14.10.11 Arytmetyka na adresach w C

W C wolno liczbę całkowitą dodać do lub odjąć od adresu, w tym również przy pomocy operatorów `++` i `--`. Adres może być bezpośrednio zapisany we wskaźniku, albo być wartością obliczonego wyrażenia.

Ostatnia z poniższych instrukcji

```
double *wsk = ...;
int n = 2;
wsk = wsk + n;
```

zwiększa wartość `wsk` (zapisany tam adres) **nie o wartość `n`**, czyli 2, ale o:

`n` razy rozmiar pamięci zmiennej mającej typ równy dziedzinie wskaźnika, tu **`double`**.

Operacja odejmowania działa w analogiczny sposób.

Nie wolno dokonywać operacji arytmetycznych na wskaźnikach, które nie mają określonej dziedziny.

Przykład W poniższym programie

```
main() {
   float tab[3], *wsk;  /*1*/
   wsk = tab;           /*2*/
   wsk = wsk + 1;       /*3*/
   wsk = &tab[1];       /*4*/
}
```

operacje przypisania z linii 3 i 4 są równoważne. W linii 3 adres zostaje zwiększony o tyle, ile zajmuje słów zmienna typu `float`. Skoro przed linią 3, `wsk` zawierał adres elementu `tab[0]`, to po linii 3 zawiera adres `tab[1]`. ■

Konwencję drugą oraz operację adresowania pośredniego można stosować nie tylko do zmiennych typów wskaźnikowych, ale do wszystkich wyrażeń, których wartość jest adresem o konkretnej dziedzinie.

Zachęcam do wykonania Zadania 30.

Przykład W następującym programie wszystkie przypisania zmiennej `x` można stosować zamiennie — prawe strony przypisań to wartość `tab[1]`, są one aliasami.

```
main()
{
 int tab[3] = { 1,2,3 };  // 1
 int x, *wsk;             // 2
 wsk = tab;               // 3

 x = tab[1];            x = wsk[1];            //4
 x = *(tab + 1);        x = *(wsk + 1);        //5
 x = *(&tab[1]);        x = *(&wsk[1]);        //6
 x = *(&tab[0] + 1);    x = *(&wsk[0] + 1);    //7
 x = *(&tab[2] - 1);    x = *(&wsk[2] - 1);    //8

 x = (tab + 1)[0];      x = (wsk + 1)[0];      //9
 x = (tab + 2)[-1];     x = (wsk + 1)[-1];     //10

 x = (&tab[0])[1];      x = (&wsk[0])[1];      //11
 x = (&tab[1])[0];      x = (&wsk[1])[0];      //12
 x = (&tab[2])[-1];     x = (&wsk[2])[-1];     //13

 x = (&tab[0] + 1)[0];  x = (&wsk[0] + 1)[0];  //14
 x = (&tab[0] + 2)[-1]; x = (&wsk[0] + 2)[-1]; //15

 x = (&tab[1] - 1)[1];  x = (&wsk[1] - 1)[1];  //16
 x = (&tab[1] + 1)[-1]; x = (&wsk[1] + 1)[-1]; //17

 x = (&tab[2] - 1)[0];  x = (&wsk[2] - 1)[0];  //18
 x = (&tab[2] - 2)[1];  x = (&wsk[2] - 2)[1];  //19
}
```

linia 4 Po wykonaniu linii 3, zgodnie z konwencją I, `wsk` i `tab` oznaczają ten sam adres. Jest to adres `tab[0]`. Zgodnie z konwencją II, `wsk` staje się nową nazwą tablicy.

```
    /* 4,... */
51 |  1  | tab[0] | wsk[0]
52 |  2  | tab[1] | wsk[1]
53 |  3  | tab[2] | wsk[2]
54 |  2  | x
55 |  51 | wsk
56 |     |
```

Lewa i prawa kolumna w liniach od 4 do końca zawiera niemal identyczne instrukcje. Dalej będę omawiać jedynie instrukcje stojące w lewej kolumnie.

linia 5 Wynik wyrażenia `tab + 1` jest adresem (dokładniej adresem elementu `tab[1]`). Można do niego zastosować operację adresowania pośredniego: idź pod ten adres i weź stamtąd wartość.

linia 6 Wynik wyrażenia `&tab[1]` jest adresem (dokładniej adresem elementu `tab[1]`). Można do niego zastosować operację adresowania pośredniego.

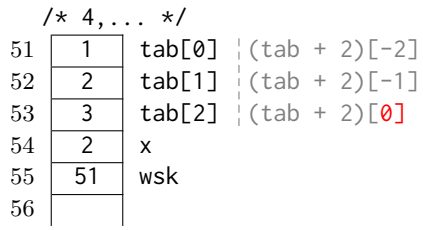
linia 7 Wynik wyrażenia `&tab[0]` jest adresem (dokładniej adresem elementu `tab[0]`). Do adresu można dodać liczbę całkowitą i wynikiem będzie adres. Wartością `&tab[0] + 1` jest adres elementu`tab[1]`. Do tego adresu można zastosować operację adresowania pośredniego.

linia 8 Wynik wyrażenia `&tab[2]` jest adresem (dokładniej adresem elementu `tab[2]`). Od adresu można odjąć liczbę całkowitą i wynikiem będzie adres. Wartością `&tab[2] - 1` jest adres elementu `tab[1]`. Do tego adresu można zastosować operację adresowania pośredniego.

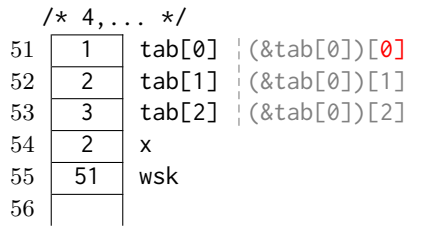
linia 9 Wynik wyrażenia `tab + 1` jest adresem (dokładniej adresem elementu `tab[1]`). Można do niego zastosować konwencję II, czyli `tab + 1` staje się nową nazwą tablicy `tab`. Zerowy element przy zastosowaniu nowej nazwy znajduje się pod adresem, który jest wartością wyrażenia `tab + 1`.

```
    /* 4,... */
51 |  1  | tab[0] | (tab + 1)[-1]
52 |  2  | tab[1] | (tab + 1)[0]
53 |  3  | tab[2] | (tab + 1)[1]
54 |  2  | x
55 |  51 | wsk
56 |     |
```

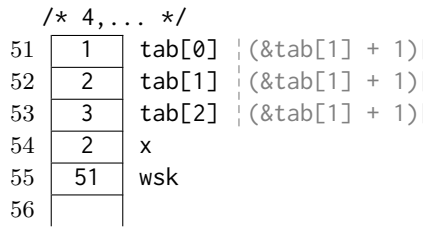
linia 10 Wynik wyrażenia `tab + 2` jest adresem (dokładniej adresem elementu `tab[2]`). Można do niego zastosować konwencję II, czyli `tab + 2` staje się nową nazwą tablicy `tab`. Zerowy element przy zastosowaniu nowej nazwy znajduje się pod adresem, który jest wartością wyrażenia `tab + 2`.



linia 11 Wynik wyrażenia `&tab[0]` jest adresem (dokładniej adresem elementu `tab[0]`). Można do niego zastosować konwencję II, czyli `&tab[0]` staje się nową nazwą tablicy `tab`. Zerowy element przy zastosowaniu nowej nazwy znajduje się pod adresem, który jest wartością wyrażenia `&tab[0]`.



linia 17 Wynik wyrażenia `&tab[1]` jest adresem (dokładniej adresem elementu `tab[1]`). Do adresu można dodać liczbę całkowitą i wynikiem będzie adres. Wartością `&tab[1] + 1` jest adres elementu elementu `tab[2]`. Można do niego zastosować konwencję II, czyli `&tab[1] + 1` staje się nową nazwą tablicy `tab`. Zerowy element przy zastosowaniu nowej nazwy znajduje się pod adresem, który jest wartością wyrażenia `&tab[1] + 1`.



■

14.10.12 Jednowymiarowe tablice dynamiczne ze sterty

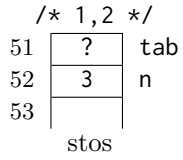
Wskaźniki są niezbędne do posługiwania się tablicami dynamicznymi ze sterty. Alokacja takiej tablicy następuje podczas wykonywania programu, wtedy też jest ustalany rozmiar tablicy. Służą do tego funkcje `malloc` lub `calloc`. Tak jak w przypadku każdej zmiennej dynamicznej, należy zwolnić pamięć przy pomocy funkcji `free`, gdy tablica jest już niepotrzebna.

Dzięki konwencji II, w momencie, gdy dokonamy alokacji, używanie tablic dynamicznych ze sterty niczym się nie różni od innych tablic. Formalnie te tablice nie posiadają nazwy, ale wskaźnik, w którym zapamiętaliśmy wartość zwróconą przez `malloc` lub `calloc` staje się nową nazwą tablicy.

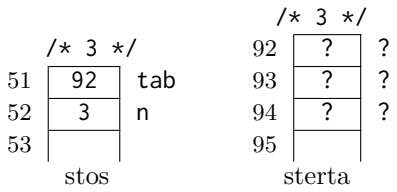
Przykład (Zadanie 31) W następującym fragmencie programu tworzona jest tablica dynamiczna ze sterty.

```
main() {                                          /*0*/
   int *tab;                                      /*1*/
   int  n = 3;                                    /*2*/
   // alokacja tablicy, zawsze trzeba
   // sprawdzic, czy alokacja sie udala
   tab = (int*)malloc(n * sizeof(int));           /*3*/
   if (tab == NULL) {                             /*4*/
      printf("Brak pamieci.");                    /*5*/
      return 0;                                   /*6*/
   }                                              /*7*/
   // uzywanie tablicy, korzystamy z konwencji II
   tab[0] = 7;                                    /*8*/
   tab[2] = 8;                                    /*9*/
   // dealokacja tablicy
   free(tab);                                    /*10*/
}                                                /*11*/
```

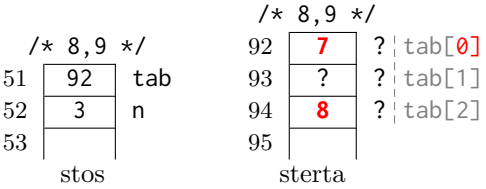
W momencie osiągnięcia przez program deklaracji zmiennej `tab`, rezerwowana jest na stosie pamięć na wskaźnik, a zaraz potem na zmienną `n`.



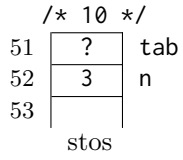
Po ustaleniu wartości zmiennej `n` (może to być bardziej skomplikowane niż w tym programie), następuje rezerwacja trzech komórek pamięci rozmiaru `sizeof(int)`. Utworzona tablica nie posiada nazwy. Zakładamy, że alokacja się udała.



Dzięki konwencji II, możemy posługiwać wskaźnikiem `tab` jako nową nazwą tablicy. W linii 8 i 9 nadawana jest wartość elementom tablicy. Zauważmy, że z praktycznego punkty widzenia, odbywa się to tak samo, jak w przypadku „zwykłych" tablic.



Funkcja `free` zwalnia pamięć zajmowaną przez tablicę dynamiczną i dane zostają bezpowrotnie utracone.



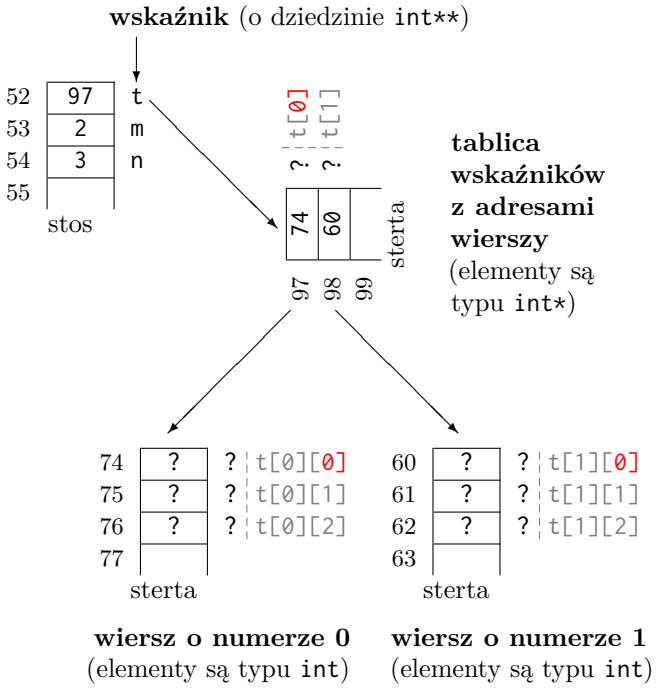
Obowiązkiem programisty jest zawsze sprawdzenie, czy przydział pamięci się udał i zaprogramowanie odpowiednich akcji w przypadku porażki (linie 4 – 7). ■

14.10.13 Dwuwymiarowe tablice dynamiczne ze sterty

Utworzenie dwuwymiarowej tablicy dynamicznej o m wierszach i n kolumnach jest bardziej skomplikowane. Nie wystarczy zaalokować na stercie $m \cdot n$ elementów tablicy, bo w ten sposób uzyskujemy tylko tablicę jednowymiarową o $m \cdot n$ elementach.

Dzięki konwencji II, da się problem rozwiązać. Ideę ilustruje poniższy przykład, który przedstawia alokację tablicy liczb całkowitych 2×3.

Przykład (Zadanie 32)



Alokację tablicy zaczyna się od utworzenia na stosie wskaźnika, który będzie przechowywał adres początku tablicy z adresami wierszy. Każdy element tablicy z adresami wierszy jest typu `int*`, stąd `int*` jest dziedziną wskaźnika ze stosu. Druga gwiazdka w deklaracji informuje, że deklarujemy wskaźnik (deklaracja `int *tab` tworzy wskaźnik o dziedzinie `int`, a nie `int*`). W tym zadaniu na stosie pojawiają się również zmienne przechowujące rozmiar tablicy.

```
int **t;
int m = 2, n = 3;
```

Alokacja tablicy z adresami wierszy odbywa się na stercie i jej adres zostaje zapisany w `t`. Jest to tablica jednowymiarowa. Każdy jej element ma przechowywać adres. Jej alokację wykonuje poniższy kod.

```
// alokacja dwuwymiarowej tablicy dynamicznej
// 1. alokacja tablicy z adresami wierszy
t = (int**)malloc(m * sizeof(int*));
if (t == NULL) {
   printf("Brak pamieci. Koniec programu.");
   return 0;
}
```

Następnie alokowane są wiersze: każdy osobno. Alokacji trzeba wykonać tyle, ile jest wierszy — należy użyć pętli. Każdy wiersz jest tablicą jednowymiarową liczb całkowitych.

```
// alokacja dwuwymiarowej tablicy dynamicznej
// 2. alokacja wierszy
for (int i = 0; i < m; i++) {
   t[i] = (int*)malloc(n * sizeof(int));
   if (t[i] == NULL) {
      printf("Brak pamieci. Koniec programu.");
      ...
      return 0;
   }
}
```

Zwróćmy uwagę, że gdy podczas alokacji któregoś wiersza zabraknie pamięci, to aby uniknąć powstawania śmieci, trzeba zwolnić już zaalokowane wiersze oraz tablicę z adresami wierszy. Ten kod pojawia się zamiast wielokropka.

Zwalnianie musi odbyć się we właściwej kolejności: najpierw wiersze, potem tablica z adresami wierszy. Odwrotna kolejność spowodowałby utratę adresów wierszy przed ich zwolnieniem.

```
// alokacja dwuwymiarowej tablicy dynamicznej
// 2. alokacja wierszy
for (int i = 0; i < m; i++) {
   t[i] = (int*)malloc(n * sizeof(int));
   if (t[i] == NULL) {
      printf("Brak pamieci. Koniec programu.");
      // dealokacja wczesniejszych wierszy
      for (int j = 0; j < i; j++) {
         free(t[j]);
      }
      // dealoakacja tablicy z adresami wierszy
      free(t);
      return 0;
   }
}
```

W tym momencie tablica jest gotowa. Można jej używać dokładnie tak, jak „zwykłej” tablicy.

```
// uzywanie tablicy
t[0][2] = 12;
```

Możliwe jest to dzięki konwencji II. Tablica z adresami wierszy nie posiada nazwy. Ponieważ jednak adres jej początku jest zapisany w `t`, to `t` staje się jej nową nazwą. Do elementów tej tablicy sięgamy pisząc `t[0]`, `t[1]`. Oba elementy są wskaźnikami.

Wiersz o numerze 0 jest zaalokowany na stercie i nie posiada nazwy. Jednak adres jego początku jest zapisany we wskaźniku `t[0]`. Dzięki konwencji II, `t[0]` staje się nową nazwą wiersza 0. Dlatego do elementów wiersza sięgamy pisząc `t[0][0]`, `t[0][1]` i `t[0][2]`. Ten sam mechanizm działa dla wszystkich wierszy.

Na koniec trzeba usunąć tablicę, by nie tworzyć śmieci.

```
// dealokacja tablicy
for (int i = 0; i < m; i++) {
   free(t[i]);
}
free(t);
```

■

Przykład (Kod z zadania 32 z podziałem na funkcje)

Ponieważ na laboratoriach zostały wprowadzone funkcje, to w tym miejscu zamieszczam kod, który pozwala na prostsze użycie dynamicznych tablic dwuwymiarowych. Zamyka on procesy alokacji i delaokacji w osobnych funkcjach. Osoba, która nawet nie do końca rozumie, co dzieje się w tych funkcjach, może je użyć.

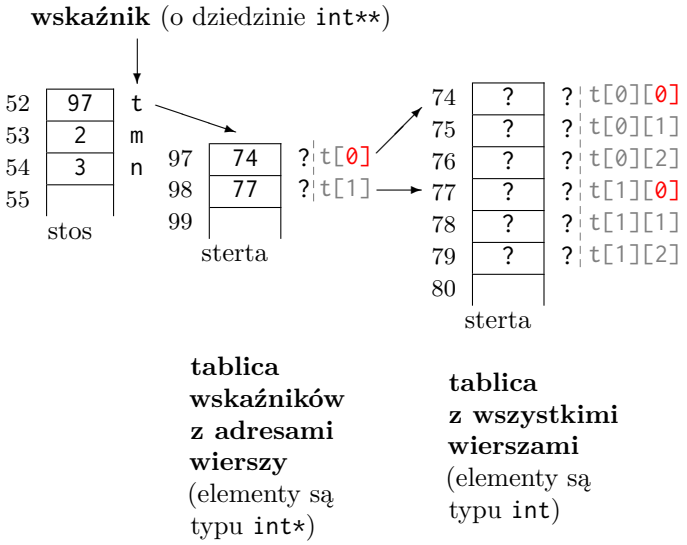
```
#include<stdio.h>
#include<stdlib.h>

void zwolnij_int_2D(int **tab, int m)
{
   for (int i = 0; i < m; i++) {
      free(tab[i]);
   }
   free(tab);
}

int** alokuj_int_2D(int m, int n)
{
   int **tab;
   tab = (int**)malloc(m * sizeof(int*));
   if (tab == NULL) {
      return NULL;
   }
   for (int i = 0; i < m; i++) {
      tab[i] = (int*)malloc(n * sizeof(int));
      if (tab[i] == NULL) {
         zwolnij_int_2D(tab, i);
         return NULL;
      }
   }
   return tab;
}

main()
{
   int **t;
   int m = 2, n = 3;
   // alokacja tablicy
   if ((t = alokuj_int_2D(m, n)) == NULL) {
      printf("Brak pamieci. Koniec programu.");
      return 0;
   }
   // uzywanie tablicy
   t[0][2] = 12;
   // dealokacja tablicy
   zwolnij_int_2D(t, m);
}
```

■

Opisana wyżej metoda, alokuje każdy wiersz osobno. Wiersze znajdują się w lokalizacjach pamięci odległych od siebie i nie mamy na to wpływu. Przypomnijmy, że w przypadku „zwykłych”, wiersze znajdują się w pamięci jeden za drugim bez przerw. Czy da się tak zrobić w przypadku tablic dynamicznych ze sterty?

Odpowiedź brzmi: tak. Należy wtedy do alokacji **wszystkich** wierszy użyć jednego wywołania funkcji `malloc` lub `calloc`. Nadal potrzebna jest tablica adresów wierszy, którą trzeba stosownie wypełnić. Opisuje to poniższy przykład.

Przykład (Zadanie 33)



Początek jest dokładnie taki sam, jak w zadaniu 32. Na stosie alokowany jest wskaźnik, który będzie przechowywał adres początku tablicy z adresami wierszy oraz zmienne przechowujące rozmiar tablicy.

Alokacja tablicy z adresami wierszy odbywa się na stercie i jej adres zostaje zapisany w `t`. Jest to tablica jednowymiarowa i każdy jej element ma przechowywać adres.

```
int **t;
int m = 2, n = 3;
// alokacja dwuwymiarowej tablicy dynamicznej
// 1. alokacja tablicy z adresami wierszy
t = (int**)malloc(m * sizeof(int*));
if (t == NULL) {
   printf("Brak pamieci. Koniec programu.");
   return 0;
}
```

Następnie na stercie alokowana jest tablica na wszystkie wiersze. Jej adres jest zapisany w `t[0]`. Jest to równocześnie adres zerowego wiersza tablicy dwuwymiarowej.

```
// alokacja dwuwymiarowej tablicy dynamicznej
// 2. alokacja tablicy ze wszystkimi wierszami
t[0] = (int*)malloc(m * n * sizeof(int));
if (t[0] == NULL) {
   printf("Brak pamieci. Koniec programu.");
   free(t);
   return 0;
}
```

Aby móc się odwoływać do elementów tablicy dwuwymiarowej przez użycie dwóch indeksów, należy uzupełnić tablicę z adresami wierszy. Zerowy jej element jest już uzupełniony, do pozostałych trzeba wpisać adresy, od których zaczynają się kolejne wiersze. Adres wiersza obliczamy dodając do adresu początku tablicy odpowiednią wartość: łączną ilość elementów we wszystkich wierszach poprzedzających dany wiersz.

```
   // alokacja dwuwymiarowej tablicy dynamicznej
   // 3. wypelnienie tablicy z adresami wierszy
   for (int i = 1; i < m; i++) {
      t[i] = t[0] + i * n;
   }
```

Tablica jest gotowa, można jej używać.

```
   // uzywanie tablicy
   t[0][2] = 12;
```

Tak jak poprzednio, wykorzystywana jest konwencja II. Na koniec tablicę trzeba zwolnić. Zwalnianie musi się odbywać w odwrotnej kolejności niż alokacja: najpierw tablica ze wszystkimi wierszami, potem tablica z adresami wierszy.

```
   // dealokacja tablicy
   free(t[0]);
   free(t);
}
```

■

Przykład (Kod z zadania 33 z podziałem na funkcje)

Proszę zwrócić uwagę, że kod funkcji `main` nie zmienił się względem kodu z Zadania 32 napisanego z podziałem na funkcje. Wymienione zostały treści funkcji alokujących i dealokujących. Gdyby duży program intensywnie wykorzystywał dwuwymiarowe tablice dynamiczne i chcielibyśmy zmienić sposób ich reprezentacji, to dzięki funkcjom zmiana dotyczyłaby tylko treści funkcji.

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


void zwolnij_int_2D(int **tab, int m)
{
   free(tab[0]);
   free(tab);
}

int** alokuj_int_2D(int m, int n)
{
   int **tab;
   tab = (int**)malloc(m * sizeof(int*));
   if (tab == NULL) {
      return NULL;
   }
   tab[0] = (int*)malloc(m*n * sizeof(int));
   if (tab[0] == NULL) {
      free(tab);
      return NULL;
   }
   for (int i = 1; i < m; i++) {
      tab[i] = tab[0] + i * n;
   }
   return tab;
}
```

```
main()
{
   int **t;
   int m = 2, n = 3;
   // alokacja tablicy
   if ((t = alokuj_int_2D(m, n)) == NULL) {
      printf("Brak pamieci. Koniec programu.");
      return 0;
   }
   // uzywanie tablicy
   t[0][2] = 12;
   // dealokacja tablicy
   zwolnij_int_2D(t, m);
}
```

■

Metoda pierwsza alokacji dynamicznych tablic dwuwymiarowych jest częściej używana. Spowodowane jest to tym, że łatwiej jest uzyskać m fragmentów o długości n, niż jeden fragment o długości $m \cdot n$.