

Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 16.03.2021

3. Sterowanie przebiegiem wykonania programu

Instrukcje

```
#include <math.h>
```

Dyrektywy

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

```
{
```

```
double p=0, r;
```

Deklaracje

```
scanf ("%lf", &p);
```

```
if (p>=0) {
```

```
    r=sqrt(p/M_PI);
```

```
    printf("Promien koła o polu %f wynosi: %f", p, r);
```

```
}else{
```

```
    printf("Blad: ujemne pole: %f", p);
```

```
}
```

```
return 0;
```

```
}
```

Instrukcje

Instrukcje w języku C/C++ pozwalają na:

- obliczanie wyrażeń (w tym wywołanie funkcji)
- sterowanie wykonaniem programu;

Składnia instrukcji

Instrukcje języka C/C++

- kończą się średnikiem (;) lub
- mają postać instrukcji blokowej { }.

Po nawiasie zamykającym bloku nie umieszcza się średnika.

```
for(i=0;i<10;i++){  
    printf("%d",i);  
}  
  
if(x>7){  
    printf("x>7", x);  
}  
  
{  
    x = z++;  
}
```

Instrukcja pusta

- Instrukcją pustą nazywamy pusty tekst, po którym umieszczony jest średnik lub pusty tekst wewnątrz bloku { }.
- Białe znaki lub komentarze też mogą być traktowane jako pusty tekst.

```
if (x>7) ; // instrukcja pusta
```

```
if (x>7) {  }
```

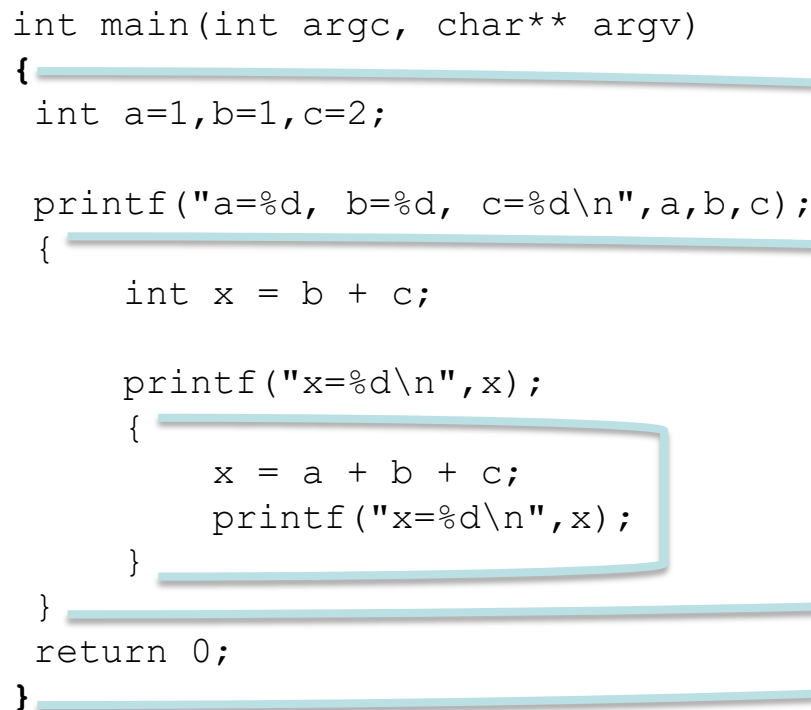
Instrukcja blokowa

- Instrukcja blokowa to dowolny (także pusty) ciąg instrukcji zawarty pomiędzy nawiasami klamrowymi
- Wewnątrz instrukcji blokowej można deklarować zmienne (na początku bloku instrukcji)
- Bloki mogą być zagnieżdżane
- Funkcja zawiera jedną główną instrukcję blokową

```
int main(int argc, char** argv)
{
    int a=1,b=1,c=2;

    printf("a=%d, b=%d, c=%d\n",a,b,c);
    {
        int x = b + c;

        printf("x=%d\n",x);
        {
            x = a + b + c;
            printf("x=%d\n",x);
        }
    }
    return 0;
}
```



Instrukcje sterujące

Instrukcje sterujące pozwalają na

- warunkowe wykonanie instrukcji (`if-else`)
- przejście do wskazanej instrukcji (`goto`)
- wykonanie instrukcji w pętli (`for`, `do`, `while`)
- wybór instrukcji do wykonania (`switch-case`)

Wartości i wyrażenia logiczne

- Warunkowe instrukcje sterujące posługują się wyrażeniami logicznymi przy wyborze ścieżki wykonania programu.
- Wyrażenie logiczne jest funkcją odwzorowującą argumenty w zbiór wartości logicznych `{true, false}`.
- W języku C nie występują słowa kluczowe `true`, `false`. Typem danych dla wartości logicznych jest typ całkowitoliczbowy `int`.

Przyjęto, że:

- **zerowa** wartość zmiennej lub wyrażenia oznacza fałsz
- **niezerowa** (nieokreślona) wartość oznacza prawdę

Wyrażenia logiczne 1

- Wyrażenia logiczne mają postać pojedynczej zmiennej całkowitej lub są konstruowane z użyciem operatorów zwracających wartości logiczne.
- Podstawowe operatory zwracające wartości logiczne:

&&	koniunkcja
	alternatywa
!	negacja
<	relacja mniejszości
>	większości
<=	mniejszy lub równy
>=	większy lub równy
!=	porównanie: nierówność
==	porównanie: równość

Wyrażenia logiczne 2

W wielu przypadkach kompilator języka C/C++ oczekując wyrażenia logicznego jest w stanie dokonać automatycznej konwersji wyrażenia do typu całkowitego nawet, jeśli nie miałyby to sensu i wynikało z błędu programisty!

```
if("ala ma kota") {...}      // wartością jest true

if(2.7){...}                  // wartością jest 2 - true

int isGood(int x){
    return x>=0;
}

if(isGood){...}               // wartością jest true
```

Wyrażenia logiczne 3

- Uwaga na operatory & oraz &&

```
int x=1
if(x) {...}      // wartością jest true

int y=2;
if(y) {...}      // wartością jest true

if(x&&y){...}     // wartością jest true

if(x&y){...}      // wartością jest 0 czyli false
```

Instrukcja if-else 1

Instrukcja if-else steruje warunkowym wykonaniem instrukcji.

Ma ona postać

```
if ( expression ) statement1
```

albo

```
if ( expression ) statement1  
else statement2
```

- Instrukcja `statement1` zostanie wykonana, jeżeli wyrażenie `expression` przyjmie wartość niezerową (`true`).
- W przeciwnym przypadku zostanie wykonana instrukcja `statement2` (dla wersji z `else`).

Instrukcja if-else 2

Przykłady

```
if(x>7)printf("x>7\n");

if(x){
    printf("x not equal 0\n");    // instr. blokowa
}

if(x>=0){
    printf("%f",sqrt(x));
}else printf("error");

if(x>=0){
    if(x>7) printf("x>7\n");
    else printf("x>=0 and x<=7\n");
}else{
    printf("x<0\n");
}
```

Instrukcja goto 1

Instrukcja goto pozwala na przejście do etykietowanej instrukcji wewnątrz funkcji.


```
#include <stdio.h>

int main(int argc, char** argv) {
    int i,suma=0;

    i=0;

    loop:
    suma=suma+i;
    i=i+1;
    if(i<10) goto loop;

    printf("Suma wynosi: %d",suma);
    return 0;
}
```



Instrukcja goto 2

Inna konstrukcja..

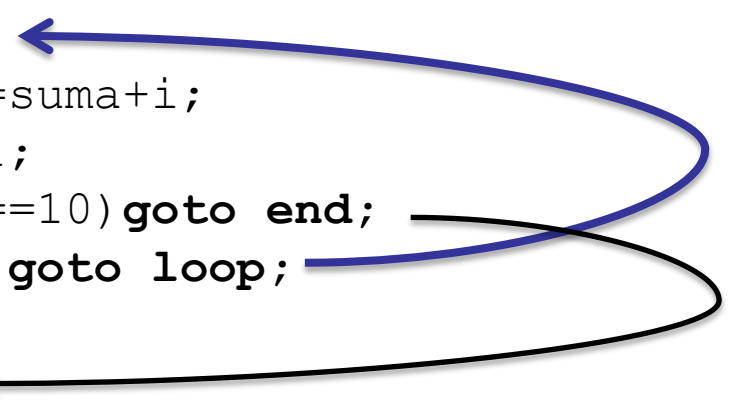
```
#include <stdio.h>

int main(int argc, char** argv) {
    int i,suma=0;

    i=0;

    loop: suma=suma+i;
           i=i+1;
           if(i==10) goto end;
           else goto loop;

    end: printf("Suma wynosi: %d",suma);
         return 0;
}
```



Instrukcja goto 3

Jaki jest rezultat wykonania programu?

```
#include <stdio.h>

int main(int argc, char** argv) {
    int i,j;

    i=0,j=0;

    one:
    printf("i=%d,j=%d\n",i,j);
    two:
    i=i+1;
    j=j+1;
    if(i%2==0) goto one;
    if(j>10) goto end;
    goto two;

    end:
    return 0;
}
```

```
graph TD
    one: --> printf
    printf --> two:
    two: --> i_plus_1
    i_plus_1 --> j_plus_1
    j_plus_1 --> if_i
    if_i --> one:
    if_j --> end:
    goto_two --> two:
    end: --> return
```


Instrukcja goto 4

```
i=0,j=0  
i=2,j=2  
i=4,j=4  
i=6,j=6  
i=8,j=8  
i=10,j=10
```

- Instrukcja goto jest łatwa do bezpośredniego odwzorowania w kodzie maszynowym

`goto two; → jmp two`

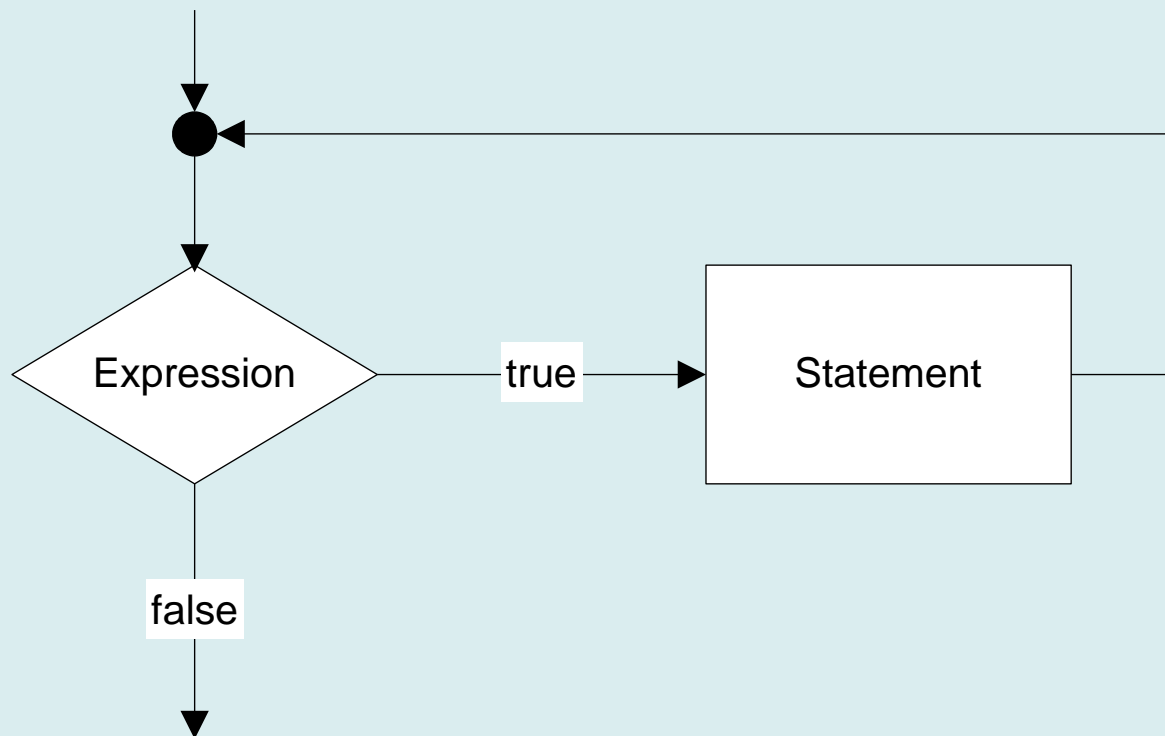
- Kod używający instrukcji skoku jest jednak
 - mało czytelny
 - podatny na błędy
 - przecinające się pętle są antywzorcem zasad konstrukcji algorytmów

Instrukcja while

Instrukcja `while` ma postać:

```
while (expression) statement
```

Instrukcja `statement` będzie wykonywana dopóki wyrażenie `expression` nie przybierze wartości zerowej (`false`).



Instrukcja while – przykłady 1

```
int i=4;
while(i){                                // = while(i>0)
    printf("%d ",i);
    i--;                                // modyfikacja i=i-1
}

while(0){ // instrukcje nieosiagalne
    printf("this should never occur");
}

while(1){ // pętla nieskończona
    printf("infinite loop ");
}
```

4 3 2 1 infinite loop infinite loop infinite loop...

Instrukcja while – przykłady 2

```
#include <stdio.h>
int main()
{
    int i=0;
    int sum=0;
    while (i<100) {
        sum=sum+i;
        i++; //i=i+1;
    }
    printf("Suma %d\n", sum);
    printf("Suma %d\n", 100*(0+99)/2);
    return 0
}
```

Suma 4950

Suma 4950

Instrukcja while – przykłady 3

```
int compare(int t1[],int t2[],int size)
{
    int i=0;
    while(i<size && t1[i]==t2[i])i++;
    if(i==size)return 0; // równe
    else return 1;      // różne
}
```

- Argumentami funkcji są dwie tablice `t1` i `t2` oraz rozmiar `size`
- Wyrażenie `t1[i]==t2[i]` porównuje elementy tablic
- Pętla jest wykonywana dopóki:
 - Elementy tablic są równe
 - Spełniony jest warunek `i<size`

Instrukcja while – przykłady 4

```
void cyfry(int k)
{
    while (k>0) {
        printf("%d", k%10);
        k=k/10;
    }
}
```

Wywołanie: `cyfry(1234);`

Wynik: 4321

Wywołanie: `cyfry(0);`

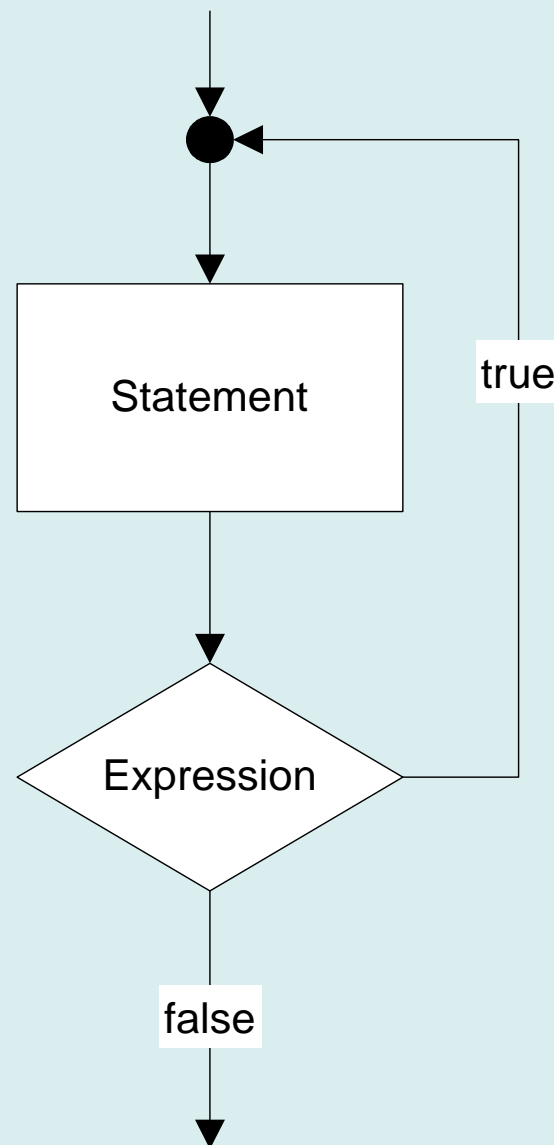
Wynik: <pusty>

Instrukcja do-while

Instrukcja ta ma postać:

```
do  
    statement  
while (expression) ;
```

Różnicą w stosunku do `while` jest to, że instrukcja `statement` jest wykonywana przed sprawdzeniem warunku, stąd będzie zawsze wykonana, co najmniej jeden raz.



Instrukcja do-while – przykłady 1

```
int passwordOk=0;
do{
    passwordOk = enterPassword();
}while (!passwordOk);
// dalsze instrukcje
```

- Funkcja `enterPassword()` jest wołana co najmniej jeden raz

Instrukcja do-while – przykłady 2

```
void cyfry2(int k)
{
    do{
        printf("%d", k%10);
        k=k/10;
    }while(k>0);
}
```

Wywołanie: `cyfry(1234);`

Wynik: 4321

Wywołanie: `cyfry(0);`

Wynik: 0

Instrukcja for

Instrukcja `for` jest równoważna instrukcji `while`. Pozwala ona na jednolity zapis wszystkich elementów sterujących iteracją:

- inicjalizację (`initialization`)
- testowanie warunku pętli (`expression`)
- modyfikację stanu iteracji (`modification`)

Składnia instrukcji `for` jest następująca

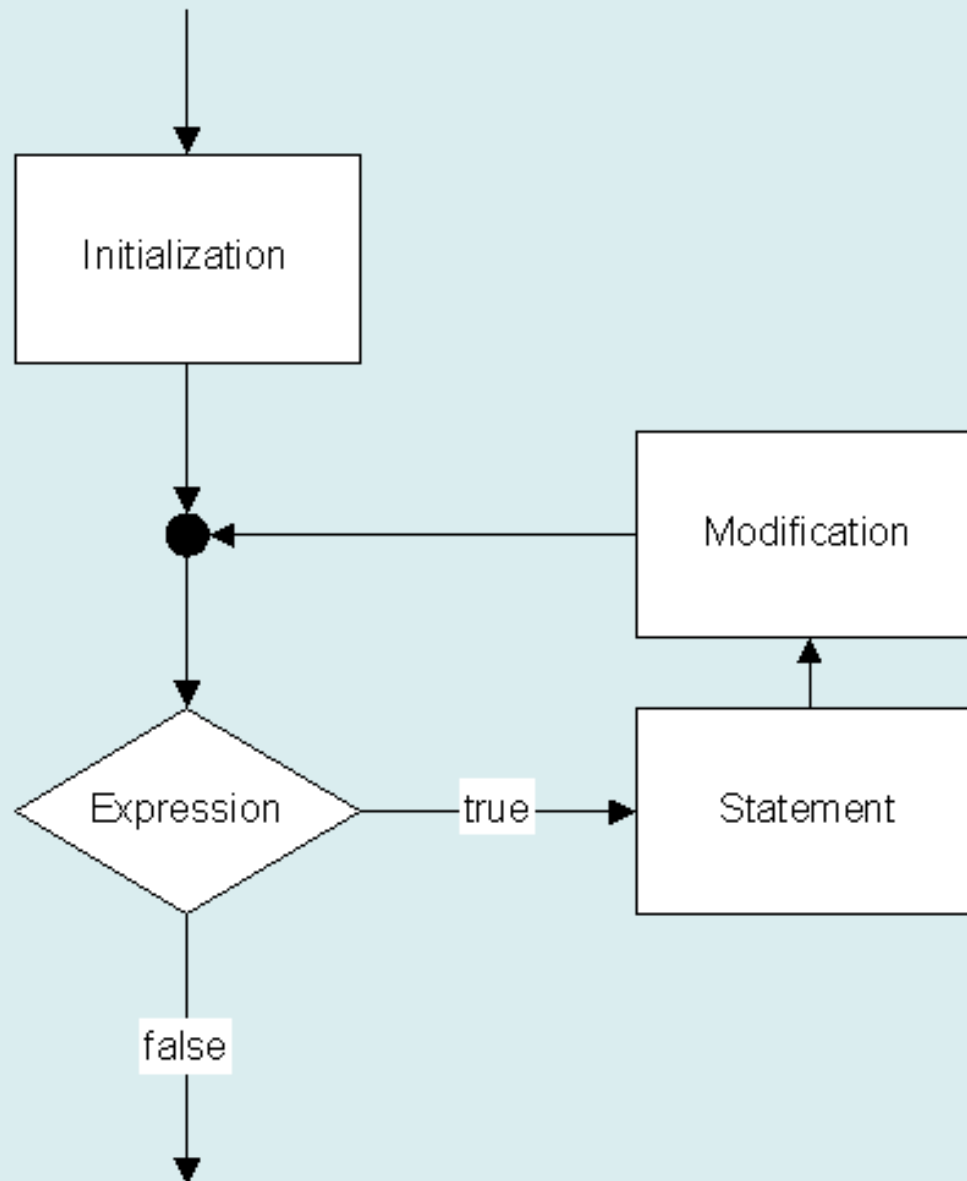
```
for(initialization; expression; modification)  
statement
```

Instrukcja for - przykład

```
#include <stdio.h>
int main()
{
    int i;
    int sum=0;

    for (i=0;i<100;i++) {
        sum=sum+i;
    }
    printf("Suma %d\n", sum);
    printf("Suma %d\n", 100*(0+99)/2);
    return 0;
}
```

Instrukcja for – schemat blokowy



Instrukcja for – algorytm

```
for(initialization; expression; modification)  
statement
```

1. Jeżeli instrukcja `initialization` nie jest pusta wykonaj ją.
2. Oblicz wartość wyrażenia logicznego `expression`.
 - a. Jeżeli wyrażenie jest puste przejdź do 3
 - b. Jeżeli wyrażenie ma wartość niezerową (`true`), to przejdź do 3.
 - c. Jeżeli wyrażenie ma wartość zerową (`false`), przejdź do 5.
3. Wykonaj instrukcję `statement`
4. Jeżeli wyrażenie `modification` nie jest puste, wykonaj `modification`. Następnie przejdź do 2
5. Przejdź do następnej instrukcji po `for`.

Instrukcja for – różne wersje 1

- Wszystkie wyrażenia instrukcji `for` są opcjonalne.

```
// Pętla nieskończona  
for(;;) {  
    ...  
}
```

```
// analogicznie
```

```
while(1) {  
    ...  
}
```

Instrukcja for – różne wersje 2

- Zarówno w wyrażeniu inicjalizacyjnym, jak i modyfikującym może pojawić się lista instrukcji oddzielonych przecinkami. Instrukcje są wykonywane w kolejności od lewej do prawej.

```
#include <stdio.h>
int main()
{
    int i;
    int sum;
    for(i=0, sum=0; i<100; i++) sum=sum+i;
/*
    niezalecane
    for(i=0, sum=0; i<100; sum=sum+i, i++);
*/
    printf("Suma %d\n", sum);
    printf("Suma %d\n", 100*(0+99)/2);
    return 0;
}
```

Czytelna pętla odróżnia kod sterujący iteracją od zadania do wykonania podczas iteracji.

Instrukcja for – różne wersje 3

Zarówno w wyrażeniu inicjalizacyjnym, jak i modyfikującym może pojawić się lista instrukcji oddzielonych przecinkami. Instrukcje są wykonywane w kolejności od lewej do prawej.

```
void reverse(int table[], int size)
{
    int i, j;
    for (i=0, j=size-1; i<j ; i++, j--)
    {
        int tmp;
        tmp=table[i];
        table[i]=table[j];
        table[j]=tmp;
    }
}
```

W jednej pętli
modyfikowane są w każdej
iteracji dwie zmienne.

Instrukcja for – wartości funkcji

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

int main(int argc, char** argv) {
    double min=0;
    double max=M_PI;
    int count = 5;
    double step=(max-min) / (count-1);
    double x;

    for (x=min; x<=max; x=x+step) {
        printf("sin(%f)=%f\n", x, sin(x));
    }
    return 0;
}
```

Modyfikowana wartość nie musi być liczbą całkowitą. Nie zawsze musimy inkrementować ją o 1.

```
sin(0.000000)=0.000000
sin(0.785398)=0.707107
sin(1.570796)=1.000000
sin(2.356194)=0.707107
sin(3.141593)=0.000000
```

Instrukcja for – minimum funkcji

```
#include <stdio.h>

int main(int argc, char** argv) {
    double min=0;
    double max=10;
    int count = 10;
    double step=(max-min)/(count-1);
    double x;
    double min_value=1e10;
    double arg_min=min;
    for(x=min;x<=max;x=x+step){
        double y;
        //x^3-2x^2-x+1
        y=x*x*x-2*x*x-x+1;
        if(min_value>y){
            min_value=y;
            arg_min=x;
        }
        printf("f(%f)=%f\n",x,y);
    }
    printf("Minimum funkcji f(%f)=%f\n",arg_min,min_value);
}
```

Obliczanie minimum
funkcji w przedziale

```
f(0.000000)=1.000000
f(1.111111)=-1.208505
f(2.222222)=-0.124829
f(3.333333)=12.481481
f(4.444444)=44.840878
f(5.555556)=105.183813
f(6.666667)=201.740741
f(7.777778)=342.742112
f(8.888889)=536.418381
f(10.000000)=791.000000
Minimum funkcji f(1.111111)=-1.208505
```

Instrukcja for – ciąg Fibbonaciego 1

```
#include <stdio.h>

int main(int argc, char** argv) {
    int a=1,b=1,c;
    int n;

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

    for (n=2;n<10;n++) {
        c=a+b;
        printf("%d\n",c);
        a=b;
        b=c;
    }
    return 0;
}
```

Pętla nie musi się
rozpocząć od granicznej
wartości (od 0)

1
1
2
3
5
8
13
21
34
55

Instrukcja for – ciąg Fibbonaciego 2

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

int main(int argc, char** argv) {
    int a=1,b=1,c;
    int n;
    printf("%d\n",a);
    printf("%d\n",b);
    for (;a+b<100;) {
        c=a+b;
        printf("%d\n",c);
        a=b;
        b=c;
    }
    return 0;
}
```

Pętla nie musi się mieć charakteru jawnej iteracji. Wyznaczamy wyrazy ciągu <100. (W tym przypadku bardziej przejrzyste jest użycie konstrukcji do-while.)

1
1
2
3
5
8
13
21
34
55
89

Instrukcja continue

- Instrukcja `continue` umieszczana jest w bloku instrukcji `for`, `while` oraz `do-while`. Pozwala na przejście do końca bloku z pominięciem następujących po niej instrukcji.

```
while (expression){  
    // ...  
    continue;  
    // ...  
}
```

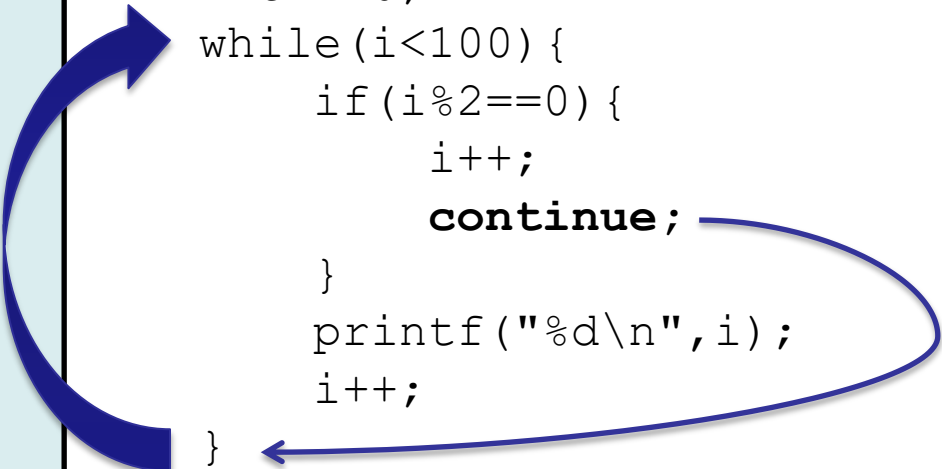
```
do {  
    // ...  
    continue;  
    // ..  
}while (expression) ;
```

- W wyniku wykonania instrukcji `continue` wewnątrz bloku instrukcji `while`, `do-while` nastąpi przejście na koniec bloku i obliczenie wartości wyrażenia warunkowego `expression`.

Instrukcja continue – przykład 1

- Instrukcja `continue` w pętli `while`

```
int main(int argc, char** argv)
{
    int i=0;
    while(i<100){
        if(i%2==0){
            i++;
            continue;
        }
        printf("%d\n",i);
        i++;
    }
    return 0;
}
```



Instrukcja continue – przykład 2

- Instrukcja continue w pętli do-while

```
int main(int argc, char** argv) {  
    int i=0; int main(int argc, char** argv)  
    {  
        int i=0;  
        do{  
            if(i%2==0) {  
                i=i++;  
                continue;  
            }  
            printf("%d",i);  
            i++;  
        }while( printf("\n%d:",i) && i<10);  
        return 0;  
    }  
}
```

Nie tu!

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

Instrukcja `continue` w pętli `for`

- Instrukcja `continue` umieszczana jest w bloku instrukcji `for` również powoduje przejście do końca bloku z pominięciem następujących po niej instrukcji.

```
for(initialization; expression; modification){  
    // ...  
    continue;  
    // ...  
}
```

- Dla instrukcji `for` nastąpi:
 - wykonanie wyrażenia modyfikującego `modification`
 - wyznaczenie wartości wyrażenia warunkowego `expression`.
- Kontynuacja pętli uzależniona jest od wartości wyrażenia warunkowego

Instrukcja continue w pętli for 2

Kolejność obliczeń

```
int x;  
for (x=0; x<100; x++) {  
    if (x%2==0) continue;  
  
    printf("%d ", x);  
}
```

Instrukcja continue - porównanie

```
int x;  
for (x=0; x<100;x++) {  
    if (x%2==0) continue;  
    printf("%d ", x);  
}
```

Instrukcja continue
wewnątrz pętli for
pozwala na skrócenie
zapisu i zapewnia
większą czytelność


```
int x =0;  
while (x<100) {  
    if (x%2==0) {  
        x++;  
        continue;  
    }  
    printf("%d ", x);  
    x++;  
}
```

Instrukcja break

Instrukcja `break` umieszczana jest w bloku instrukcji `for`, `while`, `do-while` oraz `switch-case`.

Pozwala na opuszczenie bloku i przejście do następnej instrukcji programu.

```
for(initialization; expression; modification){  
    // ...  
    break;  
    // ...  
}  
// następna instrukcja po zakończeniu bloku
```



Instrukcja break - przykład

Wyznaczanie liczb pierwszych

```
#include <stdio.h>

int main(int argc, char** argv) {
    int number=2,n;

    for(n=0;n<10;){
        int i;
        for(i=2;i<number;i++){
            if(number%i==0)break;
        }
        if(i==number) {
            printf("Liczba pierwsza (%d) : %d\n",n,number);
            n++;
        }
        number++;
    }
    return 0;
}
```

Liczba pierwsza (0): 2
Liczba pierwsza (1): 3
Liczba pierwsza (2): 5
Liczba pierwsza (3): 7
Liczba pierwsza (4): 11
Liczba pierwsza (5): 13
Liczba pierwsza (6): 17
Liczba pierwsza (7): 19
Liczba pierwsza (8): 23
Liczba pierwsza (9): 29

Przykład break i continue

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

int main()
{
    for (;;) {
        int c;
        printf( "Options:\na - option A\n"
               "b - option B\nq - quit\n");

        c=getch();
        if(c=='q') break;
        if(c=='a') {
            printf("Selected: a\n");
            continue;
        }
        if(c=='b') {
            printf("Selected: b\n");
            continue;
        }
        printf("Unknown option\n");
    }
}
```

Instrukcja switch-case

Instrukcja `switch-case` pozwala na wybór sekwencji instrukcji w zależności od wartości wyrażenia sterującego.

- Jeżeli I jest pewnym zbiorem wartości całkowitych, natomiast S zbiorem ciągów instrukcji, instrukcja `switch-case` może być traktowana jako odwzorowanie $I \rightarrow S$.
- Dodatkowy ciąg może być przypisany wszystkim wartościom spoza zbioru I .

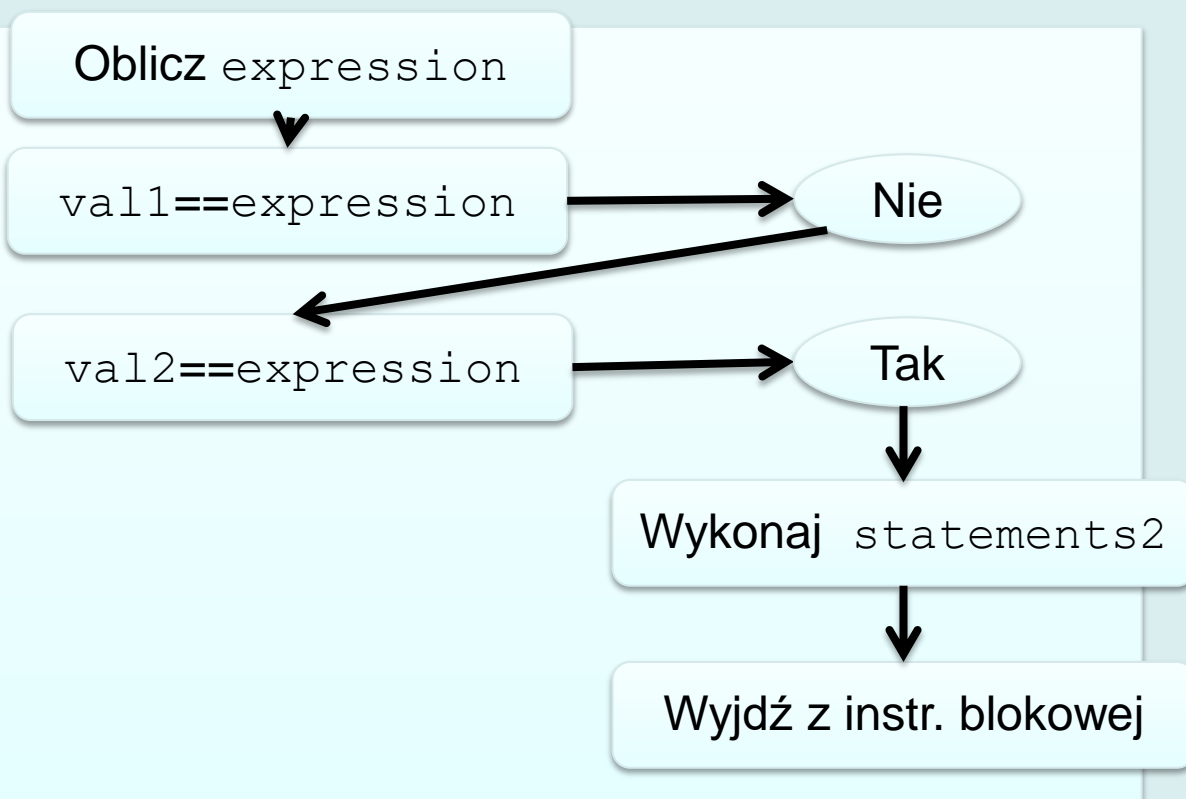
Instrukcja switch-case - składnia

```
switch (expression)
{
    case constant-expression:
        [statements]
        [break; ]
    ...
    [
    default:
        [statements]
        [break; ]
    ]
}
```

- Wyrażenie `expression` jest dowolnym wyrażeniem całkowitego typu; w szczególności może być nazwą zmiennej.
- Etykieta `constant-expression` musi być stałą reprezentowaną przez typ `int` (całkowitoliczbową lub znakową). Wartości etykiet występujące po `case` muszą być unikalne.

Instrukcja switch-case - semantyka

```
switch (expression)
{
    case val1:
        statements1
        break;
    case val2:
        statements2
        break;
    default:
        statements_d
        break;
}
```



- Instrukcja `break` powoduje wyjście z bloku `switch-case` po wykonaniu ciągu instrukcji. W przypadku jej braku nastąpiłoby przejście do następnego ciągu.
- Etykieta `default` rozpoczyna ciąg instrukcji, który będzie wykonany w przypadku, kiedy żadna wartość etykiety `case` nie odpowiada obliczonej wartości wyrażenia `expression`. Jeżeli brak sekcji `default`, wówczas żadna instrukcja z bloku `switch-case` nie zostanie wykonana.

Instrukcja switch-case – przykład 1

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

void main() {
    for(;;) {
        int c;
        printf( "Options:\na - option A\n"
               "b - option B\nq - quit\n");
        c=getch();
        switch(c) {
            case 'q':
                return;
            case 'a':
                printf("Selected: a\n");
                break;
            case 'b':
                printf("Selected: b\n");
                break;
            default:
                printf("Unknown option\n");
                break;
        }
    }
}
```

Instrukcja switch-case – przykład 2

```
int main() {
    for(;;) {
        int c;
        printf( "Options:\na - option A\n"
                "b - option B\nq, x  - quit\n");
        c=getch();
        switch(c) {
            case 'Q':
            case 'X': printf("Use q or x to quit\n");
                     continue; // kontynuacja pętli
            case 'q':
            case 'x': return;
            case 'A': printf("Used capital letter ");
            case 'a': printf("Selected: a\n");
                     break;
            case 'B': printf("Used capital letter ");
            case 'b': printf("Selected: b\n");
                     break;
            default: printf("Unknown option\n");
                    break;
        }
    }
}
```

Wartościom 'Q' i 'X' oraz 'q' i 'x' przypisano te same listy instrukcji

Instrukcja `continue` jest traktowana tak samo, jak umieszczona w bloku `for`

Dla wartości 'A' lub 'B' wypisane jest ostrzeżenie, a następnie opcja jest przetwarzana tak, jakby użyto małych liter 'a', 'b'. (Brak instrukcji `break`.)

Jeszcze raz goto

Jedynym racjonalnym powodem użycia goto jest konieczność opuszczenia, co najmniej dwóch bloków instrukcji (np.: podwójnej pętli lub instrukcji switch-case umieszczonej w pętli)

```
int main() {
    for (;;) {
        int c;
        printf( "Options:\na - option A\n"
                "b - option B\nq - quit\n");

        c=getch();
        switch(c) {
            case 'q':
                goto thankyou;
            case 'a':
                printf("Selected: a\n");
                break;
            case 'b':
                printf("Selected: b\n");
                break;
            default:
                printf("Unknown option\n");
                break;
        }
    }

thankyou:
    printf("Thank you\n");
}
```

Zamiast goto

```
void main()
{
    int end=0;
    for(;!end;) {
        int c;
        printf("Options:\na - option A\n"
               "b - option B\nq - quit\n");

        c=getch();
        switch(c) {
            case 'q': end=1;
                     break;
            case 'a': printf("Selected: a\n");
                     break;
            case 'b': printf("Selected: b\n");
                     break;
            default: printf("Unknown option\n");
                    break;
        }

        }
    printf("Thank you\n");
}
```

Dzięki zastosowaniu dodatkowej zmiennej można uzyskać podobny efekt używając wyłącznie instrukcji strukturalnych.

Czego nie można zrobić?

```
int main(){
    char*ptr="Ala";
    switch(ptr){
        case "Ala":
            printf("%s ma kota\n",ptr);
            break;
        case "Ola":
            printf("%s nie ma kota\n",ptr);
            break;
        default:
            printf("Nie wiem czy %s ma kota\n",ptr);
    }
}
```

error: switch quantity not an integer

```
switch(ptr){
```

^~~

error: case label does not reduce to an integer constant

```
case "Ala":
```

^~~~

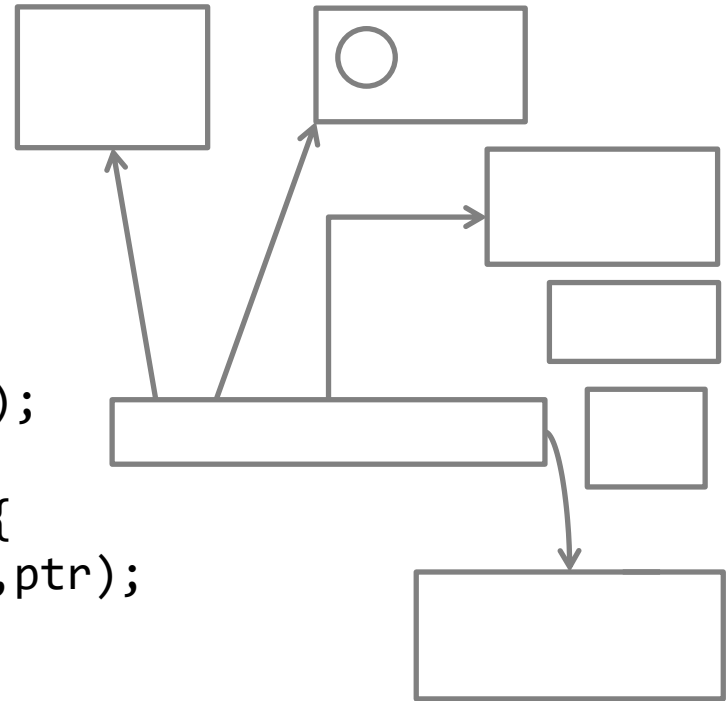
: error: case label does not reduce to an integer constant

```
case "Ola":
```

^~~~

```
#include <time.h>
#include <stdlib.h>
const char*losuj(){
    char*tab[]={ "Ala", "Ola", "Kasia", "Mateusz", "Szymon", "Wojtek" };
    int n = sizeof(tab)/sizeof(tab[0]);
    return tab[rand()%n];
}
int main(){
    srand(time(0));

    for(int i=0;i<10;i++){
        const char*ptr=losuj();
        if(! strcmp(ptr, "Ala")){
            printf("%s ma kota\n", ptr);
        }
        else if (! strcmp(ptr, "Ola")){
            printf("%s nie ma kota\n", ptr);
        }
        else {
            printf("Nie wiem czy %s ma kota\n", ptr);
        }
    }
}
```



Typowe błędy 1

- Typowe błędy związane z instrukcjami sterującymi, to przypadkowo umieszczone średniki lub brak instrukcji blokowej!

```
int main(int argc, char** argv) {  
    int i=0;  
  
    for(i=0;i<1000;i++){  
        if(i==100);break;  
        printf("%d\n",i);  
    }  
    return 0;  
}
```

Nic nie zostanie wypisane

Typowe błędy 2

```
int main(int argc, char** argv) {  
    int i=0,sum_p=0,sum_np=0;  
  
    for(i=0;i<10;i++){  
        if(i%2==0) sum_p=sum_p+i;continue;  
        sum_np=sum_np+i;  
    }  
    printf("Suma liczb parzystych < 10 : %d\n",sum_p);  
    printf("Suma liczb nieparzystych < 10 : %d\n",sum_np);  
    return 0;  
}
```

Suma liczb parzystych < 10 : 20
Suma liczb nieparzystych < 10 : 0

Do zapamiętania

- Składnia instrukcji sterujących przebiegiem programu:
 - `if`
 - `while` i `do-while`
 - `for`
 - `switch-case`
- Semantyka (kolejność wykonania i obliczania wyrażeń)

Dygresja – perfect numbers 1

```
7 int is_perfect(int n){
8     int sum = 1;
9     int i;
10    for(i=2; i*i<n;i++) {
11        if (n%i == 0) {
12            sum+=i;
13            sum+=n/i;
14        }
15        i++;
16    }
17    if (i*i==n)sum+=i;
18    return sum == n;
19 }
```

```
21 void find_perfect(int*tab,int N){
22     tab[0]=0;
23     for(int i=1;i<N;i++){
24         tab[i]=is_perfect(i);
25     }
26 }
27
28 void print_perfect(int*tab,int n){
29     int cnt=0;
30     for(int i=0;i<n;i++){
31         if(tab[i]>0){
32             printf("%d ",i);
33             cnt++;
34         }
35     }
36     printf("\nFound %d numbers\n",cnt);
37 }
```

Dygresja – perfect numbers 2

Wersja sekwencyjna:

- Tworzymy tablicę o rozmiarze N
- Wywołujemy find_perfect(), która wypełnia tablicę wartościami 0 lub 1
- Drukujemy zawartość tablicy

```
39 int main(){
40     int N=36000000;
41     int*tab = (int*)malloc(N*sizeof(int));
42     clock_t start = clock();
43     find_perfect(tab,N);
44     clock_t end = clock();
45     double seconds = (double)(end - start) / CLOCKS_PER_SEC;
46     print_perfect(tab,N);
47     printf("\nt=%f",seconds);
48     free(tab);
49 }
```

```
☞ 1 6 28 496 8128 33550336
   Found 6 numbers
```

```
t=286.523575
```

Dygresja – perfect numbers 3

Wersja równoległa na CUDA

```
--  
22 __global__ void find_perfect(int*tab,int N) {  
23     int tid = blockIdx.x;    // this thread handles the data at its thread id  
24     if (tid >= N) return;  
25  
26     tab[tid]=0;  
27     if(tid==0){  
28         return;  
29     }  
30     tab[tid]=is_perfect(tid);  
31 }
```

- Kernel (funkcja wykonywana przez rdzeń CUDA) find_perfect wykona się tylko dla jednego elementu tablicy przekazanego przez blockIdx.x
- Informację, którym elementem ma się zająć kernel przekazuje środowisko wykonawcze

Dygresja – perfect numbers 4

```
44 int main(){
45     int N=36000000;
46     int*tab = (int*)malloc(N*sizeof(int));
47     int*dev_tab;
48     clock_t start = clock();
49     cudaError_t err = cudaMalloc( (void**)&dev_tab, N * sizeof(int) );
50     if(err!=cudaSuccess){
51         printf( "%s in %s at line %d\n",
52             cudaGetErrorString( err ),__FILE__, __LINE__ );
53         exit( -1 );
54     }
55     find_perfect<<<N,1>>>(dev_tab,N);
```

- Alokujemy pamięć na zwracaną tablicę tab i na urządzeniu (dev_tab)
- Obie mają N elementów

Dygresja – perfect numbers 5

```
57 err = cudaMemcpy( tab, dev_tab, N * sizeof(int),cudaMemcpyDeviceToHost );
58 if(err!=cudaSuccess){
59     printf( "%s in %s at line %d\n",
60         cudaGetErrorString( err ),__FILE__, __LINE__ );
61     exit( -1 );
62 }
63 clock_t end = clock();
64 double seconds = (double)(end - start) / CLOCKS_PER_SEC;
65 print_perfect(tab,N);
66 printf("\nN=%d t=%f",N,seconds);
67 free(tab);
68 }
```

Kopiuujemy wyniki z pamięci urządzenia dev_tab do tab za pomocą cudaMemcpy()

1 6 28 496 8128 33550336
Found 6 numbers

N=36000000 t=15.825811

$286.523575 / 15.825811 = 18.1$

Dygresja – gdzie?

- Colaboratory
- https://github.com/pszwed-ai/wyklad-imperatywne/blob/main/CUDA_perfect_numbers.ipynb