Oj, jak leje i leje.
A tam jest Ala.
I jej lalki i kotki.
Oj, mamo, mamo.
Jak leje, jak leje.
O, moje lalki.
O, kotki moje.

oj • jak • leje • jej • moje

43

Instrukcje sterujące

wer. 13 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki Inżynierii Materiałowej i Biomedycznej

2021-02-28 19:09:35 +0100

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska

Część I

Prosty przykład

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   $Ala\ ma\ i\ kot\{a|y|ów\}$
   gdy i zmienia się w zakresie od 0 do 100.

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   $Ala\ ma\ i\ kot\{a|y|ów\}$
   gdy i zmienia się w zakresie od 0 do 100.
2. Po co?

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   Ala ma i kot{a|y|ów}
   gdy *i* zmienia się w zakresie od 0 do 100.
2. Po co?
   - zapoznanie się z instrukcją **if—then—else**,

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   Ala ma i kot$\{a|y|ów\}$
   gdy *i* zmienia się w zakresie od 0 do 100.
2. Po co?
   - zapoznanie się z instrukcją **if—then—else**,
   - zapoznanie się z ideą rozgałęziania algrytmów,

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   Ala ma i kot$\{a|y|ów\}$
   gdy *i* zmienia się w zakresie od 0 do 100.
2. Po co?
   - zapoznanie się z instrukcją **if—then—else**,
   - zapoznanie się z ideą rozgałęziania algrytmów,
   - a, ponieważ to pierwszy program, zapoznanie się z podstawowymi konstrukcjami programistycznymi
     oraz

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis:
   Ala ma i kot{a|y|ów}
   gdy *i* zmienia się w zakresie od 0 do 100.
2. Po co?
   - zapoznanie się z instrukcją **if—then—else**,
   - zapoznanie się z ideą rozgałęziania algrytmów,
   - a, ponieważ to pierwszy program, zapoznanie się z podstawowymi konstrukcjami programistycznymi
     oraz
   - zapoznanie się z instrukcjami dzielenia całkowitoliczbowego...

Oj, jak leje i leje.
A tam jest Ala.
I jej lalki i kotki.
Oj, mamo, mamo.
Jak leje, jak leje.
O, moje lalki.
O, kotki moje.

oj • jak • leje • jej • moje

Ala ma kota

Zerowa dziesiątka

- Ala ma 0 kotów.
- Ala ma 1 kota.
- Ala ma 2 koty.
- Ala ma 3 koty.
- Ala ma 4 koty.
- Ala ma 5 kotów.
- Ala ma 6 kotów.
- Ala ma 7 kotów.
- Ala ma 8 kotów.
- Ala ma 9 kotów.

Ala ma kota

Zerowa dziesiątka

- Ala ma 0 kotów.
- Ala ma 1 kota.
- Ala ma 2 koty.
- Ala ma 3 koty.
- Ala ma 4 koty.
- Ala ma 5 kotów.
- Ala ma 6 kotów.
- Ala ma 7 kotów.
- Ala ma 8 kotów.
- Ala ma 9 kotów.

Ala ma kota

Pierwsza dziesiątka

- Ala ma 10 kotów.
- Ala ma 11 kotów.
- Ala ma 12 kotów.
- Ala ma 13 kotów.
- Ala ma 14 kotów.
- Ala ma 15 kotów.
- Ala ma 16 kotów.
- Ala ma 17 kotów.
- Ala ma 18 kotów.
- Ala ma 19 kotów.

Ala ma kota

Pierwsza dziesiątka

- Ala ma 10 kotów.
- Ala ma 11 kotów.
- Ala ma 12 kotów.
- Ala ma 13 kotów.
- Ala ma 14 kotów.
- Ala ma 15 kotów.
- Ala ma 16 kotów.
- Ala ma 17 kotów.
- Ala ma 18 kotów.
- Ala ma 19 kotów.

Ala ma kota

Kolejna dziesiątka

- Ala ma 20 kotów.
- Ala ma 21 kotów.
- Ala ma 22 koty.
- Ala ma 23 koty.
- Ala ma 24 koty.
- Ala ma 25 kotów.
- Ala ma 26 kotów.
- Ala ma 27 kotów.
- Ala ma 28 kotów.
- Ala ma 29 kotów.

Ala ma kota

Kolejna dziesiątka

- Ala ma 20 kotów.
- Ala ma 21 kotów.
- Ala ma 22 koty.
- Ala ma 23 koty.
- Ala ma 24 koty.
- Ala ma 25 kotów.
- Ala ma 26 kotów.
- Ala ma 27 kotów.
- Ala ma 28 kotów.
- Ala ma 29 kotów.

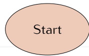
Idea algorytmu — zmienne pomocnicze

Niech zmienna typu **int**, i oznacza liczbę kotów ($0 \leq i \leq 100$).

- Wówczas $i/10$ oznacza numer dziesiątki (cd — cyfra dziesiątek), a
- $i\%10$ oznacza „numer kota w dziesiątce” (cj — cyfra jednostek).

Idea Algorytmu

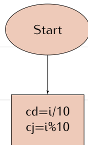
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

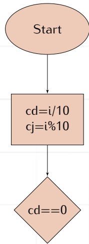
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

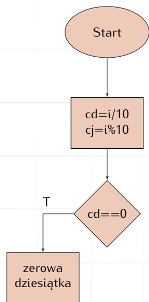
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

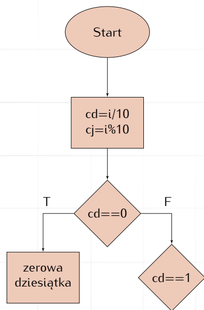
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

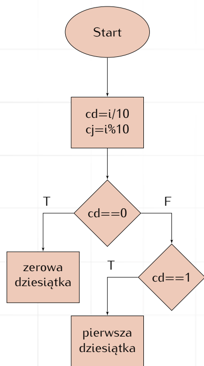
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

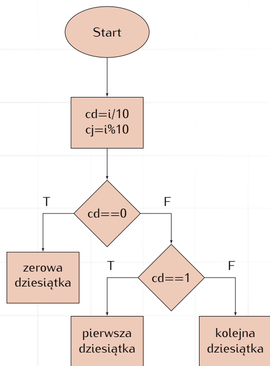
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

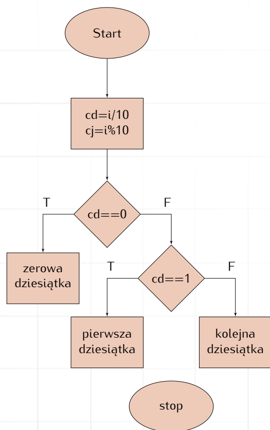
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

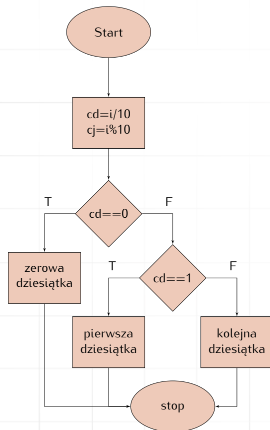
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

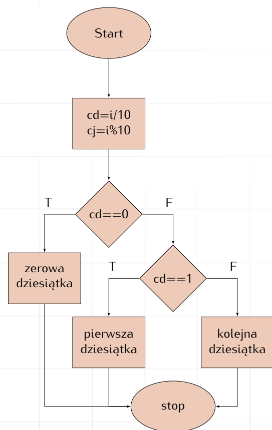
Schemat blokowy i zarys kodu



Rysunek: Schemat blokowy i idea

Idea Algorytmu

Schemat blokowy i zarys kodu




```
cd = i / 10;
cj = i % 10;
printf("Ala ma %d kot", i);
if ( cd == 0 )
{
//    Zerowa dziesiątka
}
else if ( cd == 1 )
{
//    Pierwsza dziesiątka
}
else
{
//    Kolejna dziesiątka
}
```

Rysunek: Schemat blokowy i idea

Idea algorytmu

Zerowa dziesiątka

Przyrostek dla zerowej dziesiątki może być określony następującym „wzorem matematycznym”:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } cj = 0 \cup 4 < cj < 10 \\ \text{a} & \text{gdy } cj = 1 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

Idea algorytmu

Zerowa dziesiątka

Przyrostek dla zerowej dziesiątki może być określony następującym „wzorem matematycznym”:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } cj = 0 \cup 4 < cj < 10 \\ \text{a} & \text{gdy } cj = 1 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

```
//   Zerowa dziesiątka
if (cj == 0 || (4 < cj && cj < 10))
   printf("ow\n");
else if (cj == 1)
   printf("a\n");
else
   printf("y\n");
```

Idea algorytmu

Pierwsza dziesiątka

Na dobrą sprawę nie ma o czym mówić:

```
//    Pierwsza dziesiątka
printf("ow\n");
```

Idea algorytmu

Kolejna dziesiątka

Przyrostek dla każdej następnej dziesiątki może być określony następującym wzorem matematycznym:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } 0 <= cj < 2 \cup 4 < cj < 10 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

Idea algorytmu

Kolejna dziesiątka

Przyrostek dla każdej następnej dziesiątki może być określony następującym wzorem matematycznym:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } 0 <= cj < 2 \cup 4 < cj < 10 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

```
//    Kolejna dziesiątka
if (1 < cj && cj < 5)
    printf("y\n");
else
    printf("ow\n");
```

Program (prawie kompletny) I

```
cd = i / 10;
cj = i % 10;
printf("Ala ma %d kot", i);
if ( cd == 0 )
{
//    Zerowa dziesiątka
    if ( cj == 0 || ( 4 < cj && cj < 10 ) )
        printf("ow\n");
    else if ( cj == 1 )
        printf("a\n");
    else
        printf("y\n");
}
else if ( cd == 1 )
```

Program (prawie kompletny) II

```
//    Pierwsza dziesiątka
    printf("ow\n");
else
{
//    Kolejna dziesiątka
    if ( 1 < cj && cj < 5 )
        printf("y\n");
    else
        printf("ow\n");
}
```

Część II

Teoria

Instrukcje proste

Każde wyrażenie typu `a = b` lub `puts("ala")` staje się instrukcją gdy dodamy na końcu średnik.

Instrukcje złożone I

Grupa instrukcji prostych, zamknięta w bloku i traktowana przez kompilator jak jedna instrukcji (w pewnym sensie!).

```
{
   a = b + c;
   d = e * (f + a);
}
```

Uwaga: W ramach każdego bloku można deklarować zmienne **lokalne**. Ich zawartość nie jest dostępna poza blokiem! Natomiast dostępna jest wartość wszystkich zmiennych zadeklarowanych w nadrzędnym bloku (chyba, że „przykryjemy” je lokalną deklaracją).

Instrukcje złożone II

```
{
    int d = 1;
    printf("%d\n", d);
    {
        int d = 2;
        printf("%d\n", d);
    }
    printf("%d\n", d);
}
```

Co pojawi się na wyjściu programu?

Instrukcje warunkowe

- Wariant „**if**–then”

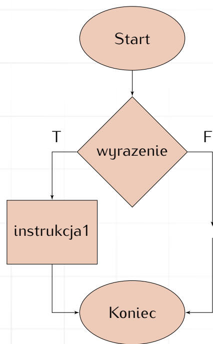
```
if (wyrazenie)
    instrukcja1
```

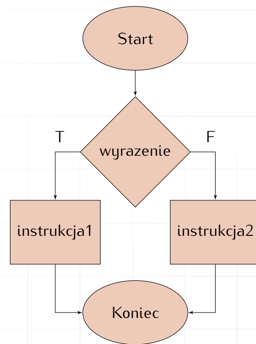
- Wariant „**if**–then–else”

```
if (wyrazenie)
    instrukcja1
else
    instrukcja2
```

Instrukcje warunkowe

Schemat blokowy





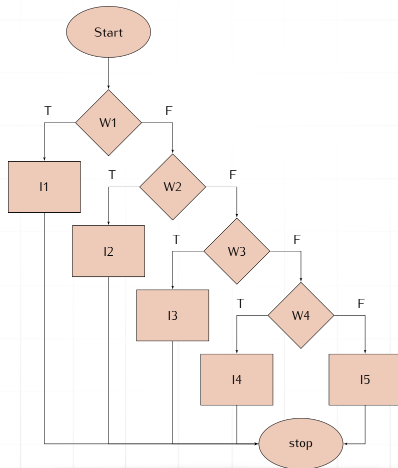
Instrukcje warunkowe

- Wariant „**if**–`then`–**else if**"

```
if (W1)
    I1
else if (W2)
    I2
else if (W3)
    I3
else if (W4)
    I4
else
    I5
```

Instrukcje warunkowe

Schemat blokowy



Instrukcje warunkowe

Uwagi:

1. Wyrażenie warunkowe **musi** być zapisane w nawiasach okrągłych.
2. Słowo „`then`” nie występuje (w odróżnieniu od innych języków programowania).
3. C (w wersji ANSI) właściwie **nie zna** typu logicznego (w odróżnieniu od innych języków programowania).
4. Instrukcja **if** sprawdza numeryczną wartość wyrażenia; zamiast (*wyrazenie!=0*) piszemy (możemy pisać!) (*wyrazenie*).
5. Wyrażenie `(a > b)` ma wartość 1 gdy istotnie a jest większe od b i 0 w przeciwnym razie.

Wyrażenia warunkowe

Wyrażenie

```
if (a > b)
    m = a;
else
    m = b;
```

powoduje wstawienie do m większej z liczb a i b.
Powyższe może być zastąpione wyrażeniem:

```
m = (a > b) ? a : b;
```

Dowcip

Jest taki dowcip (o programistach)

Żona programisty wysyła go do sklepu:

- Idź do sklepu i kup bochenek chleba, jak będą jajka — kup tuzin...

Wraca programista z dwunastoma bochenkami chleba...

Zadanie domowe:
Narysuj schemat blokowy działania programisty oraz schemat blokowy (prawdopodobnych) oczekiwań jego małżonki. Jakie są różnice?

Rozgałęzienia — instrukcja switch I

```
switch (wyrażenie){
   case wyrazenie-stale1: instrukcje1
   case wyrazenie-stale2: instrukcje2
   ...
   case wyrazenie-stale3: instrukcje3
   default: instrukcje
}
```

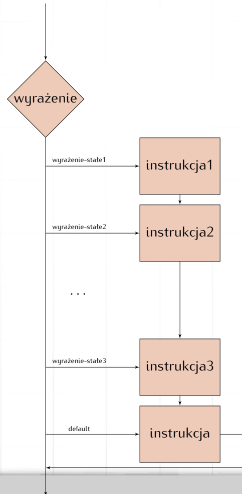
1. Instrukcja **switch** służy do podejmowania decyzji wielowariantowych.

Rozgałęzienia — instrukcja switch II

5. Wszystkie *wyrażenia-stałe* muszą być różne.
6. Przypadek **default** zostanie wykonany gdy *wyrażenie* nie jest zgodne z żadnym przypadkiem.
7. **default** nie jest obowiązkowy: jeżeli nie występuje, a wyrażenie nie jest zgodne z żadnym przypadkiem — nie podejmuje się żadnej akcji.
8. Klauzula **default** może wystąpić na dowolnym miejscu.

case — schemat blokowy

wariant bez break



case

- Jeżeli nie podoba nam się przedstawione działanie (po Instrukcji 1 Wykonywana jest Instrukcja 2 i tak dalej)...

case

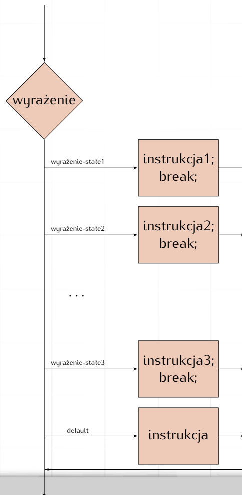
- Jeżeli nie podoba nam się przedstawione działanie (po Instrukcji 1 Wykonywana jest Instrukcja 2 i tak dalej)...
- Powinniśmy dodać instrukcję **break**!

case

- Jeżeli nie podoba nam się przedstawione działanie (po Instrukcji 1 Wykonywana jest Instrukcja 2 i tak dalej)...
- Powinniśmy dodać instrukcję **break**!
- Wówczas schemat blokowy będzie nieco inny.

case — schemat blokowy

(z break)



switch — przykład

```
char keystroke = getch();
switch( keystroke ) {
    case 'a':
    case 'b':
    case 'c':
    case 'd':
        KeyABCDPressed();
        break;
    case 'e':
        KeyEPressed();
        break;
    default:
        UnknownKeyPressed();
        break;
}
```

1. Zwracam uwagę na instrukcje **break** po rozpatrzeniu każdego przypadku!

switch — przykład

```
char keystroke = getch();
switch( keystroke ) {
    case 'a':
    case 'b':
    case 'c':
    case 'd':
        KeyABCDPressed();
        break;
    case 'e':
        KeyEPressed();
        break;
    default:
        UnknownKeyPressed();
        break;
}
```

1. Zwracam uwagę na instrukcje **break** po rozpatrzeniu każdego przypadku!
2. Gdy nie zostanie ona umieszczone — po rozpatrzeniu jednego z przypadków wykonywane będą kolejne instrukcje (z kolejnych przypadków).

switch — przykład

```
char keystroke = getch();
switch( keystroke ) {
    case 'a':
    case 'b':
    case 'c':
    case 'd':
        KeyABCDPressed();
        break;
    case 'e':
        KeyEPressed();
        break;
    default:
        UnknownKeyPressed();
        break;
}
```

1. Zwracam uwagę na instrukcje **break** po rozpatrzeniu każdego przypadku!
2. Gdy nie zostanie ona umieszczone — po rozpatrzeniu jednego z przypadków wykonywane będą kolejne instrukcje (z kolejnych przypadków).
3. Napis **case** (aż do dwukropka) może być traktowany jako etykieta; nie ogranicza wykonywania poleceń.

Pętla while

```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.

Pętla while

```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.

Pętla while

```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.

Pętla while

```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Pętla while

```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Pętla while

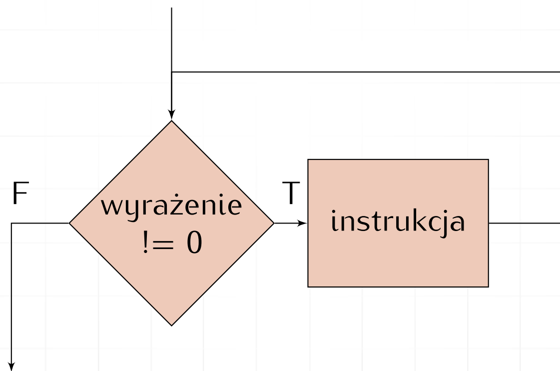
```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Uwaga! Aby pętla się skończyła **coś musi spowodować** zmianę wartości wyrażenia.

Pętla while

Schemat blokowy



Pętla for

Pętla

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

jest równoważna rozwinięciu:

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

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.

Pętla for

Pętla

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

jest równoważna rozwinięciu:

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

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.
2. Najczęściej wyr1 i wyr3 są przypisaniami lub wywołaniami funkcji

Pętla for

Pętla

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

jest równoważna rozwinięciu:

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

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.
2. Najczęściej wyr1 i wyr3 są przypisaniami lub wywołaniami funkcji
3. wyr2 to wyrażenie warunkowe.

Pętla for

Pętla

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

jest równoważna rozwinięciu:

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

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.
2. Najczęściej wyr1 i wyr3 są przypisaniami lub wywołaniami funkcji
3. wyr2 to wyrażenie warunkowe.
4. Każdy ze składników można pominąć — wówczas znika on też z rozwinięcia. Średnik pozostaje!

Pętla for

Przykłady

```
for(i = 1; i < n; i++)
    printf("%d\n", i);
```

Pętla for

Przykłady

```
for(i = 1; i < n; i++)
    printf("%d\n", i);
```

```
i = 1;
while ( i < n ) {
    printf("%d\n", i);
    i++;
}
```

Pętla for

Przykłady

```
for(i = 1; i < n; i++)
    printf("%d\n", i);
```

```
i = 1;
while ( i < n ) {
    printf("%d\n", i);
    i++;
}
```

```
for( ; ; )
    printf("%d\n", i);
```

Pętla do—while

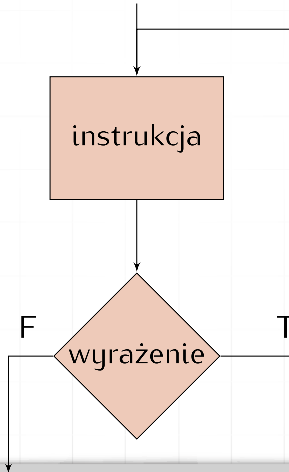
1. Konstrukcja używana stosunkowo najrzadziej:

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

2. Najpierw wykonuje się instrukcję...
3. ...a następnie wyznacza wartość wyrażenia.
4. Pętla jest powtarzana gdy wyrażenie jest prawdziwe, czyli...
5. ...pętla zostanie zatrzymana gdy wyrażenie okaże się fałszywe.

Pętla do—while

Schemat blokowy



Instrukcja **break**

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.

Instrukcja **break**

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.
2. Polecenie może być stosowane w przypadku pętli **while**, **for**, **do** oraz instrukcji **switch**; gdzie indziej jego użycie będzie błędem.

Instrukcja **break**

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.
2. Polecenie może być stosowane w przypadku pętli **while**, **for**, **do** oraz instrukcji **switch**; gdzie indziej jego użycie będzie błędem.
3. W przypadku zagnieżdżonych pętli wyskakujemy tylko jeden poziom wyżej.

Instrukcja **break**

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.
2. Polecenie może być stosowane w przypadku pętli **while**, **for**, **do** oraz instrukcji **switch**; gdzie indziej jego użycie będzie błędem.
3. W przypadku zagnieżdżonych pętli wyskakujemy tylko jeden poziom wyżej.

Instrukcja break

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.
2. Polecenie może być stosowane w przypadku pętli **while**, **for**, **do** oraz instrukcji **switch**; gdzie indziej jego użycie będzie błędem.
3. W przypadku zagnieżdżonych pętli wyskakujemy tylko jeden poziom wyżej.

```
while( x < 100 ) {
  if( x < 0 )
    break;
  printf(" %d \n", x);
  x++;
}
```

W przypadku gdy x jest mniejsze od zera — nie realizujemy pętli.

Instrukcja **continue**

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.

Instrukcja **continue**

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.
2. Może być stosowana wyłącznie wewnątrz pętli!

Instrukcja **continue**

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.
2. Może być stosowana wyłącznie wewnątrz pętli!
3. Powoduje przerwanie przetwarzania bieżącego kroku pętli i przejście do kroku następnego.

Instrukcja **continue**

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.
2. Może być stosowana wyłącznie wewnątrz pętli!
3. Powoduje przerwanie przetwarzania bieżącego kroku pętli i przejście do kroku następnego.

Instrukcja **continue**

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.
2. Może być stosowana wyłącznie wewnątrz pętli!
3. Powoduje przerwanie przetwarzania bieżącego kroku pętli i przejście do kroku następnego.

```
for(i = 0; i < n; i++){
    if (a[i] < 0) /* pomiń element ujemny */
        continue;
    ...       /* przetwarzaj element nieujemny */
```

W przypadku gdy element tablicy jest ujemny — pomijamy przetwarzanie.

Instrukcja skoku I

1. Język C oferuje instrukcję skoku **goto** (pisane **bez** odstępu!) oraz etykiety pozwalające oznaczyć różne miejsca programu.
2. Instrukcja skoku formalnie **nie** jest potrzebna.
3. W praktyce, **prawie** zawsze można się bez niej obejść.
4. Idea programowania strukturalnego sugeruje, żeby z niej nie korzystać.
5. Czasami zdarzają się sytuacje (awaryjne!), gdzie zastosowanie instrukcji skoku może być bardzo przydatne.

Przykład:

Instrukcja skoku II

```
...
if (warunek)
  goto error; /* Skok do obsługi błędow */
...
...
error:
  /* Jakis komunikat o bledzie lub próba
     naprawy sytuacji */
```

Część III

Przykład algorytmu z użyciem instrukcji **goto**

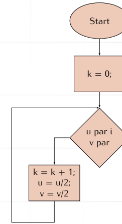
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.

Algorytm B

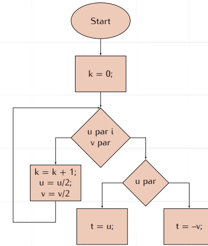
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.



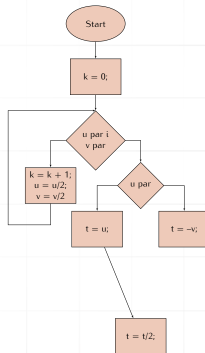
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.



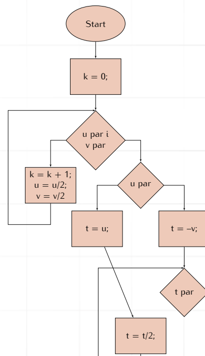
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.



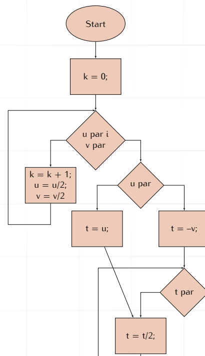
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.



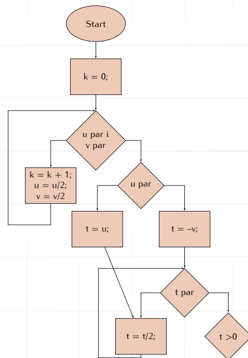
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.



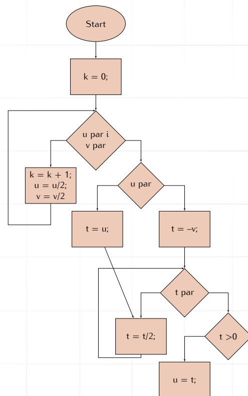
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.



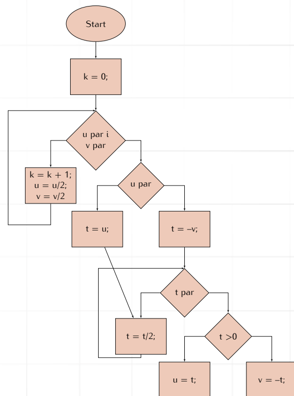
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.



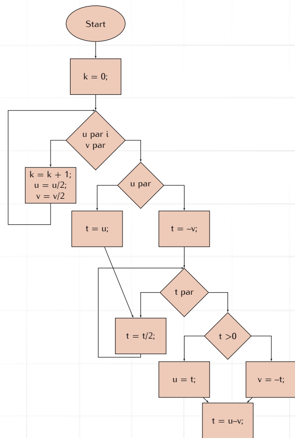
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.



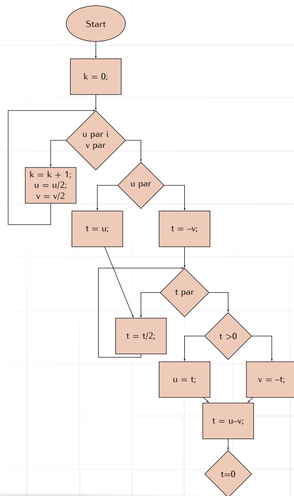
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3.
   W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.



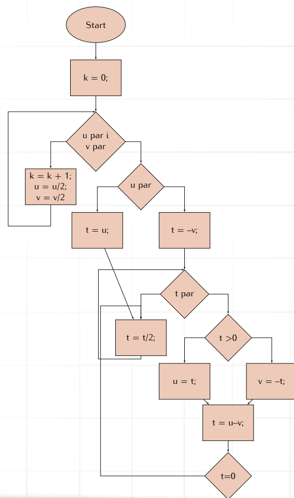
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.



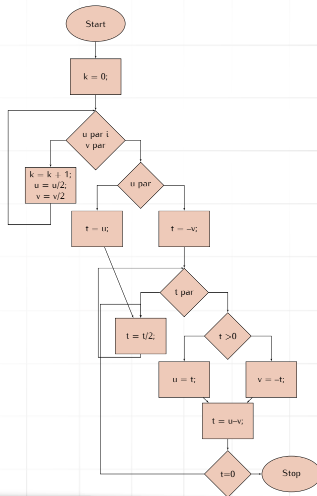
Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.



Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.
5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.



Podsumowanie

Instrukcje warunkowe

1. Wariant „if–then”

```
if (wyrazenie)
    instrukcja1
```

2. Wariant „if–then–else”

```
if (wyrazenie)
    instrukcja1
else
    instrukcja2
```

Podsumowanie

Rozgałęzienia — instrukcja switch

```
switch (wyrazenie){
   case wyrazenie-stale1: instrukcje1
   case wyrazenie-stale2: instrukcje2
   ...
   case wyrazenie-stale3: instrukcje3
   default: instrukcje
}
```

1. Instrukcja `switch` służy do podejmowania decyzji wielowariantowych.
2. Sprawdza się czy wartość wyrażenia pasuje do jednej z kilku stałych wartości.
3. Wszystkie *wyrażenia-stałe* muszą być różne.
4. Przypadek **default** zostanie wykonany gdy *wyrażenie* nie jest zgodne z żadnym przypadkiem.
5. **default** nie jest obowiązkowy: jeżeli nie występuje, a wyrażenie nie jest zgodne z żadnym przypadkiem — nie podejmuje się żadnej akcji.

Podsumowanie

Pętla while

```
while (wyrazenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Podsumowanie

Pętla while

```
while (wyrazenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Uwaga! Aby pętla się skończyła **coś musi spowodować** zmianę wartości wyrażenia.

Podsumowanie

Pętla

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

jest równoważna rozwinięciu:

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

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.
2. Najczęściej wyr1 i wyr3 są przypisaniami lub wywołaniami funkcji
3. wyr2 to wyrażenie warunkowe.
4. Każdy ze składników można pominąć — wówczas znika on też z rozwinięcia. Średnik pozostaje!

Podsumowanie

Pętla do—while

1. Konstrukcja używana stosunkowo najrzadziej:

```
do
    instrukcja
while (wyrazenie)
```

2. Najpierw wykonuje się instrukcję...
3. ...a następnie wyznacza wartość wyrażenia.
4. Pętla jest powtarzana gdy wyrażenie jest prawdziwe, czyli...
5. ...pętla zostanie zatrzymana gdy wyrażenie okaże się fałszywe.

Podsumowane

Pozostałe

```
etykieta:
    goto etykieta;
```

Wyłącznie w pętli (**while**, **do-while**, **for**) i w instrukcji **switch**:

```
    continue;
    break;
```