Literatura:
(1) Kernighan, Ritchie "Język ANSII C"
(2) Schildt "Język C"
(3) Cormen, Leiseron, Rivest "Wprowadzenie do algorytmów"

Algorytm - jednoznaczny przepis na rozwiązanie pewnego problemu w skończonej liczbie kroków (występujących w ściśle ustalonej kolejności) zawierający opis obiektów wraz z opisem czynności jakie należy wykonać na tych obiektach, aby osiągnąć zamierzony cel.

Formy algorytmu:
1. opis w języku naturalnym
2. schemat blokowy
3. opis w metajęzyku (pseudokodzie)
4. kod w konkretnym języku programowania ($\sim$ program)

Elementy składowe **schematu blokowego**:
start
stop
wejście (wczytanie danych)
wyjście (wypisanie wyników)
instrukcja
warunek

Przykład. Schemat blokowy dla rozwiązywania równania $ax^2+bx+c=0$.

Przykład. **Kod źródłowy** programu dla rozwiązywania równania $ax^2+bx+c=0$.

```
#include <stdio.h> //printf, scanf
#include <conio.h> //getch
#include <math.h>  //sqrt

main()
{ float a, b,c; //deklaracja zmiennych
  float delta;
  float x0,x1,x2;

  printf("Podaj a, b, c");
  scanf("%f%f%f",&a,&b,&c);

  if(a!=0) //a!=0 oznacza a różne od 0
  { delta=b*b-4*a*c;
    { if(delta>=0)
       { if(delta>0)
          { x1=(-b-sqrt(delta))/(2*a);
            x2=(-b+sqrt(delta))/(2*a);
            printf("x1=%f, x2=%f",x1,x2);
          }
         else
          { x0=-b/(2*a);
            printf("x0=%f",x0);
          }
       }
     else //delta<0
     {
      printf("Brak rozwiązań");
     }
  }
```

```
    else //a=0
    {
    ...
    }
  }
    getch(); //w starej wersji DevC++ konieczne
}
```

Po **kompilacji** kodu źródłowego program można **uruchomić**.

DZIAŁANIE PROGRAMU
Podaj a,b,c: 1 2 -3
x1=-3.000000, x2=1.000000

Typy danych w C

I STAŁE

1) liczbowe (int oraz float)
2) znakowe (char)
3) tekstowe (char*)

Definiowanie stałej:

```
const typ_stałej nazwa_stałej = wartość;
```

Przykłady:

```
const float pi=3.14;
const int rozmiar=1000;
const char gwiazdka='*';
```

II ZMIENNE

A. PROSTE

1) liczbowe
   a) **int** (integer = całkowite) - 4-bajtowe
   b) zmiennoprzecinkowe
   - pojedynczej precyzji **float**
   - podwójnej precyzji **double**

   Modyfikatory:

   ```
   long (np. long int, long)\\
   short (np. short int)\\
   signed (ze znakiem)\\
   unsigned (bez znaku)\\
   ```

2) znakowe (kod ASCII)
3) wyliczeniowe
4) typ pusty (**void**)

B. ZŁOŻONE

1) tablice (ciągi elementów tego samego typu)
2) struktury (zbiory elementów różnych typów)

Definiowanie (deklarowanie) zmiennych:

typ_zmiennej nazwa_zmiennej;

typ_zmiennej nazwa_zmiennej=wartość; //zadeklarowanie zmiennej z nadaniem jej początkowej wartości

typ_zmiennej nazwa_zmiennej1, nazwa_zmiennej2, ..., nazwa_zmienejn;

typ_zmiennej nazwa_zmiennej1=wartość1, ..., nazwa_zmienejn;

Przykłady.

```
int n=1000;
float max=3e10; //3·10^10
double mama=312.85;
float a=1,b,c=0;
```

W języku C nazwy zmiennych, stałych i innych obiektów definiowanych przez programistę określa się nazwą **identyfikatora**, który może się składać z liter alfabetu angielskiego, cyfr i znaku podkreślenia _ (przy czym pierwszy znak nie może być cyfrą).

Przykłady.

```
int brat;
int Brat; //dobrze, bo w C rozróżniamy małe i DUŻE litery!
int 2brat; //ŹLE!
char znak='a';
```

Zmienne deklarujemy w jednym z trzech miejsc:

a/ wewnątrz funkcji - wtedy mówimy o zmiennych **lokalnych**,
b/ w nagłówku definicji funkcji (wewnątrz nawiasu) - wtedy nazywa się je **parametrami formalnymi funkcji**, np.

```
float potega (float x, int n)
```

c/ poza wszelkimi funkcjami - wtedy nazywa się je zmiennymi **globalnymi**.

TABLICA - ciąg skończony elementów tego samego typu.

Deklaracja tablicy:

typ_elementu nazwa_tablicy[rozmiartablicy];

Uwaga. Pierwszy element w tablicy ma indeks 0 (a nie 1).

np.

```
int tab[10]; //deklaracja tablicy o nazwie tab o 10 elemetach typu całkowitego
tab[0]=1; //przypisanie wartości 1
tab[4]=16;
tab[10]=1024;// ŹLE, bo nie ma indeksu 10 w tab
scanf("%d", &tab[3]);
printf("%d",tab[3]);
```

Deklaracja tablicy z nadaniem wartości początkowych elementom tablicy:

typ_elementu nazwa_tablicy $[rozmiar] = \{w_0, w_1, \ldots, w_{rozmiar-1}\}$; /*$\Leftrightarrow$ nazwa_ tablicy $[0] = w_0$;
nazwa_ tablicy $[1] = w_1$; ...; nazwa_ tablicy $[rozmiar-1] = w_{rozmiar-1}$;*/

np.

```
int tab[10]={1,2,4,8,...,512}; //te 3 kropki są niepoprawne
```

STRUKTURA - zbiór elementów różnych typów.

Deklaracja struktury:

```
struct nazwa_struktury
  {typ_pola_1 nazwa_pola_1;
   typ_pola_2 nazwa_pola_2;
    ⋮

   typ_pola_n nazwa_pola_n;
  }[nazwy zmiennych typu struktura]; //opcjonalnie
```

np.

```
struct zespolona
    {float re;
     float im;
    } z1;
struct zespolona z2,z3;
z1.re=0;
z1.im=1;
z2.re=3.0;
z2.im=-4.666666;
printf("z=%f+%fi",z1.re,z1.im); // z=0+1i
```

```
np. struct wyborca
     {char imie[20];
      char nazwisko [30];
      int rok_urodzenia;
     };

struct wyborca w1,w2;
w1.rok_urodzenia=1996;
```

TYP WYLICZENIOWY

```
np. enum dni_tygodnia ={pn,wt,sr,cz,pt,sb,nd};//
⇔ pn=0, wt=1, sr=2, ..., nd=6
```

Definiowanie NAZW TYPÓW danych

```
np. typedef float rzeczywiste;
    rzeczywiste x,y,z;  //zamiast float x,y,z;
np. typedef struct wyborca str_wyb;
    str_wyb w1,w2;// zamiast struct wyborca w1, w2;
```

INSTRUKCJE

1) Instrukcja PUSTA ;

2) Instrukcja PRZYPISANIA (podstawienia) (=)

```
np. float x,y;
          y=5;
          x=0.5*3.1415;
          y=y+x;
```

Instrukcja przypisania (podstawienia) podstawia wartość wyrażenia stojącego po prawej stronie do zmiennej stojącej po lewej stronie.

```
np. float x,y,z;
    y=5;
    3=x; // BARDZO ŹLE!!!
    y=x; // bez sensu bo x ma wartość losową
    z=y+1;
    x+y=5; //ŹLE
    x+1=6; //ŹLE
    x=y=5; // ⇔ x = (y = 5)
```

3) Instrukcja ZŁOŻONA (blok instrukcji)

```
{ I1;
  I2;
   ⋮
  In;
}
```

4) Instrukcja WARUNKOWA

- KRÓTKA
  if (warunek) I_1;
  Jeśli warunek jest prawdziwy (ma wartość różną od 0), to wykonywana jest I_1 (a w przeciwnym razie instrukcja I_1 jest pomijana).

```
np. int x,y,z;
        x=...
        y=...
    if(x!=0) z=y/x; //⇔ if(x) z=y/x
```

- DŁUGA

```
if(warunek) I1;
   else I2;
```

  Jeśli warunek jest prawdziwy (różny od 0), to wykonywana jest I_1 a jeśli warunek nie jest prawdziwy (równy 0), to wykonywana jest instrukcja I_2.

```
np. if(x) z=y/x;
      else printf("Nie dzieli się przez 0")
```

Uwaga! Jeśli y=5 a x=2, to z=2 (a nie 2.5) // y/x - dzielenie całkowite, gdy y i x są typu całkowitego (int)

Instrukcje warunkowe można zagnieżdżać (wielokrotnie) jedna wewnątrz drugiej:

```
if(w1) I1;
  else
   if(w2) I2;
    else
     if (w3) I3;
      else I4;
```

Przykład. Wczytać ciąg *n* liczb całkowitych i obliczyć sumę tych liczb. Narysować schemat blokowy algorytmu.

5) Instrukcje ITERACYJNE

a/ while (warunek) I;
//dopóki warunek jest prawdziwy wykonywana jest instrukcja I

b/ do I while (warunek);

c/ for (wyr1; wyr2; wyr3) I;

Najczęstsza postać pętli for:

for(inicjalizacja zmiennej sterującej; warunek końcowy na zmienną sterującą; zmiana wartości zmiennej sterującej) {I_1; ..., I_n;}

Przykład. Wczytać ciąg *n* liczb całkowitych i obliczyć sumę tych liczb.

```
main()
{int n, tab[20], i, suma=0;
printf("Podaj liczbę elementów ciągu:");
 scanf("%d",&n);

  for(i=0; i<n; i++) //i++; ⇔ i=i+1;
  {printf("tab[%d]=" ,i); //tab[0]=
   scanf("%d",&tab[i]);
  }

  for(i=0; i<n; i++)
     suma=suma+tab[i];
  printf("suma=%d",suma);
}
```

Przykład. Sprawdzanie czy liczba naturalna jest pierwsza. Narysować najpierw schemat blokowy algorytmu.

Wersja 1.

```
main()
{ int n, i;
  int licznik=0; //licznik dzielników
printf("podaj n (>1): ");
scanf("%d",&n);
i=2;
while(i<n))
  {if (n%i==0) licznik++;
   i++;
  }
 if (licznik==0) printf("Liczba %d jest pierwsza.", n);
    else printf("Liczba %d nie jest pierwsza.", n);
}
```

Wersja 2.

```
main()
{ int n, i;
 int pierwsza=1; //zmienna "logiczna"
printf("podaj n (>1): ");
scanf("%d",&n);
i=2;
while(i<=sqrt(n))
  {if (n%i==0) pierwsza=0;
   i++;
  }
 if (pierwsza==1) printf("Liczba %d jest pierwsza.", n);
    else printf("Liczba %d nie jest pierwsza.", n);
}
```

Oczywiście, zamiast

```
if (pierwsza==1) ...    można napisać   if(pierwsza) ...
```

i zamiast

```
while(i<=sqrt(n))   while(i<=sqrt(n) && pierwsza==1)
```

6) Instrukcje SKOKU

- **goto** nazwa_etykiety;
  nazwa_etykiety: I; //nie używamy
- **break** - przerywa działanie aktualnie wykonywanego bloku programu i przechodzi do pierwszej instrukcji poza tym blokiem. Najczęściej używany w pętlach, instrukcji wyboru (switch), ...
- **continue** - przerywa wykonywanie aktualnego kroku w pętli i przechodzi do sprawdzenia warunku w **while** lub war2 w **for**.

  Przykład. Sprawdzanie czy liczba naturalna jest pierwsza.

  Wersja 3.

```
main()
{ int n, i;
printf("podaj n: ");
scanf("%d",&n);
i=2;
while(i<=sqrt(n))
  {if (n%i==0) break;
   i++;
  }
 if (i>sqrt(n)) printf("Liczba %d jest pierwsza.", n);
    else pritf("Liczba %d nie jest pierwsza.", n);
}
```

- **return** (**return** lub **return** wartosc lub **return** (wartosc)). Używa się wewnątrz funkcji. Powoduje przekazanie sterowania do punktu, w którym wywołano funkcję. Funkcja zwraca wtedy wyrażenie "wartosc"(jako jej wartość). Wartość wyrażenia staje się wtedy wartością funkcji.

Przykład. Sprawdzanie czy liczba naturalna jest pierwsza.

Wersja 4.

```
int pierwsza(int n) //nagłowek funkcji
 {int i;
  for (i=2; i<=sqrt(n); i++)
   if(n%i==0) return 0;

  return 1;
 }
main()
{int n;
 printf("podaj n: ");
 scanf("%d",&n);
 if (pierwsza(n)) // lub if(pierwsza(n)==1)
    printf("Liczba %d jest pierwsza.", n);
  else printf("Liczba %d nie jest pierwsza.", n);
}
```

7) Instrukcja WYBORU (przełącznik)

```
switch (wyrazenie)
{case wyr_stale1: I1; [break;]
 case wyr_stale2: I2; [break;]
  ⋮
 case wyr_stalen: In; [break;]
 default: In+1;
};
```

Wyrażenie jest typu całkowitego lub znakowego.

```
np. int nr_dnia;
     ...
    switch(nr_dnia)
    {case 1: printf("Poniedziałek"); break;
     case 2: printf("Wtorek"); break;
      ⋮
     case 6: printf("Sobota"); break;
     case 7: printf("Niedziela"); break;
     default: printf("???");
    };
```

Przykład. Zamiana polskich liter (ze znakami diakrytycznymi) na litery bez znaków diakrytycznych.

```
   char znak
    ...
    switch(znak)
    {case 'ą': printf("%c", 'a'); break;
     case 'Ą': printf("%c", 'A'); break;
      ⋮
     case 'ż': printf("%c", 'z'); break;
     case 'Ż': printf("%c", 'Z'); break;
     default: printf(znak);
    };
```

Przykład. Rozkład liczby naturalnej na czynniki pierwsze.

Program A1

```
main()
{int n, i;
scanf("%d",&n);
i=2;
while(n>1)
 if(n%i==0)
  {printf("%d*",i);
   n=n/i;
  }
  else i++;
}
```

Program A2

```
{...
  while(n%2==0)
   {printf("2*");
    n=n/2
   }
  i=3;
  while ((n>1)&&(i<=sqrt(n)))
```

```
    if (n%i==0)
     {printf("%d*",i);
      n=n/i;
     }
     else  i=i+2;
     if(n>1)  printf("%d",n);
}
```

liczba cyfr liczby n	A1	A2
6	0,07s	0,0s
9	1m 7s	0,01s
12	12h 27m	0,13s
14	37d 12h	1,23s

Przykład częstego błędu.

```
main()
{int i,x=2,y=3;
  for(i=1;i<5;i++)
    x=2*x;
    y=3*y;  //(*)
  printf("x=%d,  y=%d",  x,y);
}
```

Co wypisze ten program?
2*2*2*2*2=32 x=32
3*3=9 y=9
Instrukcja (*) nie jest w pętli. Jeśli miałaby być w pętli należy dopisać nawiasy {}.

OPERATORY

1) ARYTMETYCZNE

- JEDNOARGUMENTOWE
  - `-` zmienia znak na przeciwny
  - `++` zwiększa wartość liczby całkowitej o 1 (inkrementacja)
  - `--` zmniejsza wartość liczby całkowitej o 1 (dekrementacja)

  Przykład

```
i++;  //  i=i+1;
--i;  //  i=i-1;
x=i++;  //x=--i;  ???
```

- DWUARGUMENTOWE
  - `+,-,*`
  - `/` dzielenie (całkowite, gdy argumenty są typu całkowitego)
  - `%` modulo - reszta z dzielenia liczb całkowitych

2) RELACYJNE

`==` porównanie
`!=` nieprawda, że równe
`<, <=, >, >=`

3) LOGICZNE

`!` negacja
`&&` koniunkcja
`||` alternatywa

4) BINARNE

<< , >> , ~, & , | , ^

5) ADRESOWE

& udostępnia adres zmiennej

* udostępnia wartość zmiennej na podstawie jej adresu

DODATKI

1) Rozszerzone operatory przypisania (podstawienia)

```
x op=y ⇔ x=x op y, gdzie op∈{*,/,%,+,-,...}
Przykład.
x+=2; //x=x+2
x=*5; //ŹLE
```

2) Operator warunkowy ?

```
wyrażenie_warunkowe ? wyr1 : wyr2;
(⇔ if(wyrażenie_warunkowe)
   wyr1;
    else
      wyr 2; )

np. x=(x>=0) ? x : -x; //x=|x|
```

3) Operator **sizeof** - udostępnia liczbę bajtów składających się na operand dla którego został wywołany.

```
sizeof operand; lub sizeof(operand);

\ \\
Przykład.
sizeof int;
sizeof(str_wyb);
```

HIERARCHIA OPERATORÓW

W języku C istnieją ściśle określone reguły kolejności w jakiej są wykonywane obliczenia (operatory) w wyrażeniu zawierającym kilka operatorów.
3 reguły:

1) Jeśli operatory mają różny priorytet, to wówczas pierwszy wykonywany jest operator o wyższym priorytecie.

2) Jeśli różne operatory mają równe priorytety, to wówczas wykonywane są w kolejności od lewej do prawej strony. Jeśli operatory są takie same, to wówczas o kolejności ich wykonywania decyduje to czy operator jest lewo- czy prawostronnie łączny.

3) Naturalną kolejność (czyli zgodną z regułami 1) i 2)) wykonywania operacji można zmienić poprzez zamykanie fragmentów wyrażenia w pary nawiasów okrągłych (). Wyrażenie zamknięte największą liczbą nawiasów jest wykonywane w pierwszej kolejności.

PRIORYTETY OPERATORÓW

najwyższy	łączność
1) (), [], →, .	L
2) !, ~, ++, - -, -, +, *, &, sizeof, (typ)	P
3) *, /, %	L
4) +, -	L
5) <<, >> - binarne	L
6) <, <=, >, >=	L
7) ==, !=	L
8) & - binarny	L
9) ^ - binarny	L
10)| - binarny	L
11) &&	L
12) ||	L
13)?:	P
14)=, +=, -=, *=, /=, ...	P
najniższy	

FUNKCJE

Funkcją nazywamy wyodrębnioną część programu stanowiącą pewną całość, posiadającą nazwę i ustalony sposób wymiany informacji z pozostałymi częściami programu.
Funkcje stosowane są do wykonywania pewnych czynności, które wykorzystywane mogą być:

- w różnych programach,
- wielokrotnie w tym samym programie.

Często program dzielimy na kilka funkcji, z których każdą pisze inny programista, a następnie składamy razem w jedną całość.
Użycie funkcji umożliwia również bardziej **efektywne wykorzystywanie pamięci** operacyjnej, gdyż obiektom określonym (zadeklarowanym) wewnątrz funkcji pamięć przydzielana jest dopiero z chwili wywołania funkcji.

DEFINICJA FUNKCJI

typ_zwracanej_wartości nazwa_funkcji (lista_parametrów_funkcji) // NAGŁÓWEK funkcji
{ wnętrze funkcji (ciało funkcji) - ciąg deklaracji zmiennych i ciąg instrukcji
return lub **return** wartość lub **return**(wartość);
}

Przykład.

```
float max(float a, float b) //nie piszemy max(float a,b)
 {if(a>=b) return a;
  else return b;
 }
```

Przykład.

```
int silnia(int n) //wersja iteracyjna
  {int s,i;
   s=1;
   for(i=1;i<=n;i++)
    s=s*i;
   return s;
  }
```

Przykład

```
  float najmniejszy(float A[100], int n)
   {int i;
    float min=A[0];
    for(i=1;i<n;i++)
      if(A[i]<min)
        min=A[i];
```

```
    return min;
  }
```

Typ zwracanej wartości może być dowolnym **prostym (!!!) typem** danych (np. **int, float, char, void, ...** ...) (czyli **NIE** może być np. tablicą, ciągiem znaków czy strukturą).
Lista parametrów to lista nazw zmiennych oraz ich typów (rozdzielonych przecinkami), które funkcja otrzymuje jako argumenty w momencie jej wywołania (uruchomienia).
Funkcja może nie mieć żadnych parametrów i wtedy piszemy: ... **f()** lub ... **f(void)**.
Jeżeli funkcja nie zwraca żadnej wartości, to piszemy **void f**(...).

void - typ pusty

W języku C jeśli przed nazwą funkcji nie ma wpisanego żadnego typu (tzn. f(...)) to oznacza, że funkcja zwraca typ **int**.

WYWOŁANIE FUNKCJI

```
Przykład.
 float max(float a, float b)
 {
  ⋮
 }

 main()
 {float najwiekszy;
 float c,d;
 c=5;
 d=3;
 najwiekszy=max(c,d); //wywołanie funkcji max() z parametrami c, d
 printf("%f", najwiekszy);
 //lub prościej:  printf("%f", max(c,d));
 }

Przykład.
 int symbolNewtona(int n, int k)
 {int silnia(int ); //deklaracja funkcji silnia() (zwana PROTOTYPEM funkcji)
                    //musi tu być ponieważ funkcja silnia() jest zdefiniowana
                    //później niż funkcja symbolNewtona()

 return silnia(n)/(silnia(k)*silnia(n-k)); //trzykrotne wywołanie funkcji silnia()
 }
 int silnia (int n)
 {
  ⋮
 }
```

ZASIĘG FUNKCJI

Każda funkcja to osobny blok kodu. Kod funkcji to jej prywatna własność i żadna inna funkcja nie może się odwołać bezpośrednio do jej kodu inaczej niż przez wywołanie jej. Kod, z którego składa się wnętrze funkcji jest ukryty przed resztą programu (chyba, że wewnątrz funkcji korzystamy korzystamy ze zmiennych globalnych).

Wniosek. **Nie można zdefiniować jednej funkcji wewnątrz innej funkcji.**

Zmienne zdefiniowane wewnątrz funkcji to zmienne lokalne. Zmienna lokalna powstaje po rozpoczęciu działania funkcji, a gdy funkcja kończy pracę, to taka zmienna przestaje istnieć.

Zmienne występujące w nagłówku funkcji (a konkretnie wewnątrz nawiasów) nazywamy parametrami (formalnymi) funkcji.

Przykład.

```
float potega(float x, int n) // x oraz n są parametrami formalnymi funkcji potega()
  {float y=1; int i;         // y oraz i są zmiennymi lokalnymi funkcji potega()
   for (i=1;i<=n;i++)
     y=y*x; //lub prościej y*=x
   return y;
  }
```

Wywołanie funkcji potega() w programie:

```
float b;       // b jest zmienną globalną w programie (dopuszczalne ale lepiej
               // gdyby była zdefiniowana jako zmienna lokalna funkcji main())
main()
   {float a;  // zmienne a oraz n są zmiennymi lokalnymi funkcji main()
    int n;
    a=2;
    n=10;
    b=potega(a,n);
   }
```

Przekazywanie argumentów do funkcji przez wartość

W tej metodzie wartości argumentów (o ile są prostym typem danych) są kopiowane do parametrów formalnych funkcji. Zmiany wartości parametrów formalnych wewnątrz funkcji nie mają wpływu na wartość argumentów.

Przykład.

```
void WczytajCiag(int n, int A[100])
  {int i;
   printf("Podaj dlugosc ciagu:");
   scanf("%d",&n);
   printf("Podaj kolejne elementy ciagu \n");
   for (i=0;i<n;i++)
     {printf("A[%d]=",i);
      scanf("%d",&A[i]);
     }
  }

main()
{int n, A[100];
WczytajCiag(n,A);
WypiszCiag(n,A);
}
```

Okazuje się, że nawet jak poprawnie wpiszemy ciąg, to funkcja WypiszCiag() nie zadziała tak ja powinna. Argument n wcale nie będzie równy takiej wartości jaką wpisaliśmy wczytując ciąg ponieważ tę wartość wczytaliśmy pod zmienną n, która jest parametrem formalnym funkcji WczytajCiag(). Zmienna n z funkcji main() i zmienna n z funkcji WczytajCiag() to dwie różne zmienne. Po wykonaniu WczytajCiag() zmienna n ma taką samą losową wartość jak zaraz po jej zadeklarowaniu w main().

Przykład.

```
main()
{int sqr(int);
 int n, n2;
 n=5;
 n2=sqr(n);
 printf("%d,%d",n,n2); //5,25 bo funkcja sqr() podstawia pod wartość k wartość
                       //zmiennej n (czyli 5) i dalsze obliczenia wykonuje
                       //na zmiennej k (a nie na zmiennej n)
}
int sqr(int n)
{n=n*n;
 return n;
}
```

FUNKCJE REKURENCYJNE

Z rekurencją (rekursją) mamy do czynienia, gdy w funkcji następuje bezpośrednie lub pośrednie wywołanie tej samej funkcji.

REKURSJA JAWNA - gdy funkcja wywołuje samą siebie.

REKURSJA NIEJAWNA - gdy funkcja wywołuje inną funkcję, która z kolei wywołuje tę pierwszą funkcję.

Przykład.

$$n!=\begin{cases} 1, & n=0 \\ (n-1)!\cdot n, & n>0 \end{cases}$$

```
int silnia2(int n)  // wersja rekurencyjna
  {if(n==0) return 1;
   else return silnia2(n-1)*n; // funkcja silnia2() w jawny sposób wywołuje samą siebie
                               //ale już dla innego parametru
  }
```

Przykład. Ciąg Fibonacci'ego.

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \forall n \geqslant 2 \end{cases}$$

```
  int F(int n)            //wersja rekurencyjna
    { if(n<=1) return 1;
      else return F(n-1)+F(n-2);
    }
```

Liczba wywołań funkcji $F(n)$ dla zadanego n jest proporcjonalna do: $\left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1,6^n$.

czyli np. dla n=50 liczba wywołań będzie $1,6^{50} > 2^{25} = (2^{10})^2 \cdot 2^5 > 1000^2 \cdot 2^5 = 32$mln

Przykład. Wersja iteracyjna funkcji $F(n)$.

```
int F2(int n)
{
int F[51];
int i;

F[0]=1;
F[1]=1;

if (n<=1) return F[n];
 else
   { for (i=2; i<=n; i++)
     F[i]=F[i-1]+F[i-2];

     return F[n];
   }
}
```

Przykład. Obliczanie największego wspólnego dzielnika dwóch liczb naturalnych.
Wersja najprostsza.

```
int NWD1(int a, int b)
{int i, d=1;
 for(i=2; i<=a; i++)
  if((a%i==0)&&(b%i==0)) d=i;

  return d;
}
```

Algorytm Euklidesa

Przykład. a=168, b=70.
a=168-70=98, b=70;
a=98-70=28, b=70;
a=28, b=70-28=42;
a=28, b=42-28=14;
a=28-14=14, b=14.
NWD(168,70)=14.

Wersja iteracyjna algorytmu Euklidesa

```
int NWD2(int a, int b)
{ while(a!=b)
    if (a>b) a=a-b;
     else b=b-a;

  return a;
}
```

Wersja rekurencyjna algorytmu Euklidesa

$$NWD(a,b) = \begin{cases} a \text{ gdy } a = b \\ NWD(a-b,b) \text{ gdy } a > b \\ NWD(a,b-a) \text{ gdy } a < b. \end{cases}$$

```
int NWD3(int a, int b)
{ if (a==b) return a;
  else
   if (a>b) return NWD3(a-b,b);
     else return NWD3(a,b-a);
}
```

Ćw. Napisać wersję rekurencyjną NWD z wykorzystaniem operacji modulo (%), która jeszcze bardziej przyspiesza działanie algorytmu.

Przykład.
Generowanie wszystkich ciągów k-elementowych o elementach ze zbioru $\{1,\ldots,n\}$ $(k \leqslant 10)$.

```
int k, n;
int P[11];    //P[1], ..., P[k]
void wariacja (int l)  //l-indeks elementu,
{ int i,j;             // któremu nadajemy wartość
  if(l > k)
    {for (i=1; i<k; i++)
     printf("%d, ", P[i]);

     printf("%d\n", P[k]);
    }
  else
    {for (j=1; j<=n; j++)
      {P[l]=j;
       wariacja (l+1);
      }
    }
}
```

Wywołanie w programie głównym: wariacja (1)

Co wypisze program dla $k = 3, n = 2$?

1, 1, 1
1, 1, 2
1, 2, 1
1, 2, 2
2, 1, 1
2, 1, 2
2, 2, 1
2, 2, 2

Tablice wielowymiarowe

```
int tab[100],  //tablice jednowymiarowe
int tab2[]; //można nie podać rozmiaru tablicy

typ nazwa_tablicy [][rozmiar2[[rozmiar3]...
                   \(tu może być podany rozmiar1,
                      ale nie musi
                      za to rozmiar2 i ewentualne
                      następne muszą być podane)
int tab[3][4];
int tab2[][3];
int A[3][2]={{1,2},{3,4},{5,6}}; //(*)
int B[2][3]={1,2,3,4};           //(**)
```

(*) oznacza, że $A[0][0] = 1, A[0][1] = 2, A[1][0] = 3, \ldots, A[2][1] = 6$
(**) oznacza, że $B[0][0] = 1, B[0][1] = 2, B[0][2] = 3, B[1][0] = 4$ a pozostałe wartości są nieokreślone (mają losową wartość)

Przykład. Sprawdzanie, czy macierz jest symetryczna.

```
int Symetryczna(int n, int m, float A[10][10])
{ int i,j;
  if (n!=m) return 0;

  for(i=0;i<n;i++)          // pętla zewnętrzna
   {
   for(j=0;j<m; j++)        //pętla wewnętrzna
     {
      if (A[i][j]!=A[j][i]) return 0;
     }
   }

  return 1;
}
```

Przykład.
Dana jest szachownica o wymiarach n × n i skoczek znajdujący się na jednym z pól. Sprawdzić, czy skoczek może przejść po wszystkich polach szachownicy tak, aby na każdym polu stanął dokładnie jeden raz.

```
#include <dos.h>      //gotoxy()

int szach[10][10]; //szach[i][j]=k, gdy skoczek
                   // na polu (i,j) w k-tym skoku
int n=8; //rozmiar szachownicy
int l_skok=0; //liczba wykonanych skoków
int max=n*n; //maksymalna liczba skoków na szachownicy
             // o wymiarze n x n
```

```
void main()
{ int xp=1, yp=1;// początkowe ustawienie skoczka
  void skacz(int, int);
  void rysuj_rozwiazanie();
  skacz(xp, yp);
  if(l_skok == max) rysuj_rozwiazanie();
  else printf("Nie ma rozwiązania");
}

void rysuj_rozwiazanie()
{ int i,j;
  for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
    {
      gotoxy(3*i, 2*j);
      printf("%d", szach[i][j]);
    }
}

void skacz (int x, int y)
{ int i,j;
  l_skok++;
  szach[x][y] = l_skok;
  if(l_skok<max)
    { for(i=-2; i<=2; i++)
      for(j=-2; j<=2; j++)
        if(abs(i*j)==2) //czy to jest ruch skoczka
          if(x+i>=1 && x+i<=n && y+j>=1 && y+j<=n)
                //czy nie wyskoczyliśmy poza szachownicę
            if(szach[x+i][y+j]==0) //czy na tym polu
                                   //już nie byliśmy
              skacz(x+i, y+j); //wykonujemy kolejny skok
      if(l_skok<max)
      { szach[x][y]=0; //cofamy się o jeden skok
        l_skok--;
      }
    }
}
```

WSKAŹNIKI

Każda zmienna ma unikalny adres wskazujący początkowy obszar pamięci zajmowany przez tę zmienną. **Wskaźnik** (na zmienną danego typu) jest to specjalny rodzaj zmiennej, która **przechowuje adres zmiennej** (danego typu). Oznacza to, że wskaźnik **wskazuje** miejsce w pamięci gdzie zapisana jest zmienna danego typu.

Deklaracja wskaźnika (zmiennej wskaźnikowej)

typ_zmiennej *nazwa_zmiennej_wskaźnikowej;

np.

int *p; //oznacza, że zmienna wskaźnikowa p będzie wskazywać na zmienną typu int

Operatory związane ze wskaźnikami

* - operator **wyłuskania** - zwraca wartość przechowywaną pod adresem, który poprzedza

& - operator adresu - zwraca adres zmiennej, którą poprzedza

Przykład.

```
main()
  {int x;
   int *p; //deklaracja wskaźnika p
   x=1;    //odwolanie jawne do zmiennej x
   p=&x;   //inicjalizacja wskaźnika p
   *p=2;   //odwołanie niejawne do zmiennej x
           // bo poprzez wskaźnik p
   printf("x=%d",x);// x=?
  }
```

Uwaga 1.

```
  int *p;  \     //tu * oznacza, że deklarujemy wskaźnik
            te gwiazdki znaczą coś innego
  *p = 10; /    // tutaj * jest operatorem wyłuskania
```

Uwaga 2.

```
main()
  {int x=2;
   int *p=x;      // ŹLE

   printf("x=%d",*p);// *p=?
  }
```

Poprawna wersja.

```
main()
  {int x=2;
   int *p=&x;      // OK

   printf("*p=%d",*p);// *p=?
  }
```

Przykład nieprawidłowego użycia wskaźnika.

```
main()
  {
   int *p;
   *p=3;
   printf("*p=%d",*p);
  }
```

Przykład.

```
 void wczytaj(int n)
  {printf("Podaj n");
   scanf("%d",&n);
  }

  main()
  {int n;
   wczytaj(n);
   printf("n=%d",n); //n=?
  }
```

Przekazywanie argumentów funkcji przez wskaźniki

Wersja poprawna (z użyciem wskaźnika jako parametru funkcji).

```
void wczytaj(int *n)
{
 printf("Podaj n");
 scanf("%d",n); //brak AMPERSANDA
                //bo n jest wskaźnikiem
}

main()
{int n;
 wczytaj(&n); // AMPERSAND przed zmienną!!!
 printf("n=%d",n); //n=?
}
```

Przykład. Zamiana wartości dwóch zmiennych.

```
void zamiana(int a, int b)
{ int c;
  c=a;
  a=b;
  b=c;
}
 main()
{int a, b;
 a=1;
 b=2;
 zamiana(a,b);
 printf("a=%d, b=%d",a, b); //a=?, b=?
}
```

Wersja poprawna (z użyciem wskaźników jako parametrów funkcji).

```
void zamiana(int *a, int *b)
{ int c;
  c=*a;
  *a=*b;
  *b=c;
}
 main()
{int a, b;
 a=1;
 b=2;
 zamiana(&a,&b); //AMPERSANDY!!!
 printf("a=%d, b=%d",a, b); //a=?, b=?
}
```

Użycie wskaźników z tablicami

Nazwa tablicy jest wskaźnikiem wskazującym na początek tablicy.

Przykład.

```
int tab[10];
tab[0]=3;
printf("%d", *tab); //co wypisze?
```

Przykład.

```
int tab[10];
```

```
int *wsk;
wsk=tab;
*wsk=5;
printf("%d", tab[0]); //co wypisze?
```

ARYTMETYKA WSKAŹNIKÓW

Wskaźników można używać podobnie jak innych zwykłych zmiennych, ale są pewne ograniczenia.
Oprócz operatorów & i * są tylko 4 operatory arytmetyczne, które można stosować ze wskaźnikami. Są to: +, ++, -, - -.
Co więcej dodawać i odejmować można tylko liczby całkowite.
Arytmetyka wskaźników różni się również tym od zwykłej arytmetyki (na liczbach), że jest wykonywana względem typu podstawowego wskaźnika.

Przykład.

```
int *wsk;
⋮
wsk++;  // przesuwamy się nie o 1 bit,
        //  nie o jeden bajt
        //  ale o tyle bajtów ile zajmuje
        //  jedna liczba typu int

double *p;
⋮
p=p+3; // przesuwamy się o tyle bajtów ile
       // zajmują trzy liczby typu double
```

Przykład. Wersja 1 sumowania liczb w tablicy.

```
float Suma(int n, float A[1000000])
{ int i;
  float s=0;
  for (i=0; i<n; i++)
      s=s+tab[i];

 return s;
}
```

Przykład. Wersja 2 sumowania liczb w tablicy.

```
 float Suma(int n, float A[1000000])
{ int *wsk;
  float s=0;
  wsk=tab;
  while(wsk<=&tab[n-1])
  { s=s+*wsk;
      wsk++;
  }
 return s;
}
```

Przykład. Wersja 3 sumowania liczb w tablicy.

```
float Suma(int n, float A[1000000])
{ int i;
  float s=0;
  for(i=0; i<n;i++)
    s=s+*(tab+i);

 return s;
}
```

Uwaga! Operacja dodawania i odejmowania na wskaźnikach nie są sprawdzane przez kompilator i możemy przesunąć wskaźnik poza zadeklarowany obszar tablicy i zniszczyć inne istniejące (i ważne) dane.

TABLICE DYNAMICZNE

Są tworzone dynamicznie (na bieżąco) podczas działania programu i wymagają ręcznego przydziału i zwalniania pamięci.
Przykład.

```
int *tab;
printf("Podaj długość ciągu: );
scanf("%d", &n);
tab=malloc (n*sizeof(int));
//lub tab=(int *) malloc(n*sizeof(int));

if (tab!=NULL)
 {tab[0]=1; // lub    *tab=1;
  tab[n-1]=n; // lub    *(tab+n-1)=n;

  free(tab); //zwalnia pamięć
  }
```

Uwaga.
Jeśli funkcja malloc() zwróci NULL (wskaźnik pusty), to oznacza, że nie udało się przydzielić pamięci na tablicę.

ŁAŃCUCHY ZNAKÓW (tablice znaków, napisy, stringi)

Przykład.

```
char slowo1[30]="mama";
char *slowo2="tata";

printf("%s",slowo1);
printf("Podaj słowo: ");
scanf("%s",slowo2) // tu nie ma i nie
                   //powinno być ampersanda
```

Biblioteka <string.h>

- strlen(slowo) - zwraca długość słowa //strlen("mama") jest równe 4
- strcpy(slowo1, slowo2) - kopiuje slowo2 do slowo1
- strcmp(slowo1, slowo2) =
  $\begin{cases} 0 & \text{jeśli slowo1 i slowo 2 są takie same} \\ - & \text{jeśli slowo1 występuje wcześniej w porządku leksykograficznym niż slowo2} \\ + & \text{jw, z tą różnicą, że slowo1 występuje później niż slowo2} \end{cases}$

Przykład. Napisać funkcję, która dla dwóch zadanych słów sprawdza, czy są one swoimi anagramami.

```
int LWLWS(char litera, char slowo[30])// liczba wystąpień
{int LW=0;                            // litery w słowie
 int DlugoscSlowa=strlen(slowo);
 for(i=0;i<DlugoscSlowa;i++)
 if(slowo[i]==litera)LW++;
 return LW;
}

int Anagram(char slowo1[30], char slowo2[30])
{int i;
if(strlen(slowo1)!=strlen(slowo2))return 0;

for (i=0; i<strlen(slowo1); i++)
if (LWLWS(slowo1[i],slowo1)!= LWLWS(slowo1[i],slowo2))
```

```
        return 0;

return 1;
}
```

STANDARDOWE WEJŚCIE/WYJŚCIE PROGRAMU

Biblioteka <stdio.h> zawiera podstawowe funkcje wykonujące znakowe operacje wejścia/wyjścia.

I Funkcje tworzenia i interpretowania sformatowanych ciągów znaków:
int printf(const *format, arg1, arg2,..., argn)

int scanf(const *format, &arg1, arg2,..., &argn)

Format (łańcuch sterujący) składa z trzech rodzajów obiektów:

- znaki, które należy wypisać na ekranie (konsoli) (np. "Podaj n: "),

- specyfikatory formatu, określające sposób wypisywania argumentów,

- znaki specjalne.

Format:

%[flaga][szerokość][.precyzja][modyfikator] **typ**

typ:
- d - liczba całkowita w postaci dziesiętnej
- c - argument jest znakiem (np. 'a')
- s - argument jest ciągiem znaków (np. "mama")
- f - liczba zmiennoprzecinkowa z kropką dziesiętną (np. 3.1415)
- e - liczba zmiennoprzecinkowa w notacji wykładniczej z e (np. 3.14e10)
- p - argument jest wskaźnikiem

flaga:

- oznacza wyrównanie do lewej
+ oznacza wypisanie znaku liczby (np. +1 zamiast po prostu 1)
modyfikator:

l - liczba w postaci długiej (long (int) lub double);

Przykład.

```
%8.2f
szerokość=8 //liczba zapisana na polu
            //o co najmniej 8 znakach
precyzja=2  //liczba zapisana z dokładnością
            //do 2 miejsc po przecinku

printf("%8.2f",123); //_ _ 123.00
```

ZNAKI SPECJALNE

\n - przejście do nowej lini
\a - dzwonek
\b - backspace (usuwa znak z lewej strony kursora)
\r - powrót na początek linii
\t - tabulator poziomy
\r - tabulator pionowy
\\ - wypisuje \ (backslash)
\' - wypisuje ' (apostrof)
\"- wypisuje "(cudzysłów)

II Inne funkcje wykonujące znakowe operacje wejścia/wyjścia:

int putchar(char ch) - funkcja wypisania jednego znaku (na ekranie)

char getchar() - funkcja wczytania jednego znaku(z klawiatury, po wciśnięciu enter)

char *gets(char *napis) - funkcja wczytuje znaki z klawiatury aż do momentu wciśnięcia enter

int puts(char *napis) - funkcja wypisuje napis i zwraca niezerową wartość gdy wykonanie funkcji zakończy się powodzeniem i zwraca **EOF** w przeciwnym przypadku

W bibliotece <conio.h> znajdują się jeszcze funkcje:

char getche() - funkcja wczytania jednego znaku (z klawiatury, bez wciśnięcia enter) z natychmiastowym wypisaniem go na ekranie

char getch() - funkcja wczytania jednego znaku (z klawiatury, bez wciśnięcia enter) ale NIE wypisuje go na ekranie

int kbhit() - zwraca wartość niezerową jeśli został wciśnięty jakiś znak i zwraca 0 w przeciwnym przypadku

PLIKI

Plik to pozbawiony konkretnej struktury zbiór znaków i może być zarówno plikiem dyskowym, jak i urządzeniem zewnętrznym (np. ekran, klawiatura, drukarka, ...).

Korzystanie z pliku z poziomu języka C odbywa się poprzez specjalną strukturę **FILE** (z biblioteki <stdio.h>), do której dostęp uzyskujemy za pomocą wskaźnika do tej struktury.

Definicja zmiennej plikowej

```
FILE *nazwa_zmiennej_plikowej;
```

Nazwa zmiennej plikowej jest wskaźnikiem do pewnej struktury zawierającej informacje o pliku (nazwa, lokalizacja, tryb otwarcia, wielkość, położenie bufora, ...) potrzebne do komunikacji z fizycznym plikiem w pamięci zewnętrznej.

Otwarcie pliku

```
FILE *fopen(char*nazwa, char*tryb)
```

```
np. FILE*f;
    f=fopen("zbiór.txt","w+");
```

Otwarcie pliku oznacza powiązanie fizycznego pliku (zbiór) ze zmienną plikową.
Funkcja fopen() zwraca wskaźnik do pliku chyba, że nie uda się otworzyć pliku, to wówczas zwraca wartość NULL.

Tryby otwarcia

Pliki tekstowe:
r otwieranie pliku do czytania
w otwieranie pliku do pisania (jeśli pliku nie ma, to tworzy go, a jeśli jest to kasuje go i tworzy nowy)
a otwieranie pliku do dopisywania na końcu pliku
r+ do odczytu i pisania
w+ do pisania, ale gdy plik nie istnieje, to tworzy nowy (ten) plik
a+ do dopisywania (jeśli pliku nie ma, to tworzy nowy (ten) plik)

Pliki binarne:
rb
wb
ab
r+b(rb+) j.w.
w+b(wb+)
a+b(ab+)

Zamykanie pliku

```
fclose(nazwa_zmiennej_plikowej);
np. fclose(f);
```

Jeśli operacja zamknięcia wykonana została poprawnie, to funkcja fclose() zwraca 0, a w przeciwnym razie zwraca EOF. Zamykanie pliku oznacza usunięcie powiązania zmiennej plikowej z fizycznym plikiem.

Czytanie z pliku jednego znaku

```
fgetc(zm_plikowa)
```

Wpisanie do pliku jednego znaku

```
fputc(znak, zm_plikowa)
```

Funkcja

```
feof(zm_plikowa)
```

zwraca wartość niezerową, jeśli w pliku został osiągnięty koniec pliku. W przeciwnym razie zwraca 0.

Przykład. Usuwanie polskich znaków diakrytycznych z pliku tekstowego.

```
main(){
  FILE *f1, *f2;
  char znak;
  f1=fopen("test_z_ogonkami.txt","r");
  f2=fopen("tekst_bez_ogonkow.txt","w+");

  while (feof(f1)==0{ //(!feof(f1))
    znak=fgetc(f1);
    switch(znak){
      case 'ą':fputc('a',f2); break;
      case 'Ą':fputc('A',f2); break;
       .
       .
       .
      case 'Ż': fputc('Z',f2); break;
      default: fputc(znak, f2);
    }
  }
```

```
    fclose(f1);
    fclose(f2);
  }
```

Plik tekstowy - plik ze znakami w kodzie ASCII podzielony na wiersze.

Można użyć wtedy funkcji:

```
  fprintf(zm_plikowa,...);
  fscanf(zm_plikowa),...) // to samo co w zwykłych funkcjach
                          // printf() ,scanf().
```

```
np. fprintf(f1,"%f",x);
    fscanf(f2,"%d",&a);
```

Przykład.
Załómy, że w zbiorze slownik.txt znajduje się zbiór słów, z których każde jest w innym wierszu.
ABAŻUR
ABECADŁO
.
.
.
ŻUR
ŻYWIEC
Zadanie. Zapisać w w nowym zbiorze palindromy.txt wszystkie słowa ze slownik.txt, które są palindromami.

```
int palindrom(char slowo[30]{
  int n=strlen(slowo);
  int i;                                    // [ K A J A K ]
                                               0       n-1
  for (i=0;i<n/2;i++)
    if slowo[i]!=slowo[n-1-i] return 0;
  return 1;
}

main(){
  FILE *f1,*f2;
  char slowo[30];
  f1=fopen("slownik.txt", "r");
  f2=fopen("palindromy.txt","w+");

  while(!feof(f1)){
    fsacnf(f1,"%s",slowo);
      if(palindrom(slowo)==1)
        fprintf(f2,"%s\n",slowo);
  }

  fclose(f1);
  fclose(f2);
}
```

Zadanie.
Zapisać w nowym zbiorze anagramy.txt wszystkie pary anagramów występujących w slownik.txt

np. KOT - TOK

slownik2.txt - kopia slownik.txt

```
int anagram(char slowo[30], char slowo2[30])
⋮ BYŁO

main(){
FIlE *f1,*f2,*f3;
char slowo1[30], slowo2[30];
f1=fopen("slownik.txt", "r");
f3=fopen("anagram.txt", "w+");
}

  while(!feof(f1)){
    fscanf(f1,"%s",slowo1);
    f2(fopen("slownik2.txt","r");
    while(!feof(f2)){
      fscanf(f2;"%s",slowo2);
      if(strcmp(slowo1,slowo2)<0 && anagram(slowo1,slowo2)==1)
      fprintf(f3,"%s %s\n",slowo1,slowo2);
    }
  fclose(f2);
  }
fclose(f1);
fclose(f3);
}
```

Pliki binarne

fread(& bufor, rozmiar, liczba_elementów, zm_plikowa)

fwrite(& bufor, rozmiar, liczba_ elementów, zm_plikowa)

Przykład.

```
  struct poborowy
    {char nazwisko [30];
     int wiek;
    };

   main()
   {FILE *f;
   f=fopen("BazaPoborowych.txt","wb+");
   struct poborowy pob;
   strcpy(pob.nazwisko,"Kowalski");
   pob.wiek=19;
   fwrite(&pob,sizeof(pob),1,f);
   fclose(f);
   }


   ⋮
   f=fopen("BazaPoborowych.txt","rb");
   fread(&pob,sizeof(pob),1,f);
   printf("Nazwisko: "%s, wiek: %d\n",pob.nazwisko,pob.wiek);
   fclose(f);
   }
```

Funkcje swobodnego dostępu do pliku

- long ftell (zm_plikowa) - zwraca pozycję bieżącą w pliku (wyrażoną w liczbie bajtów),
- int fseek (zm_plikowa, przesunięcie, początek) - powoduje przejście w pliku skojarzonym ze zm_plikową do pozycji o "przesunięcie"dalej (w bajtach) w stosunku do "początku", który może przyjąć wartość:
  SEEK_SET - początek pliku
  SEEK_CUR - bieżąca pozycja w pliku
  SEEK_END - koniec pliku

```
np. fseek (f,0,SEEK_END); //powoduje przejście na koniec pliku;
```

Funkcje systemu plików

- int remove (char *nazwa_pliku); - funkcja usuwa plik o zadanej nazwie
- int rename (char *stara_nazwa, char *nazwa_nazwa); - funkcja zmienia nazwę pliku ze star_nazwa na nowa_nazwa
- void rewind (zm_plikowa); - funkcja przewija (otwarty) plik związany ze zm_plikową i przesuwa jego bieżącą pozycję na początek pliku

Pliki (strumienie) standardowe

- standardowe wejście – **stdin** (klawiatura)
- standardowe wyjście – **stdout** (ekran monitora)
- standardowy strumień błędów – **stderr** (ekran monitora)

```
np. fprintf(stdout,"%d",liczba); //⇔ printf("%d",liczba)
```

Przekazywanie argumentów do funkcji main(...)

np.
kopiuj.exe tekst1.txt tekst1kopia.txt

usuńogonki.exe tekst2.txt

```
int main(int argc, char *argv[])
```

Dygresja. Często na końcu main() wpisujemy return 0. Jeśli system operacyjny otrzymuje 0 oznacza to, że program się poprawnie wykonał.

argc - określa liczbę parametrów występujących w wierszu poleceń (jego minimalna wartość to 1, bo nazwa_programu to też parametr).

argv - wskaźnik do tablicy wskaźników do znaku czyli tablicy, której elementami będą łańcuchy znaków (napisy). Każdy z tych łańcuchów to pojedynczy argument w wierszu poleceń.

Przykład.

```
np. kopiuj.exe tekst1.txt tekst1kopia.txt
```

argc=3;
argv[0] - wskazuje na "kopiuj.exe"
argv[1] - wskazuje na "tekst1.txt"
argv[2] - wskazuje na "tekst1kopia.txt"

argv[arcgc]=NULL,(='\0')

Przykład.
usuńogonki.exe inwokacja

```
int main(int argc, char *argv[])
 {FILE *f1,*f2;
 f1=fopen(argv[1],"r");
 strcpy(nazwawynikowego, argv[1]);
                  //nazwawynikowego = "inwokacja"
 strcat(nazwawynikowego, "bezogonkow");
                  //nazwawynikowego="inwokacjabezogonkow"
 f2=fopen(nazwawynikowego,"w+");
 :
 :(ćw.)
```

Dyrektywy preprocesora

1) `#include <nazwa.pliku>`
   `#include "nazwa_pliku"`

Dyrektywa #include instruuje kompilator (a właściwie jego część zwaną preprocesorem), że należy wczytać plik źródłowy (nagłówkowy, bibliotekę) o zadanej nazwie.

Jeśli #include<nazwa.pliku>, to plik nagłówkowy jest szukany w standardowych podkatalogach (utworzonych specjalnie do przechowywania takich plików, np. \include).

Jeśli #include"nazwa_pliku", to plik nagłówkowy jest szukany w katalogu bieżącym (a gdy tam go nie znajdzie, to jest przeszukiwany katalog z plikami nagłówkowymi, np. \include).

```
np. #include "mojpliknaglowkowy.h".
```

Jeśli sami stworzyliśmy plik nagłówkowy i nie jest on umieszczony w katalogu bieżącym, to oprócz samej nazwy pliku musimy podać również całą ścieżkę dostępu do tego pliku.

2) #define

Dyrektywa #define definiuje identyfikator jako ciąg znaków, które zostaną podstawione w każdym miejscu występowania identyfikatora w pliku źródłowym.

```
    #define identyfikator ciąg_znaków
np. #define TRUE 1
    #define FALSE 0
    #define MaxRozmiar 1000
```

Definiowanie tzw. makr "funkcjopodobnych"

```
np.         #define SREDNIA(a,b) (((a)+(b))/2)
            #define ABS(a)       (a)<0?-(a):(a)

gdyby       #define ILOCZYN(a,b)  a*b
to wtedy    ILOCZYN(2+3,4) daje   2+3*4=14    ŹLE!

dlatego     #define ILOCZYN(a,b)  (a)*(b)
i  wtedy    ILOCZYN(2+3,4) daje (2+3)*4=20
```

KOMENTARZE

- wielowierszowy

```
/* to jest
komentarz
3 - linijkowy*/
```

Uwaga. Nie wolno umieszczać komentarzy wewnątrz komentarzy (zagnieżdżać komentarzy).

- jednowierszowy

```
//to jest komentarz jednolinijkowy
```

KONWERSJA TYPÓW

Gdy w jednym wyrażeniu występują zmienne (i/lub stałe) różnych typów to wszystkie są przekształcane do jednego typu.
Automatycznie dokonywane są konwersje, które mają ewidentny sens.

char ⊂ int ⊂ long int ⊂ float ⊂ double

Przykład.

```
char ch;
int i;
float f;
double d;
double wynik;

wynik = (ch/i)+f*d-f*i
```

Przykład.

```
int i;
float x;
i=x; //źle
x=i; //dobrze
```

Przykład.

```
float srednia(int n, int tab[])
 {
 int i;
 int s=0;  // float s=0; dobrze
 for (i=0; i<n; i++)
     s=s+tab[i];
 return s/n;  // źle gdy s typu int ;
 }
```

RZUTOWANIE (konwersja jawna, wymuszona przez programistę) wykonujemy, gdy chcemy zmienić typ obiektu.

```
zmienna_typu_1 = (typ_docelowy) wyrażenie_typu_2;
```

Przykład.

```
int i=5;
float x;
x=i/2; //x=5/2=2 źle

x=(float)i/2; //zmiana typu i (z int) na float
```

```
x=(float)(i/2); //źle bo zmieni typ wyniku
                //już po podzieleniu

x=i/2.0; //⇔ x=0.5*i
```

KLASY PAMIĘCI ZMIENNYCH

Modyfikatory określające klasę przechowywania zmiennych:

- auto,
- extern,
- static,
- register.

Modyfikatory te informują kompilator jak i gdzie przechowywać przyszłe wartości zmiennych w pamięci operacyjnej.

```
modyfikator_klasy_pamięci typ nazwa_zmiennej;
```

1) Automatyczna klasa pamięci - zmienna istnieje od momentu zdeklarowania do chwili, gdy program opuści blok, w którym została zdeklarowana.
   Przykład.

```
int silnia(int n)
{ int s=1; //⇔ auto int s=1;
   ⋮
}
```

2) Zewnętrzna klasa pamięci - zmienna istnieje w czasie działania całego programu. Dostęp do tej zmiennej jest w całym programie. Zmienną taką nazywa się **globalną**.
   Przykład.

```
int n;
main()
{
 ⋮ n=10;
}
f1(...)
{
 ⋮ n++;
}
```

   Modyfikatora **extern** używa się gdy nasz program składa się z kilku plików źródłowych i trzeba w każdym pliku źródłowym przekazać informację o tej samej zmiennej globalnej.

Przykład.

- pierwszy plik źródłowy:

```
#include "mój_plik_naglowkowy.h"
  int x,y;
  char ch;

  main()
  {x=10;
   y=x/5;
     ⋮
  }

  int f11(...)
  {
   ⋮
  }
```

-drugi plik źródłowy: mój_plik_naglowkowy.h

```
  extern int x,y; //nie wolno int x,y;

  int f21(...)
  {
   x=x+5;
     ⋮
  }

  void f22(...)
  {
   ⋮
  }
```

Program dzielimy na kilka plików, gdy jest bardzo długi.
Najlepiej stworzyć jeden plik nagłówkowy z samymi deklaracjami typu extern i dołączyć je do każdego z plików źródłowych.

3) Statyczna klasa pamięci - zmienna istnieje od momentu zadeklarowania do końca działania programu, ale dostęp do tej zmiennej możliwy jest tylko w bloku, w którym została zadeklarowana.

   Jeśli zmiennej lokalnej przypiszemy modyfikator **static**, to kompilator utworzy dla niej **stałe** miejsce w pamięci operacyjnej (analogicznie, jak dla zmiennej globalnej, z tym, że lokalna zmienna statyczna jest widoczna tylko w bloku, w którym znajduje się deklaracja).

   Zmienna lokalna statyczna to zmienna lokalna, która **zachowuje swoją wartość między wywołaniami funkcji** (inaczej niż zwykła zmienna lokalna, która przy kolejnym wywołaniu tej samej funkcji jest tworzona na nowo i nie "pamięta"jaka była jej wartość w poprzednim wywołaniu).

   Przykład.

```
  int LWF=0 // liczba wywołań funkcji – zmienna globalna
  int Fib1(int n)
  { LWF++;
     ⋮
  }
```

```
int Fib2(int n)
{static int LWF=0;
LWF++;
⋮
}
```

Wartość początkowa zmiennej LWF zostanie nadana tylko przy pierwszym wywołaniu funkcji Fib2(...).

Jeśli **zmiennej globalnej** przypiszemy modyfikator **static**, to będzie to oznaczało, że zmienna globalna będzie znana tylko w pliku, w którym wystąpiła jej deklaracja (tzn. że w innym pliku źródłowym można zadeklarować zmienną o tej samej nazwie i nie będą one ze sobą kolidować i mieć ze sobą cokolwiek wspólnego).

4) Rejestrowa klasa pamięci

Modyfikator **register** instruuje kompilator, żeby zmienną umieścić w pamięci operacyjnej tak, aby zapewnić jak najszybszy dostęp do jej wartości (najczęściej w rejestrach procesora).
Ten modyfikator możemy stosować tylko do **zmiennych lokalnych** oraz **parametrów formalnych funkcji**.

np.

```
int f1(register int n)
{register int s=0;
 ⋮
}
```

MAKSYMY I PORADY PROGRAMISTYCZNE

1) Programy mają być również czytane przez ludzi.
2) Czytelność jest często ważniejsza niż sprawność.
3) Wystarczy jedna instrukcja w wierszu.
4) Stosuj odstępy do poprawienia czytelności.
5) Stosuj wcięcia dla uwidocznienia struktury programu i danych.
6) Używaj dobrych nazw mnemonicznych.
7) Nawiasy kosztują mniej niż błędy.
8) Dawaj więcej komentarzy niż będzie Ci, jak sądzisz, potrzeba.
9) Stosuj komentarze wstępne.
10) Komentarz ma dawać coś więcej niż tylko parafrazę tekstu programu.
11) Najpierw projekt, potem kodowanie.