UNIWERSYTET KOMISJI EDUKACJI NARODOWEJ
w Krakowie
Instytut Bezpieczeństwa i Informatyki

---

dr ROMAN CZAPLA

PROGRAMOWANIE PROCEDURALNE

wykład nr I

Kraków 23.02.2024

ostatnia aktualizacja: 26 lutego 2024

Wskaźnik do null

- Przypisanie wartości `NULL` zmiennej wskaźnikowej skutkuje tym, że zmienna ta nie będzie niczego wskazywała. Dokładniej, wskaźnik o wartości `NULL`, gwarantuje, że taki wskaźnik przechowuje określoną wartość, która nie jest równa żadnemu innemu wskaźnikowi niezerowemu. Wskaźnik pusty (o wartości `NULL`) nie wskazuje adresu pamięci, a dwa puste wskaźniki są zawsze równe. Pustym wskaźnikiem można uczynić wskaźnik dowolnego typu. Możemy przypisać wskaźnikowi wartość zerową, ale nie możemy mu przypisać żadnej innej wartości typu `integer`.

  Makro `NULL` jest zerową stałą typu `integer` rzutowaną na wskaźnik na `void`, przeważnie zdefiniowane jako:

  ```
  #define NULL ((void *) 0)
  ```

  Nigdy nie należy stosować operatora wyłuskania (dereferecji) do niezainicjalizowanej zmiennej wskaźnikowej - spowoduje to niezidentyfikowane działanie programu. Ponadto, nie należy dokonywać dereferencji pustych wskaźników, ponieważ nie zwierają one prawidłowego adresu - takie działanie będzie skutkować niezidentyfikowanym działanie programu.

Wskaźnik na `void`

- Wskaźnik na **`void`** jest wskaźnikiem ogólnego stosowania i jest on przeznaczony do przechowywania odniesień do danych dowolnego typu. Przykładowy wskaźnik na **`void`**:

```
void *p;
```

- Wskaźnik na **`void`** ma taką samą reprezentację i organizację pamięci jak wskaźnik na **`char`**.
- Wskaźnik na **`void`** jest pozbawiony informacji o typie tj. reprezentuje adres obiektu lecz nie jego typ, nie może służyć do odczytania miejsca, na które pokazuje i nie można do niego stosować praw arytmetyki wskaźnikowej.
- Każdy wskaźnik może zostać przypisany do wskaźnika na **`void`**. Następnie taki wskaźnik można z powrotem rzutować na jego początkowy typ. Po takiej operacji wartość wskaźnika będzie równa wartości wskaźnika przed zmianami. Wskaźniki na **`void`** stosowane są w kontekście wskaźników na dane, a nie wskaźników na funkcje.
- Należy mieć świadomość, że gdy przeprowadzimy operację rzutowania dowolnego wskaźnika na **`void`**, nic nie będzie zabezpieczać przed ewentualnym rzutowaniem go na inny typ wskaźnika.

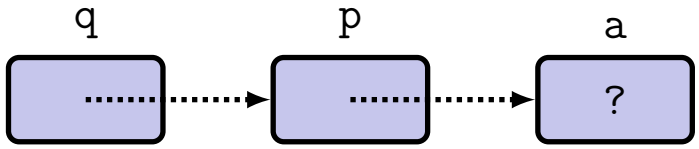
Wskaźniki podwójne

- Ponieważ zmienna wskaźnikowa również posiada adres, więc musi istnieć sposób na deklarację zmienne wskaźnikowej `p` której wartością będzie adres innej zmiennej wskaźnikowej. Ogólna deklaracja wygląda następująco:

```
type **p;
```

W takim wypadku: `p` jest wskaźnikiem do wskaźnika do obiektu typu `type`, `*p` jest wskaźnikiem typu `type`, a `**p` jest obiektem typu `type` (l - wartość).

```
int a, *p, **q;
p = &a;
q = &p;
```



PRZYKŁAD 1

Tablice wielowymiarowe a wskaźniki

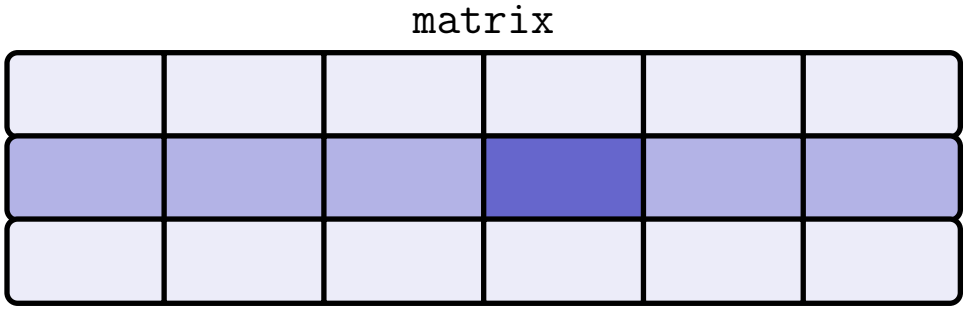
- Należy mieć świadomość, że w języku C tablice są tylko jednowymiarowe. Tablica dwuwymiarowa to tak naprawdę tablica tablic jednowymiarowych. Na przykład deklaracja:

  ```
  int matrix[3][6];
  ```

  powoduje utworzenie jednowymiarowej tablicy o nazwie **matrix**, która zawiera trzy elementy. Każdy z tych elementów jest tablicą sześciu liczb typu **int**. Wartość **matrix** jest stałym wskaźnikiem na pierwszy element tablicy - wskazuje on tym samym na tablicę sześciu liczb typu **int**!

Tablice wielowymiarowe a wskaźniki

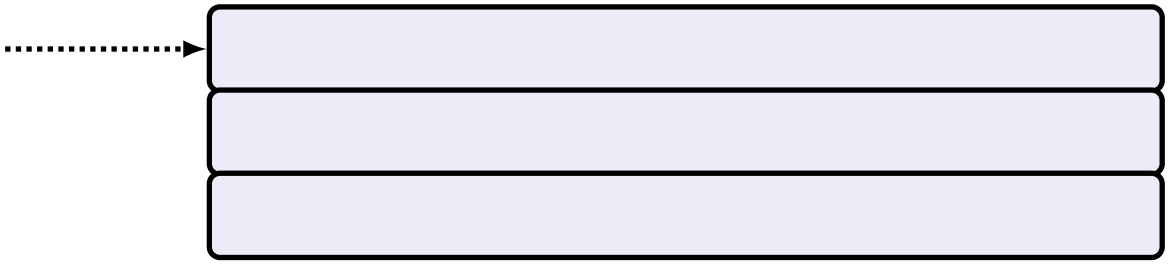
- Do elementów tablicy dwuwymiarowej możemy odwoływać się poprzez indeksy, np. wyrażenie `matrix[1][3]` odwołuje się do zaznaczonego elementu:



Tak naprawdę indeksy są w rzeczywistości ukrytymi wyrażeniami dereferencji, nawet w przypadku tablic wielowymiarowych.

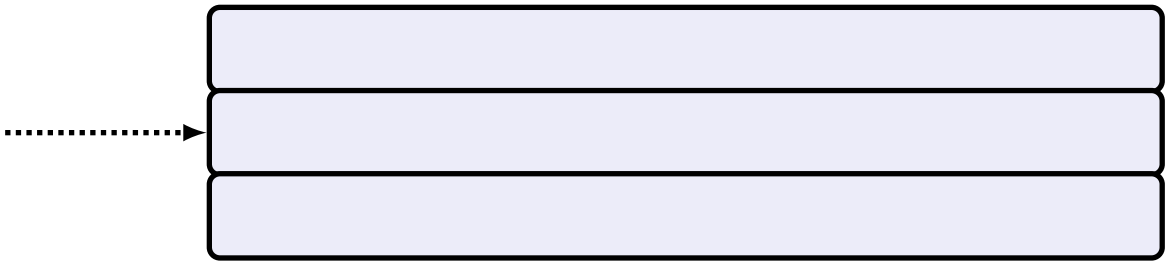
Tablice wielowymiarowe a wskaźniki

- Rozważmy wyrażenie `matrix`. Jego typ to wskaźnik na tablicę sześciu liczb całkowitych, a jego wartością jest:



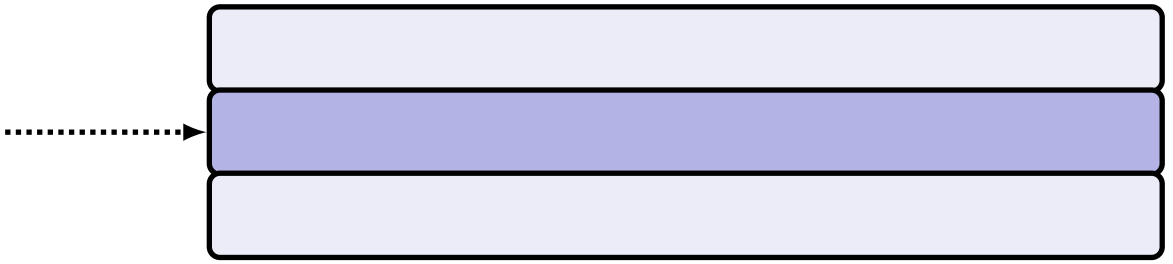
Wskazuje on na pierwszy wiersz, czyli pierwszą tablicę.

Wyrażenie `matrix + 1` też jest wskaźnikiem na tablicę sześciu liczb całkowitych, ale wskazuje on na inny wiersz tablicy (wartość `1` jest skalowana wielkością tablicy sześciu liczby typu `int`):

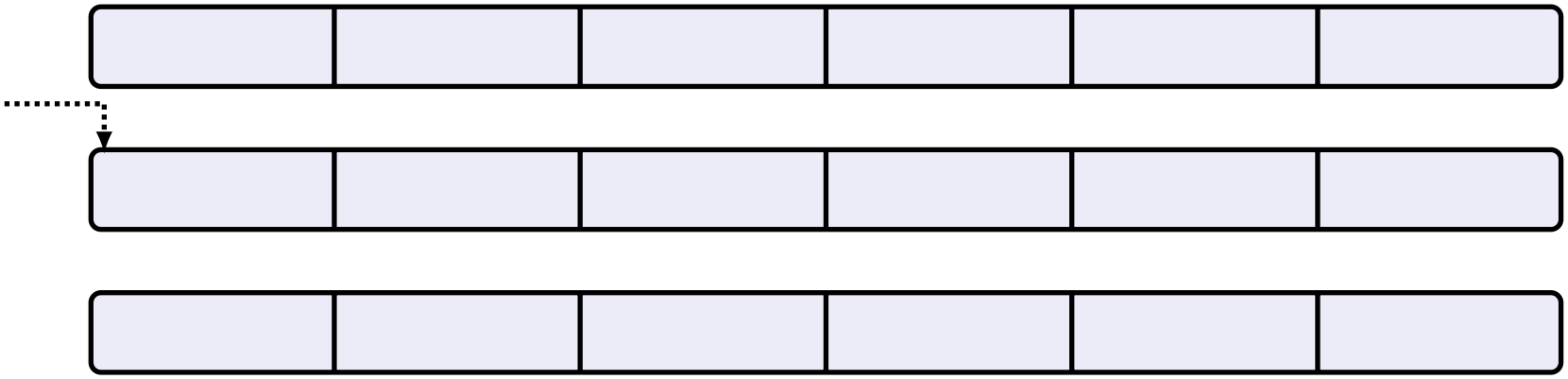


Tablice wielowymiarowe a wskaźniki

- Stosując dereferencję możemy wybrać wiersz który wskazuje wskaźnik `matrix + 1`:

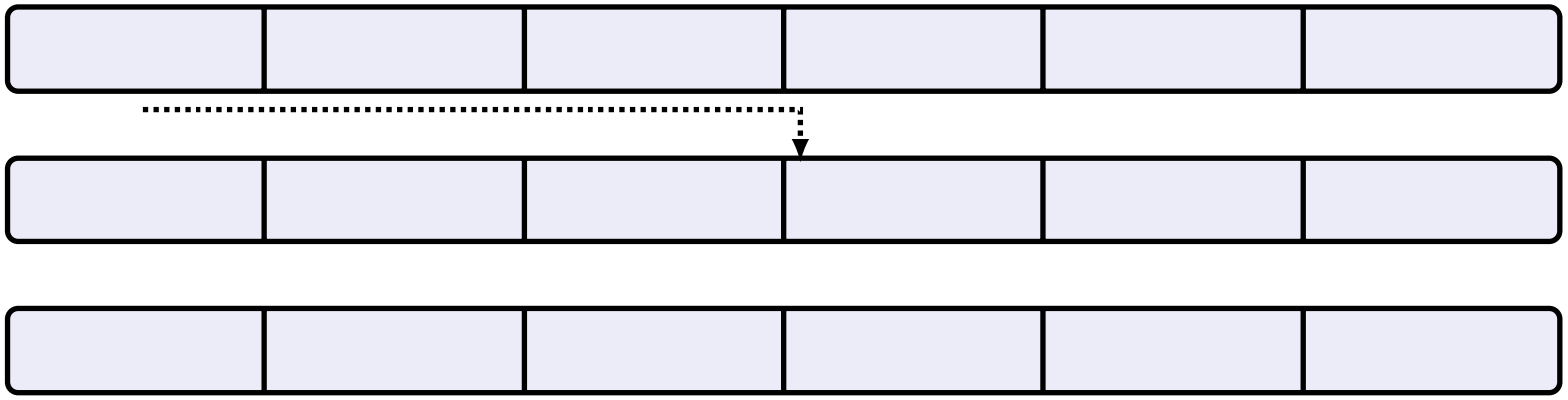


Wyrażenie `*(matrix + 1)` identyfikuje określoną tablicę sześciu liczb. Jest to nazwa tablicy będącej drugim wierszem (inaczej `matrix[1]`). Wiemy, że wartością nazwy tablicy jest stała wskaźnikowa na pierwszy element tablicy - tak jest w naszym wyrażeniu. Typem tego wskaźnika jest wskaźnik na `int`, a sam wskaźnik wskazuje na pierwszy element wiersza:



Tablice wielowymiarowe a wskaźniki

- Rozważmy dalej `*(matrix + 1) + 3`. Poprzedni wskaźnik wskazywał na `int`, więc `3` jest skalowane wielkością typu `int`, a wynikowy wskaźnik wskazuje na trzeci element naszego wiersza.



Zastosowanie dereferencji `*(*(matrix + 1) + 3)` zwraca wartość elementu na który wskazuje wskaźnik `*(matrix + 1) + 3`.

W drugą stronę mając wyrażenie `*(*(matrix + 1) + 3)` widzimy tu indeksowanie: wrażenie `*(matrix + 1)` jest tym samy co `matrix[1]`, a wyrażanie `*(matrix[1] + 3)` tym samy co `matrix[1][3]`.

Tablice wielowymiarowe a wskaźniki

- Podsumowując, do elementów tablicy dwuwymiarowej `t` możemy odwoływać się poprzez indeksy oraz notację wskaźnikową:

  $$\texttt{t[i][j]} \Leftrightarrow \texttt{*(*(t + i) + j)}$$

  Przykładowo:

  $\texttt{t[0][0]} \Leftrightarrow \texttt{*(*(t + 0) + 0)} \Leftrightarrow \texttt{**t}$

  $\texttt{t[1][0]} \Leftrightarrow \texttt{*(*(t + 1) + 0)} \Leftrightarrow \texttt{**(t + 1)}$

  $\texttt{t[0][1]} \Leftrightarrow \texttt{*(*(t + 0) + 1)} \Leftrightarrow \texttt{*(*t + 1)}$

Deklaracja wskaźnika do tablicy

- Ogólna deklaracja wskaźnika `p` do N-elementowej tablicy o elementach typu `type` wygląda tak:

```
type (*p)[N];
```

W powyższej deklaracji nawiasy są istotne, gdyż deklaracja:

```
type *p[N];
```

spowodowałaby utworzenie N-elementowej tablicy wskaźników do `type`.

PRZYKŁAD 2

PRZYKŁAD 3

Dynamiczne zarządzanie pamięcią w języku C

- Tak naprawdę w języku C wskaźniku ujawniają swoją największą użyteczność w dużej mierze dzięki możliwości śledzenia dynamicznie alokowanej pamięci. Program napisany w języku C jest wykonywany w systemie wykonawczym. Zwykle środowisko to jest zapewniane przez system operacyjny. System wykonawczy zapewnia programowi między innymi obsługę stosu i sterty.
- *Sterta* jest obszarem pamięci, udostępnionym na wyłączność uruchomionemu programowi (procesowi). Na stercie przechowywane są dynamicznie (podczas działania programu) tworzone struktury danych oraz dynamicznie przydzielane obszary pamięci.
- Alokowanie i dealokowanie pamięci przez program pozwala na bardziej elastyczne i wydajne zarządzanie pamięcią. Zamiast alokować pamięć dla największej możliwej struktury danych, możliwe staje się alokowanie tylko wymaganej ilości pamięci.

Dynamiczna alokacja pamięci - funkcje `malloc()` i `free()`

- Dynamiczna alokacja pamięci w języku C przebiega w trzech podstawowych krokach:
  - alokowanie pamięci za pomocą funkcji typu `malloc()`,
  - wykorzystanie alokowanej pamięci przez program,
  - dealokowanie pamięci za pomocą funkcji `free()`.
- Funkcja `malloc()`:

```
void* malloc (size_t size)
```

  Argument funkcji `malloc()` określa, ile bajtów pamięci funkcja ta ma alokować na stercie. Jeżeli alokacja przebiegnie pomyślnie, funkcja ta zwraca wskaźnik do pamięci alokowanej na stercie. Jeżeli alokacja nie zakończy się sukcesem, funkcja zwróci pusty wskaźnik tj. `NULL`.

- Funkcja `free()`:

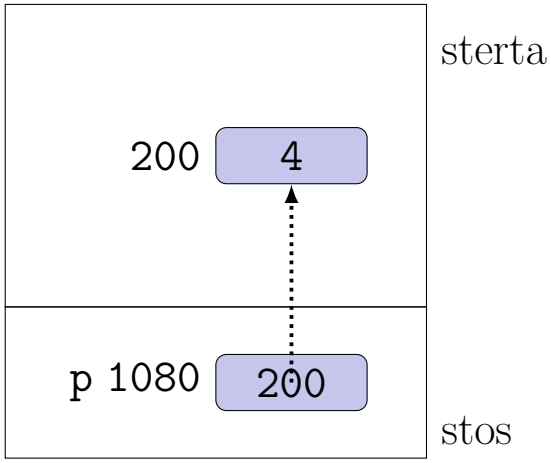
```
void free(void *p)
```

  Funkcja `free()` działa w połączeniu z funkcją `malloc()`. Służy do zwalniania (dealokacji) pamięć przydzieloną wcześniej przez funkcję `malloc()` i wskazywanej przez wskaźnik `p`, jeśli `p` jest różne od `NULL`.

Dynamiczna alokacja pamięci

- Przykładowe alokacja i dealokacja pamięci dla obiektu typu `int`:

```
int *p = (int*) malloc(sizeof(int));
*p = 4;
printf("*p: %d\n", *p);
free(p);
```



- Należy pamiętać, że każdorazowo, gdy wywoływana jest funkcja `malloc()` (lub funkcja podobna do niej), należy wywołać funkcję `free()`. Funkcję `free()` należy wywołać po tym, jak program skończy operacje na danym fragmencie alokowanej pamięci. Zapobiegnie to powstawaniu tzw. *wycieków pamięci*.

Dynamiczna alokacja pamięci

- Wywołanie funkcji `malloc()` jest żądaniem uzyskania od sytemu operacyjnego nowego ciągłego bloku pamięci - możemy zatem tworzyć tablice dynamiczne wykorzystując arytmetykę wskaźnikową (również korzystając z notacji tablicowej).
- Podczas alokacji pamięci, zapisywane są także dodatkowe informacje które są przechowywane jako część struktury danych. Są one obsługiwane przez menedżera sterty. Wśród tych danych znajdują się między innymi informacje o rozmiarze bloku. Dane te są zwykle umieszczane w miejscu przylegającym do alokowanego bloku. Jeżeli program dokona zapisu poza tym blokiem pamięci, struktura danych może ulec uszkodzeniu, a sam program awarii.
- Prototypy funkcji do dynamicznej alokacji pamięci znajdują się m.in. w pliku nagłówkowym `stdlib.h`.

Dynamiczna alokacja pamięci dla tablic jednowymiarowych

- Przykład:

  dynamiczna alokacja pamięci dla 20-elementowej tablicy liczb typu `int`.

alokacja pamięci	dealokacja pamięci
`int *p;`	
`int n = 20;`	`free(p);`
`p = (int *) malloc(n * sizeof(int));`	

PRZYKŁAD 4

Wycieki pamięci

- Do wycieku pamięci dochodzi, gdy alokowana pamięć nie jest już wykorzystywana, a nie jest zwolniona. Może do tego dojść, gdy:
  - utracono adres pamięci;
  - funkcja `free()` nie została wywołana pomimo tego, że powinna być wywołana (mówimy wtedy o tzw. *ukrytym wycieku*).
- Niebezpieczeństwo zajęcia całej dostępnej pamięci:

```
char *pc;
while (1) {
  pc = (char*) malloc(2000000);
  printf("Alokacja\n");}
```

- Utrata adresu:

```
int *p = (int*) malloc(sizeof(int));
*p = 5;
p = (int*) malloc(sizeof(int));
```

Inne funkcje dynamicznej alokacji pamięci

- Oprócz funckcji `malloc()` i `free()` mamy jeszcze do dyspozycji funkcje `realloc()` oraz `calloc()`.
- Funkcja `realloc()`:

```
void* realloc (void *p, size_t size)
```

Służy do zmiany rozmiaru zaalokowanego bloku pamięci. Zawartość pamięci jest zachowywana (poza przypadkiem zmniejszania bloku pamięci - wtedy część informacji jest tracona). W przypadku powiększania bloku pamięci nowy, dodany, obszar ma przypadkową zawartość. Jeżeli `p = NULL` operacja jest analogiem funkcji `malloc()`. Jeżeli `size = 0`, operacja jest analogiem wywołania funkcji `free()`. Jeżeli zmiana rozmiaru nie była możliwa funkcja zawraca wartość `NULL`, jednak oryginalny wskaźnik wciąż jest prawidłowy, a oryginalny blok pamięci nie ulega zmianie.

Uwaga: Ponieważ w efekcie powiększania bloku może zmienić się jego adres (wartości oryginalnego bloku będą skopiowane) zawsze należy korzystać z nowego wskaźnika zwracanego przez `realloc()`.

PRZYKŁAD 5

Inne funkcje dynamicznej alokacji pamięci

- Funkcja `calloc()`:

```
void* calloc (size_t n, size_t size)
```

Wywołanie funkcji `calloc()` jest żądaniem od sytemu operacyjnego nowego bloku pamięci o liczbie elementów równym `n` i wielkości pojedynczego elementu równego `size` wyrażonej w bajtach. Funkcja gwarantuje, że zwrócony blok pamięci jest zapełniony bajtami o wartości zero. Ponadto zachowuje się podobnie jak funkcja `malloc()`.

PRZYKŁAD 6

Zwalnianie pamięci przy użyciu funkcji `free()`

- Programista, mając pamięć alokowaną dynamicznie, jest w stanie ją zwolnić, gdy nie jest już używana przez dany obiekt. Zwykle jest to przeprowadzane za pomocą funkcji **`free()`.**
- Argument funkcji **`free()`** powinien zawierać adres pamięci alokowanej przy użyciu funkcji typu **`malloc()`**. Pamięć ta jest zwracana z powrotem stercie, a wskaźnik może wciąż wskazywać na ten rejon pamięci. Dlatego należy założyć, że w takiej sytuacji wskaźnik wskazuje na bezużyteczne dane. Zwrócony stercie obszar pamięci może być później realokowany i zapełniony innymi danymi.
- Funkcja **`free()`** zwykle nie podejmuje żadnych działań, gdy przekażemy jej pusty wskaźnik (**`NULL`**). Jeżeli funkcji **`free()`** przekażemy wskaźnik do pamięci alokowanej funkcją innego typu niż **`malloc`**, wtedy zachowanie funkcji **`free()`** jest nieokreślone.

```
int number;
int *p = &number;
free(p); // zachowanie nieokreślone
```

Zwalnianie pamięci przy użyciu funkcji `free()`

- Próba wyłuskania już uwolnionego wskaźnika spowoduje niezdefiniowane zachowanie programu. W związku z tym programiści przypisują wartość `NULL` do wskaźnika, aby go unieważnić. Późniejsze skorzystanie z takiego wskaźnika doprowadzi do błędu wykonywania programu.

```
int *p = (int*) malloc(sizeof(int));
free(p);
p = NULL;
```

- Należy unikać tzw. *podwójnego uwalniania wskaźnika*, czyli dwukrotnego zwolnienia danego bloku pamięci. Uruchomienie funkcji `free()` po raz drugi będzie skutkowało wywołaniem wyjątku (program zakończy się błędem).

```
int *p1 = (int*) malloc(sizeof(int));
int *p2 = p1;
free(p1);
free(p2); // podwójne uwolnienie
```

Literały złożone - tablice

- Przed standardem C99 w języku C nie był odpowiedników stałych dla typów złożonych tj. tablic i struktur[1]. Wraz z nastaniem standardu C99 wypłoniono tę lukę za pomocą *literałów złożonych*. Literały są stałymi, które nie są jednak symbolami np. `2` jest literałem typu `int`, `3.14` jest literałem typu `double`, `'U'` literałem typu `char`, a `"kot"` jest literałem łańcuchowym. Literały złożone służą do reprezentacji zawartości tablic i struktur.
- W przypadku tablic literały złożone wyglądają jak inicjalizacja tablicy poprzedzona przez ujętą w nawiasy nazwę typu ale pozbawionej nazwy. Poniżej literał złożony, tworzący tablicę bez nazwy zaiwaniającą trzy wartości całkowite (`int`):

```
(int [3]) {3, 5, 7};
```

  Oczywiście, tak jak w przypadku zwykłych tablic rozmiar w literale złożonym możemy pominąć:

```
(int []) {3, 5, 7};
```

[1] O strukturach będzie mowa na jednym z kolejnych wykładów.

Literały złożone - tablice

- Ponieważ literały złożone nie mają nazwy, nie można ich stworzyć w jednej instrukcji, a następnie korzystać z nich dalej. Możemy je wykorzystać w miejscu ich tworzenia. Możemy jednak wykorzystać wskaźniki do przechowywania ich lokalizacji:

```
int *t;
t = (int []) {3, 5, 7};
```

  Teraz możemy korzystać z `t` analogicznie jak ze zmiennej tablicowej[2].

  PRZYKŁAD 7

- Literały złożone możemy przekazywać jako argument faktyczny funkcji o odpowiednim argumencie formalnym. Takie zastosowanie, w którym przekazujemy informacje do funkcji bez faktycznego stworzenia np. tablicy, jest typowym zastosowaniem literałów złożonych.

[2] Pamiętać należy, że wskaźnik `t` będzie miał przypisany adres pierwszego elementu literału złożonego ale nie będzie nigdy zmienną tablicową np. `sizeof(t)` zwróci rozmiar wskaźnika, a nie tablicy.

Literały złożone - tablice

- Literały złożone w analogiczny sposób możemy wykorzystać do tworzenia stałych tablicowych o większych rozmiarach. Poniżej przykład utworzenia tablicy dwuwymiarowej z wykorzystaniem techniki literałów złożonych wraz z zachowaniem jej adresu:

```
int (*t)[3];  // deklaracja wskaźnika
              // do tablicy o 3 elementach typu int
t = (int [2][3]) {{3, 5, 7}, {11, 13, 17}};
```

  PRZYKŁAD 8

- Literały złożone stanowią mechanizm odwołania się do wartości potrzebnych tylko tymczasowo.. Mają one zasięg bloku - ich istnienie jest ograniczone do czasu, kiedy program wykonuje dany blok kodu w którym ów literał jest zdefiniowany.

Tablice wskaźników

- Poza swoim typem zmienne wskaźników zachowują się identycznie z pozostałymi zmiennymi. Nic nie stoi na przeszkodzie aby tworzyć tablice wskaźników. Ogólna postać deklaracji tablicy `t` jednowymiarowej jest następująca:

  `type* t[size];`

  Utworzona została tablicy o rozmiarze `size`, której każdy element jest wskaźnikiem do typu `type`.

- Poniżej przedstawiono przykład, w którym zadeklarowano tablicę wskaźników typu `int`, alokowano pamięć na każdy element tablicy i zainicjalizowano tę pamięć wartościami indeksów tablicy:

```
int* array[4];
for(int i=0; i<4; i++) {
  array[i] = (int*) malloc(sizeof(int));
  *array[i] = i;
}
```

Tablice wskaźników

- Korzystając z tablic wskaźników oczywiście możemy stosować notacje wskaźnikową

```
1 int* array[4];
2 for(int i=0; i<4; i++)
3 {
4   *(array + i) = (int*) malloc(sizeof(int));
5   **(array + i) = i;
6 }
```

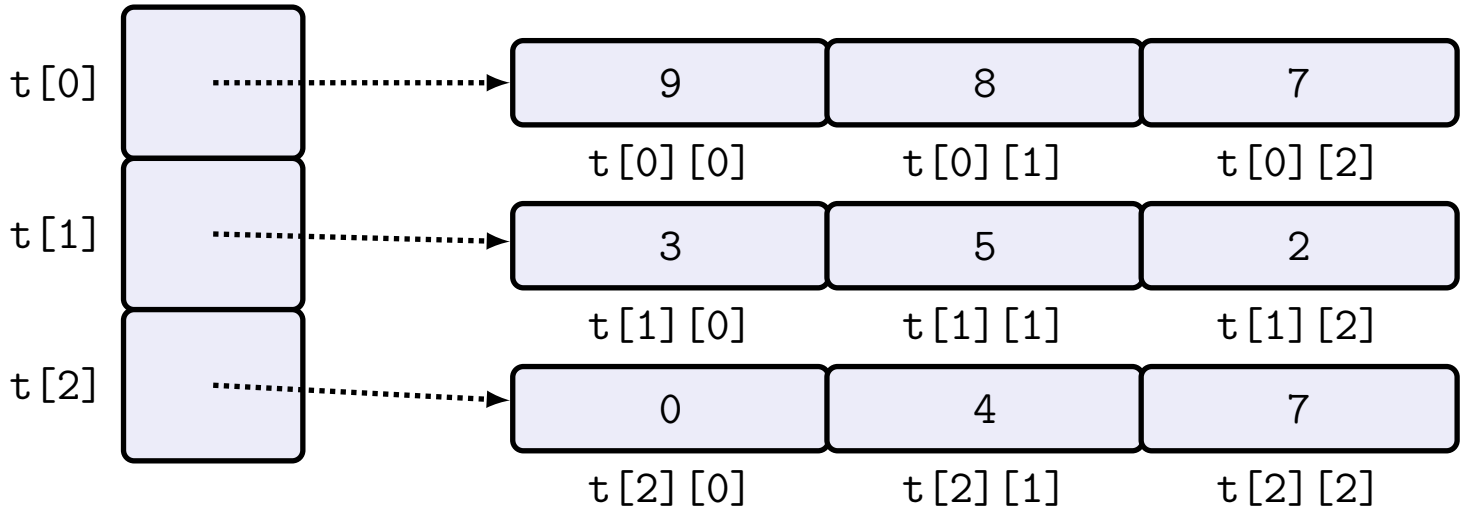
  Wyrażenie `(array + i)` to adres i-tego elementu tablicy. Musimy zmodyfikować zawartość znajdującą się pod tym adresem, a więc stosujemy operator wyłuskania tj. `*(array + i)`. W czwartej linii kodu dokonano przypisania alokowanej pamięci do tego adresu. Wyłuskując wyrażenie po raz drugi, co robimy w piątej linii kodu, uzyskamy adres alokowanej pamięci. Następnie pod ten adres przyporządkujemy wartość zmiennej `i`.

- Przykład: Jakie są wartości poniższych wyrażeń (zakładamy deklaracje jak powyżej)?
  `*array[0];` `**array;` `**(array + 1);` `array[0][0];` `array[3][0];`

Tablice wskaźników

- Możemy wykorzystać tablice wskaźników i np. tablice jednowymiarowe tworzone za pomocą literałów złożonych do budowy tablicy dwuwymiarowej[3]:

```
int* t[] = {
(int[]) {9, 8, 7},
(int[]) {3, 5, 2},
(int[]) {0, 4, 7}};
```



PRZYKŁAD 9

[3]Zamiast literałów złożonych można zastosować zwykłe tablice przypisując ich nazwy jako wartość kolejnych elementów tablicy wskaźników.

Dynamiczna alokacja pamięci dla tablic dwuwymiarowych

- Przykład:

  dynamiczna alokacja pamięci dla tablicy o wymiarach `3×5` o elementach typu `int` (ciągły blok pamięci, ale stała liczba kolumn!).

alokacja pamięci	dealokacja pamięci

`int (*t)[5]; // wskaźnik do tab. 5 el. int`
`int rows = 3;`
`t = (int (*)[5]) malloc(rows * 5 *sizeof(int));`
`for (int i = 0; i < rows; i++)`
`  for (int j = 0; j < 5; j++)`
`  t[i][j] = rand() % 100;`
	`free(t);`

PRZYKŁAD 10

Dynamiczna alokacja pamięci dla tablic dwuwymiarowych

- Przykład:

  dynamiczna alokacja pamięci dla tablicy o wymiarach `3×5` o elementach typu `int` (potencjalnie nieciągły blok pamięci).

alokacja pamięci

```
int **t;// wskaźnik do wskaźnika
int r = 3;
int c = 5;
t = (int **) malloc(r * sizeof(int *));
for(int i = 0; i < r; i++)
  t[i]=(int *) malloc(c * sizeof(int));
for (int i=0; i < r; i++)
  for (int j=0; j < c; j++)
    t[i][j]=rand() % 100;
```

dealokacja pamięci

```
for (int i = 0; i < r; i++)
    free(t[i]);
  free(t);
```

PRZYKŁAD 11

Dynamiczna alokacja pamięci dla tablic dwuwymiarowych

- Przykład:

  dynamiczna alokacja pamięci dla tablicy o wymiarach `3×5` o elementach typu `int` (ciągły blok pamięci).

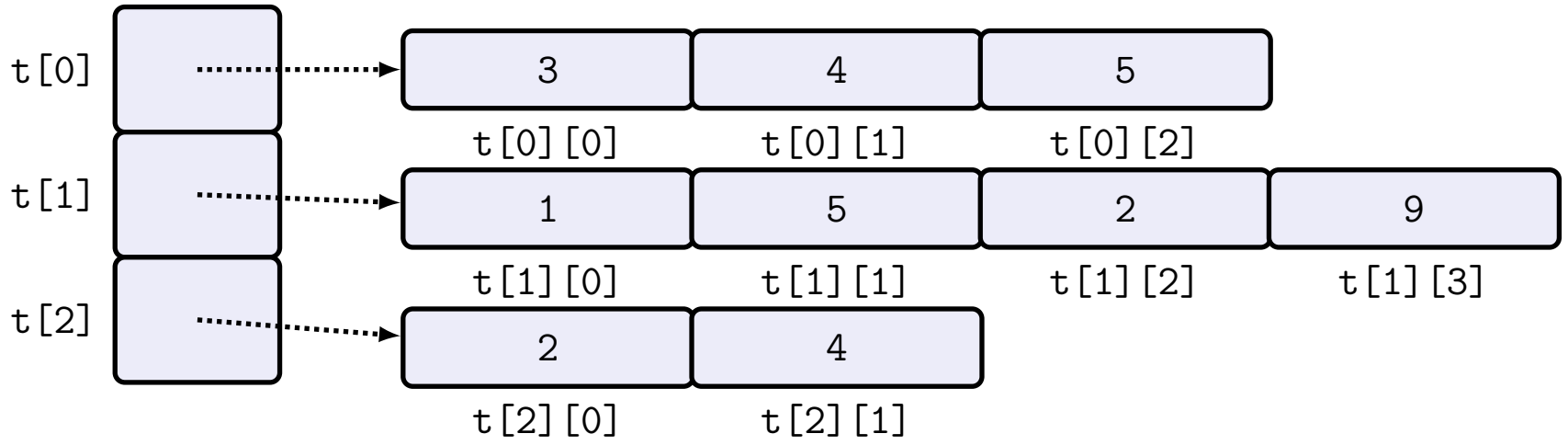
alokacja pamięci	dealokacja pamięci
`int **t;// wskaźnik do wskaźnika`	
`int r = 3;`	
`int c = 5;`	
`t = (int **) malloc(r * sizeof(int *));`	
`t[0]=(int *) malloc((c * r) * sizeof(int));`	`free(t[0]);`
`for(int i = 0; i < r; i++)`	`free(t);`
`  t[i] = t[0] + i * c;`	
`for (int i=0; i < r; i++)`	
`  for (int j=0; j < c; j++)`	
`    t[i][j]=rand() % 100;`	

PRZYKŁAD 12

Tablice postrzępione

- *Tablica postrzępiona* jest to taka dwuwymiarowa tablica, która posiada różną liczbę kolumn w każdym rzędzie. Do jej utworzenia wykorzystuje się tablice wskaźników.
- Tworząc tablice postrzępioną możemy wykorzystać literały złożone, zwykłe tablice lub tablice alokowane dynamicznie:

```
int* t[] = {
(int[]) {3, 4, 5},
(int[]) {1, 5, 2, 9},
(int[]) {2, 4}};
```



Tablice postrzępione

- Uzyskiwanie dostępu do elementów tablicy postrzępionej może być skomplikowanym procesem, gdyż każdy wiersz może mieć różną liczbę elementów. Proces ten może być uproszczony dzięki zastosowaniu oddzielnej tablicy przechowującej rozmiary każdego z wiersza tablicy postrzępionej.

  PRZYKŁAD 13

  PRZYKŁAD 14

Pytania

1. Czym jest wskaźnik do null i wskaźnik na `void`?
2. Jaki jest związek tablicy dwuwymiarowej ze wskaźnikami?
3. Jakie są funkcje języka C służące do alokacji pamięci? W jaki sposób alokujemy pamięć dynamicznie dla obiektów i tablic jednowymiarowych?
4. Czy jet problem wycieków pamięci i w jaki sposób prawidłowo zwalniać pamięć?
5. Czym są tablicowe literały złożone?
6. Jak tworzymy tablice wskaźników?
7. W jaki sposób alokujemy i dealokujemy dynamicznie pamięć dla tablic dwuwymiarowych?
8. Czym są tablice postrzępione?