

Stefan Sokołowski

JĘZYKI FORMALNE I METODY KOMPILACJI

wykład 11

**Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025**

W realnych komputerach...

W realnych komputerach...

...odchodzimy od tego, że każda liczba zajmuje **jedną** komórkę.

W realnych komputerach...

... odchodzimy od tego, że każda liczba zajmuje **jedną** komórkę.

Programowanie w języku wewnętrznym

- jest **podobne** do przedstawionego na poprzednich wykładach,
- ale nie jest **dokładnie takie**.

W realnych komputerach...

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$)

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą
z przedziału $[0..255]$ ($2^8 = 256$), albo
z przedziału $[-128..127]$

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą
z przedziału $[0..255]$ ($2^8 = 256$), albo
z przedziału $[-128..127]$

MAŁO!

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą
z przedziału $[0..255]$ ($2^8 = 256$), albo
z przedziału $[-128..127]$
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

MAŁO!

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

MAŁO!

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą
 z przedziału $[0..255]$ ($2^8 = 256$), albo
 z przedziału $[-128..127]$
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

MAŁO!

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

MAŁO!

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$
float	4	

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$
float	4	niska precyzja, 6–7 cyfr znaczących

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$
float	4	niska precyzja, 6–7 cyfr znaczących UNIKAĆ!

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0 \dots 255]$ ($2^8 = 256$), albo z przedziału $[-128 \dots 127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0 \dots 2^{32} - 1] \simeq [0 \dots 4 \text{ mld}]$
int	4	zakres: $[-2^{31} \dots 2^{31} - 1] \simeq [-2 \text{ mld} \dots 2 \text{ mld}]$
float	4	niska precyzja, 6–7 cyfr znaczących UNIKAĆ!
double	8	15–16 cyfr znaczących

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$
float	4	niska precyzja, 6–7 cyfr znaczących UNIKAĆ!
double	8	15–16 cyfr znaczących
tablice	...	

W realnych komputerach...

- adresowane są nie **komórki** tylko **bajty**; bajt zawiera 8 bitów — to wystarcza na liczbę całkowitą z przedziału $[0..255]$ ($2^8 = 256$), albo z przedziału $[-128..127]$ **MAŁO!**
- na ogół operujemy na danych znacznie wykraczających poza 1 bajt

Zwykle (choć nie zawsze):

typ	bajty	
unsigned int	4	zakres: $[0..2^{32} - 1] \simeq [0..4 \text{ mld}]$
int	4	zakres: $[-2^{31}..2^{31} - 1] \simeq [-2 \text{ mld}..2 \text{ mld}]$
float	4	niska precyzja, 6–7 cyfr znaczących UNIKAĆ!
double	8	15–16 cyfr znaczących
tablice	...	
struktury	...	
...	...	

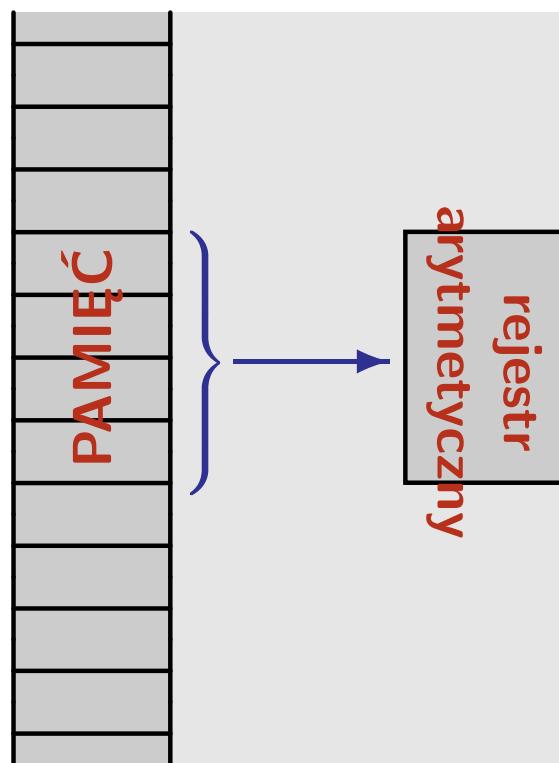
W realnych komputerach...

Rejestry arytmetyczne również są większe niż 1 bajt...

W realnych komputerach...

Rejestry arytmetyczne również są większe niż 1 bajt...

Przykładowe komendy:

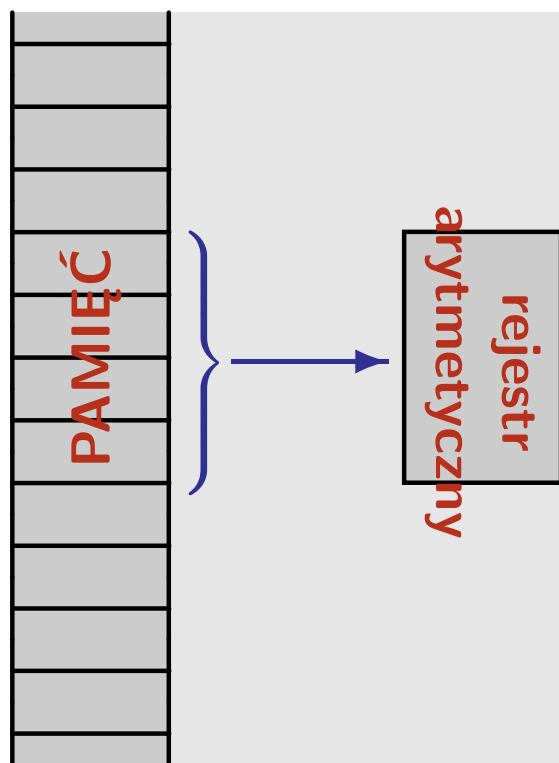


skopiować blok bajtów
z pamięci do rejestru

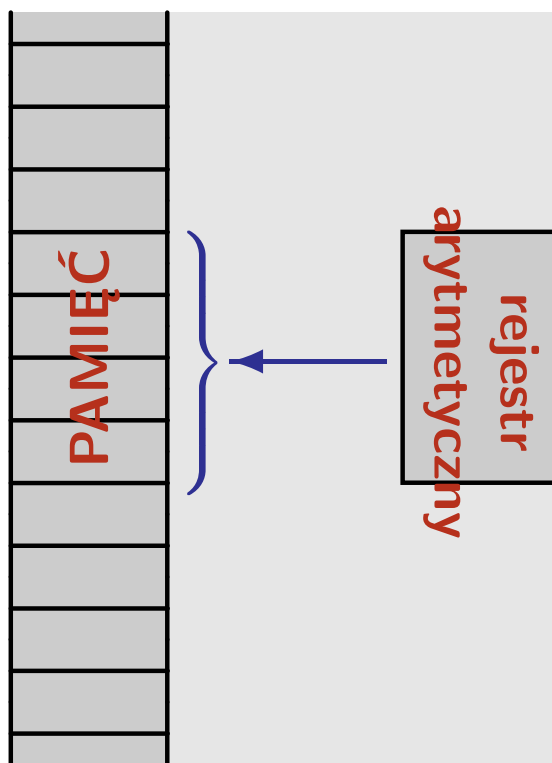
W realnych komputerach...

Rejestry arytmetyczne również są większe niż 1 bajt...

Przykładowe komendy:



skopiować blok bajtów
z pamięci do rejestru

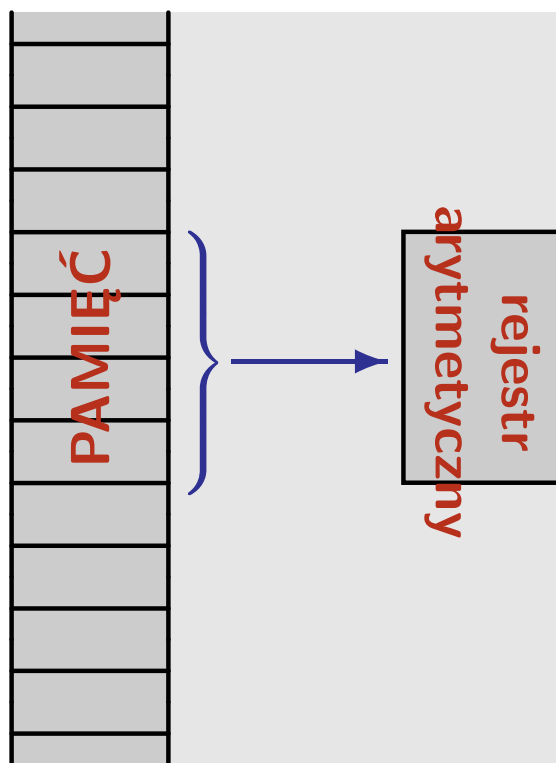


skopiować blok bajtów
z rejestru do pamięci

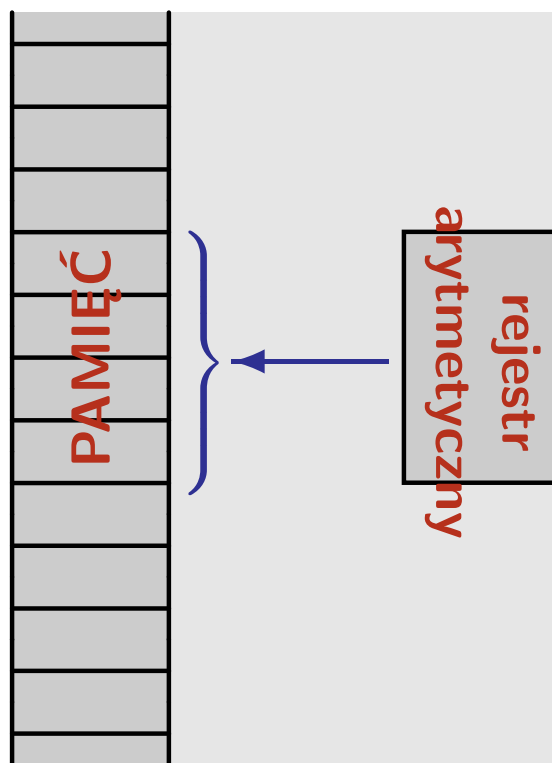
W realnych komputerach...

Rejestry arytmetyczne również są większe niż 1 bajt...

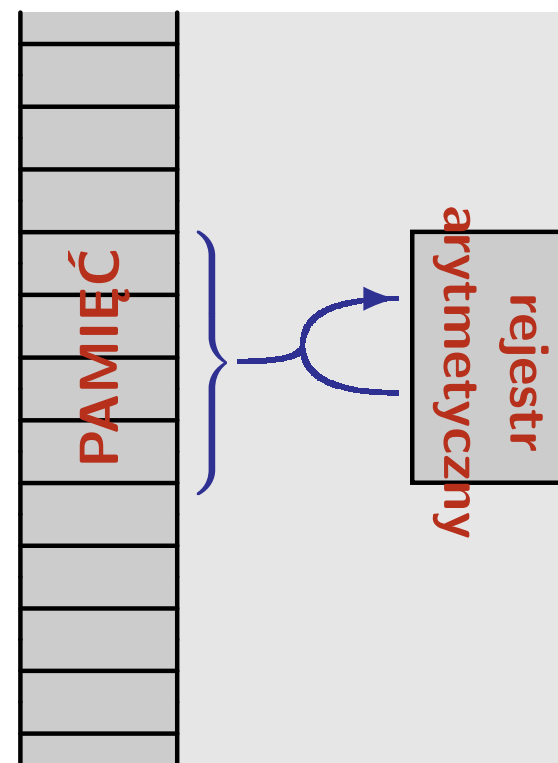
Przykładowe komendy:



skopiować blok bajtów
z pamięci do rejestru



skopiować blok bajtów
z rejestru do pamięci



dodać blok bajtów
z pamięci do rejestru

Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

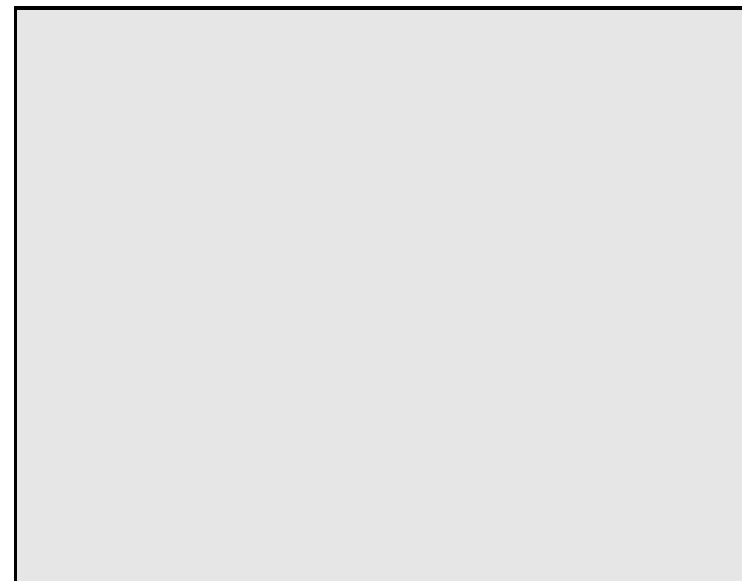
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

PRZEKŁAD



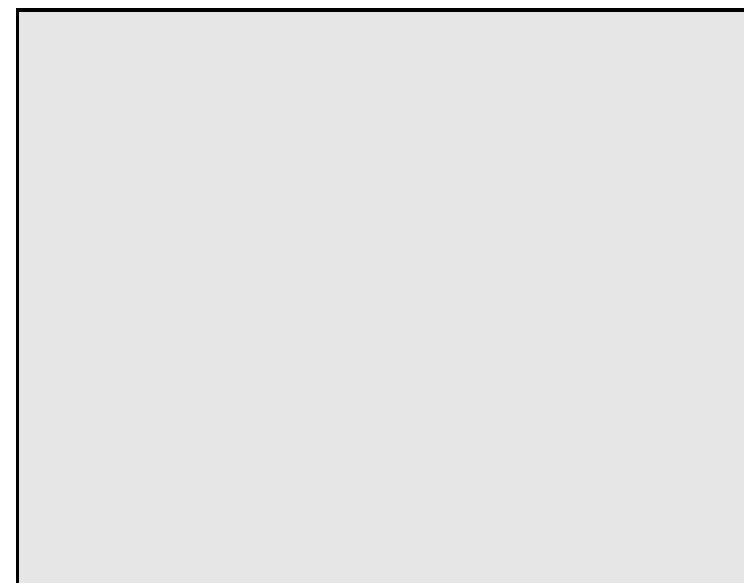
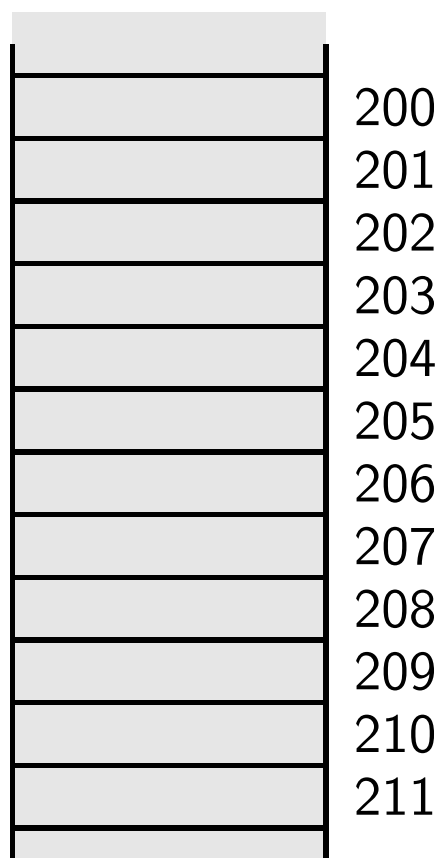
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

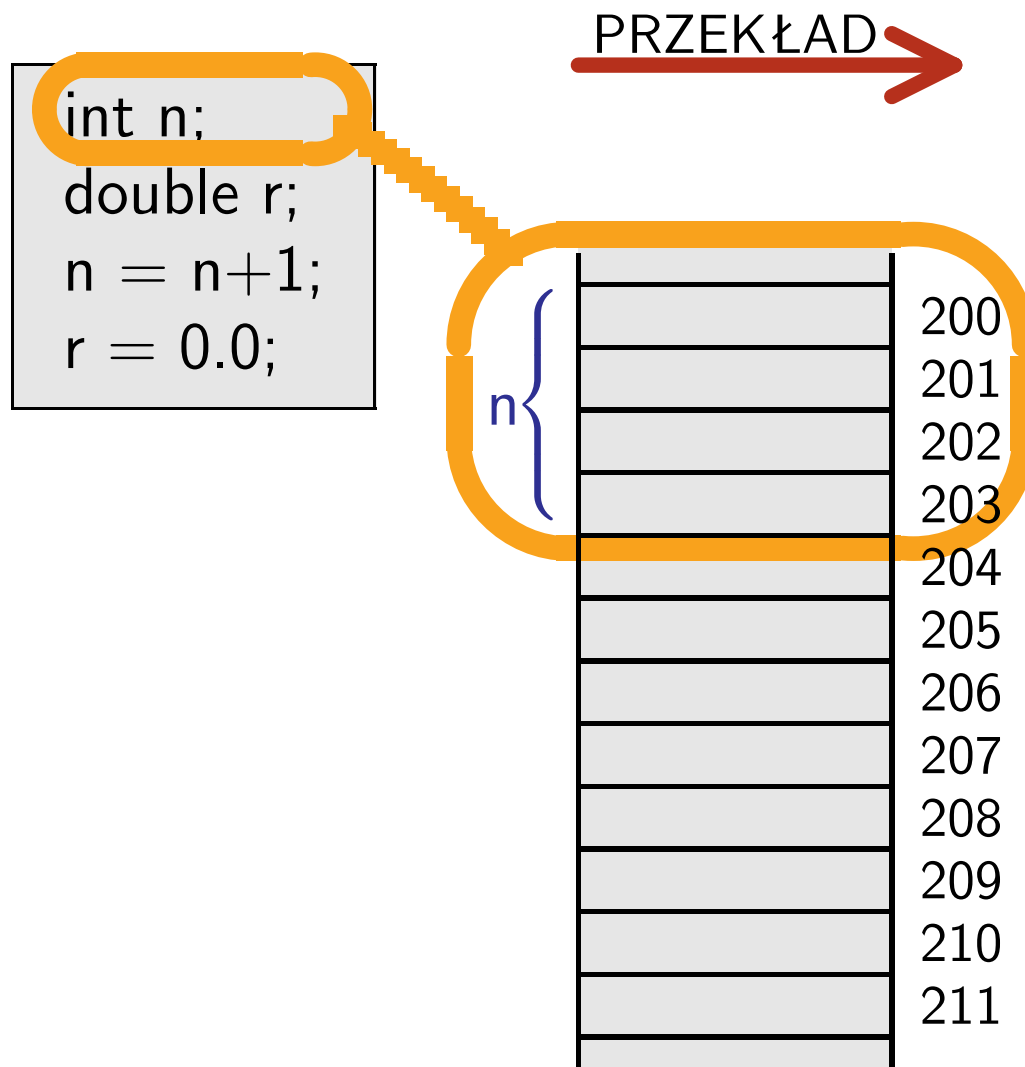
PRZEKŁAD →



Organizacja pamięci

Założenie:

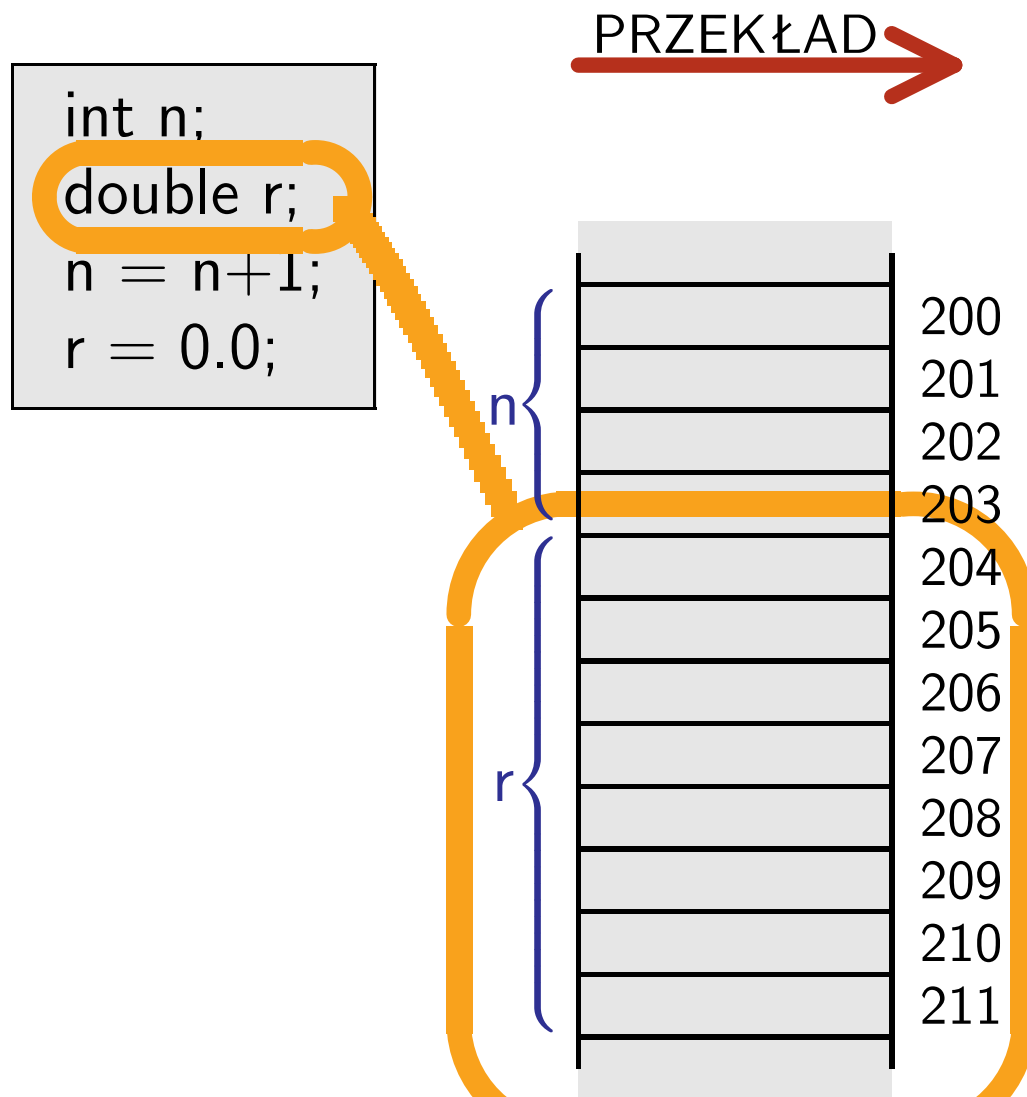
- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów



Organizacja pamięci

Założenie:

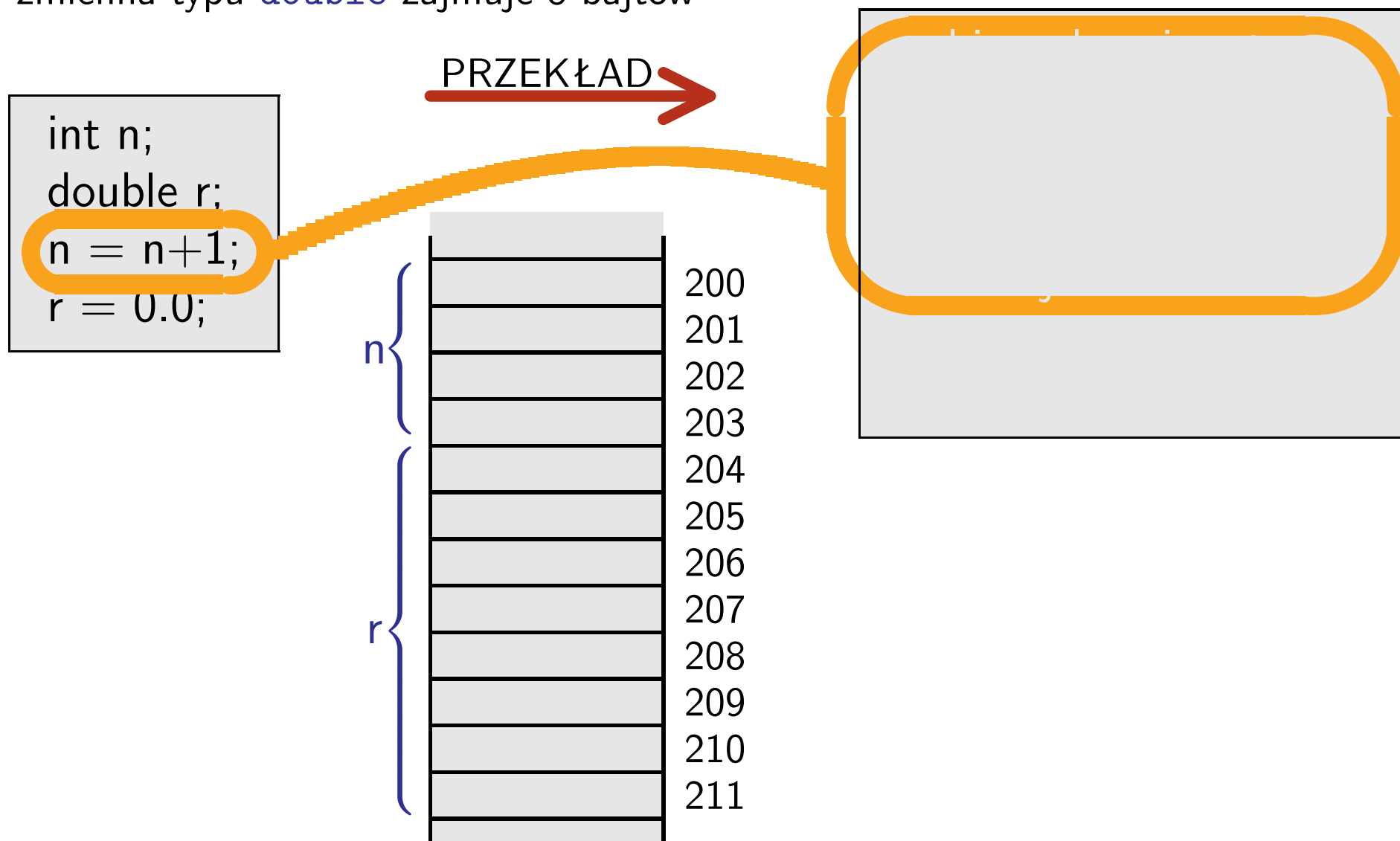
- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów



Organizacja pamięci

Założenie:

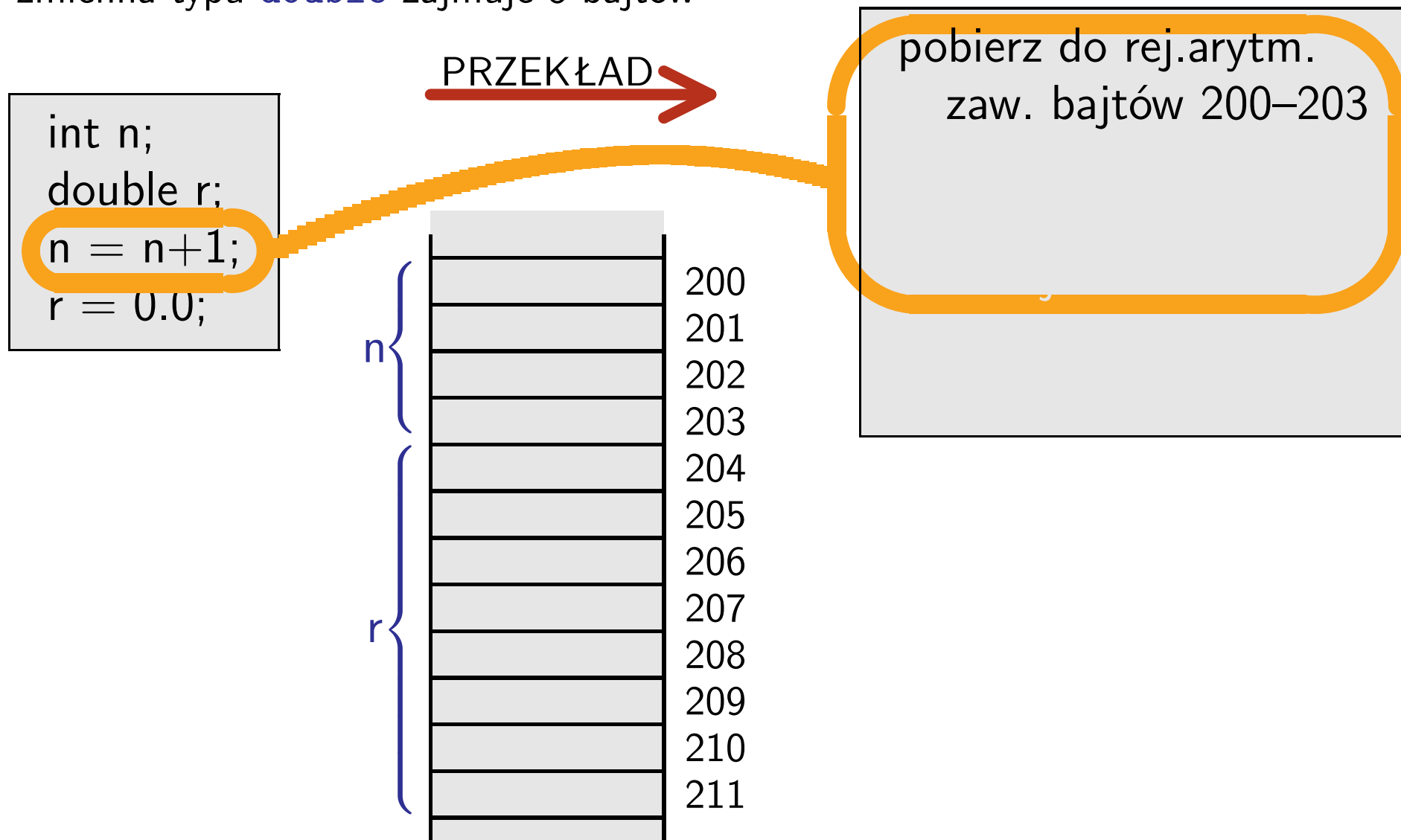
- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów



Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów



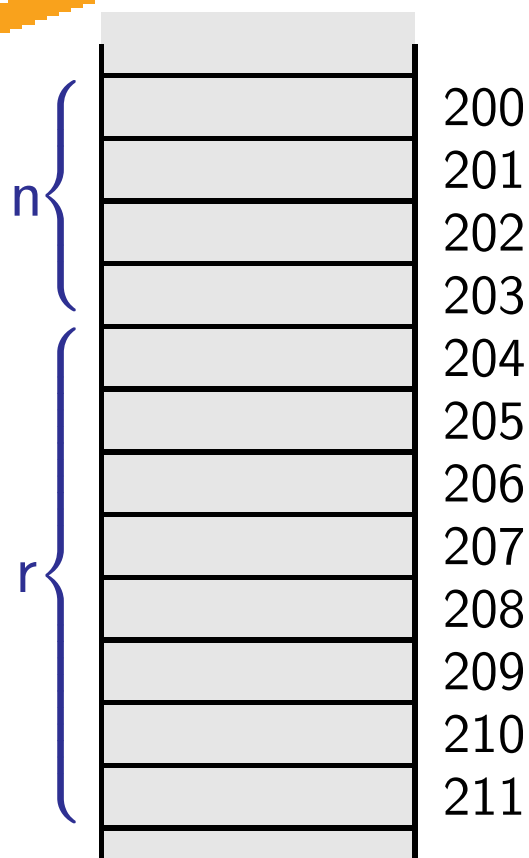
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n + 1;  
r = 0.0;
```

PRZEKŁAD →



pobierz do rej.arytm.
zaw. bajtów 200–203
dodaj 1 do rej.arytm.

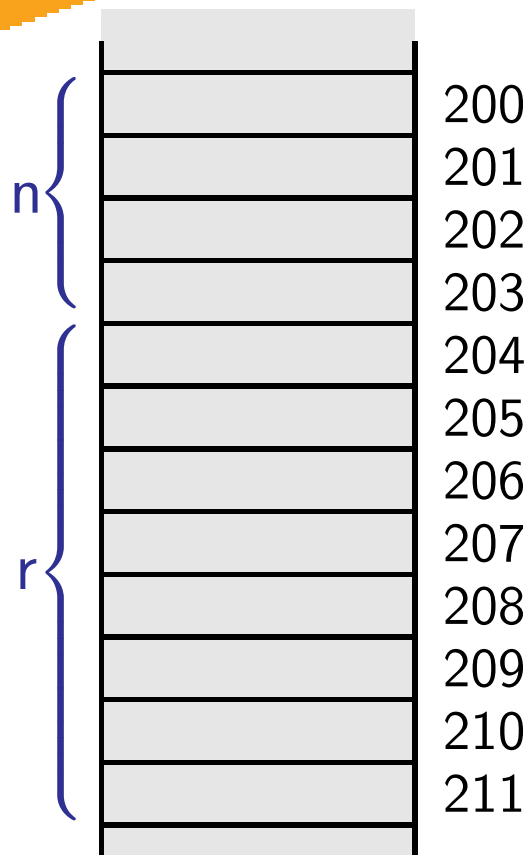
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

PRZEKŁAD →



pobierz do rej.arytm.
zaw. bajtów 200–203
dodaj 1 do rej.arytm.
zapisz stan rej.arytm.
w bajtach 200–203

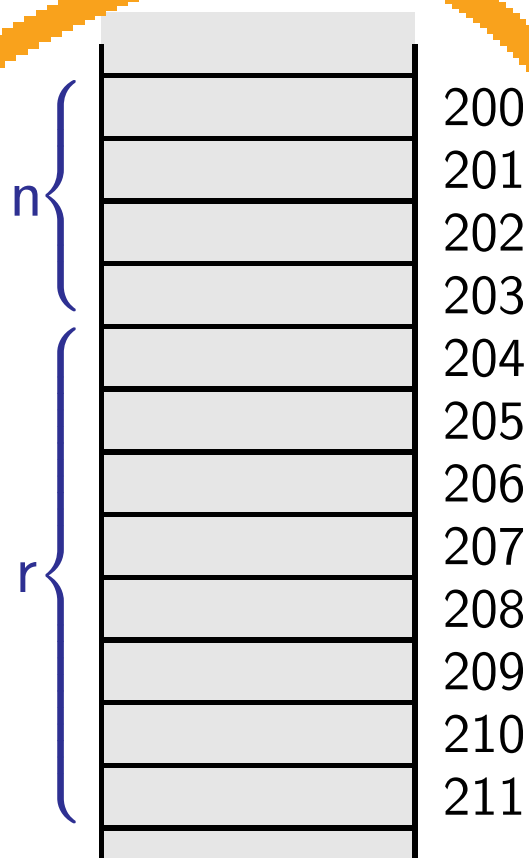
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

PRZEKŁAD →



pobierz do rej.arytm.
zaw. bajtów 200–203
dodaj 1 do rej.arytm.
zapisz stan rej.arytm.
w bajtach 200–203

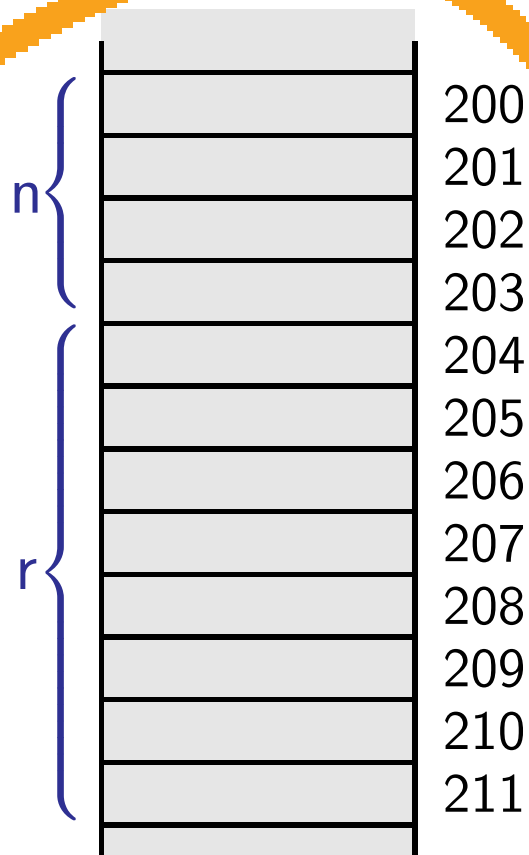
Organizacja pamięci

Założenie:

- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów

```
int n;  
double r;  
n = n+1;  
r = 0.0;
```

PRZEKŁAD →



pobierz do rej.arytm.
zaw. bajtów 200–203
dodaj 1 do rej.arytm.
zapisz stan rej.arytm.
w bajtach 200–203
wyzeruj
bajty 204–211

Organizacja pamięci

Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej

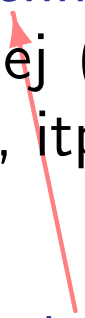
Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);

Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
 - instrukcje
 jęz. źródłowego dotyczące tej zmiennej
- 

Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
 języ. źródłowego dotyczące tej zmiennej
są tłumaczone na
 instrukcje języ. wewnętrznego dotyczące tego bloku

Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
 jęz. źródłowego dotyczące tej zmiennej
są tłumaczone na
 instrukcje jęz. wewnętrznego dotyczące tego bloku
 (odnoszące się do adresów zawartych w tym bloku)

Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
 języ. źródłowego dotyczące tej zmiennej
są tłumaczone na
 instrukcje języ. wewnętrznego dotyczące tego bloku
 (odnoszące się do adresów zawartych w tym bloku);
- tablice zajmują spójne obszary pamięci o kolejnych adresach

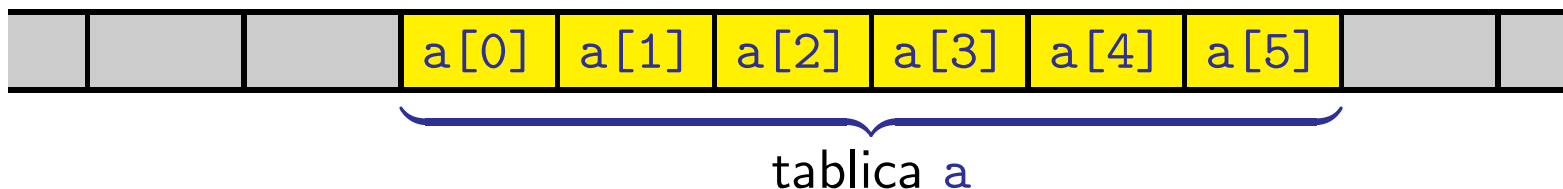
- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
 jęz. źródłowego dotyczące tej zmiennej
są tłumaczone na
 instrukcje jęz. wewnętrznego dotyczące tego bloku
 (odnoszące się do adresów zawartych w tym bloku);
- tablice zajmują spójne obszary pamięci o kolejnych adresach:



Organizacja pamięci

W czasie kompilacji:

- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
 jęz. źródłowego dotyczące tej zmiennej
są tłumaczone na
 instrukcje jęz. wewnętrznego dotyczące tego bloku
 (odnoszące się do adresów zawartych w tym bloku);
- tablice zajmują spójne obszary pamięci o kolejnych adresach:



Organizacja pamięci

Tablice wielowymiarowe

Organizacja pamięci

Tablice wielowymiarowe:

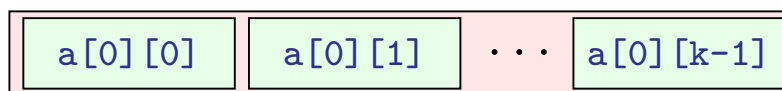
punkt widzenia programisty

Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:

wiersz $a[0]$:



Organizacja pamięci

Tablice wielowymiarowe:

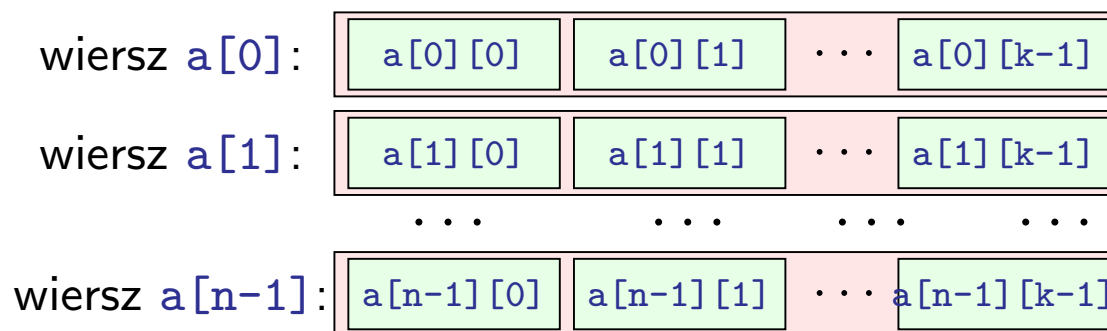
punkt widzenia programisty:

wiersz $a[0]$:	$a[0][0]$	$a[0][1]$	$\dots$	$a[0][k-1]$
wiersz $a[1]$:	$a[1][0]$	$a[1][1]$	$\dots$	$a[1][k-1]$

Organizacja pamięci

Tablice wielowymiarowe:

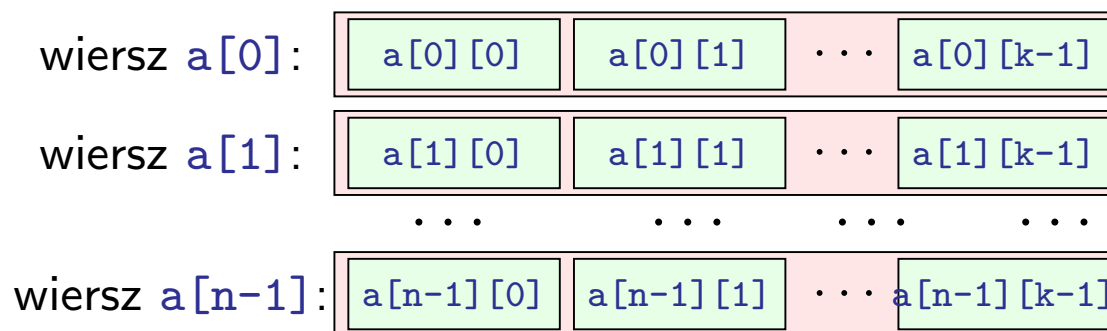
punkt widzenia programisty:



Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:

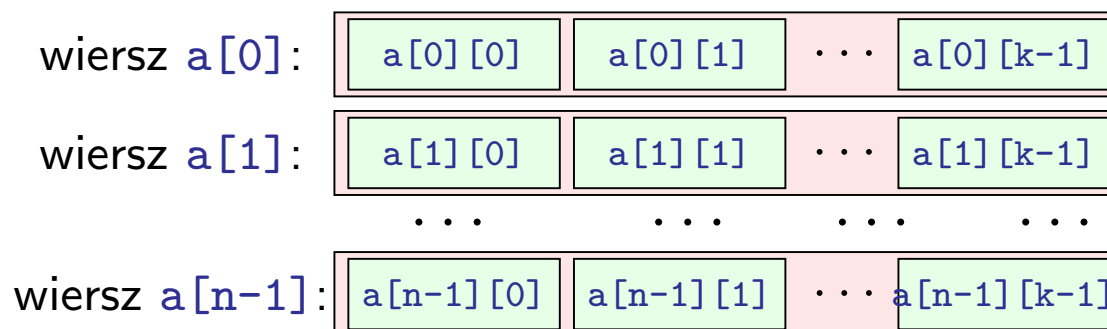


realizacja komputerowa

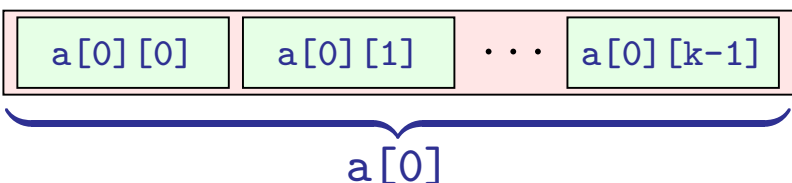
Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:



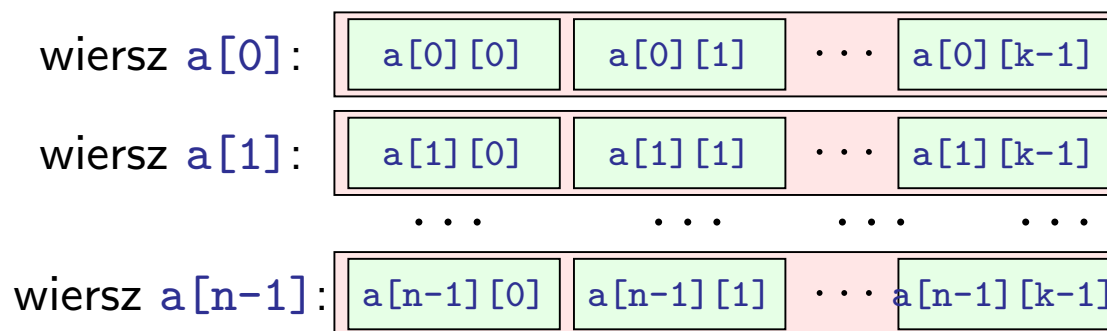
realizacja komputerowa:



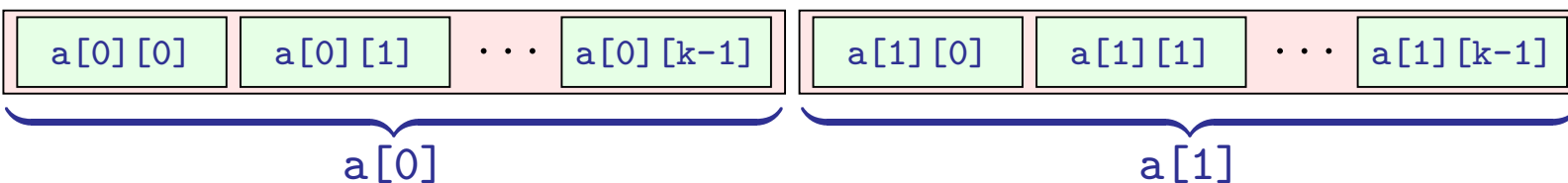
Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:



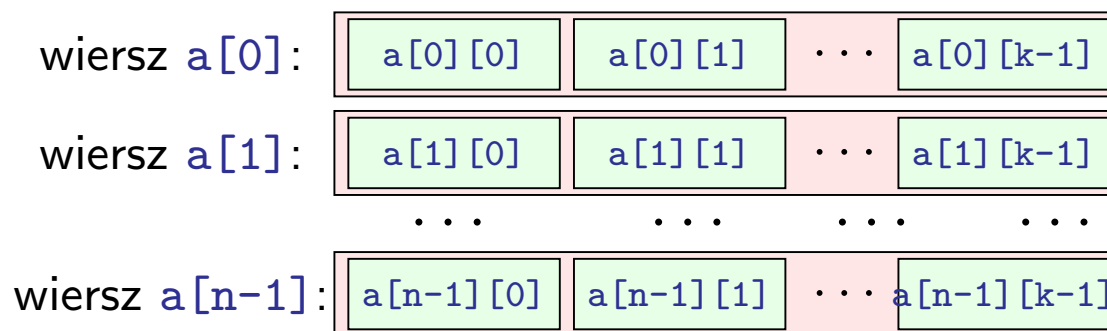
realizacja komputerowa:



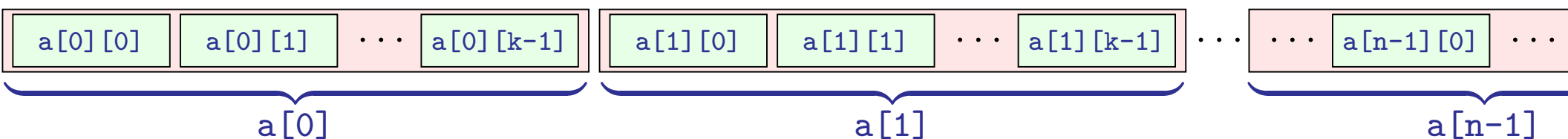
Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:



realizacja komputerowa:



Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0; ?`

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0; ?`

	0	1	2	...	18	19
0				...		
1				...		
2				...		
...	...	...	...	...	...	...
8				...		
9				...		

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb.

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza;

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza; a więc jest poprzedzony $2 \cdot 20 + 1$ liczbami tablicy;

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza; a więc jest poprzedzony $2 \cdot 20 + 1$ liczbami tablicy; a więc jest odsunięty od początku obszaru poświęconego na tablicę o $(2 \cdot 20 + 1) \cdot 4$ bajtów;

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza; a więc jest poprzedzony $2 \cdot 20 + 1$ liczbami tablicy; a więc jest odsunięty od początku obszaru poświęconego na tablicę o $(2 \cdot 20 + 1) \cdot 4$ bajtów; wobec tego zajmuje bajty od $1000 + (2 \cdot 20 + 1) \cdot 4$ do $1003 + (2 \cdot 20 + 1) \cdot 4$

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza; a więc jest poprzedzony $2 \cdot 20 + 1$ liczbami tablicy; a więc jest odsunięty od początku obszaru poświęconego na tablicę o $(2 \cdot 20 + 1) \cdot 4$ bajtów; wobec tego zajmuje bajty od $1000 + (2 \cdot 20 + 1) \cdot 4$ do $1003 + (2 \cdot 20 + 1) \cdot 4$ czyli 1164–1167.

Organizacja pamięci

Przykład:

Mamy deklarację tablicy: `int tab [10][20];`

Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000.

Jak zostanie przetłumaczona komenda `tab[2][1] = 0;` ?

	0	1	2	...	18	19
0	1000	1004	1008	...	1072	1076
1	1080	1084	1088	...	1152	1156
2	1160	1164	1168	...	1232	1236
...	...	...	...	...	...	...
8	1640	1644	1648	...	1712	1716
9	1720	1724	1728	...	1792	1796

Odpowiedź: Każdy wiersz tablicy zawiera 20 liczb. Element `tab[2][1]` jest poprzedzony 2 całymi wierszami tablicy po 20 liczb oraz 1 liczbą własnego wiersza; a więc jest poprzedzony $2 \cdot 20 + 1$ liczbami tablicy; a więc jest odsunięty od początku obszaru poświęconego na tablicę o $(2 \cdot 20 + 1) \cdot 4$ bajtów; wobec tego zajmuje bajty od $1000 + (2 \cdot 20 + 1) \cdot 4$ do $1003 + (2 \cdot 20 + 1) \cdot 4$ czyli 1164–1167.

A więc tłumaczenie komendy: **wyzerować bajty 1164–1167** .

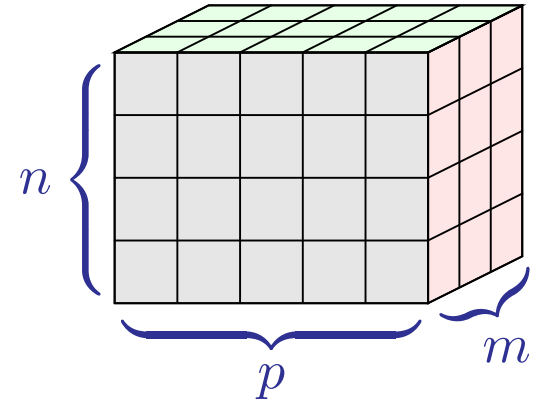
Organizacja pamięci

Tablice wielowymiarowe

Organizacja pamięci

Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

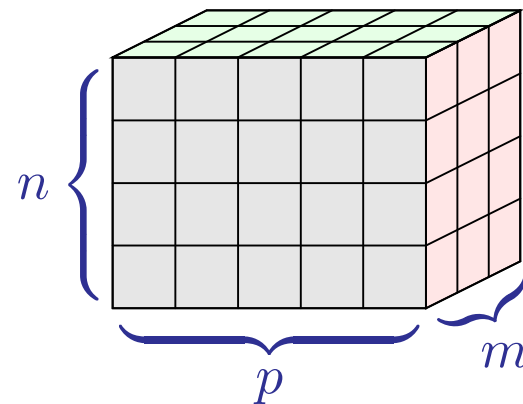


Organizacja pamięci

Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw



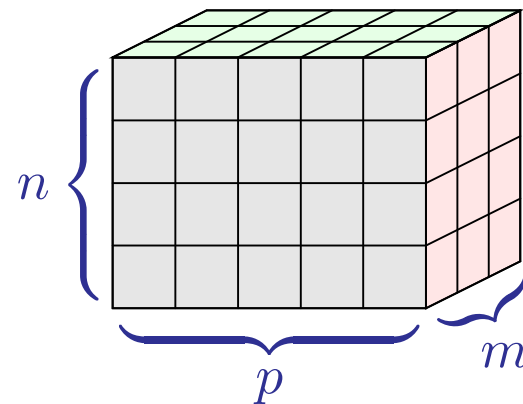
Organizacja pamięci

Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie



Organizacja pamięci

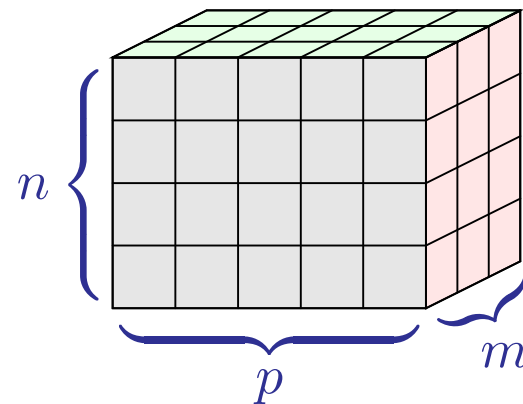
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Organizacja pamięci

Tablice wielowymiarowe:

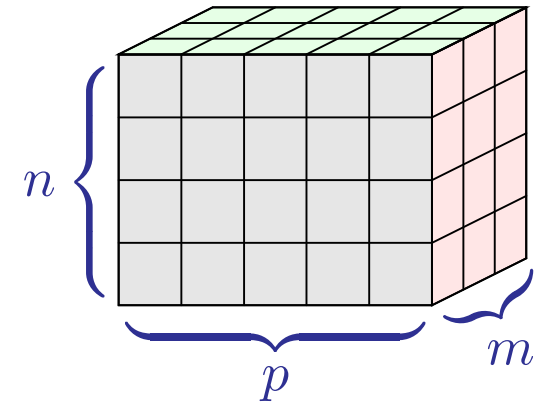
Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu

Założmy, że pojedynczy element typu T zajmuje b bajtów



Organizacja pamięci

Tablice wielowymiarowe:

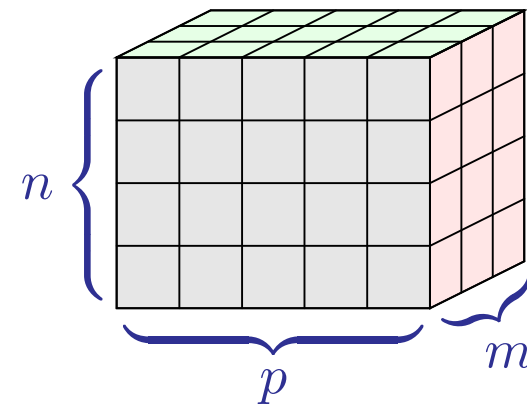
Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu

Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .



Organizacja pamięci

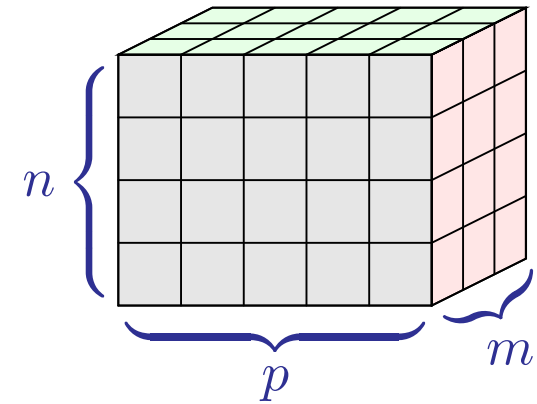
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Organizacja pamięci

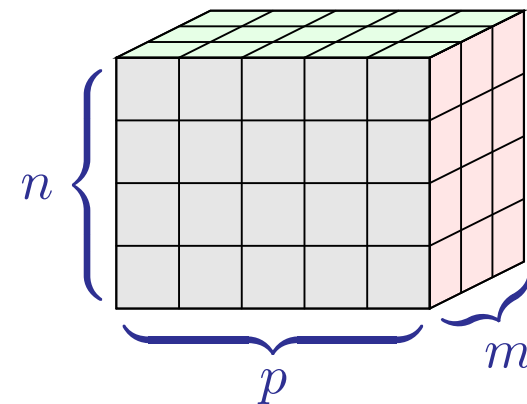
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go

Organizacja pamięci

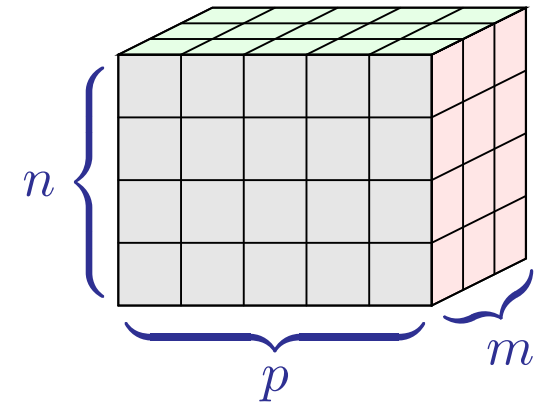
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

Organizacja pamięci

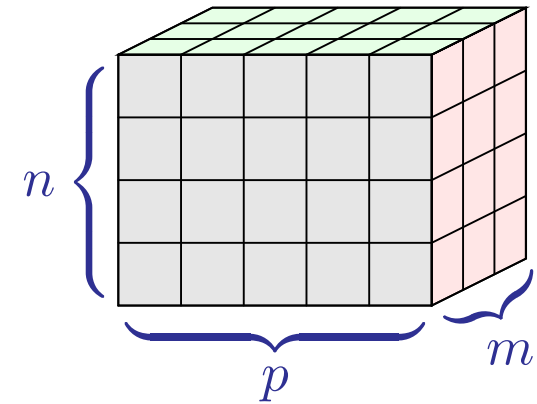
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

Organizacja pamięci

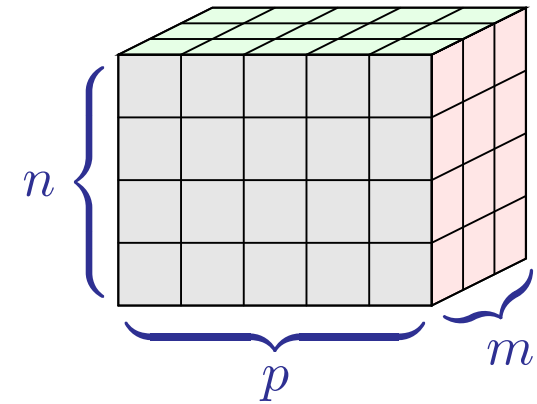
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

Organizacja pamięci

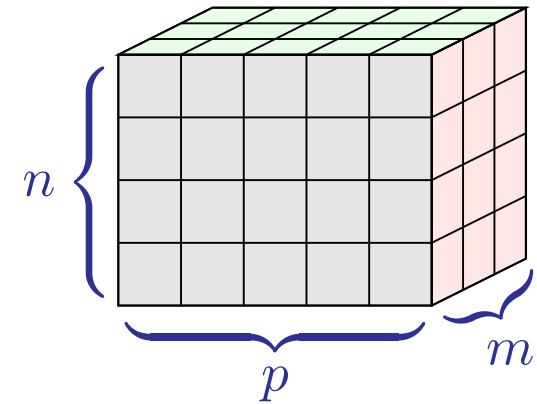
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

$$\text{Razem : } i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$$

Organizacja pamięci

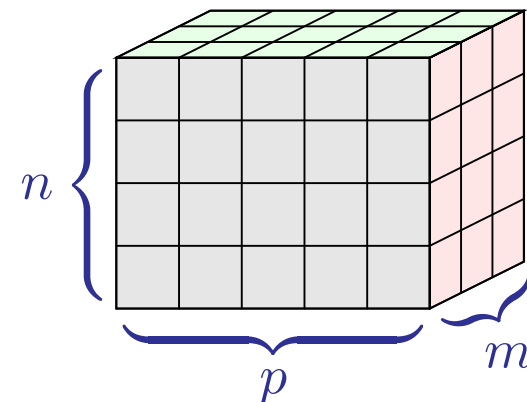
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

Organizacja pamięci

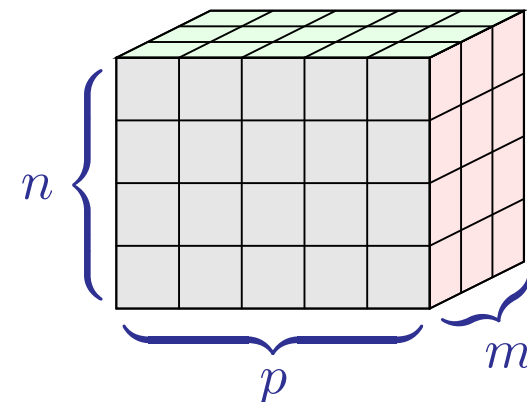
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci

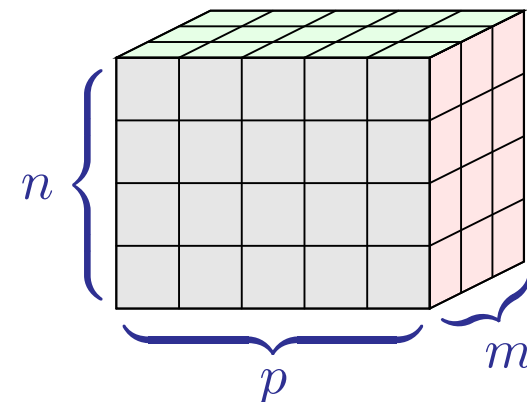
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

deklaracja: $\text{tab}[n][m][p]$

element: $\text{tab}[i][j][k]$

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci

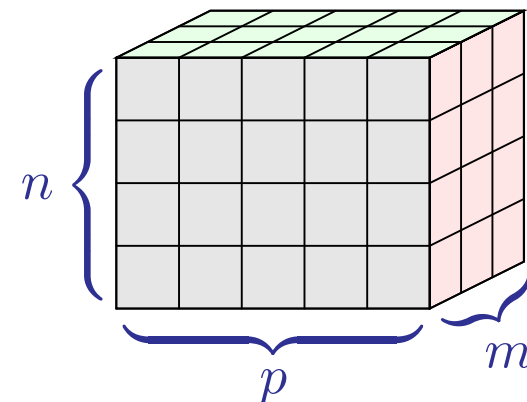
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

deklaracja: $\text{tab}[n][m][p]$

element: $\text{tab}[i][j][k]$

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci

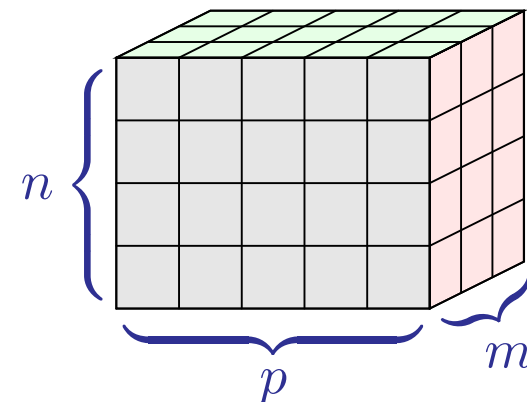
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

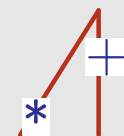
Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

deklaracja: $\text{tab}[n][m][p]$



element: $\text{tab}[i][j][k]$

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci

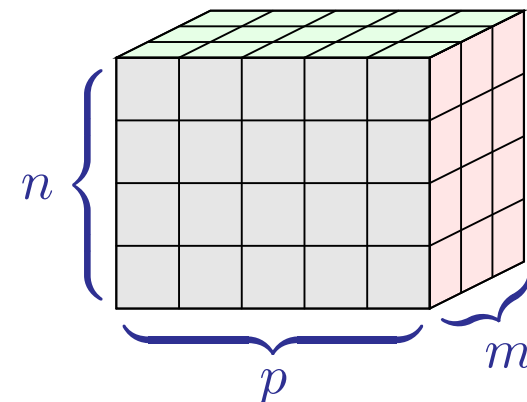
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

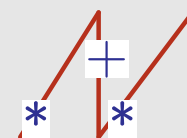
Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

deklaracja: $\text{tab}[n][m][p]$



element: $\text{tab}[i][j][k]$

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci

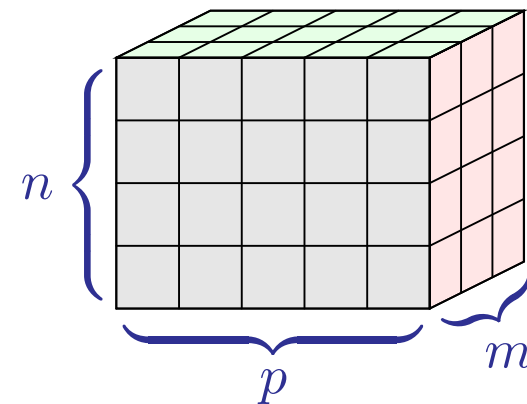
Tablice wielowymiarowe:

Deklaracja: $T \text{ tab}[n][m][p];$

czyli n warstw po

m wierszy w warstwie po

p elementów typu T w wierszu



Założmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a .

Które adresy należą do elementu $\text{tab}[i][j][k]$?

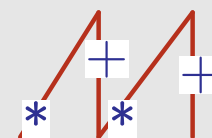
Poprzedza go:

i warstw po $m \cdot p$ w warstwie : $i \cdot m \cdot p$

j wierszy własnej warstwy po p w wierszu : $j \cdot p$

k elementów własnego wiersza : k

deklaracja: $\text{tab}[n][m][p]$



element: $\text{tab}[i][j][k]$

Razem : $i \cdot m \cdot p + j \cdot p + k = (i \cdot m + j) \cdot p + k$

czyli $((i \cdot m + j) \cdot p + k) \cdot b$ bajtów

więc **zaczyna się** w bajcie $((i \cdot m + j) \cdot p + k) \cdot b + a$

Organizacja pamięci w czasie działania

```
typedef struct os {  
    char nazw[26];  
    int zar;  
} Osoba;  
  
Osoba kart[1000]; int p;  
  
int main () {  
    p = 500;  
  
    kart[p].zar =  
        kart[p+1].zar + 200;  
  
    p = 720;  
  
    kart[p].nazw[5] = 'a';  
  
    return 0;  
}
```


Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

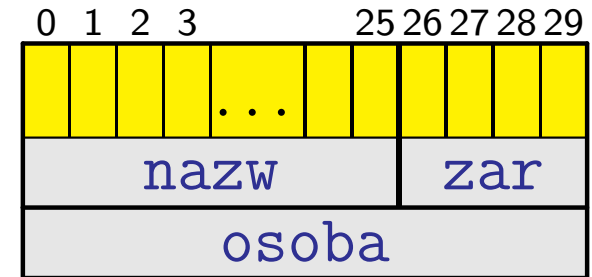
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

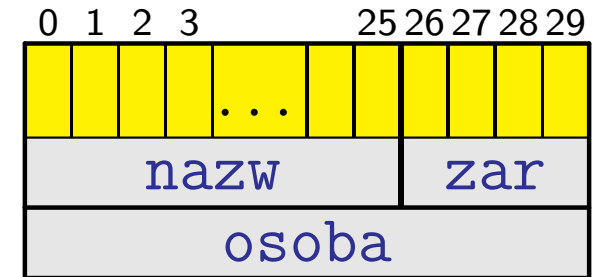
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

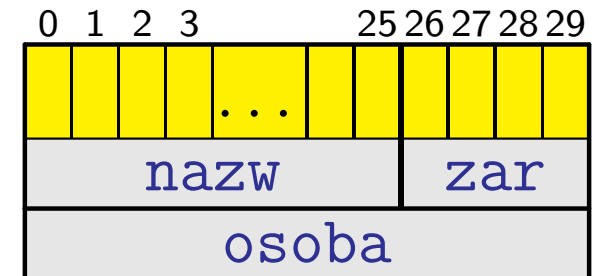
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

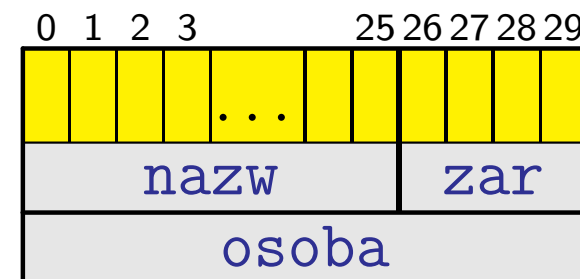
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

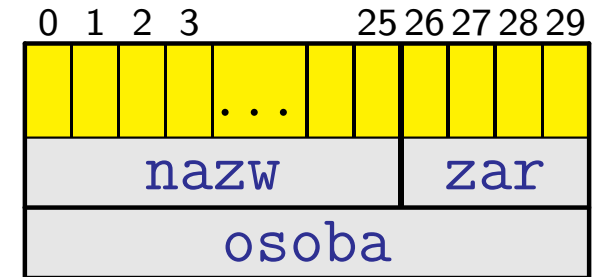
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

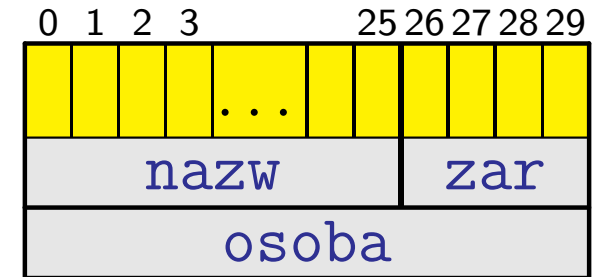
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
<code>x.nazw</code>	$(\text{adres } x)$
<code>x.nazw[i]</code>	$(\text{adres } x) + i$
<code>x.zar</code>	$(\text{adres } x) + 26$
<code>kart[0]</code>	$(\text{adres } \text{kart})$
<code>kart[j]</code>	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

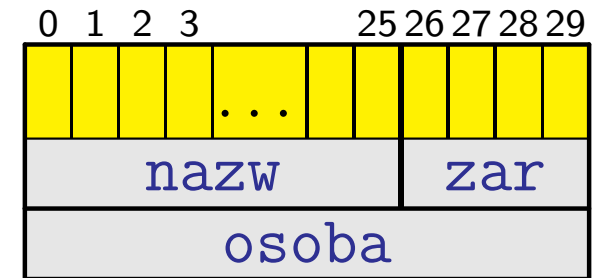
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
<code>x.nazw</code>	$(\text{adres } x)$
<code>x.nazw[i]</code>	$(\text{adres } x) + i$
<code>x.zar</code>	$(\text{adres } x) + 26$
<code>kart[0]</code>	$(\text{adres } \text{kart})$
<code>kart[j]</code>	$(\text{adres } \text{kart}) + j \cdot 30$
<code>kart[j].nazw[i]</code>	

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

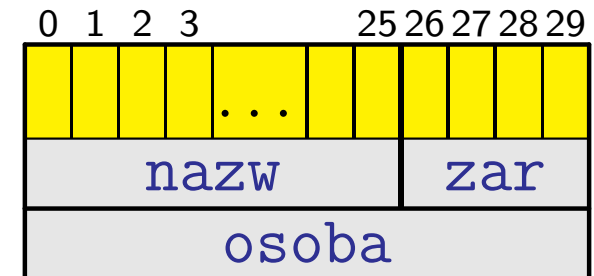
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
<code>x.nazw</code>	$(\text{adres } x)$
<code>x.nazw[i]</code>	$(\text{adres } x) + i$
<code>x.zar</code>	$(\text{adres } x) + 26$
<code>kart[0]</code>	$(\text{adres } \text{kart})$
<code>kart[j]</code>	$(\text{adres } \text{kart}) + j \cdot 30$
<code>kart[j].nazw[i]</code>	$(\text{adres } \text{kart}) + j \cdot 30 + i$

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

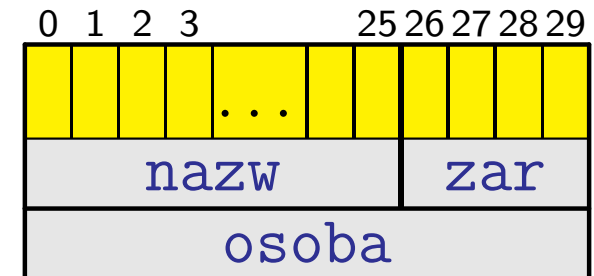
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

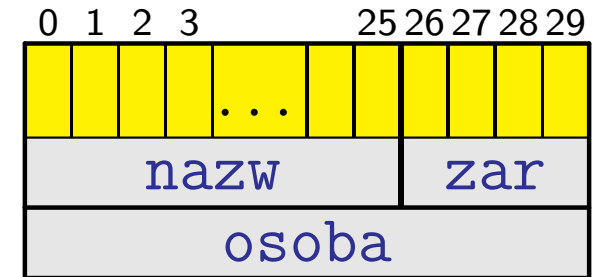
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

pobierz zaw. 4 bajtów od (adres p)

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

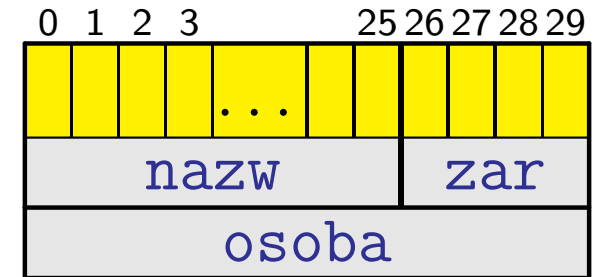
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

pobierz zaw. 4 bajtów od (adres p)
pomnóż przez 30

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

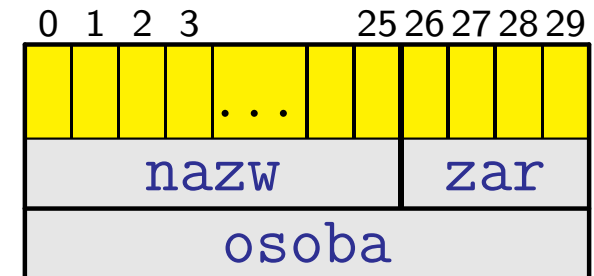
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

pobierz zaw. 4 bajtów od (adres p)
 pomnóż przez 30
 dodaj 5

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

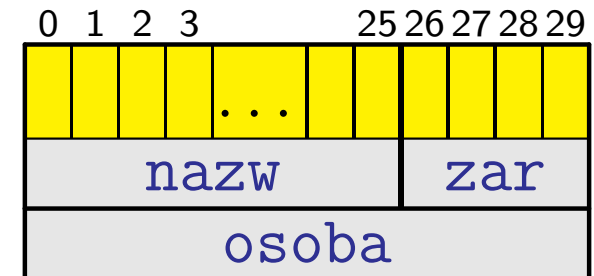
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

pobierz zaw. 4 bajtów od (adres p)
 pomnóż przez 30
 dodaj 5
 dodaj (adres kart)

Organizacja pamięci w czasie działania

```
typedef struct os {
    char nazw[26];
    int zar;
} Osoba;

Osoba kart[1000]; int p;

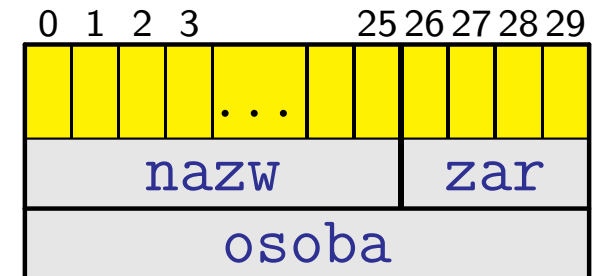
int main () {
    p = 500;

    kart[p].zar =
        kart[p+1].zar + 200;

    p = 720;

    kart[p].nazw[5] = 'a';

    return 0;
}
```



zmienna	adres w pamięci
x.nazw	(adres x)
x.nazw[i]	(adres x) + i
x.zar	(adres x) + 26
kart[0]	(adres kart)
kart[j]	(adres kart) + j·30
kart[j].nazw[i]	(adres kart) + j·30 + i

pobierz zaw. 4 bajtów od (adres p)
 pomnóż przez 30
 dodaj 5
 dodaj (adres kart)
 pod wyliczony adr. włóż kod ASCII znaku 'a'

Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.

Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.
- Adres początku tego obszaru ustala się w czasie **wykonania** programu.

Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.
- Adres początku tego obszaru ustala się w czasie **wykonania** programu. Jego wielkość i rozmieszczenie (przesunięcie) poszczególnych zmiennych w stosunku do jego początku ustala się w czasie **tłumaczenia** programu.

Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.
- Adres początku tego obszaru ustala się w czasie **wykonania** programu. Jego wielkość i rozmieszczenie (przesunięcie) poszczególnych zmiennych w stosunku do jego początku ustala się w czasie **tłumaczenia** programu.

Przykład:

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else {  
        sil(n-1,&w1); *w = n*w1;  
    }  
}
```

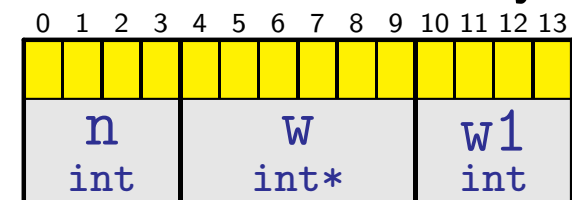
Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.
- Adres początku tego obszaru ustala się w czasie **wykonania** programu. Jego wielkość i rozmieszczenie (przesunięcie) poszczególnych zmiennych w stosunku do jego początku ustala się w czasie **tłumaczenia** programu.

Przykład:

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else {
        sil(n-1,&w1); *w = n*w1;
    }
}
```

W czasie **tłumaczenia** ustala się kształt obszaru danych:



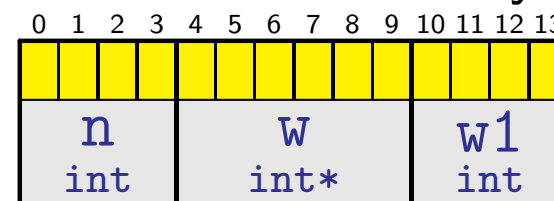
Organizacja pamięci — funkcje

- Każdemu wywołaniu funkcji odpowiada *obszar danych* zawierający miejsce na lokalne zmienne oraz parametry tej funkcji.
- Adres początku tego obszaru ustala się w czasie **wykonania** programu. Jego wielkość i rozmieszczenie (przesunięcie) poszczególnych zmiennych w stosunku do jego początku ustala się w czasie **tłumaczenia** programu.

Przykład:

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else {
        sil(n-1,&w1); *w = n*w1;
    }
}
```

W czasie **tłumaczenia** ustala się kształt obszaru danych:



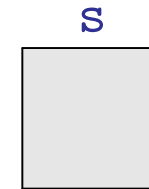
Potem w czasie **wykonania** przy każdym wywołaniu funkcji rezerwuje się jedną taką „paczkę” zmiennych.

Organizacja pamięci — funkcje

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); *w = n*w1; }  
}
```

Organizacja pamięci — funkcje

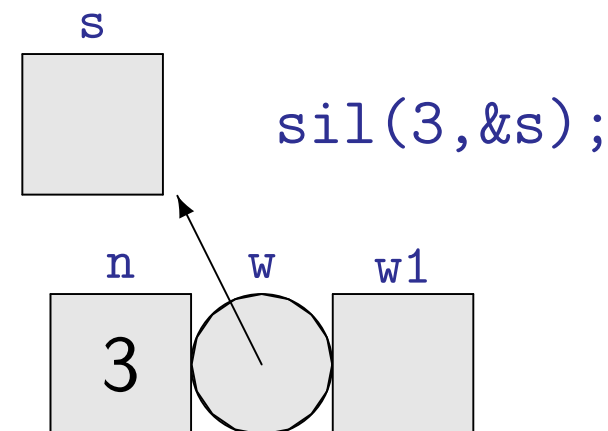
```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); *w = n*w1; }  
}
```



`sil(3,&s);`

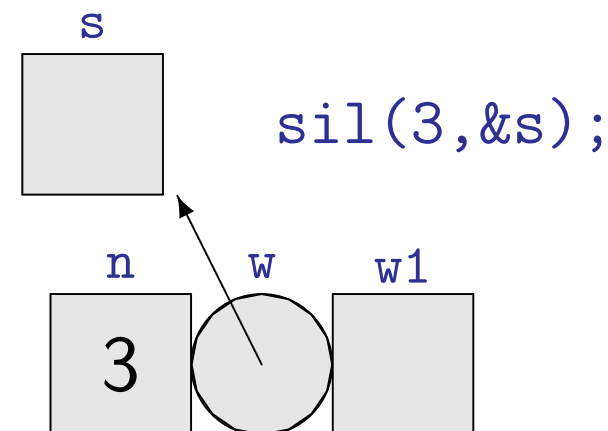
Organizacja pamięci — funkcje

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); *w = n*w1; }  
}
```



Organizacja pamięci — funkcje

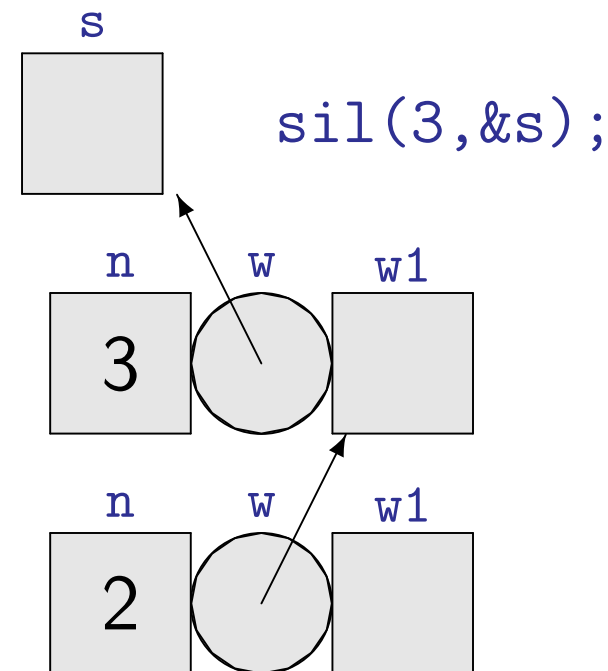
```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else {●sil(n-1,&w1); *w = n*w1; }  
}
```



Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

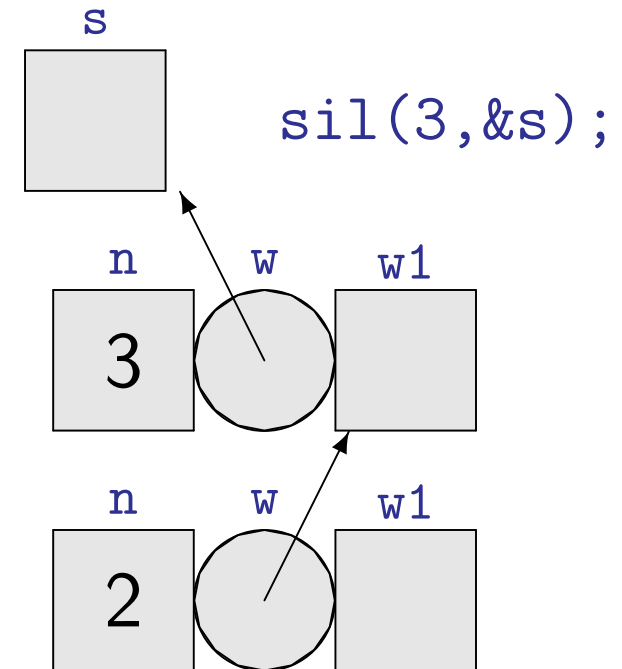
```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else {●sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else {●sil(n-1,&w1); *w = n*w1; }
}
```

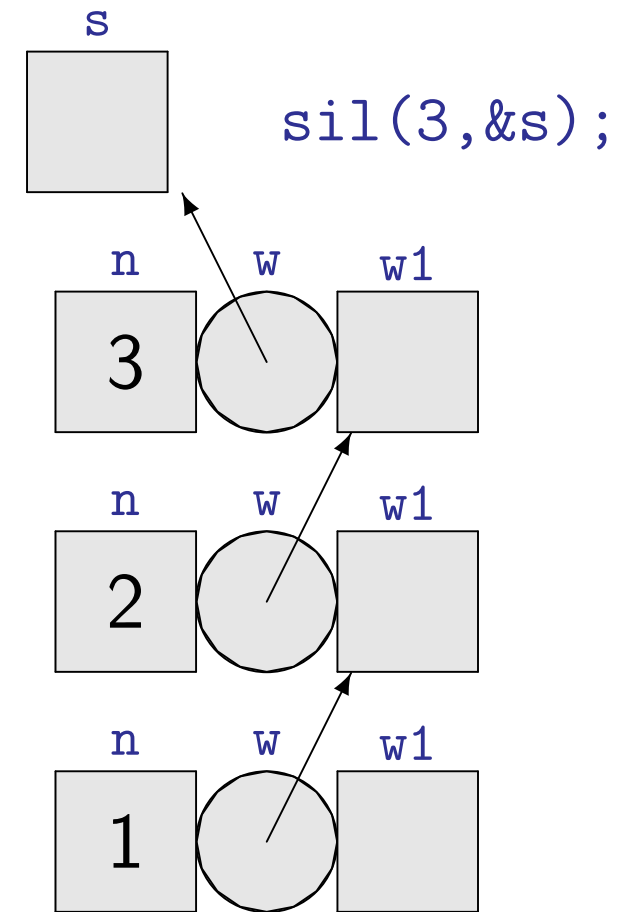


Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

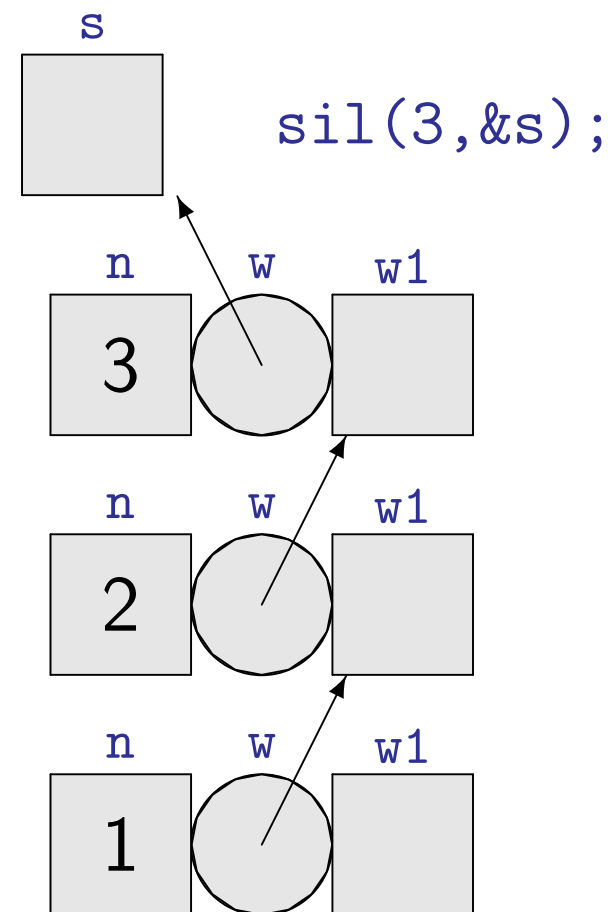


Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



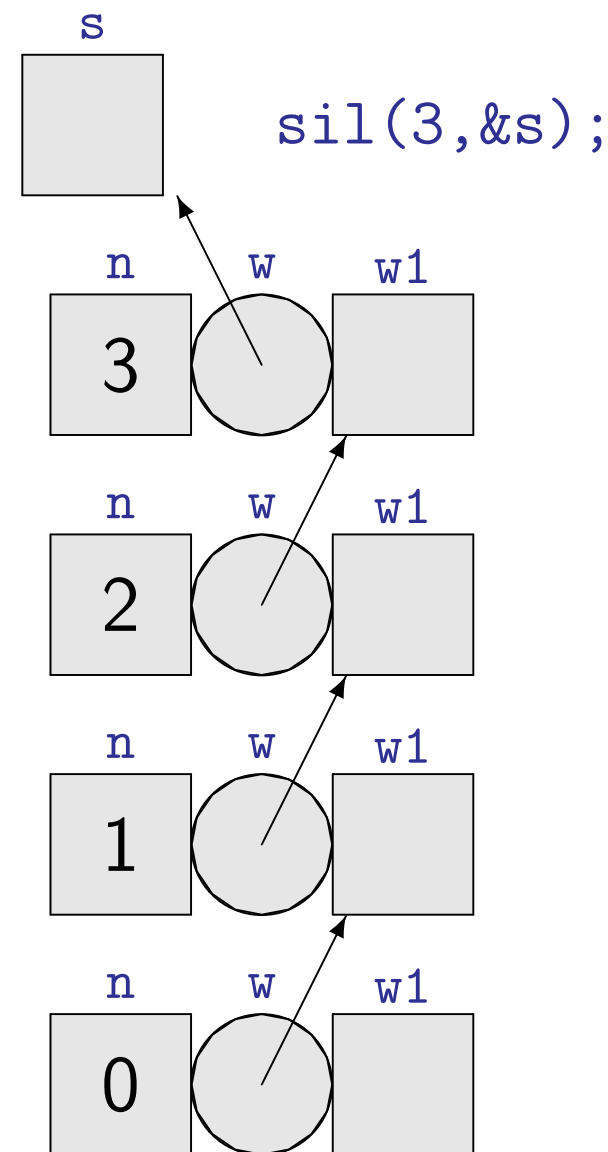
Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



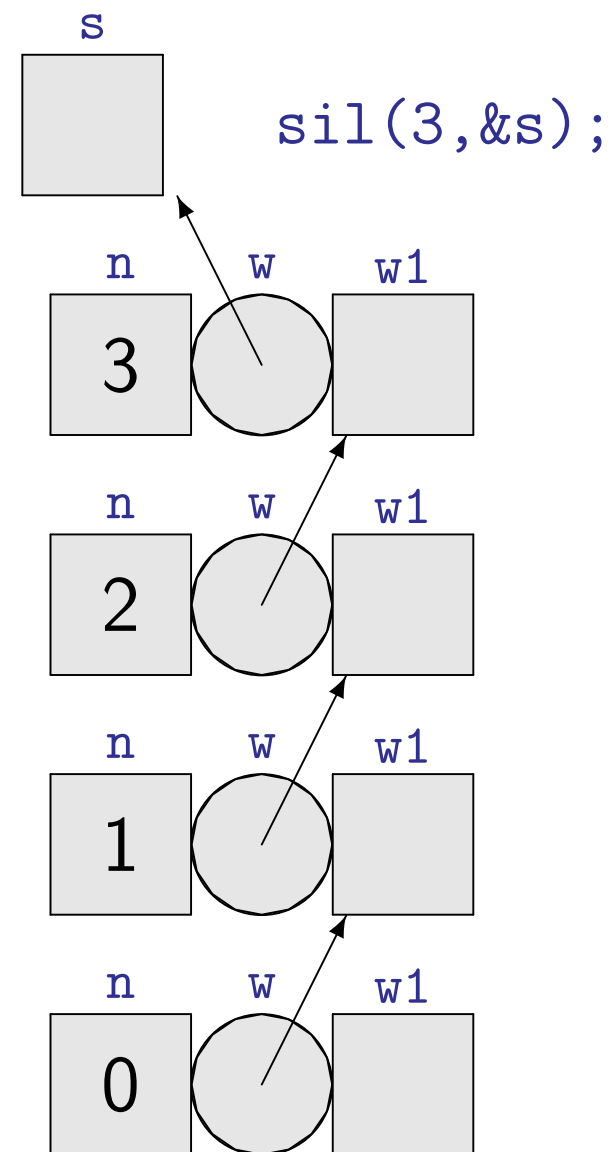
Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



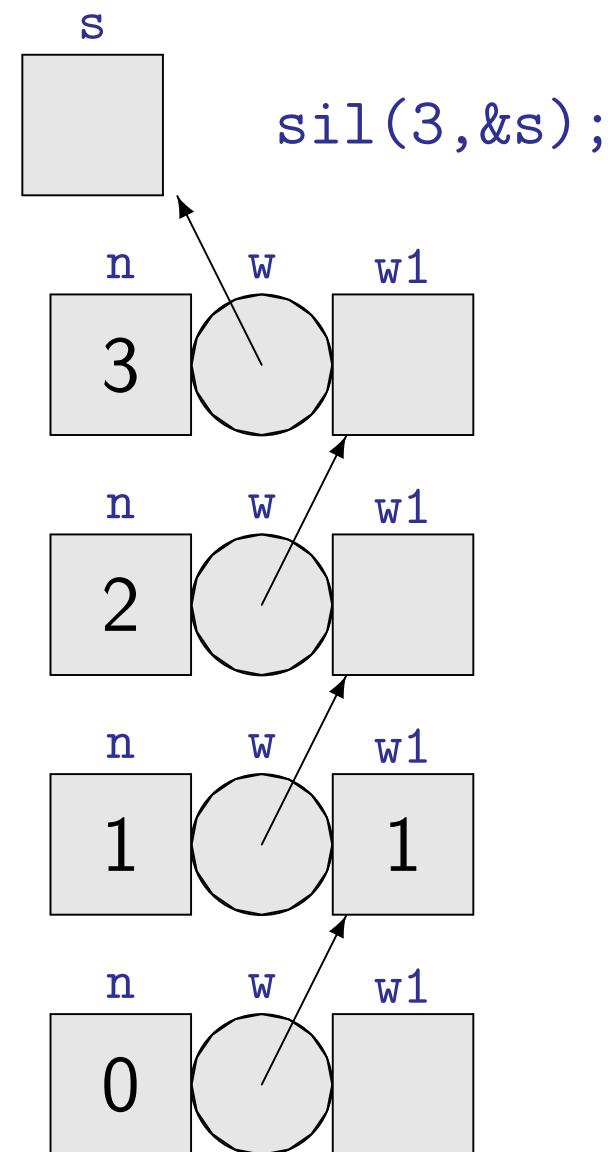
Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

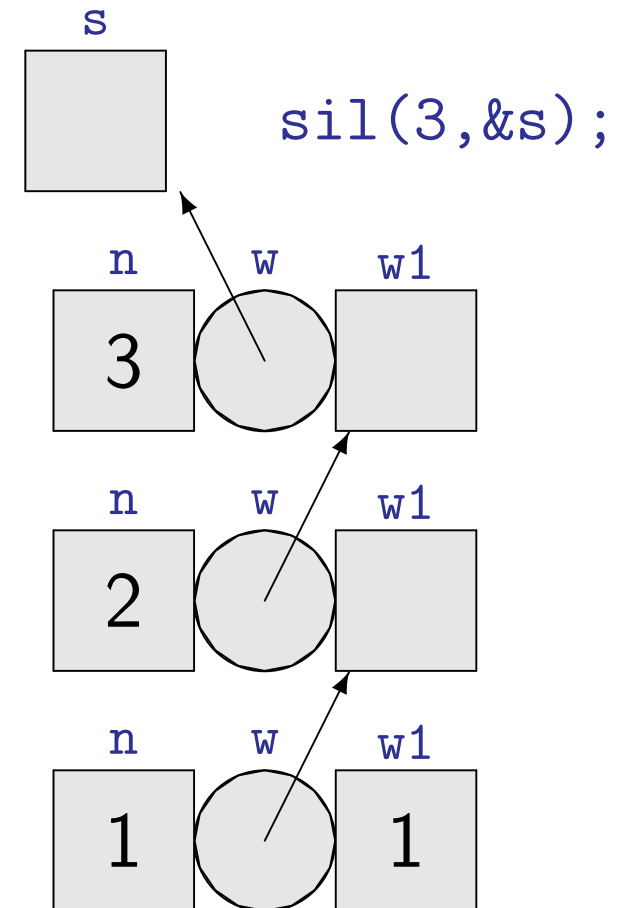


Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

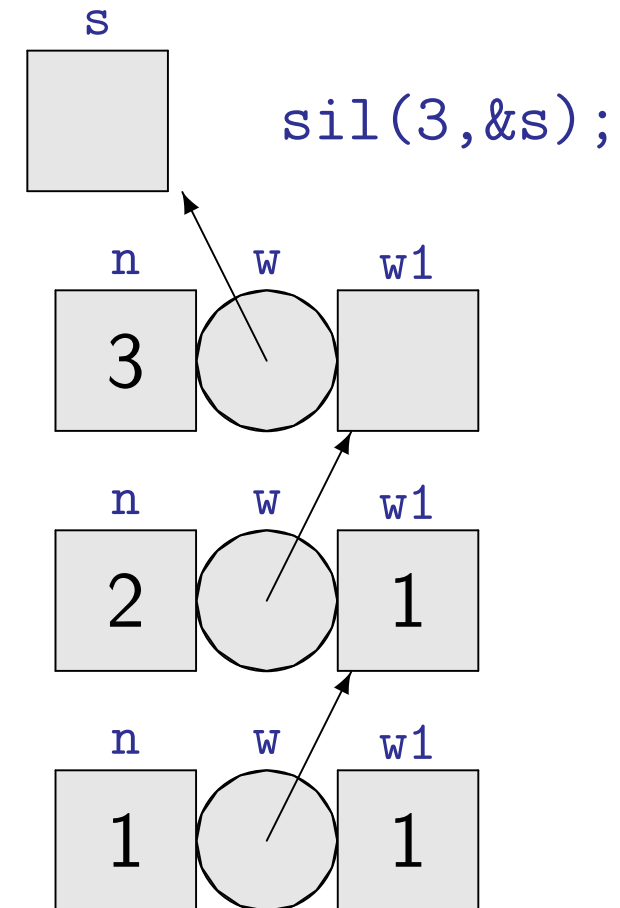


Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

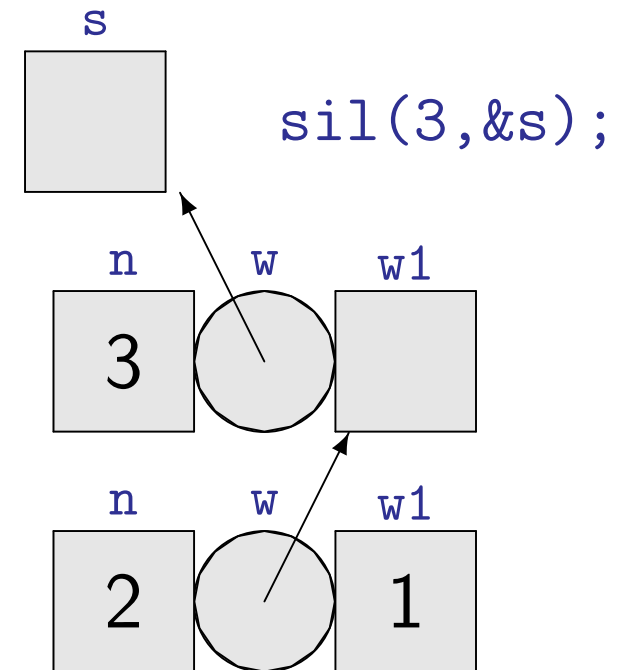
```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



Organizacja pamięci — funkcje

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else {●sil(n-1,&w1); *w = n*w1; }
}
```

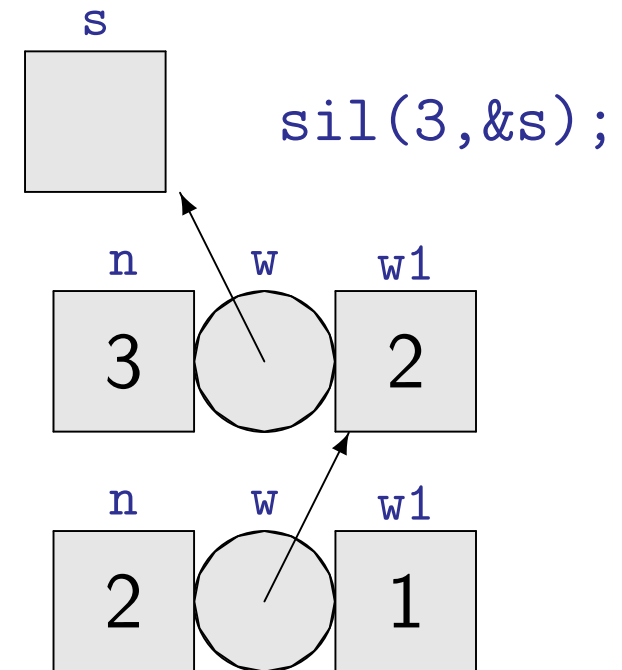
```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1);●*w = n*w1; }
}
```



Organizacja pamięci — funkcje

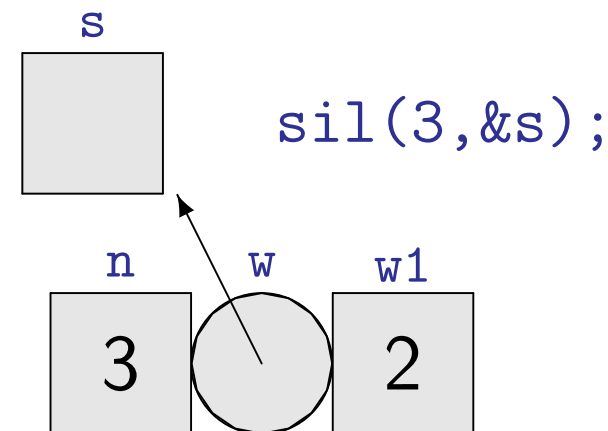
```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```

```
void sil(int n, int* w) {
    int w1;
    if (n==0) *w = 1;
    else { sil(n-1,&w1); *w = n*w1; }
}
```



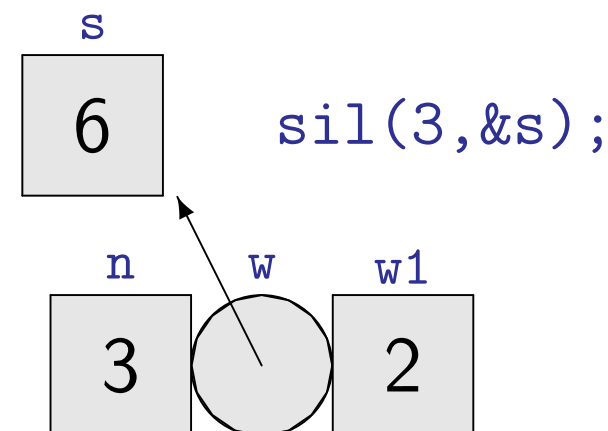
Organizacja pamięci — funkcje

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); ●*w = n*w1; }  
}
```



Organizacja pamięci — funkcje

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); *w = n*w1;●}  
}
```



Organizacja pamięci — funkcje

```
void sil(int n, int* w) {  
    int w1;  
    if (n==0) *w = 1;  
    else { sil(n-1,&w1); *w = n*w1; }  
}
```

s

6

sil(3,&s);