# Podstawy programowania: Laboratorium nr 13
## Wskaźniki. Ręczna alokacja i dealokacja pamięci

2017-2018

*mgr inż. Przemysław Walkowiak*
*dr inż. Michał Ciesielczyk*

# Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. umieszczeniu wszystkich rozwiązań w jednym pliku, chyba, że w poleceniu napisano inaczej.
5. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie).

# Wprowadzenie

## Wskaźniki

Każdy zmienna jaką deklarujemy, każdy obiekt jaki tworzymy w języku C++ zajmuje pewien obszar pamięci. Na początkowych zajęciach laboratoryjnych, były przeprowadzane eksperymenty z operatorem `sizeof`, który zwracał rozmiar obszaru zajmowanego przez daną zmienną. I tak na przykład wynikiem działania `sizeof(int)` jest liczba 4. Czyli typ całkowity `int` zajmuje 4 bajty pamięci. Analogicznie możemy zmierzyć rozmiar zmiennej `long long x`, gdzie `sizeof(x)` zwróci nam wartość 8 bajtów.

Oprócz rozmiaru, każda zmienna ma jeszcze drugą cechę: miejsce w pamięci. Kompilatorowi/programowi nie wystarczy wiedza jaki duży jest obiekt, ale potrzebuje jeszcze informację, gdzie ten obiekt jest. Tak samo jak nie wystarcza nam wiedza o tym jak wygląda jakaś konkretna książka. By z niej skorzystać potrzebujemy wiedzieć, gdzie ona się znajduje.

To miejsce w pamięci nazywamy po prostu 'adresem'. Adres jest dodatnią liczbą całkowitą. W zależności od architektury komputera jest reprezentowana przez typ całkowity bez znaku 4 lub 8 bajtowy, odpowiednio dla 32 i 64 bitowego komputera/systemu operacyjnego-/aplikacji.

W języku C++, by pobrać adres zmiennej używa się operatora `&`, np. `&x` – zwraca adres zmiennej `x`. Adres możemy również przechować w innej zmiennej, zmiennej typu wskaźnikowego. Np.

```
int x;
int *px = &x; // zmienna typu wskaźnikowego na typ int
```

Typ wskaźnikowy jest tworzony poprzez dodanie modyfikatora `*` do typu zmiennej, której adres chcemy przechowywać. Ponieważ każda zmienna znajduje się w pamięci i zajmuje jakiś jej obszar, to samo tyczy się zmiennej `px`. Ma ona rozmiar 4 lub 8 bajtów (32 lub 64bit) oraz również ma swój adres. Czyli możemy kolejny raz zaaplikować operator pobrania adresu `&`.

```
int x;
int *px = &x; // zmienna typu wskaźnikowego na typ int
int **ppx = &px; // zmienna typu  wskaźnikowego na wskaźnik typu int
```

Oczywiście sama znajomość adresu zmiennej w pamięci nie pomaga zbyt wiele. Musimy jeszcze mieć możliwość sięgnięcia pod ten adres. Do tego służy operator `*` (operator, nie modyfikator typu).

```
int x = 1337;
int *px = &x; // zmienna typu wskaźnikowego na typ int
int **ppx = &px; // zmienna typu  wskaźnikowego na wskaźnik typu int

std::cout << x; // wyswietli 1337
std::cout << px; // wyswietli adres zmiennej x
std::cout << *px; // wyswietli wartosc wskazywana przez adres przechowywany
                  // adres (czyli 1337)

std::cout << ppx; // wyswietli adres zmiennej px (wartosc ppx)
std::cout << *ppx; // wyswietli adres zmiennej x (wartosc px)
std::cout << **ppx; // wyswietli wartosc zmiennej x (1337)
```

## Do czego nam wskaźniki?

### Parametry funkcji

Przez wzgląd na fakt, że adres ma zawsze stały rozmiar (4/8 bajtów), taki adres jest dużo łatwiej kopiować procesorowi, aniżeli bardziej złożone obiekty (np. tablica 100 elementów typu `int`). Ten fakt wykorzystuje się przy definiowaniu listy parametrów jakie chcemy przekazać na przykład do funkcji.

Rozważmy trzy funkcje:

```
void fun1(std::array<int, 100> a); // przyjmuje tablicę przez wartość
                                   // (jest robiona kopia)
void fun2(std::array<int, 100> &a); // przyjmuje tablicę przez referencję
                                    // (nie ma kopii)
void fun3(std::array<int, 100> *a); // przyjmuje tablicę przez wskaźnik
                                    // (nie ma kopii)

std::array<int, 100> tablica;
fun1(tablica);
fun2(tablica);
fun3(&tablica);
```

W przypadku funkcji `fun1`, nasza tablica jest kopiowana – przy dużych obiektach niekoniecznie to jest to czego chcemy. Funkcja `fun2` przyjmuje jako argument referencję do tablicy, czyli nie będzie żadnej kopii, jednakże jednocześnie jesteśmy zmuszeni (w pozytywny sposób), by przekazać do funkcji istniejący obiekt.

Trzecia funkcja przyjmuje jako argument adres tablicy i częściowo zachowuje się tak jak reference (tj. nie kopiuje danych), jednakże zezwala dodatkowo na przekazanie adresu zerowego:

```
std::array<int, 100> *ptablica = nullptr; // wskaźnik o wartosci nullptr (0)
                                          // na tablicę
fun3(ptablica);
fun3(nullptr);
```

Taka możliwość w zależności od sytuacji może być, lub wręcz przeciwnie, pożądana. Pierwszy wariant jest prawdziwy, gdy chcemy mieć możliwość przekazywania danych do funkcji w sposób opcjonalny – czyli spodziewamy się, że użytkownik funkcji `fun3` może nie chcieć przekazać nam danych i traktujemy to jako poprawne zachowanie. Jeżeli chcemy mieć pewność, że użytkownik zawsze przekaże nam poprawny obiekt (drugi wariant), wówczas powinniśmy się zdecydować na funkcję z parametrem typu referencyjnego (to pokrywa większość przypadków).

### Przechowywanie uchwytów do pamięci zaalokowanej dynamicznie w sposób ręczny. Tzw. owning-pointers

W języku C++ mamy możliwość ręcznego alokowania pamięci (tj. rezerwowania pamięci). Daje to nam bardzo dużą kontrolę nad czasem życia danych. Czas życia obiektów o typach z biblioteki standardowej (np. `std::vector<T>`) jest ściśle związany z blokiem instrukcji, gdzie zostały one zdefiniowane (są usuwane wraz z klamrą zamykającą dany blok).

Do dynamicznej alokacji pamięci używa się operatora `new` służącego do rezerwacji pamięci na jeden element danego typu oraz `new[]` jako tablicowy odpowiednik – rezerwuje ciągła pamięć na zadaną liczbę elementów danego typu. Oba te operatory zwracają zawsze adres na początek zarezerwowanej pamięci (dlatego ten wstęp o wskaźnikach ;) ).

```
int *px = new int; // alokuje pamięć dla jednego integera
const int N = 5; // stała, znana w czasie kompilacji
int *py = new int[N]; // alokuje pamięć dla tablicy 5 integerów
int n = 4;  // wartość znana tylko w czasie działania programu
int *pz = new int[n]; // alokuje pamięć dla tablicy 3 integerów

*px = 1337; // przypisanie wartości 1337 do wskazywanego obszaru pamięci
px = 42; // nadpisujemy adres - w 99.9999% przypadków to jest błąd
py = 42; // j.w.
py[3] =  1337; // przypisanie wartości elementowi o indeksie 3
pz[3] = 1337; // j.w.
```

Z operatorami `new, new[]`, powiązane są operatory `delete, delete[]`, służące do zwalniania zarezerwowanej/zaalokowanej pamięci. Oba operatory przyjmują jako parametr adres pamięci. Jednakże należy uważać, by zastosować poprawny, sparowany operator.

```
delete px;
delete[] py; // Ok. Wewnątrz [] nic nie ma.
delete pz; // Błąd. pz przechowuje adres zaalokowany za pomocą new[],
           // powinien zostać użyty delete[]
```

Problemem z ręcznym zarządzaniem pamięcią jest to, że trzeba to robić w sposób precyzyjny. To znaczy, nigdy nie możemy pozwolić sobie na utratę adresu zaalokowanej pamięci oraz zawsze powinien być zastosowany prawidłowy operator delete. Jeżeli stracimy informację o adresie, nie jesteśmy w stanie jej odzyskać i mamy do czynienia z wyciekiem pamięci.

```
int *px = nullptr; // na razie nic tutaj nie przechowujemy
{
    px = new int[5]; // alokujemy pamięć
    int *py = new int[10] // j.w.
    int *pz = new int[10] // j.w.

    delete[] py; // poprawnie zwalniamy pamięć
} // zmienne px i pz są niszczone, ale nie obszar na który wskazują
delete[] px; // poprawnie zwalniamy pamięć - px przechowuje tylko adres
delete[] py; // błąd, Zmienna py nie istnieje już w tym miejscu.
                // Nie będziemy w stanie tej operacji wykonać.
                // Mamy wyciek pamięci.
```

Jak widać w powyższym przykładzie, dopóki posiadamy uchwyt/zmienną z adresem zaalokowanej pamięci, zasób możemy swobodnie zwolnić. W przeciwnym przypadku doprowadzamy do wycieku pamięci.

### Owning i non-owning pointers

Wskaźniki z poprzedniej sekcji możemy nazwać z ang. owning pointers – "wskaźniki posiadające", czyli wszystkie wskaźniki, które przechowują adresy ręcznie zaalokowanej pamięci.

Jednocześnie wyróżniamy tzw. non-owning pointers – "wskaźniki nieposiadające", będące tylko uchwytami/przechowujące adresy obiektów zarządzanych w sposób automatyczny (lub przez "kogoś innego").

Obecnie zalecane jest unikanie zarządzania pamięcią na własną rękę i zamiast używać operatorów `new, delete`, lepiej jest korzystać z tzw. smart pointers: `std::unique_ptr, std::shared_ptr` wraz z towarzyszącymi im funkcjami `std::make_unique, std::make_shared`.

# Zadania

## Zadanie 1

Zainicjalizuj dwa wektory o długości $n$ wartościami losowymi z przedziału $\langle 3; 27\rangle$. Oblicz ich iloczyn skalarny oraz wyświetl całe działanie na ekranie. Wartość $n$ wczytaj od użytkownika. Napisz funkcję do wypisywania wektora na ekranie. Skorzystaj z ręcznie alokowanej pamięci.

**Wskazówka 1** iloczyn skalarny:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i, \tag{1}$$

gdzie $a, b \in \mathbb{R}^n, n = 1, 2, \ldots,$

**Dodatkowe informacje:**

- operatory `new` i `delete` - `http://www.cplusplus.com/reference/std/new/`

## Zadanie 2

Zaimplementuj poniższe funkcje:

1. Wczytującą od użytkownika $n$ liczb zmiennoprzecinkowych.
2. Liczącą poniższe statystyki.
3. Wyświetlającą statystyki na ekranie.

Statystyki do obliczenia:

a) średnia,
b) suma,
c) maksimum,
d) minimum.

Tablicę oraz strukturę ze statystykami utwórz z wykorzystaniem operatora `new`. Dostosuj resztę funkcji.

## Zadanie 3

Utwórz dwie macierze losowe $A$ i $B$ o wartościach z przedziału $\langle -16; -4\rangle$ i wymiarach $m \times n$, a następnie wykonaj operacje $A + B$ i $A - B$. Wartość $m, n$ wczytaj od użytkownika.

## Zadanie 4

Utwórz dwie macierze losowe $A$ i $B$ o wartościach z przedziału $\langle 1; 14\rangle$ i wymiarach odpowiednio $m \times n$ oraz $n \times p$, a następnie wykonaj operację $A * B$. Wartość $m, n, p$ wczytaj od użytkownika.

**Dodatkowe informacje:**

- Mnożenie macierzy - `http://pl.wikipedia.org/wiki/Mno%C5%BCenie_macierzy`