J.NAWROCKI, M. ANTCZAK, H. ĆWIEK, W. FROHMBERG, A. HOFFA, M. KIERZYNKA, S. WĄSIK

WPROWADZENIE DO INFORMATYKI

PROGRAMOWANIE IMPERATYWNE

WPROWADZENIE DO JĘZYKA C

1. Podstawowa struktura prostego programu

```
#include <stdio.h>
                        /* biblioteka standardowa */
main()
                        /* definicja funkcji main */
{                                /* blok instrukcji */
  printf("ahoj, przygodo\n");
                      /* wywolanie funkcji printf */
  return 0;
}
```

Wyjście:

```
ahoj, przygodo
```

2. Deklarowanie zmiennych

definicje zmiennych (deklaracja + inicjalizacja),
typ ident = wartość [, ident = wartość]*;

```
int licznik=125, suma=0;
float dokladnosc=0.001, blad=0.1;
double moc=15e6, straty=1500;
```

3. Ważniejsze operatory

+	dodawanie
-	odejmowanie
*	mnożenie, operator dostępu pośredniego (wyłuskanie)
/	dzielenie, dzielenie całkowite
%	dzielenie modulo
++	inkrementacja (zwiększenie o 1)
--	dekrementacja (zmniejszenie o 1)
=	przypisanie
==	równe
!=	różne
<	mniejsze
<=	mniejsze lub równe
>	większe
>=	większe lub równe

```
int a, b, c;
a = 1;
b = 2;
c = a+b;
a = b = c = 0;          /* wielokrotne podstawienie */
a = b/c; /* część całkowita z dzielenia b przez c */
a = b%c            /* reszta z dzielenia b przez c */
a++;                              /* powiększ a o 1 */
++a;                              /* powiększ a o 1 */
a=a+1;                            /* powiększ a o 1 */
```

4. Podstawowe pętle

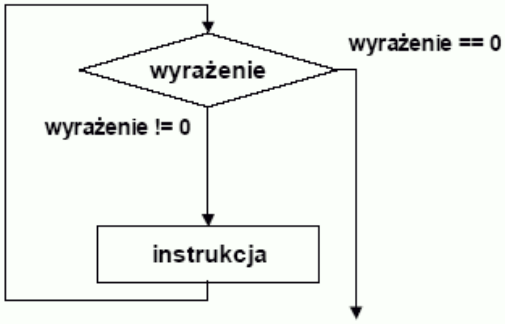
➢ **pętla *while***

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

albo:

```
while(wyrażenie)
  {
    instrukcja1;
    instrukcja2;
    instrukcjan;
  }
```

- **warunek jest testowany przed wejściem do pętli**
- **ciało pętli może nie zostać ani razu wykonane**



przykład

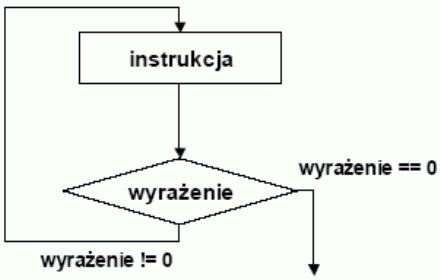
```
int n=10;
while(n>0)
{
  printf("%d\t",n);
  n--;
}
```

wyjście:

```
10   9   8   7   6   5   4   3   2   1
```

➢ **pętla do-while**

- **warunek jest testowany po wykonaniu ciała pętli**
- **ciało pętli musi być choć raz wykonane**



przykład:

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

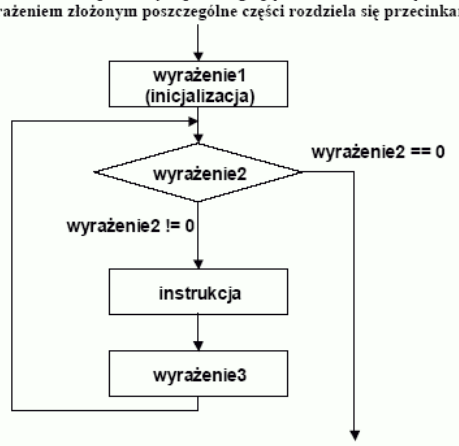
wyjście:

```
10   9   8   7   6   5   4   3   2   1
```

➢ **pętla for**

```
for(wyrażenie1;wyrażenie2;wyrażenie3)instrukcja;
```

- **dowolne (wszystkie) wyrażenia można pominąć,**
- ***wyrażenie1* jest częścią inicjującą pętli, jest obliczane tylko raz; jeśli jest wyrażeniem złożonym poszczególne wyrażenia składowe rozdziela się przecinkami,**
- ***wyrażenie2* to warunek kontynuacji pętli, pominięcie go jest równoważne z zastąpieniem stałą różną od zera (warunek zawsze prawdziwy),**
- ***wyrażenie3* jest obliczane po każdym przebiegu pętli i określa zmianę stanu pętli; jeśli jest wyrażeniem złożonym poszczególne części rozdziela się przecinkami.**



przykład:

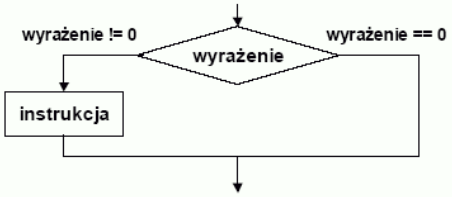
```
int i, j;
for(i=1,j=2;
    j<1000;
    i++,j*=i)printf("[%d,%d]\t",i,j);
  return 0;
}
```

wyjście:

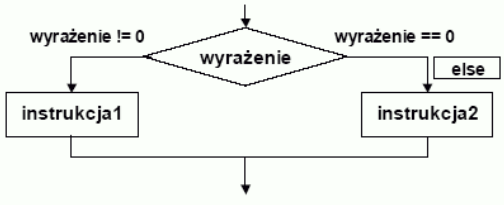
```
[1,2]    [2,4]    [3,12]   [4,48]   [5,240]
```

➢ **instrukcja warunkowa**

```
if(wyrażenie)instrukcja;
```




```
if(wyrażenie)instrukcja;
else instrukcja;
```




```
if((n % 2)==0)printf("parzysta");
else printf("nieparzysta");
```

5. Tablice

Tablice

tablica jednowymiarowa:

```
typ id [ rozmiar ];
```

z inicjalizacją

```
typ id [ opcjonalny_rozmiar ] = { lista_stałych };
```

Przykład:

- **wektor (tablica jednowymiarowa) liczb całkowitych zawierający 5 elementów**

```
int tab[5];
/* elementy tab[0], tab[1], ..., tab[4] */
```

Przykład:

- **4-elementowy wektor liczb całkowitych, rozmiar określony na podstawie listy inicjalizacji**

```
int tab[] = { 1, 2, 3, 5 };
/* tab[0] = 1, tab[1] = 2, tab[2] = 3, tab[3]=5 */
```

Przykład:

- **wektor częściowo zainicjalizowany (list inicjalizacji krótsza niż zadeklarowana liczba elementów)**

```
int tab[4] = { 1, 2 };
/* tab[0] = 1, tab[1] = 2,
   pozostałe elementy są inicjalizowane zerami*/
```

Odwołania do elementów tablic

```
nazwa_tablicy[element]
```

Przykład:

- **suma elementów w tablicy**
- **automatyczne określenie rozmiaru tablicy**

```
unsigned u;            /* w języku C indeksy tablic
                          są zawsze nieujemne */
int suma;
int tab[5] = { 1, 2, 2, 3, 4 };

for(u=0,suma=0;
    u < (sizeof(tab)/sizeof(tab[0]));
    u++)
   suma += tab[u];        /* suma=suma+tab[u]; */
printf("Suma elementow w tablicy = %d",suma);
```

tablice znakowe są specjalnym przypadkiem

reprezentacja łańcuchów

```
"\n\aHello\tworld\n"
```

wewnętrznie jest reprezentowany jako ciąg znaków zakończony wartownikiem ('\0')

'\n'	'\a'	H	e	l	l	o	'\t'	w	o	r	l	d	'\n'	'\0'

widok pod debuggerem (Turbo C++ 1.01)

[0]	'\n'	10	(0x0A)
[1]	'\a'	7	(0x07)
[2]	'H'	72	(0x48)
[3]	'e'	101	(0x65)
[4]	'l'	108	(0x6C)
[5]	'l'	108	(0x6C)
[6]	'o'	111	(0x6F)
[7]	'\t'	9	(0x09)
[8]	'w'	119	(0x77)
[9]	'o'	111	(0x6F)
[10]	'r'	114	(0x72)
[11]	'l'	108	(0x6C)
[12]	'd'	100	(0x64)
[13]	'\n'	10	(0x0A)
[14]	'\x0'	0	(0x00)

Przykład:

- **5-elementowa tablica znakowa**

```
char tab[5]; /* tab[0], ..., tab[4],
                brak inicjalizacji */
char tab[] = "tekst"; /* tab[0]='t', tab[1]='e',
                         ..., tab[5]='\0' */
char tab[] = {'t', 'e', 'k', 's', 't', '\0' };
           /* alternatywny sposób inicjalizacji */
```

typowe błędy

```
char tab[5] = "tekst"; /* raczej nie o to chodziło
                          - brak wartownika */
char tab[4] = "tekst"; /* błąd kompilacji, za mała
                          tablica */
char tab[] = {'t', 'e', 'k', 's', 't' }; /* brak
       inicjalizacji elementu tab[5] */
```

Tablice wielowymiarowe

Przykład:

```
int tab2d[4][2]; /* 4 wiersze po 2 elementy */
```

rozmieszczane w pamięci wierszami,

tab2d[0][0]	tab2d[0][1]
tab2d[1][0]	tab2d[1][1]
tab2d[2][0]	tab2d[2][1]
tab2d[3][0]	tab2d[3][1]

inicjalizacja

```
int tab2d[4][2]={{0,1},{2,3},{4,5},{6,7}};
int tab2d[4][2]={0,1,2,3,4,5,6,7}; /*niezalecane*/
```

6. Wskaźniki

Wskaźniki

wskaźnik to zmienna, która zawiera adres innej zmiennej

Przykłady:

```
int *ptr_2_int;
                /* wskaźnik na zmienne typu int */
double *ptr_2_double;
             /* wskaźnik na zmienne typu double */
int *pa, *pb, a, b;    /* 2 wskaźniki na zmienne
                       typu int, 2 zmienne typu int */
void *ptr1;       /* "wskaźnik do niczego", nie ma
                          zmiennych typu void */
```

- **pobranie adresu zmiennej – operator &**
- **uzyskanie dostępu do zmiennej – operator ***

```
int i = 5, j = 7, res;
int *pi = &i;     /* pi wskazuje na i */
*pi = 3;          /* równoważne i=3; */
pi = &j;          /* pi wskazuje na j */
*pi = 4;          /* równoważne j=4; */
{                 /* nowy blok instrukcji */
  int *p1=&i,     /* p1 wskazuje na i */
      *p2=&j,     /* p2 wskazuje na j */
      *p3=&res;   /* p3 wskazuje na res */
                  /* zmienne p1, p2, p3 istnieją
                     tylko w tym bloku instrukcji */
  *p3 = *p2**p1; /* res = j * i; */
  *p2*=*p1;       /* j *= i; */
}            /* zmienne p1, p2, p3 już nie istnieją
                  ale wartości w res, j pozostają */
```

- **uwaga na klasę register**

```
register int b;
int a, *pa;

pa = &a;
/* pa = &b; nie można -> klasa register */
```

- **wskaźniki i liczby całkowite nie są ze sobą wymienne (zła praktyka programistyczna)**
- **jedynym wyjątkiem jest zero (0), stałą zero można przypisać wskaźnikowi, wskaźnik można porównać ze stałą 0, zazwyczaj zamiast zera (0) używa się nazwy symbolicznej NULL (0 w kontekście wskaźnika), implementacja gwarantuje, że żaden wskaźnik wskazujący na zmienną nie może przyjąć wartości NULL**

Przykład:

- **kopiowanie wskaźników, zgodność typów**

```
int aa, *pa=NULL, *pb=NULL;
double *pd;

pa = &aa;    /* pa wskazuje na aa */
pb = pa;     /* poprawnie, pb i pa wskazują na aa */
pd = pa;     /* potencjalny błąd !!!,
                double - 8B w formacie fp,
                int - 4B, liczba całkowita */
*pd = 2.0;  /* zamaże 8B zamiast 4B */
pd = &aa;    /* potencjalny błąd !!! */
```

Przykład:

- **wskaźnik do stałej**

```
const int aa;
int *pa;

pa = &aa;  /* niebezpieczna konwersja */
*pa = 1;   /* błąd !!!,
              przypisanie zostanie wykonane */
```

powinno być:

```
const int aa;
const int *pa;
int b;

pa = &aa;       /* prawidłowa konwersja */
/* *pa = 1; <- kompilator zgłosi błąd */
b = *pa;        /* poprawne pobranie wartości */
```

7. Tablice a wskaźniki

Tablice a wskaźniki

w języku C występuje ścisła zależność między wskaźnikami i tablicami każdą operację, którą można wykonać indeksując tablicę można wykonać również za pomocą wskaźników

Przykład:

- **suma elementów tablicy (za pomocą indeksowania)**

```
unsigned u;
int suma, tab[5] = { 1, 2, 2, 3, 4 };
for(u=0,suma=0;
    u < (sizeof(tab)/sizeof(tab[0]));
    u++)
   suma += tab[u];
```

- **suma elementów tablicy (z przesuwaniem wskaźnika)**

```
unsigned u;
int suma, tab[5] = { 1, 2, 2, 3, 4 }, *pa;
for(u=0,suma=0,pa=tab;       /* pa=&tab[0]; */
    u < (sizeof(tab)/sizeof(tab[0]));
    u++)
{
   suma += *pa;
   pa++;
}
```

- **suma elementów tablicy (dodawanie do wskaźnika)**

```
unsigned u;
int suma, tab[5] = { 1, 2, 2, 3, 4 }, *pa;
for(u=0,suma=0,pa=tab;       /* pa=&tab[0]; */
    u < (sizeof(tab)/sizeof(tab[0]));
    u++)
  suma += *(pa+u);
```

- **różne operacje na wskaźnikach**

```
y = *ip + 1; /* wartość obiektu pod ip, powiększamy
                o 1 i podstawiamy pod y */
*ip+=1;      /* zwiększenie obiektu wskazywanego
                przez ip */
(*ip)++;     /* j.w. - nawiasy są niezbędne */
```

- **jeżeli wskaźniki wskazują na elementy tej samej tablicy to można ze sobą porównywać (==, <. <=, != ...)**
- **skutek operacji arytmetycznych lub porównań dla wskaźników odwołujących się do elementów różnych tablic jest nieokreślony**
- **wyjątek to użycie adresu pierwszego miejsca po ostatnim elemencie tablicy**

8. Formatowane wejście/wyjście

Formatowane wejście

```
int scanf(const char *format, ...);
```

- Funkcja czyta z wejścia ciąg znaków i pod kontrolą argumentu format przypisuje przekształcone wartości kolejnym argumentom. Każdy z tych argumentów musi być wskaźnikiem.
- Funkcja kończy działanie, gdy zinterpretuje cały łańcuch sterujący.
- Funkcja zwraca EOF, jeśli przed jakimkolwiek przekształceniem napotka koniec pliku lub nastąpi jakiś błąd, w przeciwnym przypadku zwraca liczbę przekształconych i przypisanych danych wejściowych.
- Format zwykle zawiera specyfikacje przekształceń, które sterują interpretacją wejściowego ciągu znaków. W formacie mogą wystąpić:
  - Odstępy i tabulacje, które są ignorowane
  - Zwykłe znaki (nie %) spodziewane jako następne "czarne" znaki w strumieniu wejściowym,
  - Specyfikacje przekształceń złożone ze znaku %, opcjonalnego znaku * wstrzymującego przypisanie, opcjonalnej liczby określającej maksymalny rozmiar pola, opcjonalnej litery h, l lub L określającej rozmiar odpowiedniego argumentu oraz ze znaku przekształcenia.
- Znaki przekształcenia d, i, n, o, u i x można poprzedzić literą h, jeśli argument jest wskaźnikiem do obiektów typu short, a nie int, lub literą l, jeśli argument jest wskaźnikiem do obiektów typu long.
- Znaki przekształcenia e, f i g mogą być poprzedzone literą l, jeśli na liście argumentów znajduje się wskaźnik do obiektów typu double, a nie float, lub literą L dla typu long double.

Znak	Dana wejściowa	Typ argumentu
d	Liczba całkowita dziesiętna	int *
i	Liczba całkowita; albo w postaci ósemkowej (wiodące zero) lub szesnastkowej (wiodące 0x lub 0X)	int *
o	Liczba całkowita ósemkowa (z wiodącym zerem lub bez)	int *
u	Liczba całkowita dziesiętna bez znaku	unsigned int *
x	Liczba całkowita szesnastkowa (z wiodącym 0x lub 0X lub bez)	int *

Formatowane wyjście

```
int printf(const char *format, ...);
```

- Wartość zwracana przez funkcję równa się liczbie wypisanych znaków lub jest liczbą ujemną w przypadku błędu.
- Liczba argumentów musi odpowiadać liczbie w formacie, jeżeli jest mniejsza rezultaty są nieprzewidywalne.

- argument format zawiera znaki dwojakiego rodzaju:
  - zwykłe znaki, które są bezpośrednio kopiowane do strumienia wyjściowego *stdout*
  - specyfikacje przekształcenia, które wskazują sposób przekształcenia i wypisania kolejnych argumentów funkcji
- modyfikatory (w dowolnej kolejności), które wpływają na postać wyniku:
  - - : dosunięcie przekształconego argumentu do lewego krańca jego pola
  - + : wypisanie liczby zawsze ze znakiem
  - odstęp : poprzedzenie wyniku znakiem odstępu, jeśli jego pierwszym znakiem nie jest plus lub minus
  - 0 - dla przekształceń liczbowych, uzupełnienie liczby wiodącymi zerami do pełnego rozmiaru pola
  - # - alternatywna postać wyniku: dla formatu o pierwszą wypisaną cyfrą będzie 0;
    dla formatu x lub X wynik różny od 0 poprzedza się 0x lub 0X;
    dla formatów e, E, f, g i G wypisana liczba zawsze będzie miała kropkę dziesiętną (dla g i G nieznaczące zera nie zostaną usunięte)
- Liczba określająca minimalny rozmiar pola. Jeśli przekształcony argument jest krótszy od rozmiaru pola, to zostanie dopełniony z lewej strony (lub z prawej, jeśli zlecono dosunięcie w lewo) do pełnego rozmiaru pola. Znakiem wypełniającym jest zwykle odstęp, chyba że polecono dopełnianie zerami - wówczas jest nim znak 0.
- Kropka oddzielająca rozmiar pola od precyzji
- Liczba (precyzja) określająca: maksymalną liczbę wypisywanych znaków tekstu; liczbę cyfr wypisywanych po kropce dziesiętnej w specyfikacjach e, E lub f; liczbę cyfr znaczących w specyfikacjach g lub G; minimalną liczbę cyfr wypisywanych dla argumentu całkowitego (dopisanie wiodących zer).
- Modyfikator długości h, l lub L. Litera h wskazuje, że odpowiedni argument jest typu short lub unsigned short; liter l wskazuje, że argument jest typu long lub unsigned long; litera L wskazuje, że argument jest typu long double.

Zamiast liczby określającej rozmiar pola lub precyzję (lub obie) można podać znak *. W tym przypadku żądaną wartość oblicza się na podstawie następnego argumentu (lub następnych argumentów) funkcji. Argument ten musi mieć typ int.

Znak	Dana wejściowa	Typ argumentu
c	Znaki; kolejne znaki zostaną wstawione do wskazanej tablicy. Liczba wpisanych znaków określona jest rozmiarem pola (domyślnie 1). Nie dodaje się znaku '\0'. Nie obowiązuje zwykłe pomijanie białych znaków.	char *
s	Ciąg czarnych znaków (bez cudzysłowów). Tablica podana jako argument musi być dostatecznie duża, aby pomieścić cały ciąg razem z dodanym na końcu znakiem '\0'.	char *
e f g	Liczba zmiennopozycyjna. Wejściowy format dla liczb zmiennopozycyjnych zawiera opcjonalny znak, ciąg cyfr ewentualnie rozdzielony kropką dziesiętną oraz opcjonalny wykładnik potęgi, składający się z litery E lub e i liczby całkowitej ewentualnie ze znakiem.	float *
p	Wartość wskaźnika w postaci zależnej od implementacji.	void *
n	Wpisuje do argumentu liczbę dotychczas przeczytanych znaków w tym wywołaniu funkcji. Nie czyta żadnych wejściowych znaków. Nie zwiększa licznika przeczytanych pól.	int *
[...]	Odpowiada najdłuższemu niepustemu ciągowi wejściowych znaków ze zbioru znaków zawartych między nawiasami. Na końcu jest dopisywany znak '\0'. Napis []...] włącza znak] do zbioru.	char *
[^...]	Odpowiada najdłuższemu niepustemu ciągowi wejściowych znaków nie należących do zbioru znaków zawartych między nawiasami. Na końcu jest dopisywany znak '\0'. Napis [^]...]włącza znak] do zbioru.	char *
%	Znak %. Nie ma żadnego przypisania.	

Przykład:

```
char tekst[10];
double zmienna;
scanf("%9s",tekst);
   /* łańcuch "abrakadabra" zostanie wczytany jako
      "abrakadab", łańcuch "Ala ma kota" zostanie
      wczytany jako "Ala" (do spacji) */

scanf("%9c",tekst); /* zostanie wczytanych 9 znaków
                       włącznie ze spacjami */

tekst[9]='\0';      /* ale wartownika trzeba wpisać
                       samemu */
scanf("%lg",&zmienna);
scanf("%c",&tekst[4]); /* czytamy jeden znak do
                          tablicy */
```

Znak	Typ argumentu	Przekształcany do postaci
d, i	int	liczby dziesiętnej ze znakiem
o	int	liczby ósemkowej bez znaku (bez wiodącego zera)
x, X	int	liczby szesnastkowej bez znaku (bez wiodącego 0x lub 0X) z użyciem liter abcdef dla 0x lub ABCDEF dla 0X
u	int	liczby dziesiętnej bez znaku
c	int	pojedynczego znaku po przekształceniu do typu *unsigned char*
s	char *	Znaki tekstu są wypisywane aż do napotkania znaku '\0' lub liczba wypisanych znaków osiągnie wskazaną precyzję
f	double	liczby dziesiętnej [-]m.ddddddd±xx lub [-]m.ddddddE±xx, gdzie liczba cyfr d zależy od zadanej precyzji. Domyślną precyzją jest 6; przy precyzji 0 opuszcza się kropkę dziesiętną
g, G	double	Jeśli wykładnik potęgi jest mniejszy niż -4 lub większy lub równy precyzji, to stosuje się specyfikację %e lub %E; w przeciwnym przypadku %f. Nie wypisuje się nie znaczących zer i zbędnej kropki dziesiętnej
p	void *	wskaźnika (reprezentacja zależy od implementacji)
n	int *	Liczbę dotychczas wypisanych znaków w tym wywołaniu funkcji printf zapisuje się do odpowiedniego argumentu. Nie ma żadnego przekształcenia argumentu
%		Nie ma żadnego przekształcania argumentu - zostanie wypisany znak %.

Przykład:

```
char tekst[]="Hello world";
double zmienna = 1.0/3.0;
printf("%+20s\t% f\t%010.3f",
      tekst,zmienna,zmienna);
```

```
        Hello world     0.333333        000000.333
```

9. Wielomodułowość

10. Podstawowe funkcje matematyczne (biblioteka math.h)

- `double sqrt(double x)` - The *sqrt()* function computes the square root of *x*, $\sqrt{x}$.
- `double ceil(double x)` - The *ceil()* function computes the smallest integral value not less than *x*.
- `double floor(double x)` - The *floor()* function computes the largest integral value not greater than *x*.
- `double pow(double x, double y)` - The *pow()* function computes the value of *x* raised to the power *y*, x^y. If *x* is negative, *y* must be an integer value.