Politechnika Wrocławska

Bardzo szybkie podsumowanie: wykład 3

wer. 9 z drobnymi modyfikacjami!

Wojciech Myszka

2022-03-22 17:00:08 +0100

HR EXCELLENCE IN RESEARCH

Uwagi

1. Obowiązuje cały materiał!
2. Tu tylko podsumowanie.

Instrukcje sterujące

wer. 13 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki Inżynierii Materiałowej i Biomedycznej

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

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...

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

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.

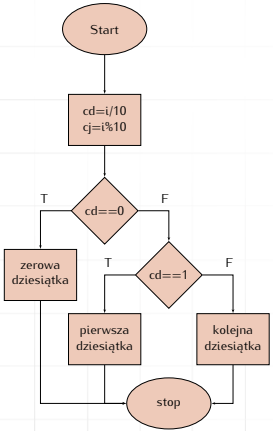
Idea algorytmu — zmienne pomocnicze

Niech zmienna typy **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




```
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”:

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

```
//   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:

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

```
//   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");
}
```

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 warunkowe

- Wariant „**if**–then”

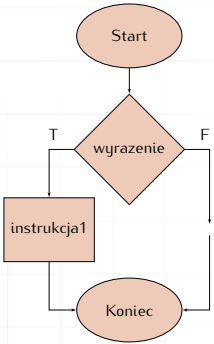
```
if (wyrazenie)
   instrukcja1
```

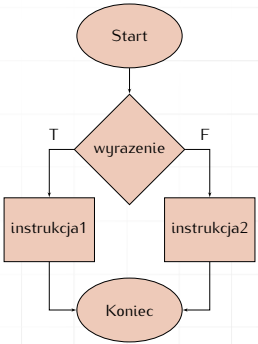
- Wariant „**if**–then–else”

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

Instrukcje warunkowe

Schemat blokowy





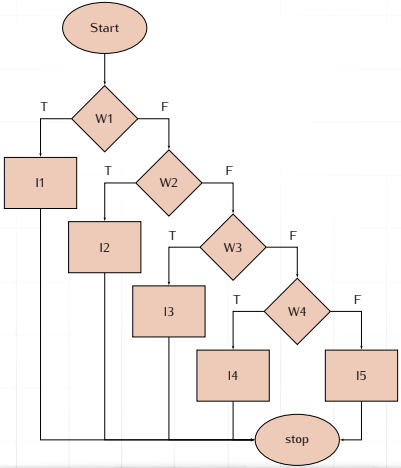
Instrukcje warunkowe

- 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;
```

Rozgałęzienia — instrukcja switch I

```
switch (wyrażenie){
   case wyrazenie_stale1: instrukcje1
   case wyrazenie_stale2: instrukcje2
   ...
   case wyrazenie_stale3: instrukcje3
   default: instrukcje
}
```

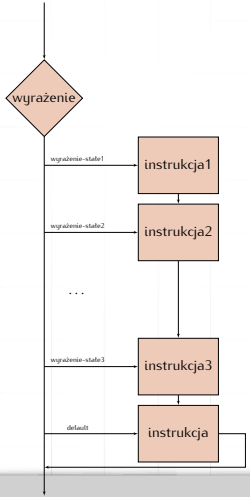
1. Instrukcja **switch** służy do podejmowania decyzji wielowariantowych.

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

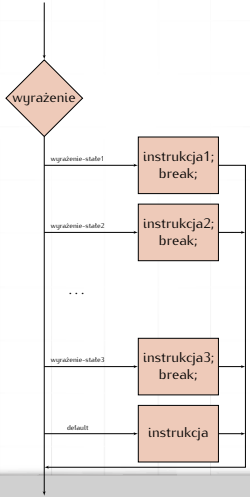


case

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

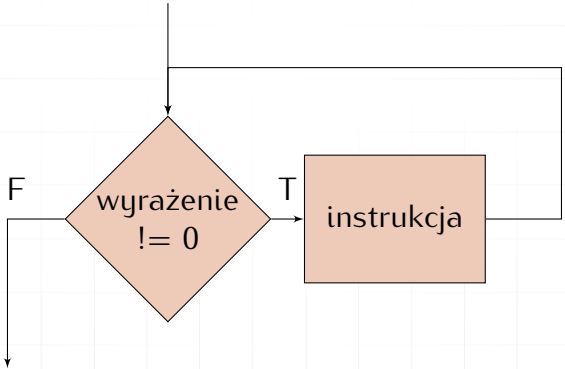
```
while (wyrażenie)
    instrukcja
```

1. Najpierw oblicza się wyrażenie.
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.
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);
```

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

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

Pętla do—while

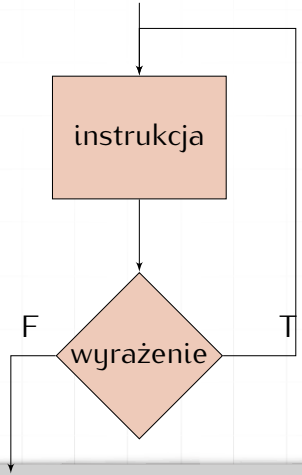
1. Konstrukcja używana stosunkowo najrzadziej:

```
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.
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**.
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.

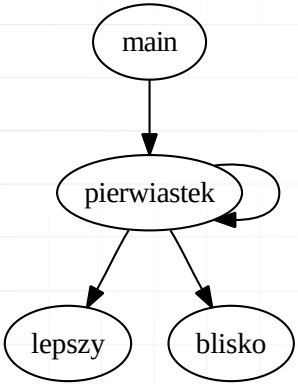
Przykład:

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**



Funkcje

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2021-04-25 10:11:06 +0200

Funkcje

1. Funkcje to sposób na podzielenie dużego programu na mniejsze, łatwiejsze w zarządzaniu fragmenty.
2. Odpowiedni (umiejętny) podział programu na moduły (funkcje) pozwala na powtórne (i wielokrotne) wykorzystanie ich w innych programach.
3. „Ukrycie" pewnych fragmentów pod postacią funkcji pozwala na uproszczenie struktury programu i uczynienie jej bardziej czytelną.
4. Funkcje to, wreszcie, podstawa programowania strukturalnego.
5. Praktycznie każdy język programowania wyposażony jest w mechanizmy podziału na moduły oraz tworzenia funkcji (i procedur).
6. W matematyce pod pojęciem funkcji rozumiemy twór, który pobiera pewną liczbę argumentów i zwraca wynik. Jeśli dla przykładu weźmiemy funkcję $\sin(x)$ to x będzie zmienną rzeczywistą, która określa kąt, a w rezultacie otrzymamy inną liczbę rzeczywistą — sinus tego kąta.

Budowa funkcji

Definicja funkcji wygląda w sposób następujący

```
typ_powrotu nazwa_funkcji ( deklaracja parametrów )
{
   deklaracje i instrukcje
}
```

1. Funkcja **musi** być *zadeklarowana* przed pierwszym jej użyciem!
2. Funkcja zwraca wartość będącą wynikiem jej działania. **Typ** zwracanej wartości zdefinowany jest podczas deklaracji funkcji i oznaczony tu jako **typ powrotu**.
3. Funkcję wywołuje się najczęściej w następujący sposób:

   ```
   a = nazwa_funkcji( parametry funkcji );
   ```

   zwłaszcza gdy zależy nam na zapamiętaniu, lub dalszym przetwarzaniu, wyniku zwracanego przez funkcję. Gdy nie jest on potrzebny (istotny) lub funkcja nie zwraca żadnych wyników można wykonać tak:

   ```
   nazwa_funkcji( parametry funkcji );
   ```

   (W ten sposób najczęściej wywoływana jest funkcja `printf`)

Budowa funkcji

Definicja funkcji wygląda w sposób następujący

```
typ_powrotu nazwa_funkcji ( deklaracja parametrów )
{
   deklaracje i instrukcje
}
```

1. Funkcja **musi** być *zadeklarowana* przed pierwszym jej użyciem!
2. Funkcja zwraca wartość będącą wynikiem jej działania. **Typ** zwracanej wartości zdefinowany jest podczas deklaracji funkcji i oznaczony tu jako **typ powrotu**.
3. Funkcję wywołuje się najczęściej w następujący sposób:

   ```
   a = nazwa_funkcji( parametry funkcji );
   ```

   zwłaszcza gdy zależy nam na zapamiętaniu, lub dalszym przetwarzaniu, wyniku zwracanego przez funkcję. Gdy nie jest on potrzebny (istotny) lub funkcja nie zwraca żadnych wyników można wykonać tak:

   ```
   nazwa_funkcji( parametry funkcji );
   ```

   (W ten sposób najczęściej wywoływana jest funkcja `printf`)
4. Jeżeli funkcja zwraca jakąś wartość wśród jej instrukcji powinno znaleźć się polecenie

   ```
   return wyrażenie;
   ```

   powoduje ona, że wartość wyrażenia przypisywana jest jako wartość funkcji!

Budowa funkcji

Najprostsza funkcja

```
dummy  ( )
{
}
```

- Funkcja nie ma parametrów.
- Funkcja nie zwraca żadnej wartości.
- Funkcja „nic nie robi”

Użycie:

```
dummy ( ) ;
```

Program z funkcją

```
void dummy(void)
{
    glupia();
}

void glupia(void)
{}

int main()
{
    dummy();
    return 0;
}
```

Program z funkcją

```
void glupia(void)
{}

void dummy(void)
{
   glupia();
}

int main()
{
   dummy();
   return 0;
}
```

Funkcje zagnieżdżone

```
int main( void )
{
    void dummy( void )
    {
        void glupia( void ) { }
        glupia();
    }
    dummy();
    return 0;
}
```

Argumenty funkcji

1. Argumenty funkcji służą do przekazania informacji z zewnątrz do jej wnętrza.
2. Według standardu ANSI C typ argumentów musi być zadeklarowany. W definicji funkcji zapisuje się to tak:

```
typ identyfikator (typ1 arg1, typ2 arg2, typn argn)
{
  /* instrukcje */
}
```

Na przykład:

```
int iloczyn ( int x, int y )
{
  int iloczyn_xy;
  iloczyn_xy = x * y;
  return iloczyn_xy;
}
```

```
int iloczyn ( int x, int y )
{
  return x * y;
}
```

3. Argumenty funkcji użyte podczas wywołania funkcji są kopiowane do odpowiednich zmiennych zadeklarowanych w definicji funkcji. Oznacza to, że jakiekolwiek modyfikacje tych argumentów nie mają wpływu na wartości zmiennych (czy wyrażeń) w wywołaniu funkcji. Mówi się, że argumenty są przekazywane przez wartość, czyli wewnątrz funkcji operujemy tylko na ich kopiach.

Argumenty funkcji

1. Funkcja nie musi mieć argumentów.

```
int smieszna()
{
   return 7;
}
```

```
int smieszna(void)
{
   return 7;
}
```

2. W takim wypadku wywołanie funkcji ma postać:

```
a = smieszna();
```

Nawiasy muszą być nawet jak nie ma argumentów!

Argumenty funkcji

1. Funkcja nie musi mieć argumentów.

```
int smieszna()
{
   return 7;
}
```

```
int smieszna(void)
{
   return 7;
}
```

2. W takim wypadku wywołanie funkcji ma postać:

```
a = smieszna();
```

Nawiasy muszą być nawet jak nie ma argumentów!

Wynik wykonania funkcji

1. Funkcja (na ogół) zwraca jakieś wyniki.
2. Do przekazania wyników na zewnątrz funkcji służy instrukcja **return**.
3. Program wywołujący może zignorować zwrócone wyniki.
4. Gdy funkcja nie zwraca wyników nazywana bywa procedurą.

„Procedury” — kilka uwag

1. Każda procedura (jak i funkcja) powinna być zadeklarowana przed pierwszym jej użyciem.
2. Deklaracja to **definicja** albo **prototyp** (a definicja później).
3. Bardzo wiele procedur systemowych deklarowanych jest w plikach nagłówkowych (o rozszerzeniu **.h**).
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:
   - wczytanie plików nagłówkowych,
   - definicje wszystkich procedur,
   - program główny.

„Procedury” — kilka uwag

1. Każda procedura (jak i funkcja) powinna być zadeklarowana przed pierwszym jej użyciem.
2. Deklaracja to **definicja** albo **prototyp** (a definicja później).
3. Bardzo wiele procedur systemowych deklarowanych jest w plikach nagłówkowych (o rozszerzeniu **.h**).
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:
   - wczytanie plików nagłówkowych,
   - definicje wszystkich procedur,
   - program główny.

Deklaracja funkcji (prototyp) wygląda (jakoś) tak:
typ nazwa (parametry i ich typ);

Definicje i deklaracje lokalne

1. Każda zmienna musi być zadeklarowana.
2. Zmienna dostępna jest tylko w bloku, w którym została zadeklarowana (i wszystkich blokach w nim zawartych. Są to zmienne lokalne.
3. Uwaga: blok to zazwyczaj wszystko co się znajduje wewnątrz nawiasów klamrowych **{ }**
4. Deklaracje w blokach niższych „przysłaniają” deklaracje z bloków wyższego poziomu.
5. Po wyjściu z bloku zmienne lokalne „znikają”. Są niedostępne, a ich zawartość jest zapominana.
6. Po powrocie do bloku **nie ma dostępu** do poprzedniej wartości zmiennej!

Definicje i deklaracje globalne

1. Zmienne zadeklarowane na zewnątrz wszystkich modułów (funkcje, procedury, funkcja **main**) nazywane są zmiennymi globalnymi.
2. Zmienne globalne dostępne są we wszystkich blokach...
3. ...chyba, że zostaną przysłonięte przez definicją lokalną.

Czym się różni?

```
int i;
int main(void)
{
    return 1;
}
```

Funkcja main

1. Każdy program w języku C musi mieć segment główny.
2. Segment główny musi nazywać się **main**...
3. ...i jest funkcją!
4. Wartość, którą zwraca funkcja main przekazywana jest do systemu operacyjnego.
5. Wartość ta zazwyczaj informuje czy program zakończył się z błędami i, czasami, o typie (rodzaju) błędu.
6. Standardowe kody zakończenia programu zdefiniowane są w pliku nagłówkowym **stdlib.h** są to

```
#define     EXIT_FAILURE    1    /* Failing exit status.  */
#define     EXIT_SUCCESS    0    /* Successful exit status.  */
```

Funkcja main

1. Każdy program w języku C musi mieć segment główny.
2. Segment główny musi nazywać się **main**...
3. ...i jest funkcją!
4. Wartość, którą zwraca funkcja main przekazywana jest do systemu operacyjnego.
5. Wartość ta zazwyczaj informuje czy program zakończył się z błędami i, czasami, o typie (rodzaju) błędu.
6. Standardowe kody zakończenia programu zdefiniowane są w pliku nagłówkowym **stdlib.h** są to

```
#define     EXIT_FAILURE    1    /* Failing exit status.  */
#define     EXIT_SUCCESS    0    /* Successful exit status.  */
```

7. Każdy segment główny powinien się kończyć poleceniem **return**.

Rekurencja

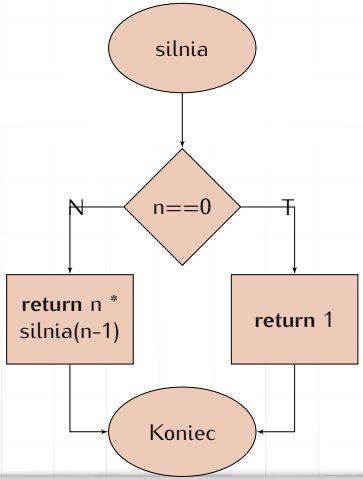
1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.
2. Nie potrafię powiedzieć, czy rekurencja to dobra czy zła technika programowania.
3. Rekurencja była bardzo pożyteczna podczas tworzenia algorytmów.
4. W realizacjach programowych (zwłaszcza bardzo skomplikowanych problemów) stwarza wiele kłopotów.

Rekurencja

1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.
2. Nie potrafię powiedzieć, czy rekurencja to dobra czy zła technika programowania.
3. Rekurencja była bardzo pożyteczna podczas tworzenia algorytmów.
4. W realizacjach programowych (zwłaszcza bardzo skomplikowanych problemów) stwarza wiele kłopotów.
5. Problemy wynikają z konieczności przechowania wszystkich argumentów i całej struktury danych używanej przez funkcję gdy wywołuje ona samą siebie.

Rekurencja

Silnia — schemat blokowy



Rekurencja

Silnia

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

float silnia(int n)
{
    if (n == 0)
    return 1.;
    else
    return n * silnia(n-1);
}
int main(int cnt, char ** arg)
{
    int n;
    n=atol( arg[1] );
    printf("%d! = %g\n", n, silnia(n));
    return 0;
}
```

Ciąg Fibonacciego

Rekurencja

$$F_n := \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

Ciąg Fibonacciego

Rekurencja

```
#include <stdio.h>
unsigned long int k;
unsigned long int fib(int n)
{
    k++;
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv)
{
    int n, m;
    for ( n = 0; n < 100; n++ )
    {
        k = 0;
        m = fib(n);
        printf("%lu, %lu, %lu\n", n, m, k);
    }
    return 0;
}
```

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Załóżmy, że mamy wyznaczyć pierwiastek stopnia n z liczby w, czyli znaleźć taką liczbę x, że:

$$x^n = w \tag{1}$$

lub inaczej:

$$x^n - w = 0 \tag{2}$$

Jeżeli oznaczymy $f(x) = x^n - w$ to zadanie to można zapisać ogólniej: należy znaleźć takie x, że $f(x) = 0$.

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Jeżeli zadanie dodatkowo uprościmy zakładając:

- funkcja ma dokładnie jedno miejsce zerowe,
- jest różniczkowalna,
- jej pochodna w całym przedziale jest albo dodatnia albo ujemna;

to możemy naszkicować następujący rysunek ilustrujący nasze zadanie:

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Zaczynamy w punkcie g; wartość funkcji w tym punkcie wynosi $f(g)$. Musimy w jakiś sposób zdecydować w którym kierunku należy wykonać następny krok. Zauważmy, że możemy w tym celu wykorzystać pochodną (czerwona, przerywana linia na poprzednim rysunku). Jeżeli przybliżymy funkcję za pomocą pochodnej (stycznej do funkcji, przechodzącej przez punkt $(g, f(g))$ to następnym przybliżeniem będzie punkt przecięcia stycznej z osią x.

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Z równania prostej mamy:

$$\frac{f(g) - 0}{g - g'} = f'(g) \tag{3}$$

czyli

$$\frac{f(g)}{f'(g)} = g - g' \tag{4}$$

i dalej

$$g' = g - \frac{f(g)}{f'(g)} \tag{5}$$

Idea programowania strukturalnego

Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Jeżeli zauważymy, że $f(x) = x^n - w$ oraz, że $f'(x) = nx^{n-1}$ to kolejne przybliżenie wyliczane będzie ze wzoru:

$$g' = g - \frac{g^n - w}{ng^{n-1}} \tag{6}$$

albo

$$g' = \frac{ng^n - g^n + w}{ng^{n-1}} = \frac{(n-1)g^n + w}{ng^{n-1}} = \frac{1}{n}\left((n-1)g + \frac{w}{g^{n-1}}\right) \tag{7}$$

Gdy $n = 2$, wówczas

$$g' = \frac{1}{2}\left(g + \frac{w}{g}\right). \tag{8}$$

Umawiamy się, że program kończy pracę gdy kolejna poprawka g' nie różni się zbytnio od poprzednio wyliczonej wartości g, czyli $|g - g'| < \varepsilon$.

Idea programowania strukturalnego

Realizacja programowa

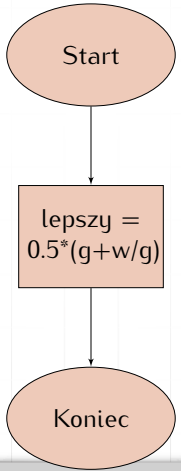
Program będzie się składał z trzech części:

1. `blisko(g, gprim)` — funkcja o wartościach logicznych sprawdzająca czy $|g - g'| \leq \varepsilon$,
2. `lepszy(n, w, g)` — funkcja rzeczywista wyliczająca następne, lepsze przybliżenie pierwiastka,
3. `pierwiastek(n, w, g)` — funkcja (rzeczywista) wyliczająca pierwiastek stopnia n z w zaczynając od przybliżenia g.

Uwaga: Dalszy przykład zakłada $n = 2$

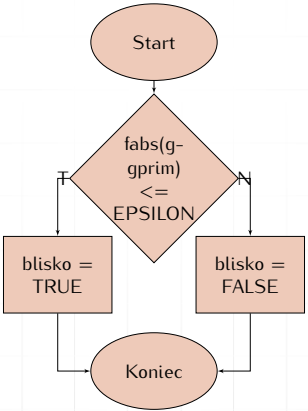
Realizacja programowa

`lepszy(w, g)`



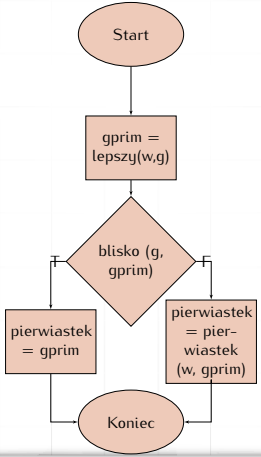
Realizacja programowa

`blisko(g, gprim)`



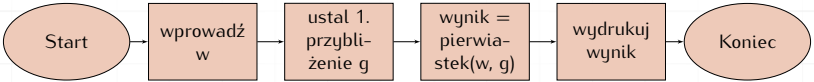
Realizacja programowa

`pierwiastek(w, g)`



Realizacja programowa

Program główny



Metoda Newtona

Realizacja programowa

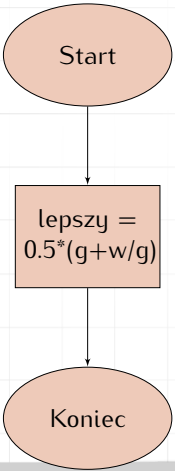
Program składa się z trzech części:

1. `blisko(g, gprim)` — funkcja o wartościach logicznych sprawdzająca czy $|g - g'| \leq \varepsilon$,
2. `lepszy(n, w, g)` — funkcja rzeczywista wyliczająca następne, lepsze przybliżenie pierwiastka,
3. `pierwiastek(n, w, g)` — funkcja (rzeczywista) wyliczająca pierwiastek stopnia n z w zaczynając od przybliżenia g.

Uwaga: Dalszy przykład zakłada n = 2

Realizacja programowa

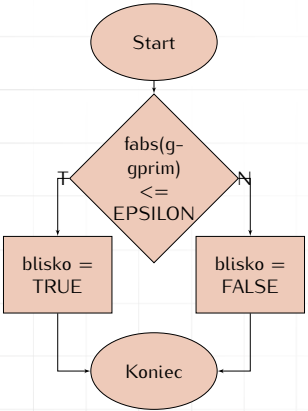
`lepszy(w, g)`




```
double lepszy(double w, double g)
{
    return 0.5 * (g + w/g);
}
```

Metoda Newtona

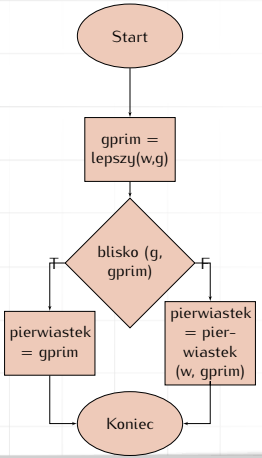
`blisko(g, gprim)`




```
int blisko(double g,          \
           double gprim)
{
    return fabs(g - gprim) \
                < EPSILON;
}
```

Metoda Newtona

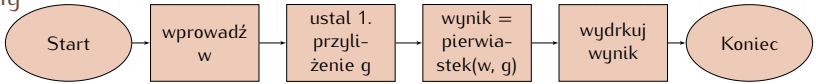
`pierwiastek(w, g)`




```
double pierwiastek(double w, \
                   double g)
{
    double gprim;
    gprim = lepszy(w, g);
    if ( blisko(g, gprim) )
    return gprim;
    else
    return pierwiastek(w, \
                       gprim);
}
```

Metoda Newtona

Program główny




```
int main(void)
{
    double w, g, wynik;
    w = 2.;
    g = 1.;
    wynik = sqrtf(w);
    printf("%f\n", wynik);
    wynik = pierwiastek(w, g);
    printf("Pierwiastek kwadratowy z liczby" \
           " %f wynosi %f\n", w, wynik);
    return 0;
}
```

Tablice (jedno i wielowymiarowe), łańcuchy znaków

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2022-03-25 07:28:03 +0100

Zmienne

Przypomnienie/podsumowanie

1. Wszystkie zmienne muszą być zadeklarowane.
2. Nazwa zmiennej składa się z liter i cyfr, a rozpoczyna się literą; znak podkreślenia zalicza się do liter.
3. Nazwy zmiennych nie powinny się zaczynać od znaku podkreślenia (tak nazywają się zmienne systemowe).
4. Deklaracja obowiązuje wewnątrz bloku (i we wszystkich blokach znajdujących się „niżej").
5. W C występują zmienne globalne (zewnętrzne) i lokalne.
6. Deklaracja lokalna przysłania deklarację globalną (jeżeli nawa zmiennej jest taka sama).

W szczególności...

...poniższa konstrukcja

```
for(int i = 0; i < 10; i++)
{
    printf("i = %d\n), i);
}
```

jest poprawna.

Ale poniższa

```
for(int i = 0; i < 10; i++)
{
    printf("i = %d\n), i);
}
printf("i = %d\n), i);
```

już nie...

...gdyz zmienna i jest zmienną lokalną tylko dla pętli!

Zmienne zewnętrzne i wewnętrzne

```
#include <stdio.h>
int a; // <—— Zmienna zewnetrzna
int main(void)
{
      int b; // <—— Zmienna wewnetrzna
      ...
```

Zmienne zewnętrzne nazywane bywają zmiennymi „globalnymi” (czyli dostępnymi dla każdej funkcji programu).

Zmienne statyczne i automatyczne

Trochę zamętu

1. Dodatkowo można zażądać od zmiennej żeby była „statyczna” (co deklaruje się dodając słowo kluczowe **static** przed nazwą typu).

```
static int x;
```

2. Zmienna statyczna zewnętrzna pozostaje zdefiniowana **tylko** dla funkcji zdefiniowanych w jednym pliku źródłowym (ukryta jest dla funkcji z innych plików źródłowych).
3. Zmienna statyczna wewnętrzna zachowuje swoją wartość pomiędzy kolejnymi wywołaniami funkcji.
4. Zmienne, które nie są statyczne **nie muszą** zachowywać wartości między wejściami do funkcji (ale mogą) — nie można na to liczyć!
5. Dodatkowo zmienne statyczne wewnętrzne inicjowane są na wartość zero (jeżeli programista nie zażąda żeby było inaczej).

Zmienne statyczne i automatyczne

```c
#include <stdio.h>
void f(void)
{
    static int x ; /* zmienna statyczna */
    int y = 0;     /* zmienna automatyczna */
    x++;
    y++;
    printf("X=%d, Y=%d\n", x, y);
}
int main()
{
    f();
    f();
    f();
    return 0;
}
```

Tablice

1. Gdy potrzebujemy przechować kilka zmiennych tego samego typu (i jakoś powiązanych ze sobą) stosujemy **tablicę**.
2. Tablica to ciąg zmiennych o tej samej nazwie; dostęp do poszczególnych elementów odbywa się przez podanie numeru zmiennej (indeksu/ów).

3. Elementy numerowane są **począwszy od zera**.
4. Deklaracja wygląda tak:
   typ nazwa_tablicy[rozmiar];

Tablice

1. Tablica jest zmienną złożoną (strukturą pewnego rodzaju).
2. Słuzy do przechowywania danych tego samego typu.
3. Jeżeli chcemy nadać elementom tablicy wartości początkowe
   int tablica[3] = {1,2,3};
4. To jest również poprawna deklaracja:
   int tablica[20] = {1,};
   (pierwszy element tablicy ma wartość 1, pozostałe mają wartość 0)
5. Nie zawsze trzeba podawać rozmiar tablicy — czasami kompilator może się domyślić sam:
   int tablica[] = {1, 2, 3, 4, 5};
   zostanie zadeklarowana tablica o pięciu elementach.

Wielkość tablic

```
1 #include<stdio.h>
2 int main(void)
3 {
4     int t[] = {1, 2, 3, 4, };
5     int i;
6     for (i = -1; i < 7; i++)
7         printf("t[%d] = %d\n", i, t[i]);
8     return 0;
9 }
```

Wielkość tablic

```
1 #include<stdio.h>
2 int main(void)
3 {
4     int t[] = {1, 2, 3, 4, };
5     int i;
6     for (i = -1; i < 7; i++)
7         printf("t[%d] = %d\n", i, t[i]);
8     return 0;
9 }
```

Ile elementów ma tablica t?

Wykonany po raz pierwszy

```
t[-1] = 11131
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -1296194160
t[5] = 32767
t[6] = 0
```

Wykonany po raz drugi

```
t[-1] = 10955
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -868000288
t[5] = 32767
t[6] = 0
```

Wykonany po raz trzeci

```
t[-1] = 11015
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -143761264
t[5] = 32767
t[6] = 0
```

Inicjowanie I

1. W deklaracji obiektu można zawrzeć wartość początkową deklarowanego identyfikatora.
2. Inicjator, który poprzedza się operatorem = jest albo wyrażeniem, albo listą inicjatorów zawartą w nawiasach klamrowych.
3. Lista może kończyć się przecinkiem.
4. Dla obiektów i tablic statycznych wszystkie wyrażenia w inicjatorach muszą być wyrażeniami stałymi.
5. Nie inicjowany jawnie obiekt statyczny jest inicjowany tak, jakby jemu, (lub jego składowym) przypisano wartość zero.
6. Początkowa wartość nie zainicjowanego jawnie obiektu automatycznego jest niezdefiniowana.
7. Inicjatorem dla obiektu arytmetycznego jest pojedyncze wyrażenie (być może ujęte w nawiasy klamrowe).

Inicjowanie II

8. Inicjatorem dla struktury jest albo wyrażenie tego samego typu albo ujęta w nawiasy klamrowe lista inicjatorów dla jej kolejnych składowych.
9. Inicjatorem dla tablicy jest ujęta w klamry lista inicjatorów dla jej kolejnych elementów.
10. Jeśli nie jest znany rozmiar tablicy — to rozmiar ten wylicza się na podstawie liczby inicjatorów.
11. Jeśli tablica ma ustalony rozmiar — liczba inicjatorów nie może przekroczyć liczby elementów tablicy; jeśli lista jest krótsza uzupełniana jest zerami.
12. Specjalnym przypadkiem jest tablica znakowa, która może być inicjowana napisem (kolejne znaki napisu inicjują kolejne elementy tablicy).

Inicjowanie III

13. Jeżeli nie jest znany rozmiar tablicy znakowej jest on wyliczany na podstawie liczby znaków w napisie (włączając w to końcowy znak zerowy).

Tablice wielowymiarowe

Przykład

```
#include<stdio.h>
int main(void)
{
    int a[4][3] = {
        {1, 3, 5},
        {2, 4, 6},
        {3, 5, 7},
    };
    int i, j;
    for (i = 0; i < 4; i++){
        for (j = 0; j < 3; j++)
        printf(" %d |", a[i][j]);
        printf("\n");
    }
    return 0;
}
```

Tablice wielowymiarowe

Wynik działania programu

```
1 | 3 | 5 |
2 | 4 | 6 |
3 | 5 | 7 |
0 | 0 | 0 |
```

Tablice wielowymiarowe

Zadanie domowe

Zmodyfikować tak przykładowy progam, żeby drukował wyniki w następującej postaci:

```
| 1 | 3 | 5 |
| 2 | 4 | 6 |
| 3 | 5 | 7 |
| 0 | 0 | 0 |
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

```
1 | 3 | 5 |
2 | 4 | 6 |
3 | 5 | 7 |
0 | 0 | 0 |
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

```
1 | 0 | 0 |
2 | 0 | 0 |
3 | 0 | 0 |
4 | 0 | 0 |
```

Napisy

1. Stała znakowa (złożona z jednego znaku) zapisywana jest tak `'c'` („c” to dowolny znak lub specjalna stała złożona ze znaku „backslash” (\) i specjalnego symbolu.
2. Stała tekstowa zapisywana jest w cudzysłowach (podwójne apostrofy) `"Ala ma kota"`
3. Sąsiadujące ze sobą napisy łączone są w jeden napis ("Ala" "ma" "kota" tworzy napis "Alamakota"; "Ala " "ma" " kota" tworzy "Ala ma kota").
4. Na końcu napisu umieszczany jest znak o kodzie ASCII równym zero ('\x00') pozwalający rozpoznać koniec tekstu.
5. W napisach można używać wszystkich symboli specjalnych dostępnych w stałych znakowych.
6. Typem do przechowywania znaków jest **char**.
7. Napisy trzeba przechowywać w tablicach typu **char**.
8. Polskie znaki — na razie proponuję o tym zapomnieć!

Tablice znakowe

To jest poprawna deklaracja. Tablica będzie miała rozmiar 14 (13 znaków napisu i znak null kończący napis). Można to sprawdzić za pomocą funkcji `sizeof(tekscik)`.

```
char tekscik[] = "Ala ma kotala";
```

Poniżej również poprawna deklaracja tablicy (o łącznym rozmiarze 21 znaków). Liczba wierszy wyliczana jest automatycznie podczas kompilacji.

```
char teksty[][7] = {
    {"Ala"},
    {"ma"},
    {"kotala"}
};
```

To niepoprawna forma deklaracji:

```
char teksty[3][] = {
    {"Ala"},
    {"ma"},
    {"kotala"}
};
```

Tablice jako argumenty funkcji

1. Trzeba bardzo uważać i myśleć zanim się coś zrobi!
2. Rzecz nie jest prosta (choć może nie aż tak skomplikowana).

Przykład

- Chcemy napisać funkcję, która zeruje tablicę (wypełnia wartością zero wszystkie elementy tablicy).
- Sam algorytm jest bardzo prosty

```
int N = 10;
double A[N];
for(int i =0; i < N; i++)
    A[i] = 0.;
```

- Musimy go tylko obudować w funkcję
  - nie zwraca parametrów
  - ma dwa parametry:
    1. tablicę
    2. jej rozmiar

Przykład c.d.

```
void zeruj(int N, double A[])
{
   for(int i = 0; i < N; i++)
      A[i] = 0.;
}
```

A używać jej będziemy tak:

```
int main()
{
  double X[1000];
  int M = 1000;
//   ...
  zeruj(M, X);
//  ...
}
```