyrażenie2)
    instrukcja-2
  else
    if (wyrażenie3)
      instrukcja-3
    else
      instrukcja-4
```

Przykład:

```
if (delta>0)
  printf("Są dwa różne pierwiastki...")
else
  if (delta==0)
    printf("Jest jeden pierwiastek podwójny...")
  else
    printf("Nie ma pierwiastków.")
```

Instrukcja switch

```
switch (wyrażenie_całkowite)
{
  case wartość1 : instrukcja1
  case wartość2 : instrukcja2
  .
  .
  .
  case wartośćn : instrukcjan
  default : instrukcja
}
```

- część `default` jest opcjonalna
- wartości przy `case` wskazują miejsce od którego zaczynamy wykonywanie instrukcji - instrukcja `break` przerywa wykonywanie kolejnych instrukcji
- Przykład:

```
znak = toupper(getchar());
switch (znak)
{
  case 'A' : Opcja(1); break;
  case 'B' : Opcja(2); break;
  case 'C' : Opcja(3); break;
  case '1' :
  case '2' :
  case '3' :
  case '4' :
  case '5' :
  case '6' :
  case '7' :
  case '8' :
  case '9' : Opcja(znak-'0'); break;
}
```

Pętla `while`

- najpierw sprawdzamy warunek, potem wykonujemy instrukcję

```
while (warunek)
  instrukcja
```

Przykład:

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma = 0, i = 0;

while (i<rozmiar))
  suma += tab[i++];
```

Pętla `do-while`

- najpierw wykonujemy instrukcję, potem sprawdzamy warunek

```
do
  instrukcja
while (warunek);
```

Przykład:

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma = 0, i=0;

do
  suma += tab[i++];
while (i<rozmiar);
```

Pętla `for`

```
for (wyr1; wyr2; wyr3)
  instrukcja
```

jest równoważna rozwinięciu

```
wyr1;
while (wyr2)
{
  instrukcja
  wyr3;
}
```

Uwaga: Dowolne z trzech wyrażeń można pominąć (brak warunku (`wyr2`) jest równoznaczny z warunkiem zawsze prawdziwym). W szczególności:

```
for (;;)
{
  ...
}
```

Przykład:

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma, i;

for (i=0,suma=0; i<rozmiar; i++)
  suma += tab[i++];
```

Uwaga: Instrukcja `break` działa także z pętlami

Instrukcja `continue` - przejście do kolejnej iteracji w pętli
Przykład:

```
for (i=0; i<n; i++)
{
  if (!przetwarzac(i))
    continue;
  przetwarzaj(i);
}
```

Instrukcja goto i etykiety

- Nie używać!
- Przykład:

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    if (macierz[i][j] < 0)
      goto znaleziono;

/* co jeśli nie znaleziono */
...

znaleziono:
/* co jeśli znaleziono (na pozycji i, j) */
...
```

Wskaźniki i tablice

- operator `&` - adres do obiektu (`px = &x`)
- operator `*` - wartość spod adresu (`x = *px`)
- tablice

```
int a[10];   /* a jest tablicą 10-elementową */
int *pa, x;  /* wskaźnik i zmienna całkowita */
pa = &a[0];  /* równoważne zapisowi pa = a; */
a[1] = 5;
x = *(pa+1); /* równoważne zapisowi x = a[1]; */
```

- `&a[i]` jest równoważne `a+i`
- tablice wielowymiarowe nie istnieją, są tablice tablic:

```
int macierz[5][5];
int *tab1[10];
```

- deklaracja tablicy z zainicjowaniem

```
int  a[5] = {1, 2, 3, 4, 5};
char c[] = {'a', 'b', 'c'};

main()
{
  int rozmiar = sizeof(c)/sizeof(*c);
  ...
}
```

Struktury i unie

- Syntaktyka struktury (poniżej `nazwa` jest opcjonalna)

```
struct nazwa
{
  lista_deklaracji_pól
};
```

- Przykład deklaracji struktury

```
struct data
{
  int dzien;
  int miesiac;
  int rok;
};
```

- Zmienne typu strukturalnego i wskaźniki do struktur

```
struct data dzis = {30, 10, 1998}, *pd = &dzis;

printf("%d-%02d-%d", dzis.dzien, dzis.miesiac, dzis.rok);
printf("%d-%02d-%d", pd->dzien, pd->miesiac, pd->rok);

++pd->dzien           /* zwiększy dzien a nie pd */
```

- Inicjowanie tablicy struktur

```
struct data daty[] = {{1,  1, 1997},
                      {2, 11, 1997},
                      {13, 5, 1998}};
```

- Struktury ze wskaźnikami do samych siebie

```
struct drzewo
{
  float wartosc_wezla;
  struct drzewo *prawe_poddrzewo, *lewe_poddrzewo;
};
```

- Pola bitowe

```
struct
{
  unsigned jest_samcem : 1;
  unsigned jest_ssakiem : 1;
  unsigned ma_rogi : 1;
} flagi;
```

- Unie

```
union opis
{
  int   ival[2];
  float fval;
  void *pval;
} wartosc_opisu;
wartosc_opisu.fval = 1.15;
```

- Deklaracja `typedef`

```
typedef float odleglosc;
typedef struct
{
  int dzien;
  int miesiac;
  int rok;
} data;

main()
{
  odleglosc d;
  data dzis;
  ...
}
```

Funkcje, argumenty, zmienne

- Syntaktyka definicji funkcji:

```
typ_wartości nazwa(lista_argumentów_jeśli_występują)
deklaracje_argumentów_jeśli_występują
{
  deklaracje_i_instrukcje_jeśli_występują
}
```

lub

```
typ_wartości nazwa(deklaracje_argumentów_jeśli_występują)
{
  deklaracje_i_instrukcje_jeśli_występują
}
```

- Instrukcja `return` kończy wykonywanie funkcji i ewentualnie zwraca wartość funkcji
- Przykłady:

```
long silnia(n)
int n;
{
  long wynik = 0;
  while (n>0) wynik *= n--;
  return wynik;
}
```

```
long silnia(int n)
{
  long wynik = 0;
  while (n>0)  wynik *= n--;
  return wynik;
}
```

- Funkcje rekurencyjne

```
long silnia(int n)
{
  return (n<2)? 1 : return n * silnia(n-1);
}
```

- Funkcja `main()` i jej argumenty

```
void main(int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
    printf("Argument %d = %s\n", i, argv[i]);
}
```

- argumenty przekazywane są zawsze przez wartość

```
void swap(int x, int y)
{
  int z = x;
  x = y;
  y = z;
}
```

```
void swap(int *x, int *y)
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

- argumenty `char napis[]` oraz `char *napis`
- zmienne lokalne i globalne
- lokalne zmienne statyczne

```
void funkcja(void)
{
  static int licznik = 0;
  printf("To jest %d wywołanie tej funkcji\n", ++licznik);
}
```

- wskaźniki do funkcji

```
void mapuj(int tablica[], int dlugosc, int (*funkcja)(int))
{
  int i;
  for (i=0; i<dlugosc; i++)
    tablica[i] = (*funkcja)(tablica[i]);
}
```

Wejście i wyjście

- Standardowe funkcje realizujące wejście/wyjście:

```
#include <stdio.h>
int putchar(int c);
int getchar(void);
```

```
#include <conio.h>
int putch(int c);
int getch(void);
int getche(void);
```

- Formatowane wejście/wyjście:

```
#include <stdio.h>
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

- Obsługa plików dyskowych
  - Struktura `FILE`

```
FILE *fp = fopen(nazwaPliku, tryb);
...
fclose(fp);
```

Możliwe tryby otwarcia:

r	otwórz, żeby czytać
w	stwórz, żeby pisać
a	otwórz, żeby pisać (stań na końcu)
r+	otwórz, żeby czytać i pisać
w+	stwórz, żeby czytać i pisać
a+	otwórz, żeby czytać i pisać (stań na końcu)
b	otwórz jako plik binarny
t	otwórz jako plik tekstowy

– deskryptory plików

```
int handle = open(nazwaPliku, dostep, tryb);
...
close(handle);
```

Flagi dostepu:

`O_RDONLY`	otwórz, żeby czytać
`O_WRONLY`	otwórz, żeby pisać
`O_RDWR`	otwórz, żeby pisać i czytać
`O_APPEND`	po otwarciu stań na końcu pliku
`O_CREAT`	stwórz plik, jeśli nie istnieje
`O_TRUNC`	jeśli plik istnieje, wyczyść zawartość
`O_BINARY`	otwórz jako plik binarny
`O_TEXT`	otwórz jako plik tekstowy

Flagi trybu:

`S_IWRITE`	prawo pisania
`S_IREAD`	prawo czytania
`S_IREAD|S_IWRITE`	prawo pisania i czytania

– inne funkcje:

```
int putc(int c, FILE *stream);
int getc(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
```

- Standardowe urządzenia: `stdin, stdout, stderr`

Dyrektywy prekompilatora

- `#include` - wkłada zawartość pliku w miejsce gdzie się pojawia

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

- `#define` - definiuje zmienne prekompilatora

```
#define __PLIK_H
#define TABSIZE 100
#define MAX(x,y) ((x>y)? (x) : (y))
```

- `#ifdef`, `#ifndef` - przetwarzanie warunkowe

```
#ifndef __PLIK_H
#define __PLIK_H
...
#endif
```

Inne ważne zagadnienia

- Biblioteki
- Zmienne statyczne (`static`) i zewnętrzne (`extern`)
- Stos i sterta
  - stos (ang. stack) - miejsce na adresy wykonywanych funkcji i ich argumentów
  - sterta (ang. heap) - pamięć do przechowywania danych
- Usuwanie błędów (debugging)
  - narzędzia i środowiska do usuwania błędów
  - makrodefinicja assert
- Przenaszalność oprogramowania
  - uwaga na rozmiary danych
  - uwaga na reprezentację danych (kolejność bajtów)

Prosta baza danych

```
#ifndef __BAZA_H
#define __BAZA_H

#define MAX_DL_NAZWISKO 25
#define MAX_DL_IMIE     15

typedef struct
{
  char nazwisko[MAX_DL_NAZWISKO+1];
  char imie1[MAX_DL_IMIE+1];
  char imie2[MAX_DL_IMIE+1];
  data dataUr;
} Osoba;

void DodajRekord(int n, Osoba os);
Osoba CzytajRekord(int n);
void ZapiszRekord(int n, Osoba os);
void PokazRekord(int n);

#endif //__BAZA_H

main()
{
  int wybor, koniec = 0;
  do
  {
    wybor = UruchomMenu();
    switch(wybor);
    {
      case 0 : koniec = 1; break;
      case 1 : DodajRekord(); break;
      ...
    }
  }
  while (!koniec);
}
```

Wskaźniki do funkcji i funkcja qsort()

```
#include <stdio.h>     // dla printf()
#include <stdlib.h>    // dla qsort()
#include <string.h>    // dla strcmp() i stricmp()

typedef int (*Funkcja)(char *, char *);

int Porownaj1(char **s1, char **s2)
{
  return strcmp(*s1, *s2);
}
int Porownaj2(char **s1, char **s2)
{
  return stricmp(*s1, *s2);
}
int Porownaj3(char **s1, char **s2)
{
  if (strlen(*s1) > strlen(*s2)) return 1;
  if (strlen(*s1) < strlen(*s2)) return -1;
  return 0;
}

void main()
{
  int i, j;
  char *ala[] = {"ALA MA KOTA", "Ala ma Kota", "alamakota", "Ala"};
  Funkcja f[] = {Porownaj1, Porownaj2, Porownaj3};
  for (i=0; i<sizeof(f)/sizeof(*f); i++)
    printf("Wynik[%d] = %d\n", i, f[i](ala, ala+1));
  for (i=0; i<sizeof(f)/sizeof(*f); i++)
  {
    qsort(ala, sizeof(ala)/sizeof(*ala), sizeof(*ala), f[i]);
    printf("\nPo sortowaniu nr %d\n", i);
    for (j=0; j<sizeof(ala)/sizeof(*ala); j++)
      printf("%s\n", ala[j]);
  }
}
```

Lokalne zmienne tablicowe

- zwracanie ich na zewnątrz jest niebezpieczne

```
#include <dir.h>  // dla mktemp()

char *NazwaTymczasowa(void)
{
  char tmp[] = "C:\\TEMP\\TMPXXXXXX";
  mktemp(tmp);
  return tmp;     // Bardzo niebezpieczne
}
```

- rozwiązanie: statyczne tablice, które „żyją” mimo wyjścia z funkcji

```
#include <dir.h>  // dla mktemp()

char *NazwaTymczasowa(void)
{
  static char tmp[] = "C:\\TEMP\\TMPXXXXXX";
  mktemp(tmp);
  return tmp;     // Teraz już w porządku
}
```

Pamięć przydzielana dynamicznie

Idea: tworzymy wskaźnik (zmienna przechowująca adres) do obiektu, rezerwujemy fragment pamięci i jego adres zapamiętujemy we wskaźniku

- język Pascal

```
var wskaznik = ^Osoba;
begin
  New(wskaznik);     // pamięć przydzielona
  wskaznik^.nazwisko = "Kowalski";
  ...
  Dispose(wskaznik); // pamięć zwolniona
end.
```

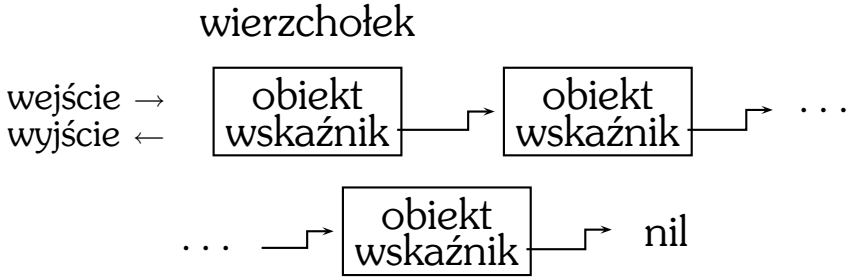
- język C

```
main()
{
  Osoba *wskaznik;
  wskaznik = (Osoba *)malloc(sizeof(Osoba));
  strcpy(wskaznik->nazwisko, "Kowaslki");
  ...
  free(wskaznik);
}
```

Sterta (ang. heap)

- obszar pamięci przeznaczony na zmienne dynamiczne
- to coś całkiem innego niż stos (ang. stack), który jest rejestrem wywołań funkcji i procedur

Dynamiczne struktury danych - stos



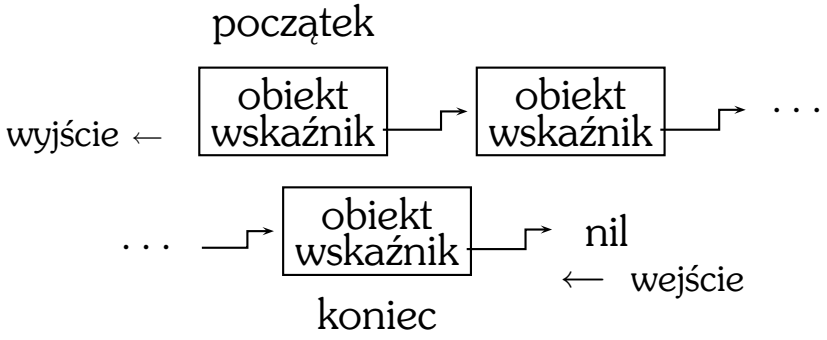
- język Pascal

```
type wskaźnik_stosu = ^skladnik_stosu;
     skladnik_stosu = record
                        obiekt   : typ_obiektu;
                        wskaznik : wskaznik_stosu;
                      end;
```

- język C

```
typedef struct skladnik_stosu
{
  typ_obiektu obiekt;
  struct skladnik_stosu *wskaznik;
} skladnik_stosu;
void NaStos(skladnik_stosu **wierzcholek, typ_obiektu o)
{
  skladnik_stosu *tmp = malloc(sizeof(skladnik_stosu));
  tmp->obiekt = o;
  tmp->wskaznik = *wierzcholek;
  *wierzcholek = tmp;
}
typ_obiektu ZeStosu(skladnik_stosu **wierzcholek)
{
  skladnik_stosu *tmp = *wierzcholek;
  typ_obiektu skladnik = tmp->obiekt;
  *wierzcholek = tmp->wskaznik;
  free(tmp);
  return skladnik;
}
```

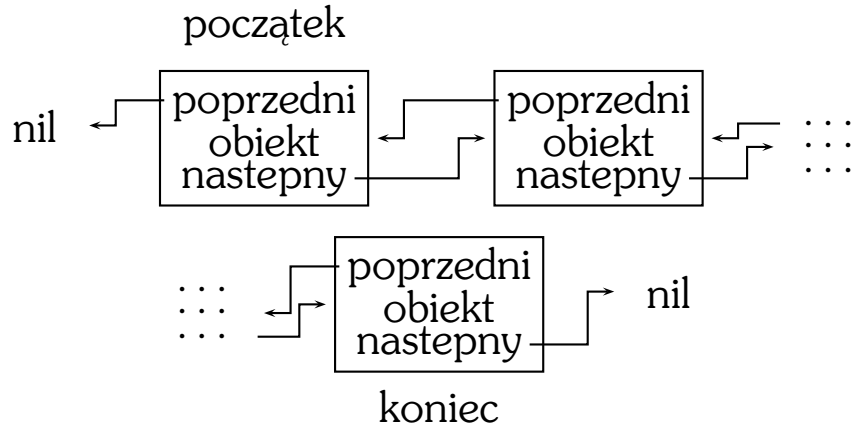
Dynamiczne struktury danych - kolejka




```
typedef struct skladnik_kolejki
{
  typ_obiektu obiekt;
  struct skladnik_kolejki *wskaznik;
} skladnik_kolejki;

void DoKolejki(skladnik_kolejki **poczatek,
               skladnik_kolejki **koniec, typ_obiektu o)
{
  skladnik_kolejki *tmp = malloc(sizeof(skladnik_kolejki));
  tmp->obiekt = o;
  tmp->wskaznik = 0;
  if (*koniec) {
    (*koniec)->wskaznik = tmp;
    *koniec = tmp;
  } else *koniec =  *poczatek = tmp;
}
typ_obiektu ZKolejki(skladnik_kolejki **poczatek,
                     skladnik_kolejki **koniec)
{
  skladnik_kolejki *tmp = *poczatek;
  typ_obiektu skladnik = tmp->obiekt;
  if (*koniec == *poczatek) *koniec = 0;
  *poczatek = tmp->wskaznik;
  free(tmp);
  return skladnik;
}
```

Dynamiczne struktury danych - lista dwukierunkowa




```
typedef struct element_listy
{
  typ_obiektu obiekt;
  struct element_listy *poprzedni;
  struct element_listy *nastepny;
} element_listy;

void DoListyZaElement(element_listy *zaKtory, typ_obiektu o)
{
  element_listy *nowy = malloc(sizeof(element_listy));
  nowy->obiekt = o;
  nowy->poprzedni = zaKtory;
  nowy->nastepny = zaKtory->nastepny;
  nowy->nastepny->poprzedni = nowy;
  zaKtory->nastepny = nowy;
}

void ZListy(element_listy *ktory)
{
  ktory->poprzedni->nastepny = ktory->nastepny;
  ktory->nastepny->poprzedni = ktory->poprzedni;
  free(ktory);
}
```

Narzędzia programowania

- kompilatory zewnętrzne

```
bcc -c plik1.c plik2.c
gcc -c plik1.c plik2.c
```

- program `make` i pliki `makefile`

```
OBJFILES = error.o rules.o feature.o hidden.o network.o main.o

%o: %cpp
        g++ -c -g $<

rules: ${OBJFILES}
        g++ -o rules ${OBJFILES} -lcurses

clean:
        rm -f rules ${OBJFILES}
```

- profesjonalne edytory (np. `Emacs`)
  - automatyczne kopie bezpieczeństwa
  - zarządzanie kompilacją (odnajdywanie błędów itp.)
  - pełna konfigurowalność (rozszerzenia w lispie)
  - współpraca z innymi narzędziami
  - kolorowanie tekstów programu
- `grep` i wyrażenia regularne, `sed` (Unix)

```
grep [Ww]yrazenie *.c *.h
grep program[A-Za-z]* *.txt
grep program[A-Za-z]+ *.txt
```

- `awk` - przetwarzanie plików tekstowych

Narzędzia programowania

- odpluskwianie (usuwanie błędów)
  - srodowiska zintegrowane
  - zewnetrzne debuggery
  - odpluskwianie przy użyciu dwóch komputerów
- `WinSight` - podgląd sygnałów przychodzących do wybranych okien
- Turbo Profiler - szukanie „wąskich gardeł” w programie, liczenie liczby wywołań poszczególnych funkcji, instrukcji itp.
- `tlib` - tworzenie i analizowanie bibliotek
- `lex` i `yacc` (`flex` i `bison`) - definiowanie gramatyk i generowanie kodu źródłowego programu analizy wyrażeń.
- zarządzanie wieloma wersjami programu i praca grupowa
  - RCS - Revision Control System
  - CVS - Concurrent Versions System
  - SCCS - Source Code Control System
  - PVCS - rozpowszechniany z Borland C++

Różnice pomiędzy Pascalem a C

- C rozróżnia wielke i małe litery a Pascal nie
- C jest zarazem językiem niskiego poziomu
- program skompilowany w C jest szybszy
- C jest zwięzły, Pascal jest bardziej czytelny
- w C łatwiej manewrować wskaźnikami, dynamicznie przydzielać pamięć
- w Pascalu dokładniejsza kontrola przed korzystaniem z niedozwolonych obszarów pamięci
- w C są unie
- w C wskaźniki do funkcji, w Pascalu typy proceduralne
- argumenty w C przekazywane zawsze przez wartość, w Pascalu przez wartość lub przez zmienną
- w C jest wygodna i szybka instrukcja `for`
- w C biblioteki, w Turbo Pascalu moduły

Podsumowania i przestrogi

- porządny projekt (przemyślenie problemu)
- nazywanie zmiennych i komentowanie źródeł
- programujemy z góry w dół (top-down)
- funkcje krótkie choć więcej
- kopie bezpieczeństwa
- wieloplatformowość oprogramowania
  - kolejności bajtów w binarnej reprezntacji
  - długości standardowych typów
  - używamy funkcji określonych standardem języka
  - dyrektywa `#pragma`
- nie używamy zmiennych globalnych
- pamięć dynamiczna
  - uwaga na rozmiary przydzielonej pamięci
  - pamiętajmy o jej zwalnianiu
- program budowany z małych kawałków łatwiej przetestować
- dobry programista, to nie ten, który pamięta najwięcej funkcji, ale ten, który wie co daje się zrobić, z łatwością odnajduje funkcje których szuka i potrafi z nich skorzystać.