

Informatyka 2 (ES1D300 017)

Politechnika Białostocka - Wydział Elektryczny
Elektrotechnika, semestr III, studia stacjonarne I stopnia
Rok akademicki 2019/2020

Wykład nr 6 (05.11.2019)

dr inż. Jarosław Forenc

Plan wykładu nr 6

- Przekazywanie argumentów do funkcji
 - przez wartość i przez wskaźnik
 - przekazywanie wektorów, macierzy i struktur
 - const przed parametrem funkcji

- Pamięć a zmienne w programie
 - zmienne automatyczne (auto)
 - zmienne rejestrowe (register)
 - zmienne zewnętrzne (extern)
 - zmienne statyczne (static)
 - struktura procesu w pamięci komputera, ramka stosu

- Programy wielomodułowe

Przekazywanie argumentów do funkcji

- Przekazywanie argumentów przez **wartość**:
 - po wywołaniu funkcji tworzone są lokalne kopie zmiennych skojarzonych z jej argumentami
 - w funkcji widoczne są one pod postacią parametrów funkcji
 - parametry te mogą być traktowane jak lokalne zmienne, którym przypisano początkową wartość

- Przekazywanie argumentów przez **wskaźnik**:
 - do funkcji przekazywane są adresy zmiennych będących jej argumentami
 - wszystkie operacje wykonywane w funkcji na takich argumentach będą odnosiły się do zmiennych z funkcji wywołującej

Przekazywanie argumentów przez wartość

```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()

Przekazywanie argumentów przez wartość

```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()
a	0x0024FAF8	20	fun()

Przekazywanie argumentów przez wartość

```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()
a	0x0024FAF8	10	fun()

fun: a = 10

Przekazywanie argumentów przez wartość

```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()

```
fun: a = 10
main: a = 20
```

Przekazywanie argumentów przez wskaźnik

```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a);
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()

Przekazywanie argumentów przez wskaźnik

```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a);
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	20	main()
a	0x0024FAF8	0x0024FBDC	fun()

Przekazywanie argumentów przez wskaźnik

```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a);
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	10	main()
a	0x0024FAF8	0x0024FBDC	fun()

fun: a = 10

Przekazywanie argumentów przez wskaźnik

```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a);
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```

Fragment pamięci komputera

	Adres zmiennej	Wartość	
a	0x0024FBDC	10	main()

```
fun: a = 10
main: a = 10
```

Parametry funkcji - wektory

- Wektory przekazywane są do funkcji przez wskaźnik
- Nie jest tworzona kopia tablicy, a wszystkie operacje na jej elementach odnoszą się do tablicy z funkcji wywołującej
- W nagłówku funkcji podaje się typ elementów tablicy, jej nazwę oraz nawiasy kwadratowe z liczbą elementów tablicy lub same nawiasy kwadratowe

```
void fun(int tab[5])  
{  
    ...  
}
```

```
void fun(int tab[])  
{  
    ...  
}
```

- W wywołaniu funkcji podaje się tylko jej nazwę (bez nawiasów kwadratowych)

```
fun(tab);
```

Parametry funkcji - wektory (przykład)

```
#include <stdio.h>

void drukuj(int tab[])
{
    for (int i=0; i<5; i++)
        printf("%3d", tab[i]);
    printf("\n");
}

void zeruj(int tab[5])
{
    for (int i=0; i<5; i++)
        tab[i] = 0;
}
```

```
float srednia(int tab[])
{
    float sr = 0;
    int suma = 0;

    for (int i=0; i<5; i++)
        suma = suma + tab[i];

    sr = (float)suma / 5;

    return sr;
}
```

Parametry funkcji - wektory (przykład)

```
int main(void)
{
    int tab[5] = {1,2,3,4,5};
    float sred;

    drukuj(tab);

    sred = srednia(tab);
    printf("Srednia elementow: %g\n", sred);
    printf("Srednia elementow: %g\n", srednia(tab));

    zeruj(tab);
    drukuj(tab);

    return 0;
}
```

```
1 2 3 4 5
srednia elementow: 3
srednia elementow: 3
0 0 0 0 0
```

Parametry funkcji - const

- Jeśli funkcja nie powinna zmieniać wartości przekazywanych do niej zmiennych, to w nagłówku, przed odpowiednim parametrem, dodaje się identyfikator **const**

```
void drukuj(const int tab[])
{
    for (int i=0; i<5; i++)
    {
        printf("%3d", tab[i]);
        tab[i] = 0;
    }
    printf("\n");
}
```

- Próba zmiany wartości takiego parametru powoduje błąd kompilacji

```
error C3892: 'tab' : you cannot assign to a variable that is const
```

Parametry funkcji - const

- Przykładowe prototypy funkcji z pliku nagłówkowego `string.h`

```
char* strcpy(char *dest, const char *source);
```

```
size_t strlen(const char *str);
```

```
char* strdup(char *str);
```


Parametry funkcji - macierze

- Macierze przekazywane są do funkcji przez wskaźnik
- W nagłówku funkcji podaje się typ elementów tablicy, jej nazwę oraz w nawiasach kwadratowych liczbę wierszy i kolumn lub tylko liczbę kolumn

```
void fun(int tab[2][3])  
{  
    ...  
}
```

```
void fun(int tab[][3])  
{  
    ...  
}
```

- W wywołaniu funkcji podaje się tylko jej nazwę (bez nawiasów kwadratowych)

```
fun(tab);
```

Parametry funkcji - macierze (przykład)

```
#include <stdio.h>

void zero(int tab[][3])
{
    for (int i=0; i<2; i++)
        for (int j=0; j<3; j++)
            tab[i][j] = 0;
}

void drukuj(int tab[2][3])
{
    for (int i=0; i<2; i++)
    {
        for (int j=0; j<3; j++)
            printf("%3d", tab[i][j]);
        printf("\n");
    }
}
```

```
int main(void)
{
    int tab[2][3] =
        {1, 2, 3, 4, 5, 6};

    drukuj(tab);
    zero(tab);
    printf("\n");
    drukuj(tab);

    return 0;
}
```

Parametry funkcji - macierze (przykład)

```
#include <stdio.h>

void zero(int tab[][3])
{
    for (int i=0; i<2; i++)
        for (int j=0; j<3; j++)
            tab[i][j] = 0;
}

void drukuj(int tab[2][3])
{
    for (int i=0; i<2; i++)
    {
        for (int j=0; j<3; j++)
            printf("%3d", tab[i][j]);
        printf("\n");
    }
}
```

```
int main
{
    int t
    {
        0 0 0
        0 0 0

        druku
        zero(
        printf("\n");
        drukuj(tab);

        return 0;
    }
}
```

1	2	3
4	5	6
0	0	0
0	0	0

Parametry funkcji - struktury

- Struktury przekazywane są do funkcji przez wartość (nawet jeśli daną składową jest tablica)

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

struct pkt
{
    float x, y;
};

float odl(struct pkt pkt1, struct pkt pkt2)
{
    return sqrt(pow(pkt2.x-pkt1.x, 2) +
                pow(pkt2.y-pkt1.y, 2));
}
```

Parametry funkcji - struktury (przykład)

```
int main(void)
{
    struct pkt p1 = {2,3};
    struct pkt p2 = {-2,1};
    float wynik;

    wynik = odl(p1,p2);

    printf("Punkt nr 1: (%g,%g)\n",p1.x,p1.y);
    printf("Punkt nr 2: (%g,%g)\n",p2.x,p2.y);
    printf("Odleglosc = %g\n",wynik);

    return 0;
}
```

```
Punkt nr 1: (2,3)
Punkt nr 2: (-2,1)
Odleglosc = 4.47214
```

Pamięć a zmienne w programie

- Ze względu na czas życia wyróżnia się w programie:
 - **obiekty statyczne** - istnieją od chwili rozpoczęcia działania programu aż do jego zakończenia
 - **obiekty dynamiczne** - tworzone i usuwane z pamięci w trakcie wykonania programu
 - automatycznie (bez udziału programisty)
 - kontrolowane przez programistę
- O typie obiektu (**statyczny** lub **dynamiczny**) decyduje klasa pamięci obiektu (ang. storage class)
 - **auto** - zmienne automatyczne
 - **register** - zmienne umieszczane w rejestrach procesora
 - **extern** - zmienne zewnętrzne
 - **static** - zmienne statyczne

Zmienne automatyczne - auto

- Miejsce deklaracji: najczęściej początek bloku funkcyjnego ograniczonego nawiasami klamrowymi { i }
- Pamięć przydzielana automatycznie przy wejściu do bloku i zwalniana po wyjściu z niego
- Zakres widzialności: ograniczony do bloku, w którym zmienne zostały zadeklarowane (**zmienne lokalne**)
- Dostęp do zmiennych z innych bloków możliwy przez wskaźnik
- Jeśli zmienne są inicjalizowane, to odbywa się ona przy każdym wejściu do bloku, w którym zostały zadeklarowane
- Nie ma potrzeby jawnego używania **auto**, gdyż domyślnie zmienne wewnątrz bloków funkcyjnych są lokalne

```
auto int x;
```

Zmienne rejestrowe - register

- Zazwyczaj o miejscu umieszczenia zmiennej automatycznej decyduje kompilator:
 - pamięć operacyjna - wolniejszy dostęp
 - rejestry procesora - szybszy dostęp
- Programista może zasugerować kompilatorowi umieszczenie określonej zmiennej automatycznej w rejestrach procesora
- Najczęściej dotyczy to zmiennych:
 - często używanych
 - takich, dla których czas dostępu jest bardzo ważny

```
register int x;
```


Zmienne zewnętrzne - extern

- Miejsce deklaracji: poza blokami funkcyjnymi, najczęściej na początku pliku z kodem źródłowym
- Pamięć na zmienne jest przydzielana, gdy program rozpoczyna pracę i zwalniana, gdy program kończy się
- Zakres widzialności: globalny - od miejsca deklaracji do końca pliku z kodem źródłowym (**zmienne globalne**)
- Jeśli inna zmienna lokalna, ma taką samą nazwę jak globalna, to lokalna przesłania widoczność zmiennej globalnej
- W większości implementacji języka C zmienne **extern** są automatycznie inicjalizowane **zerem**
- Etykieta **extern** może być pominięta (chyba, że program składa się z kilku plików z kodem źródłowym)
- Zalecane jest ograniczenie stosowania zmiennych globalnych

Zmienne statyczne - static

- Miejsce deklaracji: w bloku funkcyjnym jako automatyczne lub poza blokami funkcyjnymi, jako globalne
- Istnieją przez cały czas wykonywania programu, nawet po zakończeniu bloku funkcyjnego, w którym zostały zadeklarowane
- Zakres widzialności: zależny od sposobu deklaracji (automatyczne lub globalne)
- Zmienne **static** są automatycznie inicjalizowane zerem
- Mogą być inicjalizowane podczas deklaracji (tylko stałą wartością), inicjalizacja jest wykonywana tylko raz, podczas kompilacji programu

```
static int x = 10;
```

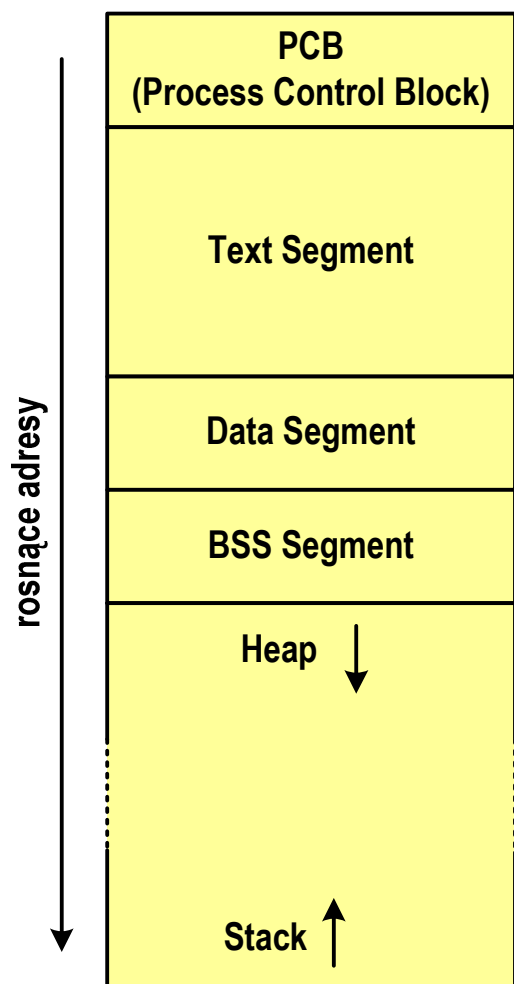
Klasy pamięci zmiennych

```
int a;                /* extern int a; - zmienna globalna */
void foo();

int main(void)
{
    int b;            /* auto int b; - zmienna lokalna */
    register float a; /* zmienna automatyczna, rejestrowa */
    foo(); foo(); foo();
    return 0;
}

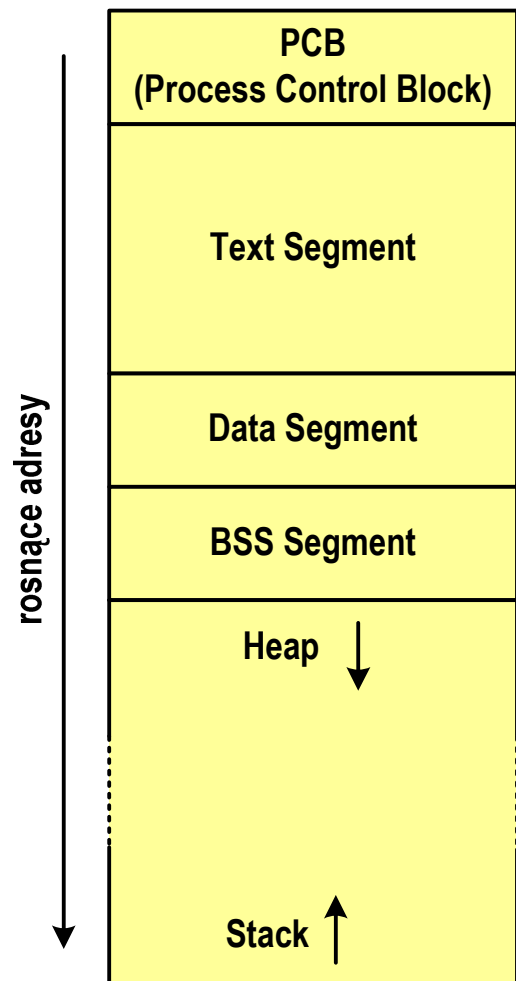
void foo()
{
    static int c = 1; /* zmienna statyczna */
    {
        double a;    /* zmienna lokalna */
    }
    c++;
}
```

Struktura procesu w pamięci komputera



- **PCB** - blok kontrolny procesu
 - obszar pamięci operacyjnej zarezerwowany przez system operacyjny do zarządzania procesem
- **Text Segment**
 - kod programu czyli instrukcje w postaci binarnej
- **Data Segment**
 - zmienne globalne i statyczne zainicjalizowane niezerowymi wartościami
- **BSS Segment** (Block Started by Symbol)
 - zmienne globalne i statyczne domyślnie zainicjalizowane zerowymi wartościami

Struktura procesu w pamięci komputera



- **Heap** - sverta
 - obszar zmiennych dynamicznych
 - pamięć w obszarze sterty przydzielana jest funkcjami **calloc()** i **malloc()**
- **Stack** - stos
 - zmienne lokalne (automatyczne)
 - parametry funkcji i adresy powrotu z funkcji (stack frame)

Zmienne w pamięci komputera

```
int a;                                /* BSS Segment */
void foo();

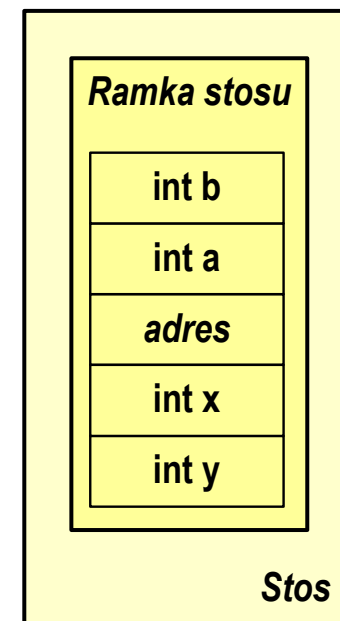
int main(void)
{
    int b;                             /* Stack */
    float *a;                          /* Stack */
    a = (float *) malloc(400);         /* Heap - 400 bajtów */
    return 0;
}

void foo()
{
    static int c = 1;                 /* Data Segment */
    {
        double a;                    /* Stack */
    }
    c++;
}
```

Ramka stosu (stack frame)

- Każde wywołanie funkcji powoduje odłożenie na stosie tzw. **ramki stosu**

```
void fun(int x, int y)
{
    int a, b;
}
```



Programy wielomodułowe

(Przykład w Visual C++ 2008)

Koniec wykładu nr 6

Dziękuję za uwagę!