Politechnika Wrocławska

Bardzo szybkie podsumowanie: wykład 2

wer. 7 z drobnymi modyfikacjami!

Wojciech Myszka

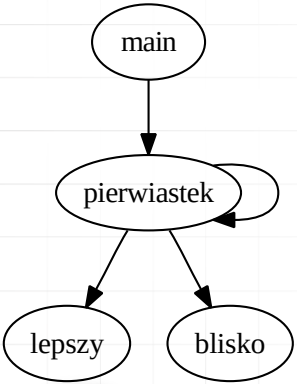
2019-04-02 09:08:11 +0200

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska

Uwagi

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



Funkcje

wer. 7 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki i Inżynierii Materiałowej

2019-03-19 07:46:40 +0100

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!
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 );
}
```

Źle !!!

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;
}
```

Źle !!!

Standard języka C na to nie pozwala!

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.
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.
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.
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;
}
```

```
int main(void)
{
    int i;
    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.  */
```

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”

3. Poprawny program powinien wyglądać jakoś tak:

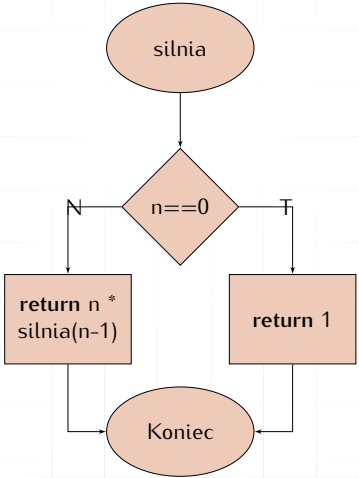
```
int main(void)
{
    return 0;
}
```

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

```c
#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

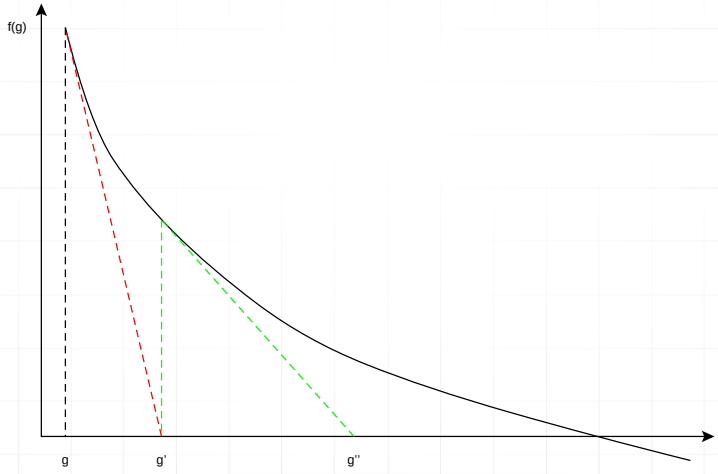
Jeżeli zadanie dodatkowo uprościmy zakładając:

- 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 punk $(g, f(g)$ 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

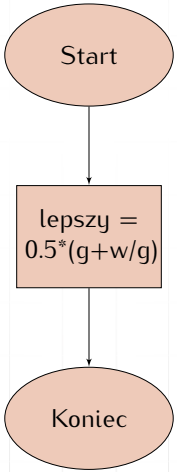
Program będzie się składał z trzech części:

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 $n = 2$

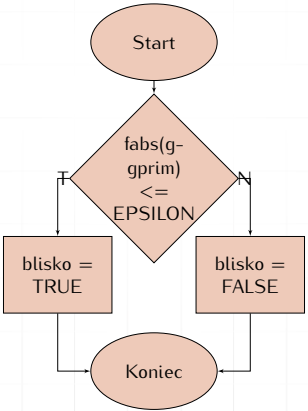
Realizacja programowa

`lepszy(w, g)`



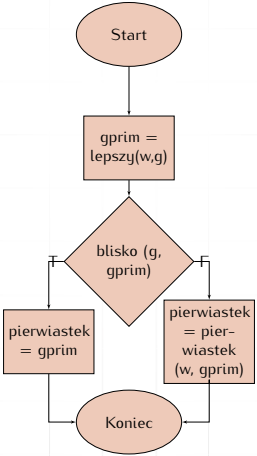
Realizacja programowa

`blisko(g, gprim)`



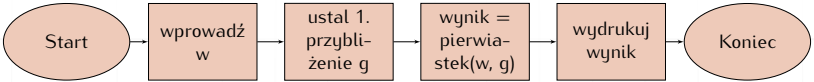
Realizacja programowa

`pierwiastek(w, g)`



Realizacja programowa

Program główny



Metoda Newtona

Realizacja programowa

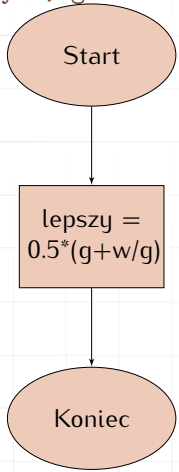
Program składa się z trzech części:

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

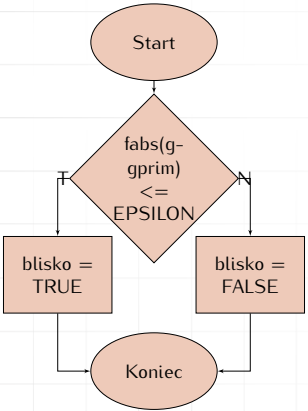
`lepszy(w, g)`




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

Metoda Newtona

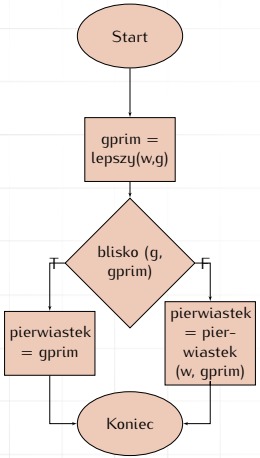
`blisko(g, gprim)`




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

Metoda Newtona

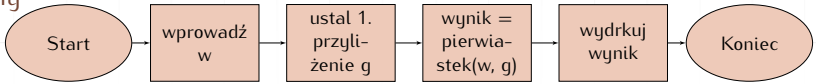
`pierwiastek(w, g)`




```
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.

Tablice (jedno i wielowymiarowe), łańcuchy znaków

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki i Inżynierii Materiałowej

2019-03-19 07:47:14 +0100

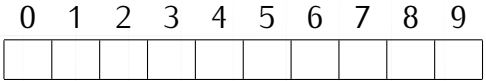
Zmienne

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).

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 }
```

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
```

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;
}
```

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.

Inicjowanie II

7. Inicjatorem dla obiektu arytmetycznego jest pojedyncze wyrażenie (być może ujęte w nawiasy klamrowe).
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.

Inicjowanie III

12. Specjalnym przypadkiem jest tablica znakowa, która może być inicjowana napisem (kolejne znaki napisu inicjują kolejne elementy tablicy).
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

```c
#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
};
```

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

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. Więcej o tym będzie później!