Podstawy Programowania

dr inż. Tomasz Marciniak

Wykład 3

- Funkcje

Po co nam funkcje?

```c
#include <stdio.h>
#include <conio.h>
int main()
{
 char nazwa[20];
 float cena;
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 getche();
 return 0;
}
```

Po co nam funkcje?

```c
#include <stdio.h>
#include <conio.h>
int main()
{
 char nazwa[20];
 float cena;
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 printf("\npodaj nazwe :");
 gets (nazwa);
 printf("podaj cene :");
 scanf("%f",&cena);
 printf("\nCena dla %s to: %.2f", nazwa,cena);
 getche();
 return 0;
}
```

Mamy 3 powtarzające się kawałki kodu
Czy nie można krócej?

Po co nam funkcje?

```
 1  #include <stdio.h>
 2  #include <conio.h>
 3  void func1(){
 4   printf("\npodaj nazwe :");
 5   gets (nazwa);
 6   printf("podaj cene :");
 7   scanf("%f",&cena);
 8   printf("\nCena dla %s to: %.2f", nazwa,cena);
 9  }
10  int main()
11  {
12   char nazwa[20];
13   float cena;
14   func1();
15   func1();
16   func1();
17   getche();
18   return 0;
19  }
```

Po co nam funkcje?

```
 1  #include <stdio.h>
 2  #include <conio.h>
 3  void func1(){
 4   printf("\npodaj nazwe :");
 5   gets (nazwa);
 6   printf("podaj cene :");
 7   scanf("%f",&cena);
 8   printf("\nCena dla %s to: %.2f", nazwa,cena);
 9  }
10  int main()
11  {
12   char nazwa[20];
13   float cena;
14   func1();
15   func1();
16   func1();
17   getche();
18   return 0;
19  }
```

Czy ten kod zadziała poprawnie?

Po co nam funkcje?

```
 1  #include <stdio.h>
 2  #include <conio.h>
 3  void func1(){
 4   printf("\npodaj nazwe :");
 5   gets (nazwa);
 6   printf("podaj cene :");
 7   scanf("%f",&cena);
 8   printf("\nCena dla %s to: %.2f", nazwa,cena);
 9  }
10  int main()
11  {
12   char nazwa[20];
13   float cena;
14   func1();
15   func1();
16   func1();
17   getche();
18   return 0;
19  }
```

NIE !!
Co ze zmiennymi?

Po co nam funkcje?

```
1   #include <stdio.h>
2   #include <conio.h>
3   char nazwa[20];
4   float cena;
5   void func1(){
6    nazwa[0]=0;
7    printf("\npodaj nazwe :");
8    scanf("%s",nazwa);
9    printf("\npodaj cene :");
10   scanf("%f",&cena);
11   printf("\nCena dla %s to: %4.2f", nazwa,cena);
12  }
13  int main()
14  {
15   func1();
16   func1();
17   func1();
18   getche();
19   return 0;
20  }
```

Zmienne muszą być globalne !

Po co nam jeszcze funkcje?

- Dają możliwość podzielenia programu na mniejsze kawałki;
- Zwiększają przejrzystość programu; np. obliczanie rozwiązań równania kwadratowego… można stworzyć funkcje delta, x1, x2 itd.

Czy musimy używać funkcji?

- Tak… ponieważ ….
- W programie musi się znajdować przynajmniej jedna funkcja (o nazwie zastrzeżonej) – **main().**

Funkcja

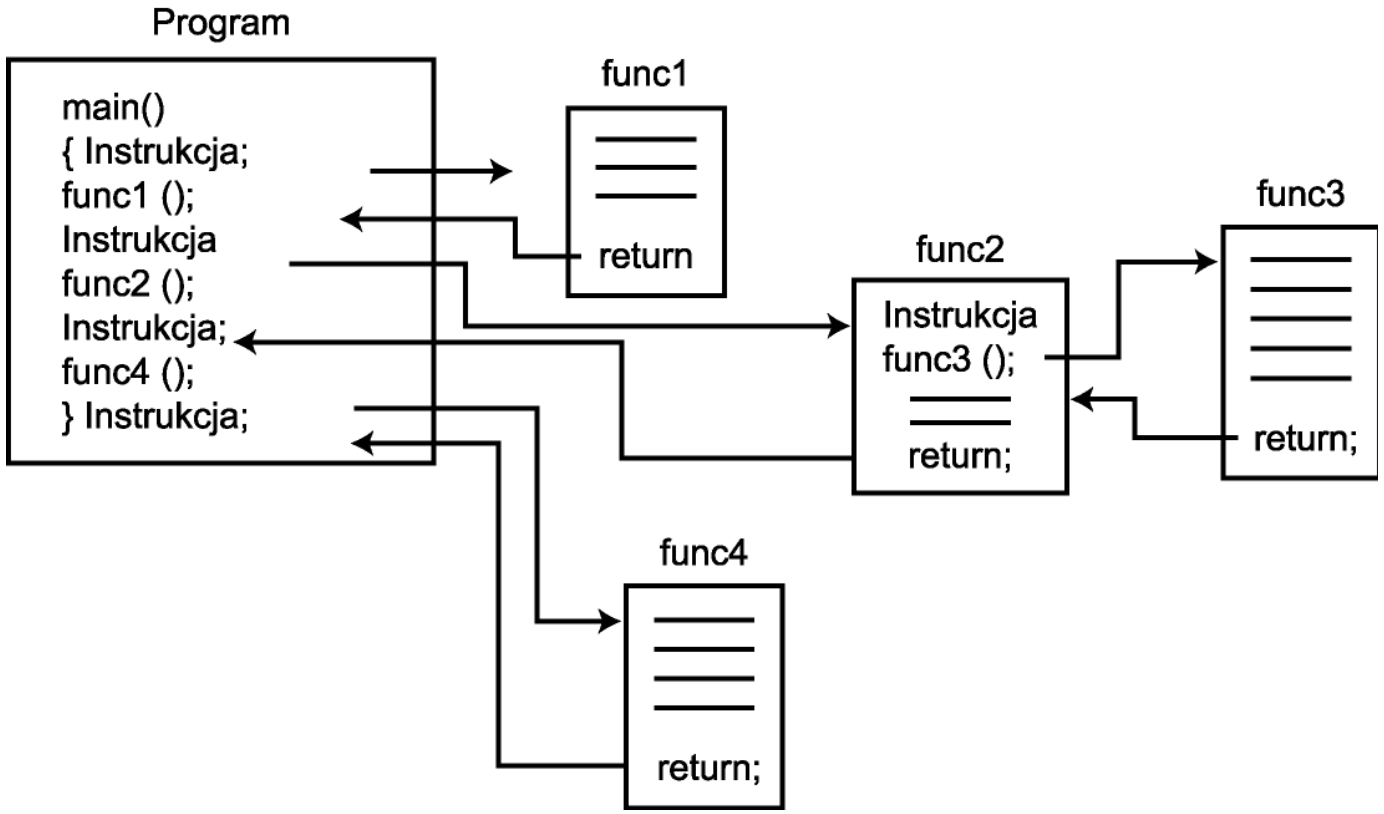
Choć w programowaniu zorientowanym obiektowo zainteresowanie użytkowników zaczęło koncentrować się na obiektach, jednak mimo to **funkcje** w dalszym ciągu pozostają głównym komponentem każdego programu:

- Funkcje **globalne** występują poza obiektami,
- Funkcje **składowe** (zwane także metodami składowymi) występują wewnątrz obiektów, wykonując ich pracę.

Czym jest funkcja ?

- Każda funkcja posiada nazwę;
- gdy ta nazwa zostanie napotkana przez program, przechodzi on do wykonywania kodu zawartego wewnątrz ciała tej funkcji. Nazywa się to *wywołaniem* funkcji;
- Gdy funkcja „wraca”, wykonanie programu jest wznawiane od instrukcji następującej po wywołaniu tej funkcji.

Schemat wykonywania funkcji w programie głównym



Funkcje

Bardzo istotny jest pierwszy wiersz definicji funkcji. Dostarcza on kompilatorowi następujące informacje:

- nazwa funkcji
- typ zwracanej przez funkcje wartości,
- listę argumentów (parametrów), przekazywanych do funkcji.

Funkcje cd.

- Funkcje występują w dwóch odmianach:
  - zdefiniowane przez użytkownika (programistę)
  - oraz wbudowane.
- Funkcje wbudowane stanowią część pakietu dostarczanego wraz z kompilatorem — zostały one stworzone przez producenta kompilatora.
- Funkcje zdefiniowane przez użytkownika są funkcjami, które pisane są samodzielnie.

Funkcje

Istnieją trzy zagadnienia związane z użyciem funkcji w programie:

- prototyp funkcji (deklaracja),
- definicja funkcji,
- wywołanie funkcji.

Deklaracja i definicja

- Aby użyć funkcji w programie, należy najpierw **zadeklarować** funkcję, a następnie ją **zdefiniować**.
- Deklaracja informuje kompilator o nazwie funkcji, typie zwracanej przez nią wartości, oraz o jej parametrach.
- Z kolei definicja informuje, w jaki sposób dana funkcja działa.
- Żadna funkcja nie może zostać wywołana z jakiejkolwiek innej funkcji, jeśli nie zostanie wcześniej zadeklarowana. Deklaracja funkcji jest nazywana *prototypem*.

Deklarowanie funkcji

Istnieją trzy sposoby deklarowania funkcji:

- zapisanie prototypu w pliku, w którym dana funkcja jest używana,
- zdefiniowanie funkcji zanim zostanie wywołana przez inne funkcje. Jeśli tego dokonasz, definicja będzie pełnić jednocześnie rolę deklaracji funkcji,
- zapisanie prototypu funkcji w pliku, a następnie użycie dyrektywy `#include` w celu dołączenia go do swojego programu.

Wartość zwracana

- Funkcja może *zwracać* wartość. Gdy wywołujesz funkcję, może ona wykonać swoją pracę, po czym zwrócić wartość stanowiącą rezultat tej pracy. Ta wartość jest nazywana wartością zwracaną, zaś jej typ musi być zadeklarowany.

  Czyli zapisując:

```
int myFunction();
```

  deklarujesz, że funkcja `myFunction` zwraca wartość całkowitą.

Pamiętamy nasz przykład?

```
1   #include <stdio.h>
2   #include <conio.h>
3   char nazwa[20];
4   float cena;
5   void func1(){
6    nazwa[0]=0;
7    printf("\npodaj nazwe :");
8    scanf("%s",nazwa);
9    printf("\npodaj cene :");
10   scanf("%f",&cena);
11   printf("\nCena dla %s to: %4.2f", nazwa,cena);
12  }
13  int main()
14  {
15   func1();
16   func1();
17   func1();
18   getche();
19   return 0;
20  }
```

Pamiętamy nasz przykład?

```
1   #include <stdio.h>
2   #include <conio.h>
3   char nazwa[20];
4   float cena;
5   void func1(){
6    nazwa[0]=0;
7    printf("\npodaj nazwe :");
8    scanf("%s",nazwa);
9    printf("\npodaj cene :");
10   scanf("%f",&cena);
11   printf("\nCena dla %s to: %4.2f", nazwa,cena);
12  }
13  int main()
14  {
15   func1();
16   func1();
17   func1();
18   getche();
19   return 0;
20  }
```

Brak prototypu funkcji !!

Definicja pełni jednocześnie rolę deklaracji funkcji

Choć możesz zdefiniować funkcję przed jej użyciem i uniknąć w ten sposób konieczności tworzenia jej prototypu, nie należy to do dobrych obyczajów programistycznych z trzech powodów:

1. Niedobrze jest, gdy funkcje muszą występować w pliku źródłowym w określonej kolejności. Powoduje to, że w razie wprowadzenia zmian trudno jest zmodyfikować taki program.
2. Istnieje możliwość, że w pewnych warunkach funkcja `A()` musi być w stanie wywołać funkcję `B()`, a funkcja `B()` także musi być w stanie wywołać funkcję `A()`. Nie jest możliwe zdefiniowanie funkcji `A()` przed zdefiniowaniem funkcji `B()` i jednoczesne zdefiniowanie funkcji `B()` przed zdefiniowaniem funkcji `A()`, dlatego przynajmniej jedna z nich zawsze musi zostać zadeklarowana.
3. Prototypy funkcji stanowią wydajną technikę debuggowania (usuwania błędów w programach). Jeśli z prototypu wynika, że funkcja otrzymuje określony zestaw parametrów lub że zwraca określony typ wartości, to w przypadku gdy funkcja nie jest zgodna z tym prototypem, kompilator, zamiast czekać na wystąpienie błędu podczas działania programu, może wskazać tę niezgodność.

Prototypy funkcji

- Wiele z wbudowanych funkcji posiada już gotowe prototypy. Występują one w plikach, które są dołączane do programu za pomocą dyrektywy `#include`. W przypadku funkcji pisanych samodzielnie, musisz stworzyć samodzielnie także ich prototypy.
- Prototyp funkcji jest instrukcją, co oznacza, że kończy się on średnikiem. Składa się ze zwracanego przez funkcję typu oraz tzw. sygnatury funkcji. Sygnatura funkcji to jej nazwa oraz lista parametrów.

```
long Area(int, int);
```

- Dodanie nazw parametrów powoduje, że prototyp staje się bardziej czytelny. Ta sama funkcja z nazwanymi parametrami mogłaby być zadeklarowana następująco:

```
long Area(int length, int width );
```

W tym przypadku jest oczywiste, do czego służy ta funkcja oraz jakie są jej parametry.

Prototypy

```
1 /* funkcje */
2 #include <stdio.h>
3 #include <conio.h>
4 void wprowadz_dane();
5 void przetwarzaj();
6 void podaj_wynik();
7 void przerwa();
8 int main()
9 {
10 clrscr();
11 wprowadz_dane();
12 przetwarzaj();
13 podaj_wynik();
14 przerwa();
15 return 0;
16 } //koniec main
17 void wprowadz_dane()
18 {
19 printf("\n funkcja wprowadz_dane().");
20 printf("\nczytam dane z klawiatury");
21}
22 void przetwarzaj()
23 {
24 printf("\n.....funkcja przetwarzaj().....");
25 printf("\nprzetwarzam dane");
26 }
27 void podaj_wynik()
28 {
29 printf("\n.....funkcja podaj_wynik().....");
30 printf("\npodaje wyniki");
31 }
32 void przerwa()
33 {
34 printf("\n.....funkcja przerwa().....");
35 printf("\nNacisnij ENTER aby skonczyc");
36 getchar();
37 }
```

Prototypy funkcji

Dopiero tutaj deklaracja funkcji

Przykłady prototypów funkcji

- void fun1(void) ;
- void fun2( ) ;
- int fun3( );
- int fun4(char c) ;
- double fun5(int x= 20, int y=20) ;

Prototypy funkcji w osobnym pliku

Plik p17.c

```
#include <stdio.h>
#include <conio.h>
#include "p17.h"

int main()
{
 funkcja1();
 funkcja2();
 funkcja3();
 getche();
 return 0;
}
```

Dołączenie pliku nagłówkowego

Plik p17.h

```
void funkcja1();
void funkcja2();
void funkcja3();

void funkcja1() {
    printf("funkcja 1 \n");
 }
void funkcja2() {
    printf("funkcja 2 \n");
 }
void funkcja3() {
    printf("funkcja 3 \n");
 }
```

Prototypy funkcji

Prototypy funkcji w osobnym pliku

Plik p17.c

```
#include <stdio.h>
#include <conio.h>
#include "p17.h"

int main()
{
 funkcja1();
 funkcja2();
 funkcja3();
 getche();
 return 0;
}
```

Dołączenie pliku nagłówkowego

Plik p17.h

```
void funkcja1();
void funkcja2();
void funkcja3();
```

Prototypy funkcji

Definicje funkcji

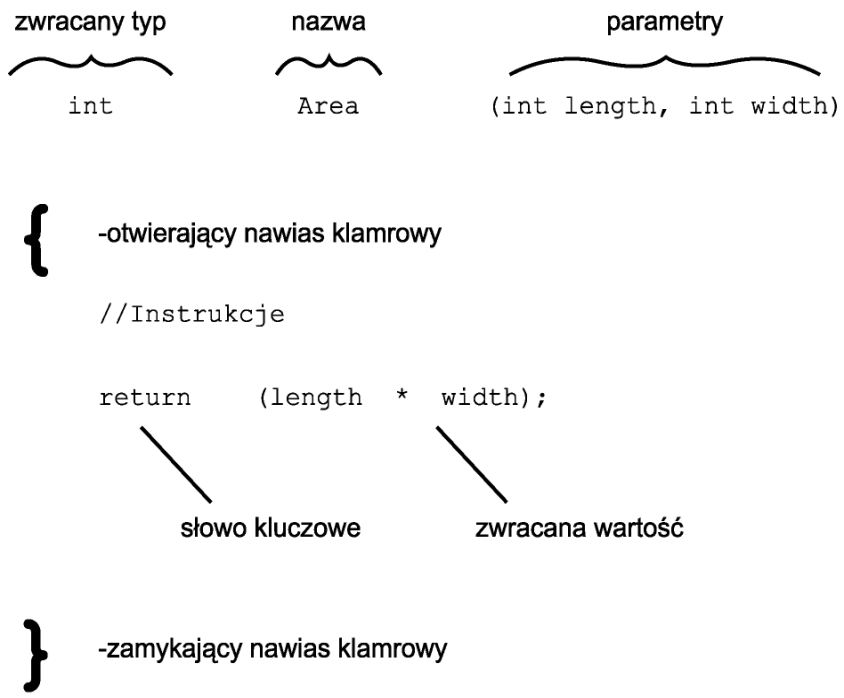
```
void funkcja1() {
    printf("funkcja 1 \n");
    return;
 }
```

```
void funkcja2() {
    printf("funkcja 2 \n");
    return;
 }
```

```
void funkcja3() {
    printf("funkcja 3 \n");
    return;
 }
```

Definiowanie funkcji

- Definicja funkcji składa się z nagłówka funkcji oraz z jej ciała. Nagłówek przypomina prototyp funkcji, w którym wszystkie parametry muszą być nazwane a na końcu nagłówka nie występuje średnik.
- Ciało funkcji jest ujętym w nawiasy klamrowe zestawem instrukcji. Rysunek przedstawia nagłówek i ciało funkcji.



Parametry funkcji

- Opis przekazywanych wartości jest nazywany listą parametrów.

```
int myFunction(int someValue, float
  someFloat);
```

- Ta deklaracja wskazuje, że funkcja `myFunction` nie tylko zwraca liczbę całkowitą ale także, że jej parametrami są: wartość `int` oraz wartość typu `float`.
- Parametr opisuje *typ* wartości, jaka jest przekazywana funkcji podczas jej wywołania. Wartości przekazywane funkcji są nazywane *argumentami*.

Argumenty funkcji

- Argumenty przekazywane funkcji są lokalne dla tej funkcji. Zmiany dokonane w argumentach nie wpływają na wartości w funkcji wywołującej. Nazywa się to przekazywaniem *przez wartość*, co oznacza, że wewnątrz funkcji jest tworzona lokalna **kopia każdego z argumentów**. Te lokalne kopie są traktowane tak samo, jak każda inna zmienna lokalna.

Argumenty funkcji

```
int funkcja1(int długość, int szerokość, int wysokość)
{
    int objętość=0;
    długość *= 2;
    if (szerokość >100)
            szerokość = 100;
    wysokość +=5;
    objętość=długość*szerokość*wysokość;
    return objętość;
}
```

Argumenty są traktowane jak zmienne lokalne, można na nich wykonywać działania

Zwracanie wartości

- Funkcja zwraca albo wartość, albo typ `void` (pusty). Typ `void` jest dla kompilatora sygnałem, że funkcja nie zwraca żadnej wartości.
- Aby zwrócić wartość z funkcji, używane jest słowo kluczowe `return`, a po nim wartości, które zostaną zwrócone. Wartość ta może być wyrażeniem zwracającym wartość. Na przykład:

```
return 5;
return (x > 5);
return (MyFunction());
```

Zwracanie wartości

- Poprzednie instrukcje są poprawne, pod warunkiem, że funkcja `MyFunction()` także zwraca wartość. Wartością w drugiej instrukcji, `return (x > 5);` będzie `false`, gdy `x` nie jest większe od `5`, lub `true` w odwrotnej sytuacji. Zwracana jest wartość wyrażenia, `false` lub `true`, a nie wartość `x`.
- Gdy program natrafia na słowo kluczowe `return`, następująca po nim wartość jest zwracana jako wartość funkcji. Wykonanie programu powraca natychmiast do funkcji wywołującej, zaś instrukcje występujące po instrukcji `return` nie są już wykonywane.
- Pojedyncza funkcja może zawierać więcej niż jedną instrukcję `return`.

Zwracanie wartości

```
int funkcja1(int długość, int szerokość, int wysokość)
{
    int objętość=0;
    długość *= 2;
    if (szerokość >100)
            return 0;
    wysokość +=5;
    objętość=długość*szerokość*wysokość;
    return objętość;
}
```

return może wystąpić kilka razy

Zmienne lokalne

- Zmienna posiada zakres, który określa, jak długo i w których miejscach programu jest ona dostępna. Zmienne zadeklarowane wewnątrz bloku mają zakres obejmujący ten blok; mogą być dostępne tylko wewnątrz tego bloku i „przestają istnieć” po wyjściu programu z tego bloku. Zmienne globalne mają zakres globalny i są dostępne w każdym miejscu programu.

Zmienne globalne

- Zmienne zdefiniowane poza funkcją mają zakres globalny i są dostępne z każdej funkcji w programie, włącznie z funkcją `main()`.
- Zmienne lokalne o takich samych nazwach, jak zmienne globalne nie zmieniają zmiennych globalnych. Jednak zmienna lokalna o takiej samej nazwie, jak zmienna globalna przesłania „*ukrywa*” zmienną globalną. Jeśli funkcja posiada zmienną o takiej samej nazwie jak zmienna globalna, to nazwa użyta wewnątrz funkcji odnosi się do zmiennej lokalnej, a nie do globalnej.

Zmienne globalne cd.

- Zmienne globalne są konieczne, gdyż zdarzają się sytuacje, w których dane muszą być łatwo dostępne dla wielu funkcji i nie chcemy ich przekazywać z funkcji do funkcji w postaci parametrów.
- Zmienne globalne są niebezpieczne, gdyż zawierają wspólne dane, które mogą być zmienione przez którąś z funkcji w sposób niewidoczny dla innych. Może to powodować bardzo trudne do odszukania błędy.

Zmienne – przykłady (p19.c)

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

void func1();

int main()
{
 int a=50;
 func1();
 printf("a=%d",a);
 getche();
 return 0;
}

void func1()
{
  int a=100;
}
```

Jaka wartość a będzie wydrukowana na ekranie?

Zmienne - przykłady

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

void func1();

int main()
{
 int a=50;
 func1();
 printf("a=%d",a);
 getche();
 return 0;
}

void func1()
{
  int a=100;
}
```

a=50

Zmienne – przykłady (p20.c)

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

void func1(int a);

int main()
{
 int a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int a)
{
  a=100;
}
```

Jaka wartość a będzie wydrukowana na ekranie?

Zmienne - przykłady

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

void func1(int a);

int main()
{
 int a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int a)
{
  a=100;
}
```

a=50

Co zrobić aby w funkcji modyfikować zmienną a ?

Zmienne - przykłady

```
#include <stdio.h>
#include <conio.h>
int a;

void func1(int a);

int main()
{
 a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int a)
{
  a=100;
}
```

Może a zadeklarować jako zmienną globalną?

Jaka wartość a będzie wydrukowana na ekranie?

Zmienne - przykłady

```
#include <stdio.h>
#include <conio.h>
int a;

void func1(int a);

int main()
{
 a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int a)
{
  a=100;
}
```

a=50

Zmienne - przykłady

```
#include <stdio.h>
#include <conio.h>
int a;

void func1(int b);

int main()
{
 a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int b)
{
  a=100;
}
```

A gdyby z parametrów funkcji wyrzucić a lub zrezygnować z parametrów?

Jaka wartość a będzie wydrukowana na ekranie?

Zmienne - przykłady

```
#include <stdio.h>
#include <conio.h>
int a;

void func1(int b);

int main()
{
 a=50;
 func1(a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int b)
{
  a=100;
}
```

Nareszcie a=100 !!!

Wnioski:

- parametr funkcji traktowany jest jak zmienna lokalna;
- zmienna lokalna przykrywa zmienną globalną o tej samej nazwie;

Czy można nasze zadanie zrealizować inaczej???

Zmienne – przykłady (p22.c)

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

void func1(int* a);

int main()
{
 int a=50;
 func1(&a);
 printf("a=%d",a);
 getche();
 return 0;
}

void func1(int* a)
{
  *a=100;
}
```

Tak – mamy wskaźniki

Zmienne lokalne

```
int f1(void)
{
    int a, b,c;
    int k;
    a = 1; b = 2; k = 10;
    c = (a + b)*k;
    return c;
}
```

- Zmienne zadeklarowane wewnątrz ciała funkcji są ***zmiennymi automatycznymi (lokalnymi);***
- Jeżeli program wchodzi do funkcji rezerwowana jest odpowiednia ilość pamięci dla zmiennych;
- Jeżeli program opuszcza funkcję, pamięć ta jest zwalniana;
- Przy ponownym wejściu następuje kolejne przydzielenie pamięci dla zmiennych -> poprzednie wartości nie są znane;
- Zmienne w różnych funkcjach mogą mieć te same nazwy.

Zmienne statyczne (p23.c)

```
int f1(char first)
{
    static int a;
    if (first==1) a=0;
    a++;
    return a;
}
```

- Wartość zmiennej static pamiętana jest pomiędzy wywołaniami funkcji;
- Przy kolejnym wejściu do funkcji mamy do dyspozycji ostatnią wartość zmiennej static;

Zmienne typu extern

```
//program1.c
#include <stdio.h>
#include <conio.h>
int counter;
void main()
{
    .
    .
    .
}
```

```
//program2.c
#include <stdio.h>
#include <conio.h>
extern int counter;
void main()
{
    .
    .
    .
}
```

Jeżeli mamy program napisany w postaci 2 plików, a zmienna globalna zadeklarowana w jednym z nich ma być używana w drugim, to w drugim pliku jej deklaracja musi być poprzedzona słówkiem **extern**

Zmienne klasy „register”

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

int func1(void);

int main()
{
 int count=0;
 count=func1();
 printf("count=%d",count);
 getche();
 return 0;
}

int func1(void)
{
    register int a=0;
    a++;
    return a;
}
```

- Tego typu zmiennych używamy, gdy chcemy przyspieszyć szybkość wykonywania programu;
- Deklaracja zmiennej, jako register informuje kompilator, że jeżeli jest to możliwe fizycznie i semantycznie, to należy zadeklarowaną zmienną zapamiętać w rejestrach;
- Pojemność rejestrów jest niewielka i często nie jest możliwe wykonanie tego zlecenia, wtedy przydzielana jest pamięć konwencjonalna;
- **deklaracja register jest tylko sugestią dla kompilatora .**

Makroinstrukcje

- tworzone są za pomocą dyrektywy preprocesora **define,** jak np.stałe

  #define PI 3.14159

- Konstrukcja makrodefinicji ma postać:

  #define NAZWA (lista_parametrów) ciąg_instrukcji

- **#define** nakazuje kompilatorami zamianę nazwy stałej na ciąg_instrukcji
- Przykłady:

  #define DANE printf("Podaj dane")

  #define KONIEC printf("Nacisnij ENTER aby skonczyc")

  #define MAX(x,y) ((x)>(y)?(x):(y))

Przykład (p25.c)

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

#define MAX(x,y) ((x)>(y)?(x):(y))
#define drukuj(x,y) printf("liczba wieksza to %d",MAX(x,y)); \
                    printf("\nto był wydruk \n")                \

int main()
{
 int count=0;
 count++;
 drukuj(7,5);
 getche();
 return 0;
}
```

- jedna makroinstrukcja może używać innej;
- makroinstrukcja może zawierać więcej niż jedno polecenie („\” na końcu linii);
- makroinstrukcje działają szybciej niż funkcje ale zwiększają ilość kodu

Funkcje rekurencyjne

```
#include <stdio.h>
#include <conio.h>
unsigned long silnia( unsigned long );
int main()
{ for ( int i = 0; i <= 12; i++)
printf("%2d ! = %ld\n",i,silnia(i));
getch();
return 0;
}
unsigned long silnia (unsigned long
    num)
{ unsigned long wal = 1;
for (int n =num; n >= 1; n--)
wal *= n;
return wal;
}
```

```
#include <stdio.h>
#include <conio.h>
unsigned long silnia( unsigned long );
int main()
{
for ( int i = 0; i <= 12; i++)
printf("%2d ! = %ld\n",i,silnia(i));
getch();
return 0;
}
unsigned long silnia (unsigned long
    num)
{
if (num <= 1) return 1;
else return num* silnia(num-1);
}
```

Argumenty funkcji main()

- Standardowo ma ona 2 argumenty **argc** i **argv**;
- argc podaje ilość argumentów wprowadzonych z wiersza poleceń;
- Tablica argv jest tablicą wskaźników do typu char;
- element argv[0] zawiera zawsze nazwę programu, wartość argc będzie zawsze wynosiła, co najmniej 1.

Przykład (p27.c)

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[])
{
   printf("Liczba argumentow = %d \n", argc);
   for (int i = 0; i < argc; i++)
   printf("Argument %d : %s\n",i,argv[i]);
   return 0;
}
```

- A teraz idziemy grzecznie na przerwę