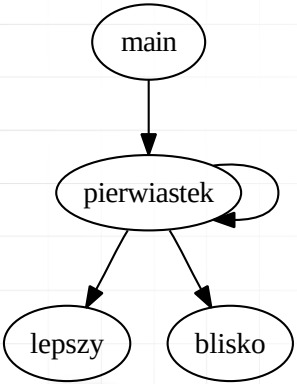


Funkcje

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.

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.

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 go bardziej czytelnym.

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 go bardziej czytelnym.
4. Funkcje to, wreszcie, podstawa programowania strukturalnego.

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 go bardziej czytelnym.
4. Funkcje to, wreszcie, podstawa programowania strukturalnego.
5. Praktycznie każdy język programowania wyposażony jest w mechanizmy podziału na moduły oraz tworzenia funkcji (i procedur).

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 go bardziej czytelnym.
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!

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.

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.
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.
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 instrukcji powinna znaleźć się instrukcja

   ```
   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)
{}

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

Program z funkcją

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

void glupia(void)
{}

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

Program z funkcją

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

void glupia(void)
{}

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

Źle !!!

Program z funkcją

```
void glupia(void)
{}

void dummy(void)
{
   glupia();
}

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

Program z funkcją

```
void glupia(void)
{}

void dummy(void)
{
   glupia();
}

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

OK !!!

Funkcje zagnieżdżone

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

Funkcje zagnieżdżone

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

Źle!!!

Standard języka C na to nie pozwala!

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”

1. Procedurę deklaruje się w następujący sposób:

```
void procedurka( int x )
{
    printf("--------\n"\
           "%d\n"\
           "--------\n",\
           x);
}
```

2. Procedurę wywołuje się w następujący sposób:

```
int main()
{
    int z = 123;
    procedurka( z + 7);
    return 1;
}
```

Kompletny program będzie wyglądał tak:

```
#include<stdio.h>

void procedurka( int x )
{
    printf("--------\n"\
           "%d\n"\
           "--------\n",\
           x);
}

int main()
{
    int z = 123;
    procedurka( z + 7);
    return 1;
}
```

„Procedury” — kilka uwag

1. Każda procedura (jak i funkcja) powinna być zadeklarowana przed pierwszym jej użyciem.

„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).

„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.

„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.
4. Pliki nagłówkowe powinny być wczytywane na początku.

„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.
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:

„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.
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:
   - wczytanie plików nagłówkowych,

„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.
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:
   - wczytanie plików nagłówkowych,
   - definicje wszystkich procedur,

„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.
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.
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.
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);

Program z funkcją i prototypy

```
void glupia(void);
void dummy(void);

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

void dummy(void)
{
   glupia();
}

void glupia(void)
{}
```

Teraz kolejność nie jest już istotna.

Definicje i deklaracje lokalne

1. Każda zmienna musi być zadeklarowana.

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.

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 { }

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.

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.

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 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!
7. Po powrocie do funkcji (w zasadzie) nie ma dostępu do poprzednich wartości zmiennych.

Definicje i deklaracje globalne

1. Zmienne zadeklarowane na zewnątrz wszystkich modułów (funkcje, procedury, funkcja **main**) nazywane są zmiennymi globalnymi.

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...

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.

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.
4. Zmienne globalne mogą być wykorzystanie do przekazywania dodatkowych wyników zwracanych przez funkcję. Nie jest to najlepsze rozwiązanie...

```
#include<stdio.h>
int v = 100;
void procedurka( int x )
{
    int v = 7;
    printf("--------\n"\
           "%d\n"\
           "--------\n",\
           x);
    printf("v = %d\n", v);
}
int main()
{
    int z = 123;
    procedurka( z + 7);
    procedurka( v );
    return 1;
}
```

Czym się różni?

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

Czym się różni?

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

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

Funkcja main

1. Każdy program w języku C musi mieć segment główny.

Funkcja main

1. Każdy program w języku C musi mieć segment główny.
2. Segment główny musi nazywać się **main**...

Funkcja main

1. Każdy program w języku C musi mieć segment główny.
2. Segment główny musi nazywać się **main**...
3. ...i jest funkcją!

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.

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.

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**.

Funkcja main

1. To jest właściwie poprawny program w języku C

```
void main(void)
{
    ;
}
```

   Nic nie robi, nie zwraca żadnej informacji. Kompilator sygnalizuje komunikat „warning: return type of ‘main’ is not ‘int’”
2. Zamiana pierwszego **void** na **int**

```
int main(void)
{
    ;
}
```

   powoduje komunikat „warning: control reaches end of non-void function” (czyli brakuje polecenia **return**).
3. Poprawny (minimalny) program powinien wyglądać jakoś tak:

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

Rekurencja

1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.

Rekurencja

1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.
2. Nie potrafię powiedzieć, czy rekurencja to dobra czy zła technika programowania.

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.

Rekurencja

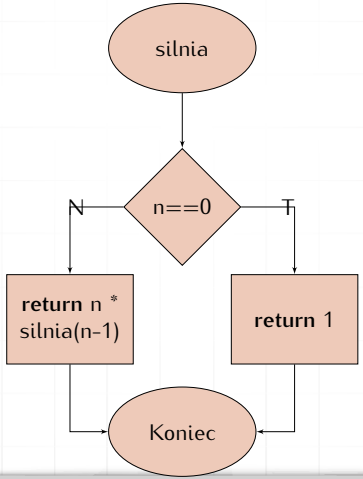
1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.
2. Nie potrafię powiedzieć, czy rekurencja to dobra czy zła technika programowania.
3. Rekurencja była bardzo pożyteczna podczas tworzenia algorytmów.
4. W realizacjach programowych (zwłaszcza bardzo skomplikowanych problemów)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

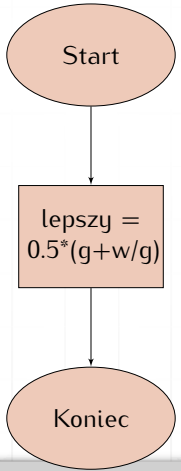
Program będzie się składał z trzech części:

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 $n = 2$

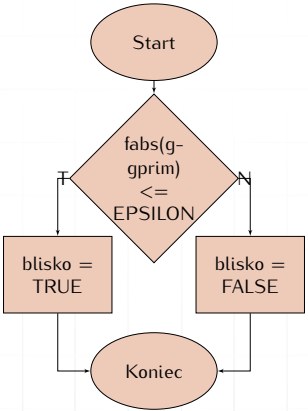
Realizacja programowa

`lepszy(w, g)`



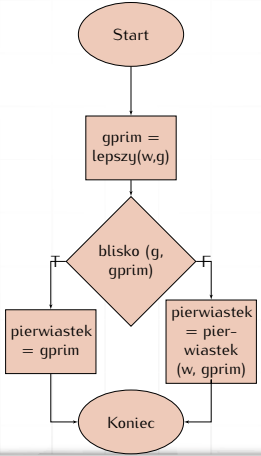
Realizacja programowa

`blisko(g, gprim)`



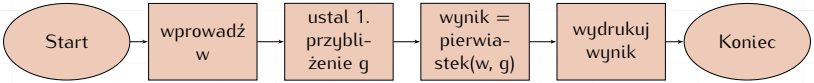
Realizacja programowa

`pierwiastek(w, g)`



Realizacja programowa

Program główny



Metoda Newtona

Realizacja programowa

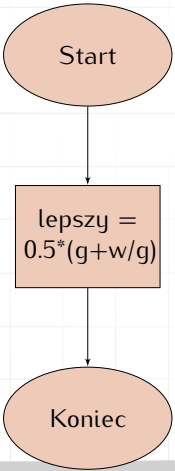
Program składa się z trzech części:

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

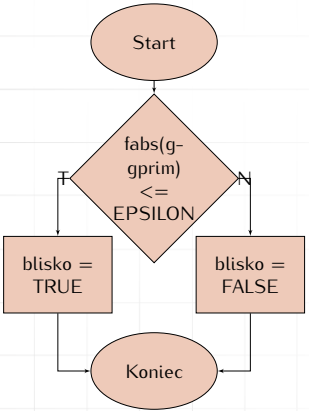
`lepszy(w, g)`




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

Metoda Newtona

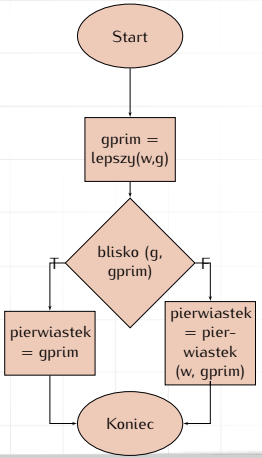
`blisko(g, gprim)`




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

Metoda Newtona

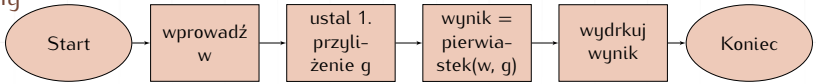
`pierwiastek(w, g)`




```
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



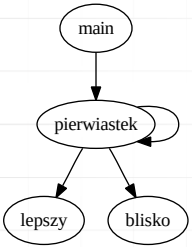

```
int main(void)
{
    double w, g, wynik;
    w = 2.;
    g = 1.;
    wynik = sqrtf(w);
    printf("%f\n", wynik);
    wynik = pierwiastek(w, g);
    printf("Pierwiastek kwadratowy z liczby" \
           " %f wynosi %f\n", w, wynik);
    return 0;
}
```

Metoda Newtona-Raphsona

Zadanie domowe

1. Narysować schemat blokowy dla dowolnego n (wszystko to co było to było dla $n = 2$.
2. Napisać program (w C) realizujący ten schemat blokowy.

Kolofon



Ilustracja na stronie tytułowej przedstawia tak zwany „call graph” programu newton opisywanego wcześniej.
Tworzony jest on automatycznie na podstawie analizy przebiegu programu.
Do jego uzyskania potrzebny są perlowy program egypt oraz program graphviz.