Stefan Sokołowski

JĘZYKI FORMALNE I METODY KOMPILACJI

wykład 11

Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025

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...

- 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

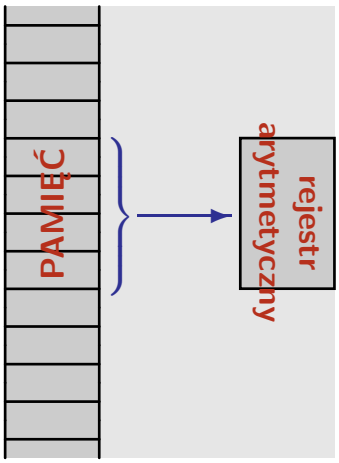
Zwykle (choć nie zawsze):

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	...	
...	...	

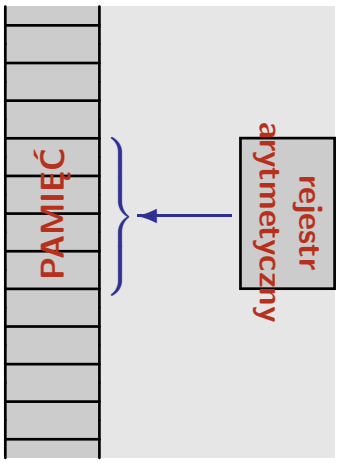
W realnych komputerach...

Rejestry arytmetyczne również są większe niż 1 bajt...

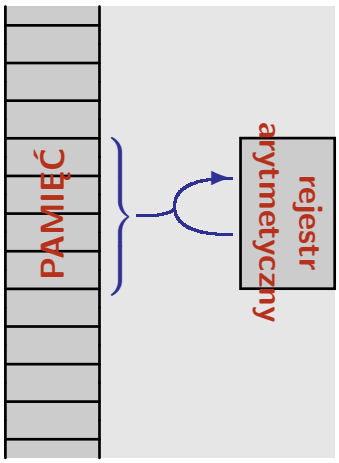
Przykładowe komendy:



skopiować blok bajtów z pamięci do rejestru



skopiować blok bajtów z rejestru do pamięci

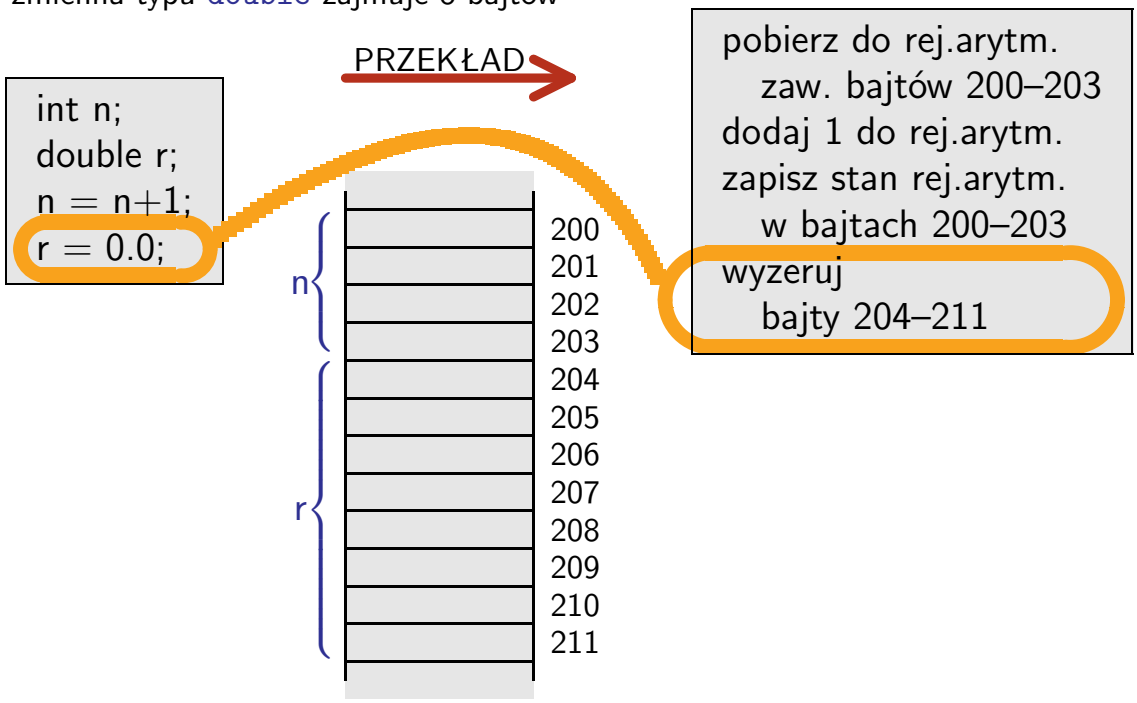


dodać blok bajtów z pamięci do rejestru

Organizacja pamięci

Założenie:

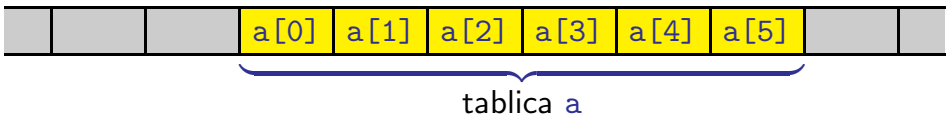
- zmienna typu `int` zajmuje 4 bajty
- zmienna typu `double` zajmuje 8 bajtów



Organizacja pamięci

W czasie kompilacji:

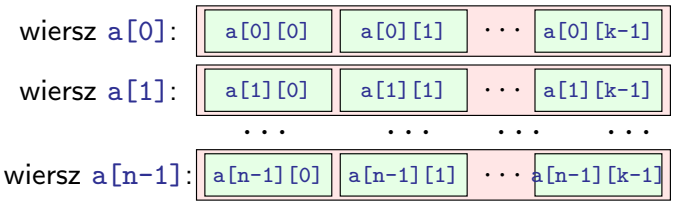
- na każdą *zadeklarowaną zmienną* rezerwuje się *blok pamięci* wielkości zależnej od *typu* tej zmiennej (np. na całkowitą zawsze 4 bajty, na rzeczywistą zawsze 8 bajtów, itp.);
- instrukcje
  jęz. źródłowego dotyczące tej zmiennej
  są tłumaczone na
  instrukcje jęz. wewnętrznego dotyczące tego bloku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



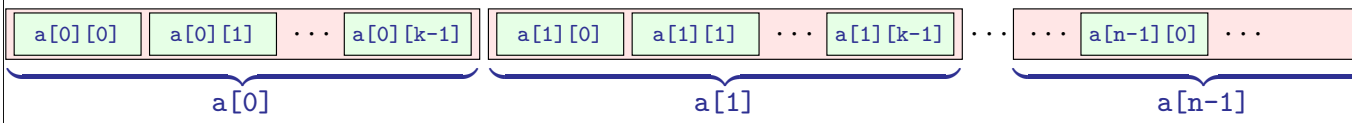
Organizacja pamięci

Tablice wielowymiarowe:

punkt widzenia programisty:



realizacja komputerowa:



Organizacja pamięci

Przykład:

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

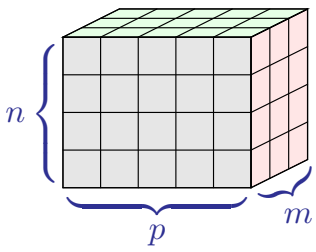
Kompilator rezerwuje dla niej blok długości 800 bajtów zaczynający się od adresu 1000. Jak zostanie przetłumaczona komenda tab[2][1] = 0; ?

	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:

Deklaracja: T `tab[n][m][p];`

czyli `n` warstw po
`m` wierszy w warstwie po
`p` elementów typu T w wierszu

Załóżmy, że pojedynczy element typu T zajmuje b bajtów; a początek tablicy jest w bajcie a.
Które adresy należą do elementu `tab[i][j][k]` **?**

Poprzedza go:

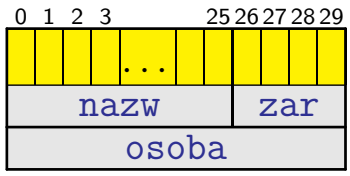
`i` warstw po `m·p` w warstwie : `i·m·p`
`j` wierszy własnej warstwy po `p` w wierszu : `j·p`
`k` elementów własnego wiersza : `k`

Razem : `i·m·p + j·p + k = (i·m+j)·p+k`
czyli $((\texttt{i·m+j})\texttt{·p+k})\cdot b$ bajtów
więc **zaczyna się** w bajcie $((\texttt{i·m+j})\texttt{·p+k})\cdot b+a$

deklaracja: `tab[n][m][p]`
element: `tab[i][j][k]`

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'

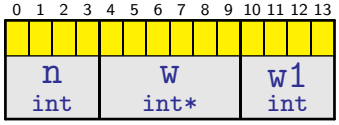
Organizacja pamięci — funkcje

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