



Politechnika Wrocławska

Wydział Elektroniki,  
Fotoniki i Mikrosystemów

---

## Laboratorium informatyki

### Ćwiczenie nr 6. Wstęp do arytmetyki wskaźników. Tablice

#### Zagadnienia do opracowania:

- wskaźniki i ich arytmetyka
- operator dereferencji i operator adresu
- tablice jedno- i wielowymiarowe
- tablice a wskaźniki
- łańcuchy znakowe
- przekazywanie argumentów do funkcji *main()*

---

## Spis treści

<b>1</b>	<b>Cel ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Wprowadzenie</b>	<b>2</b>
2.1	Wskaźniki . . . . .	2
2.2	Tablice . . . . .	10
2.3	Arytmetyka wskaźników . . . . .	15
2.4	Wskaźnik w nagłówku funkcji . . . . .	17
2.5	Łańcuchy znakowe . . . . .	20
2.6	Przekazywanie argumentów do funkcji <i>main()</i> . . . . .	26
<b>3</b>	<b>Program ćwiczenia</b>	<b>28</b>
<b>4</b>	<b>Dodatek</b>	<b>31</b>
4.1	Modyfikacja „stałej” z wykorzystaniem wskaźnika . . . . .	31

---

## 1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z podstawowymi zagadnieniami z zakresu arytmetyki wskaźników oraz ich zastosowaniem w prostych aplikacjach konsolowych w języku *C* i *C++*.

## 2. Wprowadzenie

### 2.1. Wskaźniki

*Adresem pamięci* nazywa się unikalny identyfikator, umożliwiający jednoznaczne rozpoznanie lokalizacji danego zasobu w pamięci komputera. Każda zmienna, stała czy funkcja posiada swój adres, pod którym została zainicjalizowana. Długość adresu jest zależna od **architektury komputera**, np. systemy 32-bitowe cechują się adresami o długości 4 B (32 b), natomiast systemy 64-bitowe – adresami o długości 8 B (64 b). Aby pobrać adres zmiennej należy posłużyć się operatorem adresu – *operatorem* **&**. Przykład przedstawiono na listingu 1.

```
1 int number = 123;
2 // Wyświetl wartość zmiennej
3 printf("%d\n", number);
4 // Wyświetl adres zmiennej
5 printf("%p\n", &number);
```

Listing 1. Zastosowanie operatora adresu

**Wskaźnikiem** (*ang. pointer*) nazywamy specjalny typ zmiennych, przechowujący jako wartość *adres pamięci*. Innymi słowy, **wskaźnik** zawiera informację o miejscu alokacji innego zasobu w pamięci komputera. Poza tym, wskaźnik, tak jak każda inna zmienna, posiada również swój adres, pod którym został zainicjalizowany. **Rozmiar zmiennej wskaźnikowej jest rów-**

---

ny długości adresu w danej architekturze komputera. Zatem operator *sizeof* wywołany na dowolnej zmiennej wskaźnikowej w systemie 32-bitowym zwróci wartość 4 (bajty), a w systemie 64-bitowym – 8 (bajtów). **Zmienne wskaźnikowe deklaruje się z wykorzystaniem symbolu *\**.** **Zastosowanie wskaźników w kodzie *C/C++* umożliwia:**

- pracę na oryginałach zmiennych;
- dynamiczną alokację pamięci (na *stercie*) (*ang. heap*);
- optymalizację czasu wykonania programu.

Na listingu 2. przedstawiono zmienne *x*, *y*, *z*, kolejno typów: *char*, *int* oraz *unsigned short*. Zmienne te posiadają różne rozmiary w pamięci komputera (przykładowo: *char* – 1 B, *int* – 4 B, *unsigned short* – 2 B; patrz: tabela 1 [Ćw. 2]). Zmienna *ptr* jest **zmienną wskaźnikową**. Definiując zmienną *ptr* przypisano jej, jako wartość, adres zmiennej *y* (za pomocą operatora adresu *&*).

```
1 char x = 0x16; // 22 w systemie dziesiętnym
2 int y = 0x29A; // 666 w systemie dziesiętnym
3 unsigned short z = 0x7D; // 125 w systemie dziesiętnym
4 int * ptr = &y; // adres zmiennej y
```

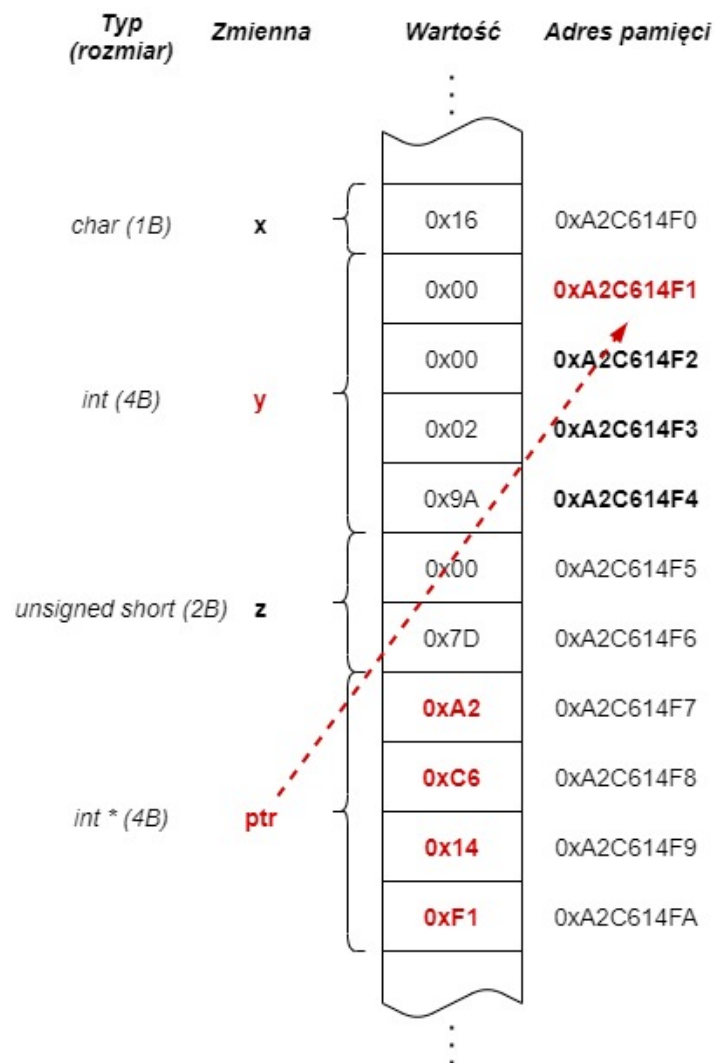
Listing 2. Deklaracja zmiennej wskaźnikowej

Należy zwrócić uwagę, że **typ zmiennej wskaźnikowej jest zgodny z typem wskazywanej zmiennej** (tu: *int* \* i *int*). Analogicznie, definiując **wskaźnik na zmienną *x*** należałoby określić jego typ jako *char* \*. Deklarację i definicję zmiennej wskaźnikowej można rozdzielić:

---

```
1 int * ptr; // deklaracja zmiennej
2 ptr = &y; // definicja zmiennej
```

Na rys. 2.1. przedstawiono fragment pamięci komputera (np. **stosu**) w 32-bitowym systemie operacyjnym. Każda komórka pamięci o rozmiarze 1 B posiada swój unikalny adres. Wartość zmiennej **y** jest rozdzielona między cztery komórki pamięci (ponieważ typ **int** w tym przypadku ma rozmiar 4 B). Analogicznie, zmienne **x** i **y** zajmują odpowiednio jedną i dwie komórki pamięci. Zarówno adresy, jak i wartości zmiennych, w pamięci komputera wyrażone są za pomocą liczb w systemie heksadecymalnym (szesnastkowym). Jest to najczęściej stosowana reprezentacja. Zmienna wskaźnikowa **ptr** o rozmiarze 4 B zajmuje cztery komórki pamięci, w których **przechowywany jest adres pierwszej komórki pamięci zmiennej y**. Widoczne jest teraz, że typ zmiennej wskaźnikowej jest bardzo istotny. Sam wskaźnik przechowuje adres pierwszej z komórek pamięci, na których została zaalokowana zmienna, natomiast jego typ (tu: **int \***) niesie informację, na ilu komórkach pamięci zapisana jest wartość zmiennej (tu: 4), tzn. ile bajtów należy pobrać w celu odczytania pełnej wartości zmiennej.



Rys. 2.1. Wizualizacja wskaźnika w pamięci komputera (z listingu 2)

Wykorzystując wskaźniki można odczytywać, jak i przypisywać wartości wskazywanym zmiennym. Służy do tego **operator dereferencji**, zwany również **operatorem wyłuskania** – **operator\***. Przykład wykorzystania operatora dereferencji przedstawiono na listingu 3. Na podstawie tego przykładu widać, że pary wyrażeń: {*variable*, *\*ptr*} oraz {&*variable*, *ptr*} mają tę samą wartość.

---

```
1 int variable = 4;
2 // ptr wskazuje na zmienna o wartosci 4
3 int * ptr = &variable;
4 std::cout << variable << std::endl;
5 std::cout << *ptr << std::endl;
6
7 // ptr wskazuje na zmienna o wartosci 2
8 // Wartosc 4 zostala nadpisana w pamieci przez
   wartosc 2
9 *ptr = 2;
10 std::cout << variable << std::endl;
11 std::cout << *ptr << std::endl;
12
13 // Wyswietl adres zmiennej variable
14 std::cout << &variable << std::endl;
15 std::cout << ptr << std::endl;
16
17 // Wyswietl adres wskaznika ptr
18 std::cout << &ptr << std::endl;
```

Listing 3. Odczytanie i nadpisanie wartości zmiennej z użyciem wskaźnika

Zarówno język *C*, jak i *C++*, dają programiście dużą swobodę w wykonywaniu operacji na pamięci komputera. Ma to swoje wady i zalety. Z jednej strony umożliwia to tworzenie szybkich, pamięciowo–wydajnych programów, ale równocześnie nie zapewnia ochrony przed przeprowadzeniem niewłaściwych operacji na pamięci. **Wskaźniki stanowią jeden z najważniejszych mechanizmów języka *C/C++*, którego zrozumienie jest niezbędne do wykorzystania pełni możliwości i pojęcia „filozofii” programowania w języku *C/C++*.** Dlatego też rozważa i świadomość wykonywanych operacji są wymagane przy posługiwaniu się zmiennymi wskaźnikowymi. Jedną z podstawowych zasad, o których musi pamiętać każdy programista,

---

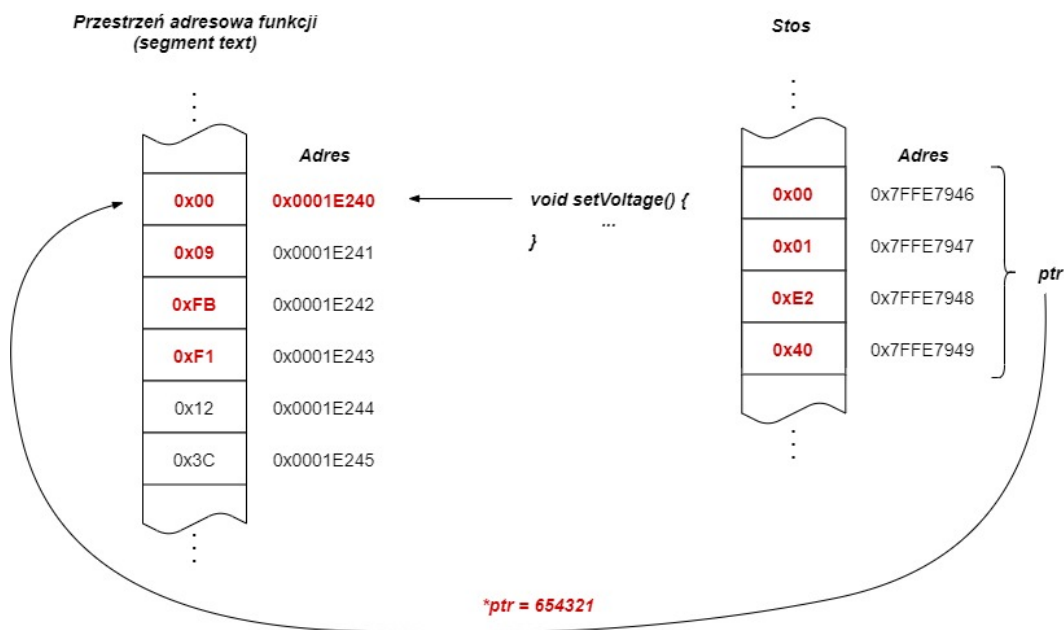
mówi: **nie wolno wykonywać dereferencji na niezainicjalizowanym wskaźniku!** Przykład przedstawiono na listingu 4., gdzie za pomocą **operatora** `*` zapisuje się wartość 654321 do pamięci wskazywanej przez lokalną zmienną wskaźnikową ***ptr***.

```
1 {  
2     int * ptr;  
3     *ptr = 654321; // 0x9FBF1  
4 }
```

Listing 4. Dereferencja na niezainicjalizowanym wskaźniku

Niezainicjalizowany wskaźnik (tak jak inne niezainicjalizowane zmienne lokalne) przechowuje losową wartość. Jest to wartość, która znajdowała się w obszarze stosu, na którym został odłożony wskaźnik. Zatem **wskaźnik przechowuje losowy adres!** Operacja `*ptr = 654321` wpisuje wartość do komórki pamięci określonej przez losowy adres. Rys. 2.2. przedstawia sytuację, wyolbrzymioną (małe prawdopodobieństwo takiego ułożenia adresów), do jakiej może doprowadzić przypisanie wartości do niezainicjalizowanego wskaźnika. W segmencie ***text*** (*code*), pod adresem **0x0001E240**, leży funkcja ***setVoltage()***, ustalająca napięcie zasilania pewnego układu elektronicznego. Kolejne komórki pamięci (0x0001E240, 0x0001E241, 0x0001E242, ...) zawierają instrukcje zawarte w ciele funkcji ***setVoltage()*** (w postaci kodu maszynowego). Niezainicjalizowany wskaźnik ***ptr*** przechowuje (losową) wartość **0x0001E240** – **adres funkcji *setVoltage()*!** Wpisując wartość 654321 do kolejnych komórek pamięci, zajmowanych przez funkcję ***setVoltage()***, zmienia się jej działanie. W wyniku tego, ustawiona zostanie nieprawidłowa wartość napięcia zasilającego układ, co w skrajnym przypadku doprowadzi do jego uszkodzenia.





Rys. 2.2. Dereferencja na niezainicjalizowanym wskaźniku

Możliwe jest zapobieganie modyfikacji danych leżących pod adresem wskazywanym przez zmienną wskaźnikową przez zastosowanie specyfikatora **const**:

```

1 float variable = 2.0;
2 const float * ptr = &variable;
3 *ptr = 5.5; // Błąd kompilacji - modyfikacja stałej

```

Zabieg taki można zastosować np. w celu optymalizacji kodu (zapobiegnięciu kopiowania zmiennej; w szczególności zmiennej o znacznym rozmiarze), przy jednoczesnym zabezpieczeniu przed nieintencjonalną modyfikacją zmiennej. Jest to szczególnie istotne w kontekście programowania mikrokontrolerów o niewielkim rozmiarze pamięci. Należy jednak mieć na uwadze, że zabieg ten jest zabezpieczeniem modyfikacji zmiennej **jedynie przed próbą zmiany jej wartości przez modyfikację wskaźnika zadeklarowanego z użyciem słowa kluczowego **const****. Kilka wskaźników zadeklarowanych zarówno bez, jak i przy użyciu specyfikatora **const**, może jednocześnie prze-

---

chowować adres tej samej zmiennej:

```
1 float variable = 2.0;
2 const float * ptr = &variable;
3 float * ptr_2 = &variable;
4 *ptr_2 = 5.5; // Poprawna modyfikacja zmiennej
   variable
```

Możliwe jest również ustalenie stałego adresu przechowywanego przez wskaźnik. Warto zwrócić uwagę na położenie słowa kluczowego **const** w obu przypadkach:

```
1 short variable = 125;
2 short * const ptr = &variable;
3 short anotherVariable = 121;
4 ptr = &anotherVariable; // Bład kompilacji - zmiana
   adresu
```

Mając na uwadze odwoływanie się do wskaźników, które wskazują na niewłaściwą (losową) lokalizację w pamięci, koniecznym stało się wprowadzenie sposobu weryfikacji adresu przypisanego do zmiennej wskaźnikowej. W tym celu wprowadzono wartość **NULL**. Jest to makro (zdefiniowane za pomocą dyrektywy preprocesora `#define`) określające, że wskaźnik nie wskazuje na adres żadnego zasobu w pamięci komputera (jego wartość wynosi 0). Należy mieć jednak na uwadze, że kompilator nie inicjalizuje wskaźników automatycznie za pomocą wartości **NULL**. Obowiązek ten ciąży na programiście. **Dereferencja wskaźnika o wartości *NULL* skutkuje niezdefiniowanym zachowaniem!**

```
1 int * ptr = NULL;
2 std::cout << *ptr; // niezdefiniowane zachowanie
```

---

Język *C++* (od standardu *C++11*) wprowadził *literał wskaźnikowy* *nullptr*. Jest to słowo kluczowe, pod którym zdefiniowana jest wartość wskaźnika niewskazującego na żaden adres zasobu w pamięci. Różnica między *NULL* a *nullptr* polega na tym, że *NULL*, jako dyrektywa preprocesora, interpretowany jest jako zmienna całkowita (*int*), a *nullptr* jako zmienna wskaźnikowa (*nullptr\_t*). Z tego względu pisząc kod w języku *C++* (od standardu *C++11*) należy używać *nullptr* zamiast *NULL*. **Należy mieć na uwadze, że dereferencja wskaźnika o wartości *nullptr* również skutkuje niezdefiniowanym zachowaniem!**

## 2.2. Tablice

Często potrzebne jest przechowywanie wielu zmiennych jednego typu, np. pobierając od użytkownika kilkanaście liczb, jedna po drugiej. W takim przypadku pomocne okazują się *tablice*. *Tablice* stanowią ciąg zmiennych, **ułożonych w sposób liniowy w pamięci komputera** (jedna po drugiej), posiadających wspólny identyfikator (nazwę). *Tablice* mogą być jedno- lub wielowymiarowe i mogą być alokowane w różnych obszarach pamięci (tablice statyczne/dynamiczne). *Tablice* deklarowane są z wykorzystaniem nawiasów kwadratowych: `[ ]`, a odwołanie do określonego elementu w ciągu przeprowadzane jest za pomocą *operatora indeksu* (`operator[]`). Przykład deklaracji jednowymiarowej tablicy przedstawiono na listingu 5. Warto zwrócić uwagę na sposób *indeksowania tablicy*. **Indeksowanie tablic w językach *C/C++* rozpoczyna się od 0**. Dlatego pętla inicjalizująca odwołuje się do elementów o numerach od 0 do 4 ( $i < 5$ ). Wywołanie na tablicy operatora indeksu (`operator[]`) z wartością 5 oznacza odwołanie do pierwszej komórki pamięci leżącej za zaalokowaną tablicą (szósty element), co skutkuje **niezdefiniowanym zachowaniem**.

```
1 // Deklaracja tablicy liczb całkowitych o rozmiarze
   5
2 int tab[5];
3 // Inicjalizacja elementów tablicy liczbami z
   zakresu 0...4
4 for (unsigned int i = 0; i < 5; ++i)
   // Przypisanie do i-tego elementu wartości i
6   tab[i] = i;
7 // Wypisanie wartości elementu o indeksie 2
8 std::cout << tab[2] << std::endl;
9 // Wypisanie wartości elementu spoza tablicy -
   niezdefiniowane zachowanie!
10 std::cout << tab[5] << std::endl;
```

Listing 5. Deklaracja tablicy i odwołanie do jej elementów

W definicji tablicy można jednocześnie zawrzeć inicjalizację kolejnych elementów:

```
1 int numbers[4] = {1, 3, 4, 8};
2 // Równoważnie - automatyczna dedukcja rozmiaru (4)
3 int tab[] = {1, 3, 4, 8};
```

Inicjalizując tablicę, możemy podać tylko część wartości poszczególnych elementów, pozostałe zostaną domyślnie zainicjalizowane zerami:

```
1 // Inicjalizacja {1, 2, 0, 0, 0, 0}
2 int tab[6] = {1, 2};
```

Wykorzystując operator indeksu można modyfikować wartość elementów na określonej pozycji:

```
1 float numbers[] = {2.0, 3.0, 4.0};
2 // Wyświetla 3.0
3 printf("%f\n", numbers[1]);
4
5 numbers[1] = 5.0;
6 // Wyświetla 5.0
7 printf("%f\n", numbers[1]);
```

Deklarowany rozmiar tablicy musi być liczbą całkowitą, a ponadto w przypadku tablic statycznych, musi być znany na etapie kompilacji programu. Jeżeli odwołujemy się do rozmiaru tej samej tablicy w wielu miejscach kodu, warto do wyrażenia rozmiaru posłużyć się stałą (lub makrem). Ewentualna zmiana rozmiaru tablicy będzie wymagała modyfikacji programu w jednym miejscu, bez konieczności przeszukiwania kodu pod kątem wszystkich przypadków użycia zmiennej. Przykład przedstawiono na listingu 6:

```
1 // Lub za pomocą makra #define tabSize 5
2 const unsigned int tabSize = 5;
3
4 int tab[tabSize] = {2, 7, -3};
5
6 for (unsigned int i = 0; i < tabSize; ++i)
7     std::cout << tab[i] << std::endl;
```

Listing 6. Deklaracja rozmiaru tablicy z użyciem stałej

Rozmiar tablicy otrzymamy również posługując się operatorem *sizeof*. Wywołanie operatora *sizeof* na tablicy zwróci całkowity rozmiar pamięci zaalokowany na tablicę (w bajtach). Dzieląc całkowity rozmiar tablicy przez rozmiar pojedynczego elementu otrzymujemy liczbę elementów:

---

```
1 int tab[] = {-2, 3, 20, 11, 8};
2 // Wyświetla rozmiar tablicy (5)
3 std::cout << sizeof(tab) / sizeof(int) << std::endl;
```

W językach *C/C++* możliwe jest definiowanie **tablic wielowymiarowych**, rozumianych jako *tablice tablic*. Inicjalizacja odbywa się z wykorzystaniem wielokrotnego układu nawiasów klamrowych:

```
1 const unsigned int tabSize = 3;
2
3 double matrix[tabSize][tabSize] = {
4     { 3.38, -7.11, 2.1 },
5     { -4.5, 9.25, -3.3 },
6     { 0.0, -0.5, 3.34 }
7 };
8
9 for (unsigned int i = 0; i < tabSize; ++i)
10     for (unsigned int j = 0; j < tabSize; ++j)
11         std::cout << matrix[i][j] << std::endl;
```

Analogicznie, jak w przypadku tablic jednowymiarowych, można zastosować automatyczną dedukcję rozmiaru, jednakże tylko dla pierwszego wymiaru:

---

```
1 // Macierz 4x4
2 unsigned short matrix[][4] = {
3     { 1, 2, 3, 4 }, // pierwszy wiersz
4     { 5, 6, 7, 8 }, // drugi wiersz
5     { 9, 10, 11, 12 }, // trzeci wiersz
6     { 13, 14, 15, 16 } // czwarty wiersz
7 };
8
9 // Przestrzen 3x3x3
10 int space[][3][3] = {
11     {
12         { -3, 1, 4 },
13         { 7, 0, -11 },
14         { 2, 4, -5 }
15     },
16     {
17         { 1, 2, 4 },
18         { -5, 9, 0 },
19         { 1, 1, 2 }
20     },
21     {
22         { 0, -8, -15 },
23         { -2, 4, -6 },
24         { 3, 7, 13 }
25     }
26 };
```

---

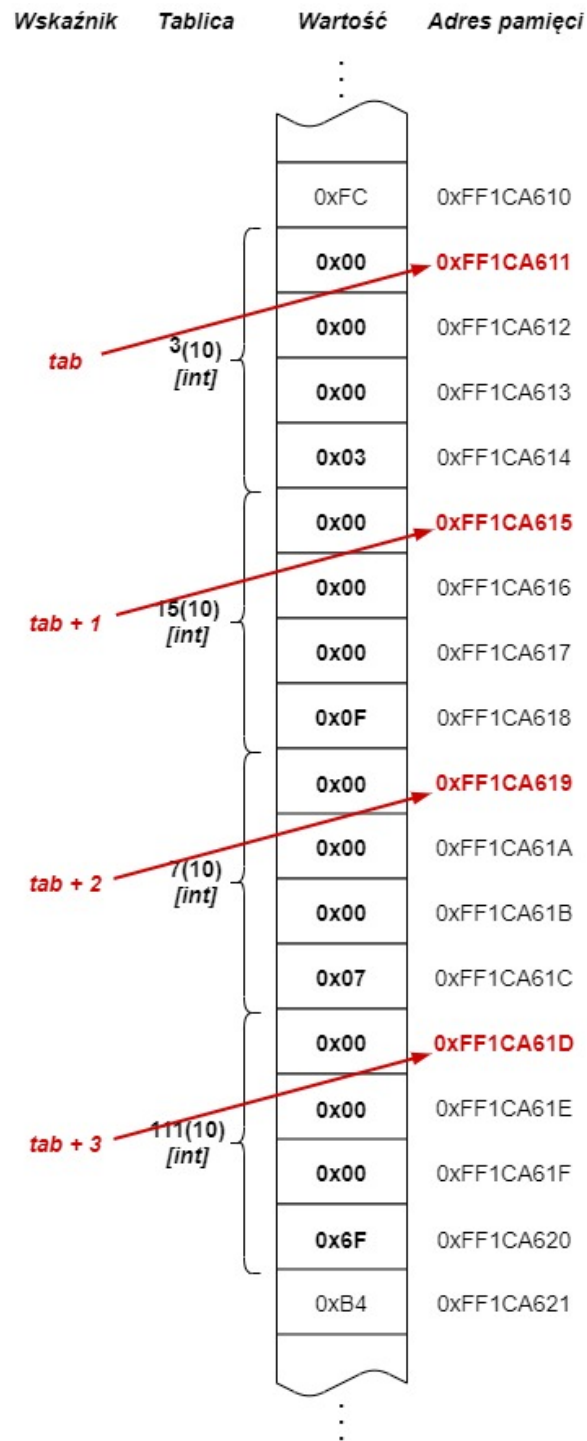
## 2.3. Arytmetyka wskaźników

W językach *C/C++* nazwa tablicy interpretowana jest jako **wskaźnik na pierwszy element tablicy**, czyli adres jej pierwszego elementu. Dlatego też odwołanie do elementu tablicy o konkretnym indeksie może być równoważnie zapisane z wykorzystaniem wskaźników:

```
1 int tab[] = {3, 15, 7, 111};
2 // Równoważne zapisy (odwołanie do pierwszego
   elementu)
3 std::cout << tab[0] << std::endl;
4 std::cout << *tab << std::endl;
5 // Równoważne zapisy (odwołanie do trzeciego
   elementu)
6 std::cout << tab[2] << std::endl;
7 std::cout << *(tab + 2) << std::endl;
```

W przedstawionym przykładzie wskaźnik (*tab*) na pierwszy element tablicy jest przesuwany o wartość 2. Zgodnie z **arytmetyką wskaźników**, **dozwolone są operacje dodawania i odejmowania wskaźników z liczbami całkowitymi** (zatem poprawna jest także operacja inkrementacji/dekrementacji). Jednakże zapis *tab + 2* **nie oznacza przesunięcia wskaźnika *tab* o 2 bajty, ale o  $2 * sizeof(int)$ !** Jest to kolejny przykład obrazujący istotność zastosowania poprawnego typu wskaźnika - **typ zmiennej przechowywanej przez wskaźnik określa rozmiar pamięci, o jaki przesuwany jest wskaźnik podczas operacji dodawania i odejmowania**. Reguła ta została schematycznie zobrazowana na rys. 2.3.





Rys. 2.3. Arytmetyka wskaźników

---

## 2.4. Wskaźnik w nagłówku funkcji

Domyślnie w języku *C*, jak i *C++*, wszystkie argumenty przekazywane do funkcji i z niej zwracane są kopiami zmiennych. Wywołując funkcję z argumentami, ich kopie odkładane są na stosie:

```
1 // Kopie x, y sa odkladane na stosie
2 void func(int x, double y) {
3     ...
4 }
5
6 int main() {
7     func(1, 2.0);
8     return 0;
9 }
```

Jeżeli argumentem funkcji będzie wskaźnik – zostanie odłożona kopia wskaźnika (wskazująca na ten sam adres), zatem funkcja będzie w stanie pracować na oryginale zmiennej. Przykład przedstawiono na listingu 7.

```
1 #include <iostream>
2
3 // Praca na kopii zmiennej
4 void increment(short val) {
5     ++val;
6 }
7
8 // Praca na oryginale zmiennej
9 void increment(short * val) {
10    ++(*val);
11 }
12
```

```

13 int main() {
14     short val = 2;
15     increment(val);
16     // Wyświetla 2
17     std::cout << val << std::endl;
18
19     increment(&val);
20     // Wyświetla 3
21     std::cout << val << std::endl;
22     return 0;
23 }

```

Listing 7. Praca na kopii i oryginale zmiennej

Analogicznie odbywa się zwracanie wartości z funkcji. Należy jednak pamiętać, aby **nie zwracać wskaźników do zmiennych lokalnych!** Zmienna lokalna zostanie usunięta po wyjściu z funkcji, a zwrócony wskaźnik będzie wskazywał na zwolniony obszar pamięci. Jest to tak zwany „*dyndający wskaźnik*” (ang. *dangling pointer*), a wszelkie operacje na takim wskaźniku skutkują *niezdefiniowanym zachowaniem*:

```

1  #include <iostream>
2
3  int * getOriginal() {
4      int val = 5;
5      // W tym miejscu usuwana jest zmienna val
6      return &val;
7  }
8
9  int main() {
10     // "Dyndający wskaźnik" ptr
11     int * ptr = getOriginal();

```

```

12
13     // Niezdefiniowane zachowanie!
14     std::cout << *ptr;
15     return 0;
16 }

```

Wskaźnik może zostać również wykorzystany do przekazania tablicy do funkcji. Należy pamiętać, że **nazwa tablicy stanowi wskaźnik na jej pierwszy element**. Jako dodatkowy parametr należy przekazać rozmiar tablicy (aby skorzystać z arytmetyki wskaźników). Zapis `tab[]` w nagłówku funkcji jest równoważny `tab *`:

```

1 #include <iostream>
2
3 // Równoważnie void printTab(const float tab[],
4   unsigned int size)
5 void printTab(const float * tab, unsigned int size)
6 {
7     for (unsigned int i = 0; i < size; ++i)
8         std::cout << tab[i] << std::endl;
9 }
10
11 int main() {
12     float tab[] = {7.3, 6.1, 9.25, 0.33};
13     printTab(tab, sizeof(tab) / sizeof(float));
14     return 0;
15 }

```

Należy pamiętać, że operator `sizeof` wywołany na tablicy zwróci jej rozmiar w bajtach (rozmiar pojedynczego elementu pomnożony przez liczbę elementów), natomiast wywołany na wskaźniku do tablicy zwróci rozmiar zmiennej wskaźnikowej (równy długości adresu w danej architekturze kom-

---

putera). Przykład przedstawiono na listingu 8.

```
1 #include <iostream>
2
3 void printTabPtrSize(const float * tab) {
4     // Wyświetli sizeof(float *), np. 8 (B) w
5     // architekturze 64-bitowej
6     std::cout << sizeof(tab) << std::endl;
7 }
8
9 int main() {
10    float tab[] = {7.3, 6.1, 9.25, 0.33};
11    // Wyświetli 4 * sizeof(float), typ. 16 (B)
12    std::cout << sizeof(tab) << std::endl;
13    printTabPtrSize(tab);
14    return 0;
15 }
```

Listing 8. Wywołanie operatora *sizeof* na tablicy i na wskaźniku do tablicy

## 2.5. Łańcuchy znakowe

*ASCII* (ang. *American Standard Code for Information Interchange*) to system kodowania znaków powszechnie stosowany w programach pisanych w języku *C* i *C++*. W swojej podstawowej wersji wykorzystuje kodowanie 7-bitowe (128 znaków), a w rozszerzonej 8-bitowe (256 znaków). Kody od 0 do 31 to tzw. **znaki sterujące** takie, jak *NUL* (0), *LF* (ang. *Line Feed* – 10) czy *ESC* (ang. *Escape* – 27). Pozostałe to tzw. **znaki drukowalne**. *Tablicę ASCII* w wersji 7-bitowej przedstawiono na rys. 2.4.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Rys. 2.4. 7-bitowa tablica ASCII [1]

W językach *C* i *C++* typem zmiennej przechowującej znaki zakodowane w standardzie **ASCII** jest **char**. Jest to typ 8-bitowy (1 B), który zakłada bezpośrednią konwersję znaku na jego reprezentację dziesiętną w kodzie ASCII, np. 'A' i 65:

---

```
1 char sign = 'A';
2 char signByNum = 65;
3 // Wyświetla AA
4 printf("%c%c\n", sign, signByNum);
```

Warto zwrócić uwagę, że litery małe są przesunięte o wartość +32 względem liter wielkich oraz, że w przeciwieństwie do napisów, znaki ASCII umieszczane są w pojedynczych cudzysłowach:

```
1 // Wyświetla 32
2 printf("%d\n", 'b' - 'B');
3 printf("%d\n", 'w' - 'W');
```

Wykorzystując tę wiedzę można przypisać łańcuch znakowy (literał łańcuchowy, napis) do tablicy zmiennych typu *char* i przekazać ją do funkcji wejścia:

```
1 char text[] = "Ala ma kota";
2 printf("%s\n", text);
```

Po uruchomieniu programu zawierającego powyższy kod, na ekranie komputera zostanie wyświetlony napis **"Ala ma kota"**. Wiadomym jest, że przekazanie zmiennej tablicowej *text* do funkcji jest równoznaczne z przekazaniem **wskaźnika na pierwszy element tablicy**, zatem wskaźnika typu *char \** na znak 'A'. Nie jest przekazywana jednak nigdzie informacja o rozmiarze tablicy. Skąd zatem funkcja *printf()* „wie”, w którym miejscu kończy się napis? Pewną sugestią może być następujący kod:

---

```
1 char text[] = "Ala ma kota";
2 // Wyświetla 12
3 printf("%lu\n", sizeof(text));
4 // Wyświetla 11
5 printf("%lu\n", strlen(text));
```

Operator ***sizeof*** zwraca całkowity rozmiar pamięci zajmowanej przez tablicę (w bajtach). Funkcja ***strlen()***, zadeklarowana w nagłówku ***string.h*** (*cstring*), oblicza długość łańcucha znakowego. Licząc znaki występujące w napisie "Ala ma kota" otrzymamy wynik 11 (włączając znaki spacji). Czyli dokładnie tyle, ile zwróciła funkcja ***strlen()***. Różnica jednego bajta pomiędzy wynikiem operatora ***sizeof*** i funkcji ***strlen()*** sugeruje, że na końcu łańcucha znajduje się dodatkowy, niewyświetlany, automatycznie umieszczany przez kompilator znak. I tak jest w rzeczywistości. **Każdy literał łańcuchowy w języku C/C++ jest zakończony symbolem '\0'**. Zarówno funkcja ***printf()***, jak i ***strlen()***, iterują po kolejnych elementach tablicy (wykorzystując zasady arytmetyki wskaźników), aż do napotkania (na ostatniej pozycji tablicy) symbolu ***\0***, o kodzie ASCII równym 0 (natomiast cyfra 0 ma kod 48). Jeżeli ręcznie podmienimy (**operacja przeprowadzona tylko na potrzeby tego przykładu!**) znak ***\0*** na inny znak z tablicy ASCII, to funkcja ***printf()*** będzie wypisywać znaki kodowane przez (najczęściej losowe) liczby znajdujące się w kolejnych komórkach pamięci leżących za tablicą, aż do natrafienia na następny symbol ***\0***:



---

```

1 #include <cstring>
2 #include <iostream>
3
4 int main() {
5     // Tablica zawiera 12 znakow na pozycjach 0...11
6     char text[] = "Ala ma kota";
7     // Podmienienie symbolu \0 na 11 pozycji tablicy
8     text[11] = '!';
9     // Wyświetla np. "Ala ma kota!?.@"
10    printf("%s\n", text);
11    return 0;
12 }

```

Domyślnym typem zmiennych przechowujących łańcuchy znaków w języku *C* jest *char \**, a w języku *C++* *const char \**. Wydaje się to logiczne, mając na uwadze, że wskaźnik na typ *char* w przypadku tablic znaków jest wskaźnikiem na konkretny element tej tablicy. Napisy przechowywane są w segmencie pamięci *text*. Kompilator mając na uwadze, że segment *text* stanowi **pamięć tylko do odczytu** może, w celu optymalizacji kodu, wykorzystywać ten sam napis w różnych miejscach programu. Z tego względu **nie powinno się modyfikować wartości łańcuchów znakowych!** Może to doprowadzić do niezdefiniowanego zachowania programu w niespodziewanych fragmentach kodu. Aby temu zapobiec należy posługiwać się wskaźnikami o **niemodyfikowalnej wartości wskazywanej zmiennej** – czyli stosować specyfikator *const*. W języku *C++* takie wskaźniki z definicji reprezentują łańcuchy znakowe:

```

1 const char * text = "Hello, world!";
2 std::cout << text << std::endl;

```

---

Ponieważ typem zmiennej przechowującej napis w językach *C* i *C++* jest zmienna wskaźnikowa, nie można porównywać dwóch napisów za pomocą operatora `==`:

```
1 // Porównanie adresów dwóch wskaźników -  
   niezdefiniowane zachowanie  
2 if ("Ala ma kota" == "Ala ma kota")
```

Aby poprawnie porównać dwa napisy, należy posłużyć się funkcją ***strncmp()***, zadeklarowaną w nagłówku ***string.h*** (w *C++* ***std::strncmp()***), zadeklarowaną w nagłówku ***cstring***). Funkcja ta przyjmuje jako argumenty dwa łańcuchy znakowe oraz maksymalną liczbę następujących po sobie znaków, które mają zostać porównane. Funkcja zwraca **0**, jeśli napisy są sobie równe, **liczbę dodatnią** – jeżeli znaki pierwszego napisu występują alfabetycznie po znakach napisu drugiego, albo **liczbę ujemną** – w przeciwnym wypadku. Przykład zastosowania funkcji ***std::strncmp()*** do poprawnego porównania literałów łańcuchowych przedstawiono na listingu 9.

```
1 #include <cstring>  
2 #include <iostream>  
3  
4 int main() {  
5     const char * text = "Ala ma kota";  
6     if (std::strncmp(text, "Ala ma kota", strlen(  
7         text)) == 0)  
8         std::cout << "Texts are equal";  
9     return 0;  
}
```

Listing 9. Poprawne porównanie napisów z wykorzystaniem funkcji ***std::strncmp()***

---

Znając naturę łańcuchów znakowych w językach *C* i *C++*, warto poświęcić uwagę kwestii tzw. **sekwencji ucieczki** (*ang. escape sequences*). Jest to sekwencja znaków, wykorzystywana do wyświetlania znaków specjalnych lub znaków, które w innych warunkach mają odmienne zastosowanie. Omówione zostały już sekwencje takie, jak `\r`, `\n`, `\0`. Innymi, często wykorzystywanymi sekwencjami są:

- `\'` – pojedynczy cudzysłów
- `\"` – podwójny cudzysłów
- `\\` – ukośnik
- `\t` – poziomy znak tabulacji
- `\v` – pionowy znak tabulacji

## 2.6. Przekazywanie argumentów do funkcji *main()*

Typowy nagłówek funkcji *main()* wymaga, żeby funkcja nie przyjmowała żadnych argumentów. Istnieje jednakże inna sygnatura funkcji, umożliwiająca wywołanie funkcji *main()* z argumentami przekazanymi podczas uruchamiania programu:

```
1 int main(int argc, char ** argv);
```

Argument *argv* stanowi **wskaźnik na wskaźnik na typ *char*** i może być równoważnie rozumiany jako tablica łańcuchów znakowych:

```
1 int main(int argc, char * argv[]);
```

Argument *argc* określa ile elementów zawiera tablica *argv*. Pierwszym elementem tablicy (*argv[0]*) **jest zawsze nazwa aplikacji**. Pozostałe elementy to argumenty, z którymi został wywołany program. Wyrażenie (*argv[argc]*)

---

ma wartość ***NULL***. Przykład obsługi parametrów wywołania programu przedstawiono na listingu 10. Program **myApp.exe** wywołano z poziomu konsoli systemowej w następujący sposób:

***myApp.exe "Hello, world!" "Goodbye, cruel world!"***

```
1 #include <iostream>
2
3 int main(int argc, char ** argv) {
4     // Sprawdź, czy do programu przekazano argumenty
5     if (argc > 1) {
6         // Wyświetl nazwę programu (myApp)
7         std::cout << "First arg: " << argv[0] << std::
8         endl;
9
10        // Wyświetl pozostałe argumenty (Hello, world!
11        Goodbye, cruel world!)
12        for (int i = 1; i < argc; ++i)
13            std::cout << "Next arg: " << argv[i] << std::
14            endl;
15    }
16    return 0;
17 }
```

Listing 10. Obsługa parametrów wywołania programu

---

### 3. Program ćwiczenia

**Zadanie 1.** W pliku nagłówkowym *tabUtils.h* zawarto deklaracje dwóch funkcji:

- *void printTab(const int \* const tab, unsigned int size)* – przesyłającej na standardowe wyjście wartości wszystkich elementów tablicy *tab* o rozmiarze *size*
- *void reverseTab(int \* const tab, unsigned int size)* – odwracającej (modyfikującej) kolejność elementów (pierwszy  $\longleftrightarrow$  ostatni, drugi  $\longleftrightarrow$  przedostatni, ...) w tablicy *tab* o rozmiarze *size*

W ramach zadania:

1. korzystając z dowolnej pętli, napisz definicje funkcji *printTab()* oraz *reverseTab()*; definicje umieść w pliku źródłowym *tabUtils.cpp*
2. wewnątrz funkcji *main()* (plik *main.cpp*) utwórz (lokalnie) tablicę kilkunastu liczb całkowitych, a następnie zainicjalizuj ją losowymi wartościami, wykorzystując funkcję *rand()* [patrz: Ćw. 2]
3. wykorzystując funkcję *reverseTab()* odwróć kolejność utworzonej tablicy
4. wykorzystując funkcję *printTab()* wyświetl zawartość tablicy przed i po odwróceniu kolejności elementów

*Uwaga: funkcje **printTab()** oraz **reverseTab()** powinny obsługiwać tablice o dowolnym rozmiarze. Funkcja **reverseTab()** modyfikuje oryginał tablicy.*

**Zadanie 2.** W pliku nagłówkowym *convertCase.h* zawarto deklarację funkcji *void convertCase(char text[])*. Funkcja ta przyjmuje tablicę znaków, a następnie, dla każdej kolejnej wczytanej litery, zmienia jej wielkość ('a'  $\longrightarrow$  'A', 'F'  $\longrightarrow$  'f', itp.). W ramach zadania:

- 
1. napisz definicję funkcji ***convertCase()*** i umieść ją w pliku źródłowym ***convertCase.cpp***. Uwaga: funkcja zmienia wielkość liter, ale ignoruje pozostałe znaki, jakie mogą znajdować się w tablicy, np. *'?' → '?'*
  2. wewnątrz funkcji ***main()*** (plik ***main.cpp***) pobierz z klawiatury kilkanaście różnych znaków ASCII (w tym różnej wielkości litery oraz znaki niealfanumeryczne) i zainicjalizuj nimi lokalną tablicę zmiennych typu *char*
  3. wywołaj funkcję ***convertCase()***, jako argument przekaz zainicjalizowaną tablicę
  4. wyświetl zawartość zmodyfikowanej tablicy na ekranie

*Uwaga: funkcja ***convertCase()*** powinna obsługiwać tablice o dowolnym rozmiarze.*

**Zadanie 3.** W pliku nagłówkowym ***calculator.h*** zawarto deklaracje pięciu funkcji:

- ***float add(float x, float y)*** – wykonującej dodawanie dwóch liczb zmiennoprzecinkowych *x, y*
- ***float subtract(float x, float y)*** – wykonującej odejmowanie dwóch liczb zmiennoprzecinkowych *x, y*
- ***float multiply(float x, float y)*** – wykonującej mnożenie dwóch liczb zmiennoprzecinkowych *x, y*
- ***float divide(float x, float y)*** – wykonującej dzielenie dwóch liczb zmiennoprzecinkowych *x, y*
- ***float calculate(float x, float y, char \* operations[], unsigned int size)*** – wykonującej podstawowe operacje arytmetyczne na dwóch liczbach zmiennoprzecinkowych *x, y*, zgodnie z kolejnością operacji określonych w tablicy ***operations*** o rozmiarze ***size***

---

W ramach zadania:

1. napisz definicje funkcji ***add()***, ***subtract()***, ***multiply()*** oraz ***divide()*** i umieść je w pliku źródłowym ***calculator.cpp***
2. napisz definicję funkcji ***calculate()*** i umieść ją w pliku źródłowym ***calculator.cpp***. Funkcja sprawdza kolejne elementy tablicy ***operations*** i, w zależności od ich wartości, wykonuje odpowiednie działanie na liczbach ***x***, ***y*** (wywołując odpowiednio funkcję ***add()***, ***subtract()***, ***multiply()*** albo ***divide()***):

- "add" –  $x + y$
- "sub" –  $x - y$
- "mul" –  $x * y$
- "div" –  $x / y$
- w każdym innym przypadku ignoruje polecenie

Wartość zwracana przez funkcję ***calculate()*** stanowi sumę wyników wszystkich operacji pośrednich

3. wewnątrz funkcji ***int main(int argc, char \*\* argv)*** (plik ***main.cpp***) wywołaj funkcję ***calculate()***, jako argumenty przekaz parametry, z którymi została uruchomiona aplikacja oraz dwie liczby zmiennoprzecinkowe pobrane z klawiatury
4. wyświetl wynik funkcji ***calculate()*** na ekranie komputera

*Przykładowy rezultat uruchomienia aplikacji (z klawiatury pobrano liczby 2.0 i 10.0:*

***calculator.exe "add" "add" "mul" "sub" "div"***

$$36.2 = (2.0 + 10.0) + (2.0 + 10.0) + (2.0 * 10.0) + (2.0 - 10.0) + (2.0 / 10.0)$$

---

## 4. Dodatek

### 4.1. Modyfikacja „stałej” z wykorzystaniem wskaźnika

**Stałe** w języku *C*, w przeciwieństwie do języka *C++*, to zmienne (zdeklarowane z użyciem słowa kluczowego **const**), dla których kompilator pilnuje, aby nie doszło do zmiany wartości. Możliwa jest jednak modyfikacja stałych w języku *C* przez bezpośrednie odwołanie do adresu pamięci. Realizowane jest to z wykorzystaniem wskaźnika. Przykład modyfikacji wartości stałej w języku *C* przedstawiono na listingu 11. W języku *C++* stałe to wyrażenia, których wartość nie może zostać zmodyfikowana. Poniższy kod, skompilowany kompilatorem *g++*, powoduje wyświetlenie wartości 2.0.

```
1 const double x = 2.0;
2 // Rzutowanie (const double *) na (double *)
3 double * ptr = (double *)&x;
4 *ptr = 5.5;
5 // Wyświetla 5.5
6 printf("%f\n", x);
```

Listing 11. Modyfikacja wartości stałej w języku *C*, z wykorzystaniem wskaźnika

## Literatura

- [1] *W świecie komputerowego tekstu*. URL: <https://www.dobreprogramy.pl/Ryan/W-swiecie-komputerowego-tekstu,20883.html>.