

# Podstawy Programowania

## Wykład nr 7: Wprowadzenie do wskaźników

dr hab. inż. Dariusz Dereniowski

Katedra Algorytmów i Modelowania Systemów  
Wydział ETI, Politechnika Gdańska

## Czym jest wskaźnik?

- Z punktu widzenia programisty, pamięć w komputerze jest zorganizowana w postaci jednego bloku pamięci, w którym komórki indeksowane są liczbami naturalnymi.
- Każda zmienna, w trakcie działania programu, zajmuje określoną liczbę komórek w pamięci operacyjnej.
- Wyjątek od powyższej zasady jest możliwy (aczkolwiek niezauważalny dla programisty): jeśli w programie nie następuje jawne odwołanie do adresu zmiennej, to zmienna ta, jeśli tak postanowi kompilator, może być przechowywana wyłącznie w rejestrach procesora.
- **Wskaźnik** jest zmienną, której wartość jest adresem w pamięci operacyjnej. Deklaracja zmiennej wskaźnikowej jest następująca:

```
void *w;
```

- Ze wskaźnikiem możemy stowarzyszyć typ, który nazywamy **typem bazowym wskaźnika**: informujemy w ten sposób kompilator jakiego typu zmiennej powinien się spodziewać pod adresem, który przechowuje wskaźnik. Wówczas deklaracja przybiera postać:

```
typ *w;
```

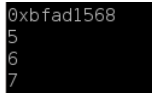
## Operator adresu & oraz operator dostępu \*

- Jednoargumentowy operator & zwraca adres zmiennej.
- Jednoargumentowy operator \* zwraca wartość zmiennej, która znajduje się pod wskazanym adresem. Operatorem tego używamy w odniesieniu do wskaźników posiadających typ bazowy. Deklaracja `typ *w;` pozwala kompilatorowi stwierdzić ile bajtów pod adresem `w` składa się na wskazywaną zmienną i jak te bajty interpretować.

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, *w;
    w = &a;

    cout << w << endl; // Wypisujemy adres zmiennej a. Zauważ, że
                        // powiedzieliśmy kompilatorowi, że w wskazuje na int.
    cout << *w << endl; // Wypisujemy wartość zmiennej a.
    a++;
    cout << *w << endl; // Wartość zmiennej a uległa zmianie, a zmienna.
                        // Zmienna w cały czas wskazuje na zmienną a.
    (*w)++;             // Modyfikujemy wartość zmiennej wskazywanej przez w.
    cout << a << endl;  // Powyższa instrukcja zmieniła wartość zmiennej a!
    return 0;
}
```



0xbfad1568  
5  
6  
7

## Arytmetyka wskaźnikowa

- Na zmiennych wskaźnikowych można wykonywać przypisania, konwersje typów i operacje arytmetyczne, których wynikiem jest inny wskaźnik: **wskaźnik + liczba**, **wskaźnik - liczba**, **wskaźnik++**, **wskaźnik--**.
- Wartość adresu w pamięci, o którą zostanie przesunięty wskaźnik w wyniku dodawania/odejmowania 1, jest równy rozmiarowi typu bazowego wskaźnika (przykładowo, dla **int\*** będzie to `sizeof(int)`, czyli np. 4!)

```
#include <iostream>
using namespace std;

#define ROZMIAR 10

int main() {
    int t[ROZMIAR] = { 3, 4, 5, 6 }; // Pozostałe elementy uzupełniane zerami.
    int *w = &(t[0]);               // w przechowuje adres pierwszego elementu tablicy t

    // Zauważ, że jeśli w zawiera adres pierwszego elementu tablicy t, to
    // w+1 jest adresem drugiego elementu, w+2 jest adresem trzeciego, itd.
    for ( int i=0; i < ROZMIAR; i++ )
        cout << *(w++) << ' ';
    cout << endl;
    return 0;
}
```

3,4,5,6,0,0,0,0,0,0,

## Wskaźniki a tablice

- W języku C nazwa tablicy jest jednocześnie adresem elementu początkowego (zerowego) tablicy. Nazwa tablicy może więc być traktowana jako wskaźnik.
- Konsekwencja: dla tablicy  $X$ , zapis  $X + i$  oznacza adres  $i$ -tego elementu tablicy (licząc od 0).

Następujące dwa wyrażenia są równoważne:  $X + 3$  i  $\&X[3]$ . (Ze względu na priorytety operatorów  $\&X[3]$  jest równoważne  $\&(X[3])$ .)

Następujące dwa wyrażenia są równoważne:  $*(X + 3)$  i  $X[3]$ .

Następujące dwa wyrażenia są równoważne:  $*X + 3$  i  $X[0] + 3$ .

- W związku z powyższymi przykładami, należy uważać na **priorytety** operatorów.

```
char a[] = "hello";  
char *p = "world";
```

a: 

h	e	l	l	o	\0
---	---	---	---	---	----

p: 

●
---

 → 

w	o	r	l	d	\0
---	---	---	---	---	----

## Wskaźniki a parametry do funkcji

- Następujący program spowoduje wypisanie liczby 0 na ekran.
- Dzieje się tak dlatego, że funkcja `f` otrzymuje na wejściu kopię wskaźnika `&a`. (Zgodnie z ogólną regułą, do funkcji w języku C zawsze przekazywana jest kopia argumentu.)
- Kopia powyższego adresu (czyli zmienna `w`) przechowuje ten sam adres co `&a`, a w konsekwencji instrukcja `*w = 0` poleca umieścić liczbę 0 pod adresem przechowywanym w zmiennej `w`, a więc poleca umieścić liczbę 0 w zmiennej `a`.
- Powyższego mechanizmu można użyć w sytuacji, aby uzyskać efekt modyfikacji argumentów przez funkcję.

```
#include <iostream>
using namespace std;

void f( int *w ) {
    *w = 0;
}

int main() {
    int a = 5;
    f( &a );
    cout << a;
    return 0;
}
```

## Manipulacja napisami – przykłady

W języku C nie ma instrukcji pozwalającej na kopiowanie tablic, w szczególności tablic znaków. Aby wykonać kopię napisu, niezbędne jest jego kopiowanie “znak po znaku”.

```
#include <iostream>
using namespace std;

/* Funkcja nie sprawdza czy tablica cel jest wystarczająco pojemna, aby
przechować cały napis (wszystkie znaki aż do wystąpienia znaku '\0')
z tablicy zrodlo */
void kopiuj_napis( const char zrodlo[], char cel[] ) {
    int i;
    for ( i=0; zrodlo[i] != '\0'; i++ )
        cel[i] = zrodlo[i];
    cel[i] = '\0'; // Nie powinniśmy zapominać o '\0' kończącym napis!
}

int main() {
    char c1[] = "Wskaźniki są fajne i bardzo przydatne!";
    char c2[100];

    kopiuj_napis( c1, c2 );
    return 0;
}
```

## Manipulacja napisami – przykłady

Deklaracje `typ *w` oraz `typ x[]` są niemalże równoważne:

- `w[i]` oraz `x[i]` są i-tymi obiektami typu `typ` od miejsca wskazywanego przez, odpowiednio, `w` oraz `x`.
- Modyfikacja zawartości `x` nie jest możliwa (tzn. instrukcja `x++` spowoduje błąd kompilacji), natomiast modyfikacja `w` jest możliwa.

Poprzedni program możemy zapisać następująco:

```
#include <iostream>
using namespace std;

/* Funkcja nie sprawdza czy tablica cel jest wystarczająco pojemna, aby
przechować cały napis (wszystkie znaki aż do wystąpienia znaku '\0')
z tablicy zrodlo */
void kopiuj_napis( const char *zrodlo , char *cel ) {
    while ( *zrodlo != '\0' )
        *(cel++) = *(zrodlo++);
    *(cel) = '\0'; // Nie powinniśmy zapominać o '\0' kończącym napis!
}

int main() {
    char c1[] = "Wskaźniki sa fajne i bardzo przydatne!";
    char c2[100];

    kopiuj_napis( c1 , c2 );
    return 0;
}
```



## Stałe wskaźniki a wskaźniki na stałą

- Deklaracja:

```
const int *w;
```

powoduje utworzenie wskaźnika `w`, który można używać jedynie do odczytu elementów, na które wskazuje, natomiast nie można użyć `w` do modyfikacji tych elementów. Na przykład, instrukcje `*w=0` lub `w[3]++` spowodują błąd kompilacji.

- Deklaracja:

```
int *const w;
```

powoduje utworzenie stałego wskaźnika. Oznacza to, że można użyć zmiennej `w` do odczytu i modyfikacji wartości wskazywanych przez `w`, lecz nie można zmieniać wartości zmiennej `w` (tzn. nie można zmieniać adresu, który `w` przechowuje). Innymi słowy, `w` wskazuje przez cały czas swojego życia na to samo miejsce w pamięci operacyjnej. Tego typu wskaźniki trzeba inicjalizować (w szczególności, mogą być używane jako parametry funkcji).

- Deklaracja:

```
const int *const w;
```

łączy obie powyższe cechy.

## Stałe wskaźniki a wskaźniki na stałą – przykłady

```
int main() {  
    int t[] = { 1, 2 };  
    const int *w;  
    int *const p; // Błąd kompilacji! Ten wskaźnik musi być zainicjalizowany.  
    int *const x=t;  
    const int *const y = x+1;  
    w = t; // Dobrze.  
    w++; // Dobrze.  
    w[0]=0; // Błąd kompilacji! Nie można modyfikować wart. wskazywanych przez w.  
    *w--; // Dobrze, gdyż równoważne: *(w--).  
    (*w)--; // Błąd kompilacji! Próba dekrementacji wart. wskazywanej przez w.  
    x[1] = 3; // Dobrze.  
    x = x+1; // Błąd kompilacji. Nie można zmienić adresu zapisanego w zmiennej x.  
    *t = y[0]; // Dobrze.  
    return 0;  
}
```

## Wybrane funkcje z biblioteki string.h

- `size_t strlen( const char *s );`

Funkcja zwraca długość napisu `s`, tzn. liczbę znaków począwszy od miejsca wskazywanego przez `s` aż do pierwszego wystąpienia znaku `'\0'`. `size_t` jest typem całkowitoliczbowym zdefiniowanym w bibliotece standardowej i jest on wystarczająco pojemny, aby przechowywać rozmiary bloków pamięci na danym komputerze. Jeśli znamy górne ograniczenie na długość napisu, to możemy rzutować wynik zwracany przez funkcję do typów całkowitoliczbowych jak `short`, `int`, `long` itp.

- `int strcmp( const char *s1, const char *s2 );`

Funkcja porównuje dwa napisy i zwraca 0 jeśli są identyczne (porównywanie dokonywane jest do napotkania znaku `'\0'`), zwraca liczbę mniejszą niż 0 jeśli `s1` jest mniejszy (alfabetycznie) od `s2`, oraz zwraca liczbę większą niż 0 gdy `s1` jest większy niż `s2`.

- `char *strcpy( char *dest, const char *src );`

Funkcja kopiuje napis wskazywany przez `src` do obszaru pamięci (tablicy) wskazywanego przez `dest`. Programista musi zadbać o to, aby tablica `dest` była wystarczająco pojemna.

**Uwaga:** zwróć uwagę na kwalifikatory `const`. Ich użycie jest przejawem dobrego stylu programowania, gdyż programista używający tych funkcji wie, że nie mogą one dokonać zmian w napisach `s`, `s1`, `s2` i `src`. **Zadanie:** napisz powyższe funkcje samodzielnie. Także, zapoznaj się z funkcjami `strncmp` i `strncpy` i także napisz je samodzielnie.

## Parametry do funkcji – tablice wielowymiarowe

W deklaracji funkcji należy wyspecyfikować:

- typ elementów przekazywanej tablicy,
- wszystkie wymiary tablicy, poza najbardziej zewnętrznym (lewym).

Poniższe deklaracje funkcji są równoważne:

```
void funkcja( int tablica[][3] );
```

```
void funkcja( int (*tablica)[3] );
```

Przykład poprawnego wywołania powyższych funkcji:

```
int t[10][3];  
funkcja( t );
```

## Stała NULL

- Czasem zachodzi potrzeba przypisania do zmiennej wskaźnikowej wartości której znaczenie jest takie, że zmienna ta nie wskazuje na żaden obszar pamięci.
- Do powyższego celu służy stała `NULL` (zdefiniowana w `stdlib.h`).
- `NULL` jest stałą typu `void *` a zatem można ją rzutować na każdy typ wskaźnikowy.
- W miejsce `NULL` można użyć `0` lub `(void *)0`, lecz używanie `NULL` jest dobrą praktyką, gdyż sprawia, że kod jest łatwiejszy w czytaniu.

## Dynamiczna alokacja pamięci

`stdlib.h` dostarcza dwóch następujących funkcji:

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

Funkcja ta przydziela programowi blok pamięci o rozmiarze `size` bajtów i zwraca adres początku tego bloku. Funkcja ta odwołuje się do systemowych mechanizmów przydziału pamięci. Zwraca `NULL` w przypadku niepowodzenia (np. brak wystarczającej ilości wolnej pamięci). Zauważmy, że funkcja zwraca wskaźnik typu `void *`. Od nas zależy jakiego typu elementy chcemy w przydzielonym bloku przechowywać i dokonujemy odpowiedniego rzutowania (patrz kolejny przykład).

```
void free( void *ptr );
```

Funkcja zwalnia pamięć przydzieloną wcześniej przez `malloc` (lub inną funkcję przydziału pamięci dostępną w C). Jeśli przekazujemy wskaźnik do bloku pamięci zwolnionej wcześniej, zachowanie funkcji `free` jest niezdefiniowane (taka sytuacja jest błędem logicznym programisty). Jeśli `free` otrzyma na wejściu wskaźnik `NULL`, to żadna akcja nie jest wykonywana.

## Dynamiczna alokacja pamięci – typowy schemat postępowania

- Założmy, że potrzebujemy zaalokować blok pamięci, w którym będziemy przechowywali `n` obiektów typu `typ`.
- Deklarujemy najpierw wskaźnik, w którym będziemy przechowywali adres początku tego bloku: `typ *blok = NULL;`
- Inicjalizacja wskaźnika wartością `NULL` nie jest konieczna.
- Używamy funkcji `malloc` aby przydzielić pamięć:  
`blok = (typ *) malloc( n*sizeof(typ) );`
- Zauważmy, że zamiast szacować ile bajtów pamięci operacyjnej zajmuje obiekt typu `typ`, pozostawiamy tą pracę operatorowi `sizeof` – w ten sposób nigdy się nie pomylimy i przydzielony blok ma właściwy rozmiar. Zwróć uwagę na rzutowanie z `void *` (typ zwracany przez `malloc`) do typu naszego wskaźnika.
- Warto sprawdzić czy blok pamięci został faktycznie przydzielony:  
`if ( blok == NULL ) { tutaj obsługa sytuacji braku pamięci }`
- Gdy pamięć nie jest już więcej potrzebna, wywołujemy:  
`free( blok );`

Powyższy mechanizm jest zilustrowany na kolejnym slajdzie, gdzie pokazany jest program wczytujący od użytkownika ciąg liczb i wypisujący te liczby w odwrotnej kolejności. Z góry nie wiemy ile liczb użytkownik chce wpisać (to użytkownik podaje na początku programu), więc musimy przydzielać pamięć w trakcie działania programu.

## Dynamiczna alokacja pamięci – przykład

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void wczytaj( int t[], int n ) {
    for ( int i=0; i < n; i++ ) {
        cout << "Podaj " << i << "-ty element: ";
        cin >> t[i];
    }
}

void wypisz_odwrotnie( const int *t, int n ) {
    for ( int i=n-1; i >= 0; i-- )
        cout << "t[" << i << "] = " << t[i] << endl;
}

int main() {
    int n, *p = NULL;
    cin >> n; // Wczytujemy liczbę elementów od użytkownika.
    p = (int *) malloc( n*sizeof(int) );
    if ( p == NULL ) { // Koniecznie sprawdzamy, czy system operacyjny
        cout << "Brak pamięci!\n"; // faktycznie przydzielił żadaną
        return 1;                // pamięć. Jeśli nie – kończymy program.
    }
    wczytaj( p, n );
    wypisz_odwrotnie( p, n );
    free( p ); // Pamiętamy o zwolnieniu pamięci – to dobry nawyk.
    return 0;
}
```

```
5
Podaj 0-ty element: 1
Podaj 1-ty element: 3
Podaj 2-ty element: 5
Podaj 3-ty element: 7
Podaj 4-ty element: 9
t[4] = 9
t[3] = 7
t[2] = 5
t[1] = 3
t[0] = 1
```

```
10000000000
Brak pamięci!
```



## Parametry linii poleceń

- Poniższy program ilustruje pewien mechanizm pozwalający na komunikację programisty z "otoczeniem".
- Mianowicie, zadeklarowanie funkcji `main` z takimi parametrami, pozwala na dostęp do parametrów (`argv`) oraz ich liczby (`argc`) podawanych z linii poleceń przy uruchomieniu programu.
- `argv[i]` jest napisem będącym i-tym spośród tych parametrów (numerujemy jak zawsze od zera). Poprawne wartości indeksu `i` są z zakresu od `0` do `argc-1`.
- `argv[0]` jest zawsze nazwą pliku wykonywalnego z nazwą programu.

```
deren@linux-nb6o:~> ./w07p08 -o -a -c ała ma kota
7
0-ty argument to: ./w07p08
1-ty argument to: -o
2-ty argument to: -a
3-ty argument to: -c
4-ty argument to: ała
5-ty argument to: ma
6-ty argument to: kota
```

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    cout << argc << endl;
    for ( int i=0; i < argc; i++ )
        cout << i << "-ty argument to: " << argv[i] << endl;
    return 0;
}
```