Politechnika Wrocławska

Bardzo szybkie podsumowanie: wykład 3

wer. 7 z drobnymi modyfikacjami!

Wojciech Myszka

2019-04-02 09:08:04 +0200

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska

Uwagi

1. Obowiązuje cały materiał!
2. Tu tylko podsumowanie.

Wskaźniki. Pamięć dynamiczna

wer. 10

Wojciech Myszka

Katedra Mechaniki i Inżynierii Materiałowej

2019-03-27 08:28:44 +0100

Literatura I

Ted Jensen.
A tutorial on pointers and arrays in C, Feb. 2000.
Dostępne jako http://pweb.netcom.com/~tjensen/ptr/pointers.htm.

Ben Klemens.
21st Century C.
O'Reilly Media, 2012.
http://shop.oreilly.com/product/0636920025108.do.

Literatura II

Richard Reese.
Wskaźniki w języku C: przewodnik.
Helion, Gliwice, 2014.
Dostęp po zalogowaniu w bazie NASBI.
http://biblioteka.pwr.wroc.pl/NASBI_Naukowa_Akademicka_Sieciowa_Biblioteka_Internetowa,161.dhtml.

Richard M Reese.
Understanding and Using C Pointers.
O'Reilly Media, 2013.

Wskaźniki

- Wskaźniki to podstawa C.
- Kto nie umie się nimi posługiwać — ten nie potrafi wykorzystać siły języka C.
- C używa wskaźników bardzo często. Czemu?
  - Bardzo trudno zaprogramować beż nich pewne operacje.
  - Pozwalają na tworzenie zwartego i efektywnego kodu.
  - Są bardzo efektywnym narzędziem.
- Korzystając z:
  - tablic,
  - struktur (będzie później),
  - funkcji

  korzystamy ze wskaźników.
- Wskaźniki to — najprawdopodobniej — najtrudniejszy fragment języka C. Nie przydadzą się tu doświadczenia uzyskane podczas programowania w innych językach.

Wskaźniki

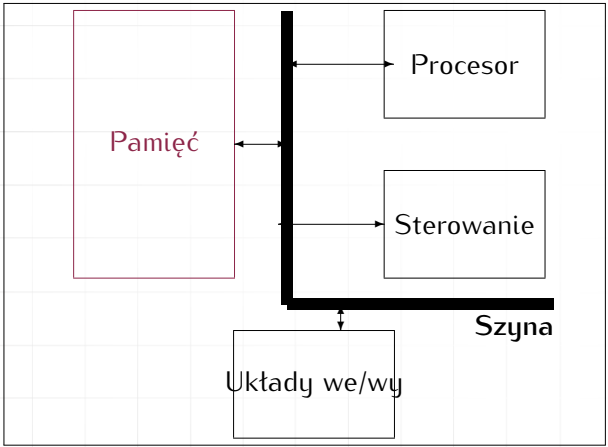
Co to jest?

Wskaźnik

to specjalna zmienna zawierająca „adres” (w pamięci) innej zmiennej.

Wskaźniki

Pamięć komputera



Pamięć komputera

1. RAM (Random Access Memory — pamięć o dostępie swobodnym).
2. ROM (Red Only Memory — pamięć tylko do odczytu).
3. Pamięć dyskowa.
4. Pamięci „zewnętrzne” (dyski USB, dyski sieciowe, . . .)

My (teraz) zajmować się będziemy tylko pamięcią RAM komputera, choć bardzo często pamieć dyskowa jest swojego rodzaju przedłużeniem pamięci RAM („systemowy plik wymiany” czy *swap*).

Pamięć operacyjna

1. Można ją przedstawić w postaci następującej:

adres	0	1	2	3	4	5	6	7	8	...	...
zawartość	·	·	·	·	·	·	·	·	·	·	·

2. Pamięć praktycznie wszystkich komputerów ma organizację **bajtową**. (Najmniejszą adresowalną jednostką pamięci jest bajt.)
3. Zmienne różnych typów zajmują w pamięci różną liczbę bajtów.
4. W chwili deklarowania zmiennych kompilator przydziela im w pamięci miejsce.
5. Każdy program uruchamiany jest w „maszynie wirtualnej" i ma dostęp tylko do przydzielonej mu pamięci.

Zmienne a pamięć operacyjna I

1. Zmienne różnych typów zajmują w pamięci różną liczbę bajtów.
2. W chwili deklarowania zmiennych kompilator przydziela im w pamięci miejsce.
3. Używając nazwy zmiennej (w jakiś) sposób wskazujemy miejsce w pamięci operacyjnej

   ```
   a = b + 3;
   ```

   należy czytać „pobierz zawartość pamięci operacyjnej przydzielonej zmiennej b, dodaj do niej 3, a wynik zapisz w miejscu pamięci operacyjnej przydzielonym zmiennej a”.
4. . . . ale (przy takim zapisie) nie mamy żadnego dostępu do adresu zmiennej!

Zmienne a pamięć operacyjna II

5. Należy natomiast pamiętać, że polecenie:

```
a = b;
```

powoduje **przekopiowanie** zawartości zmiennej b do zmiennej a.
a i b to dwa różne obiekty!

Pytanie jest takie: Czy jest nam potrzebny (i do czego) adres zmiennej?

Typy danych i zajętość pamięci

```
sizeof(char   ) = 1
sizeof(short  ) = 2
sizeof(int    ) = 4
sizeof(long   ) = 8
sizeof(float  ) = 4
sizeof(double) = 8
sizeof(long double) = 16
```

Wskaźniki I

1. Wskaźnik, w języku C to (w pewnym uproszczeniu) adres zmiennej w pamięci operacyjnej.
2. Wskaźniki, tak jak wszystko, muszą być deklarowane.
3. Deklaracja wskaźnika jest „dziwna”:

```
int *ip;
  /* ip jest wskaznikiem do obiektu typu int */
double *fp;
  /* fp jest wskaznikiem do obiektu typu double */
char* cp // wskaznik do typu char
float*Fp // wskaznik do typu float
```

mówi jakiego typu jest zmienna na którą wskaźnik wskazuje.

Wskaźniki II

4. Podstawowe operacje na wskaźnikach to
   - & pobranie adresu zmiennej na którą ma wskazywać wskaźnik.
   - $*$ pobranie zawartości zmiennej wskazywanej przez wskaźnik (gdy występuje po prawej stronie znaku równości) lub nadanie wartości zmiennej wskazywanej przez wskaźnik (gdy występuje po lewej stronie znaku równości).
   - $-$ Odejmowanie wskaźników. Wynikiem jest liczba calkowita długa.
   - $+$ Dodanie do wskaźnika stałej/zmiennej (całkowitej).
   - $-$ Odjęcie od wskaźnika stałej/zmiennej całkowitej.

   W ostatnich dwu przypadkach wynikiem jest nowy adres.

Wskaźniki

```
int x = 1, y = 2, z[10];
int *ip;     /* ip jest wskaznikiem do obiektu typu int */

ip = &x;     /* teraz ip wskazuje na x */
y = *ip;     /* y ma teraz wartosc 1 */
*ip = 0;     /* x ma teraz wartosc 0 */
ip = &z[0]; /* teraz ip wskazuje na element z[0] */
```

Aliasy

Jeżeli pa i pb to wskaźniki tego samego typu

```
int b;
int *pa, *pb;
pb = &b;
```

to polecenie

```
pa = pb;
```

tworzy coś w rodzaju aliasu: przez nazwy pa i pb możemy odwoływać się do tej samej zmiennej.

Wskaźniki i tablice I

1. Wskaźniki mogą przydawać się w przypadku organizowania dostępu do tablic.
2. Działają w tym przypadku trochę jak indeks tablicy.
3. Jeżeli mamy coś takiego:

```
int a[10];
int *pa;

pa = &a[0];
```

w **pa** mamy wskaźnik pokazujący na zerowy element tablicy **a**, czyli

```
x = *pa;
```

Wskaźniki i tablice II

jest równoważne poleceniu

```
x = a[0];
```

Natomiast zwiększenie wskaźnika o 1 da nam dostęp do następnej komórki w tablicy **a**; zapisujemy to tak:

```
y = *(pa + 1);
```

gdyż jednoargumentowy operator * ma wyższy priorytet niż dodawanie!

Wskaźniki i tablice I

1. Wszystko działa poprawnie niezależnie od tego ile miejsca w pamięci zajmuje element tablicy (jakiego jest typu) tak długo, jak poprawnie deklarujemy zmienną wskaźnikową...
2. pa+1 pokazuje „następny obiekt" tablicy
3. pa+i pokazuje element oddalony od pa o i takich obiektów.
4. Nazwa tablicy jest **stałą** typu wskaźnikowego! Zamiast pisać

```
pa = &a[0];
```

można napisać

```
pa = a;
```

Wskaźniki i tablice II

5. Natomiast odwołanie do a[i] można zapisać jako *(a+i). Nazwa tablicy pełni rolę wskaźnika do jej pierwszego elementu.
6. Identyczne są również a+i oraz &a[i].
7. Natomiast wskaźnik jest zmienną i można na nim wykonywać operacje typu pa=a czy pa++. Nazwa tablicy nie jest zmienną (raczej należy ją traktować jak stałą), zatem konstrukcja a=pa czy a++ są niedozwolone!

Tablice dwuwymiarowe

1. Język C zna pojęcie tablicy dwuwymiarowej.
2. Deklaruje się ją tak:
   <typ> <*nazwa*> [<*liczba_wierszy*>][<*liczba_kolumn*>]
3. **int** tab2 [10][20] // to tablica o 10 wierszach i 20 kolumnach.
4. Dane w tablicy przechowywane są „wierszami”: najpierw w pamięci zapisane są dane pierwszego wiersza, później drugiego,...
5. Można też wyobrazić sobie inną konstrukcję: najpierw deklarujemy tablicę jednowymiarową w której zapisujemy wskaźniki do jednowymiarowych tablic (wierszy tablicy).

Funkcje i parametry raz jeszcze

Podczas wywoływania funkcji wykonywane są następujące czynności:

1. Wyliczana jest wartość wszystkich argumentów funkcji.
2. Dokonywane są konwersje typów.
3. Wyznaczone tak wartości „przekazywane” są do wnętrza funkcji (to znaczy wyznaczone wartości przypisywane są zmiennym wewnątrz funkcji.
4. Nastepnie zostaje wykonana funkcja.
5. Ewentualy wynik przekazywany jest poleceniem **return**.
6. Jakiekolwiek zmiany wartości zmiennych lokalnych lub parametrów **wewnątrz** funkcji nie są przekazywane na zewnątrz.

Funkcje i parametry

1. Parametrem (formalnym) funkcji może być wskaźnik:

```
int funkcja(int *par)
{
    return *par;
}
```

2. Podczas wywołania funkcji parametrem „aktualnym” w tym miejscu musi być adres zmiennej (prostej lub złożonej). Na przykład:

```
int a;
...
u = fumkcja(&a);
```

Prosty program

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

int funkcja(int *par)
{
    return *par;
}

int main(void)
{
    int tablica[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int zmienna_prosta = 123;
    int wynik;
    wynik = funkcja(&zmienna_prosta);
    printf("1. wynik = %d\n", wynik);
    wynik = funkcja(&tablica[3]);
    printf("2. wynik = %d\n", wynik);
    wynik = funkcja(tablica);
    printf("3. wynik = %d\n", wynik);
    return EXIT_SUCCESS;
}
```

Wynik

1. `wynik = 123`
2. `wynik = 4`
3. `wynik = 1`

Wskaźniki i funkcje I

- Efektem ubocznym przekazywania parametrów do funkcji przez wartość jest to, że nie można w poniższy sposób stworzyć funkcji realizującej funkcję $a \leftrightarrow b$ (zamiana miejscami wartości dwu zmiennych; pierwsze, naiwne podejście, **nie działa**:

```
void swap(int a, int b) /* To jest zle!!! */
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Wskaźniki i funkcje II

gdyż żadne zmiany dokonywane na zmiennych wewnątrz funkcji nie „wydostają" się na zewnątrz.

Wskaźniki i funkcje I

- Problem można rozwiązać za pomocą wskaźników:

```
void swap(int *pa, int *pb)
{
    int temp;

    temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

- Wywołanie tej funkcji będzie takie:

```
swap(&x, &y);
```

Wskaźniki i funkcje II

- Zwracam uwagę, że próba zmiany wartości wskaźników wewnątrz funkcji również nie przenosi się poza funkcję.

Funkcje, tablice, wskaźniki

- Ponieważ nazwa tablicy (w pewnym zakresie) może być traktowana jako wskaźnik, prawidłowa jest poniższa definicja parametru formalnego: **int** x[] i jest ona równoważna **int** *x
- Wydaje się, że drugi zapis jest „lepszy" (jako bliższy prawdy?)
- Można do funkcji przekazać fragment tablicy podając jako parametr aktualny wskaźnik do początku (pod)tablicy: f(&a[2])
- Wewnątrz funkcji **f** deklaracja parametru formalnego może mieć postać:
  f(**int** arr[]) {...}
  lub
  f(**int** *arr) {...}
- Można też odwoływać się do elementów tablicy wstecz (arr[−1], arr[−2]) jeśli jest rzeczą pewną, że elementy te istnieją

Funkcje, tablice dwuwymiarowe, wskaźniki

- Tabice dwuwymiarowe mogą sprawiwć pewne problemy gdy powinny być parametrem funkcji.
- Gdy w programie głównym zadeklarowaliśmy tablice jako:
  `int tab2 [10][20]`
- W funkcji deklaracja parametru może wyglądać albo tak:
  `int f(int a [10][20])`
  albo tak:
  `int f(int a [][20])`
  albo tak:
  `int f(int (*a )[20])`

- Wywołanie funkcji zaś tak: `i = f(tab2);`

Argumenty wywołania programu II

```
#include <stdio.h>
main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
       printf("%s%s", argv[i], (i < argc - 1)?" ":"");
    printf("\n");
    return 0;
}
```

Teraz powinno być już jasne w jaki sposób można „dobierać się" do danych (o których liczbie i długości nic nie wiemy w trakcie pisania programu).

Argumenty wywołania programu

- Ale jak się dobrać do pierwszego znaku pierwszego argumentu?
- *argv jest wskaźnikiem pokazującym zerowy element tablicy argumentów (nazwa programu).
- *++argv wskazuje na pierwszy (czyli właściwy) element tekstu.
- Najprawdopodobniej zatem (*++argv)[0] to pierwszy znak.

Uwaga Nawias okrągły jest potrzebny, gdyż operator [] (pobierania elementu tablicy) ma wyższy priorytet niż operatory adresu (*) i operator zwiększenia (++).

Pamięć dynamiczna

- Jedną z funkcji Systemu Operacyjnego jest przydział pamięci.
- Jak jest to realizowane?
  - Po pierwsze — gdy użytkownik uruchamia program, SO przydziela programowi pamięć niezbędną do pracy.
  - Po drugie przydziela pamięć — niejako automatycznie — na potrzeby powstających w trakcie pracy programu zmiennych.
- Co jednak zrobić, gdy podczas pisania programu nie znamy liczby danych (a zatem ilości potrzebnej do ich przechowywania pamięci)?
- Musimy wykorzystać funkcje dynamicznego przydziału pamięci!

Pamięć dynamiczna I

malloc, calloc

1. Do dynamicznego przydzielania pamięci służy funkcja malloc
2. Sposób użycia funkcji („deklaracja" funkcji):
   extern void *malloc(size);
   - **extern** mówi, że funkcja jest „zewnętrzna" czyli zdefiniowana gdzie indziej niż nasze pliki źródłowe
   - **void** *malloc mówi, że funkcja zwraca wynik w postaci wskaźnika typu **void** (nieokreślonego typu). wskaźnik ten jest równy NULL gdy System Operacyjny nie może przydzielić pamięci lub wskazuje na początek obszaru przydzielonej pamięci.
   - size to parametr mówiący ile (bajtów) pamięci potrzebujemy.

Pamięć dynamiczna II

malloc, calloc

3. Żeby z funkcji korzystać trzeba użyć dyrektywy **#include** <stdlib.h> gdzieś na początku programu.
4. Przydzielona pamięć może zawierać przypadkowe wartości.
5. Funkcja **extern void** *calloc(nelem, elsize); może być wykorzystania do uzyskania „wyzerowanego" obszaru pamięci; pierwszy argument (nelem) określa liczbę żądanych jednostek, drugi (elsize) wielkość w bajtach każdej jednostki.

Pamięć dynamiczna I

Przykład

```
#include <stdio.h>
/* required for the malloc and free functions */
#include <stdlib.h>
int main() {
    int number;
    int *ptr;
    int i;
    printf( "How many ints would you like store? " );
    scanf( "%d", &number );
    /* allocate memory */
```

Pamięć dynamiczna II

Przykład

```
ptr = malloc( number * sizeof( int ) );
if ( ptr != NULL )
{
   for ( i = 0; i < number; i++ )
      *( ptr + i ) = i;
   for ( i = number; i > 0; i— )
      /* print out in reverse order */
      printf( "%d\n", *( ptr + ( i - 1 ) ) );
   /* free allocated memory */
   free( ptr );
   return 0;
```

Pamięć dynamiczna III

Przykład

```
    }
    else
    {
        printf( "\nMemory allocation failed –"
                " not enough \ memory.\n" );
        return 1;
    }
}
```

Dynamiczny przydział pamięci

Zwalnianie pamięci

1. Dobry obyczaj każe oddać to co się pożyczyło.
2. W zasadzie, w chwili zakończenia programu cała przydzielona pamięć powinna zostać automatycznie zwrócona...
3. ...ale różnie bywa.
4. Funkcja **extern void** free (**void** *ptr) zwraca pamięć. Jedynym argumentem jest wskaźnik początku obszaru przydzielonej pamięci.
5. Funkcja **extern void** *realloc (**void** *ptr, size) pozwala zmienić (rozszerzyć, zmniejszyć) posiadany obszar pamięci do zadanego obszaru.

Dynamiczny przydział pamięci I

Pożytek ze wskaźników

```c
#include <stdio.h>
/* required for the malloc and free functions */
#include <stdlib.h>
int main() {
    int number;
    int *ptr;
    int i;
    printf("How many ints would you like store? ");
    scanf("%d", &number);
    /* allocate memory */
```

Dynamiczny przydział pamięci II

Pożytek ze wskaźników

```
   ptr = malloc( number * sizeof( int ) );
   if ( ptr != NULL )
   {
      for ( i = 0; i < number; i++ )
//       *(ptr+i) = i;
         ptr[i] = i;
      for ( i = number; i > 0; i-- )
         /* print out in reverse order */
//       printf("%d\n", *(ptr+(i-1)));
/* print out in reverse order */
         printf("%d\n", ptr[i - 1]);
```

Dynamiczny przydział pamięci III

Pożytek ze wskaźników

```
        /* free allocated memory */
        free(ptr);
        return 0;
    }
    else
    {
        printf("\nMemory allocation failed - "
               "not enough memory.\n");
        return 1;
    }
}
```

Drobne podsumowanie

```
int T[10];
...
...
for (i = 0 ; i < 10 ; i++)
    T[i] = i;
...
for(i = 0 ; i < 10 ; i++)
    printf("%d\n", T[i]);
```

```
int * T =
    malloc(10 * sizeof(int));
...
for (i = 0; i < 10 ; i++)
    *(T + i) = i;
...
for (i = 0 ; i < 10 ; i++)
    printf("%d\n", T[i]);
```

Drobne podsumowanie I

Ciąg dalszy

Wyobraźmy sobie, że mamy następującą definicję funkcji:

```
int f(int n, double b , int c[], double * d,
                        int e[][n], double ** g);
```

1. Funkcja ma sześć (6) parametrów.
2. Funkcja zwraca wartości całkowite (to jest nieistotne).
3. Podczas wywołania funkcji:
   - pierwszym argumentem może być: stała, zmienna lub wyrażenie typu całkowitego;
   - drugim argumentem może być: stała, zmienna lub wyrażenie typu podwójnej precyzji;

Drobne podsumowanie II

Ciąg dalszy

W przypadku gdy typ pierwszych dwu argumentów będzie inny niż zadeklarowany — zostanie dokonana odpowiednia konwersja.

4. Trzecim (i czwartym) argumentem powinien być wskaźnik do (adres) tablicy typu **int** (**double**). Żadne konwersje nie będą wykonywane gdy argumentem będzie wskaźnik innego typu. Należy się spodziewać najgorszych możliwych efektów w takim przypadku.

Drobne podsumowanie III

Ciąg dalszy

Przykład

```
int t[10];
int * u = malloc( 40 );
```

Argumentem może być:

- t
- u
- &t[0] albo nawet &t[1] czy alternatywnie u + 1,

Drobne podsumowanie IV

Ciąg dalszy

5. W przypadku argumentu piątego (**int** e [][n] parametrem wywołania funkcji może być wyłącznie tablica automatyczna lub statyczna (typu **int**) zadeklarowana jako:

   ```
   int v[m][n]
   ```

   gdzie m i n to jakieś stałe. (Wówczas pierwszy parametr wywołania funkcji (n) powinien mówić o liczbie kolumn!)

6. Ostatnim argumentem funkcji (podwójny wskaźnik) powinna być tablica dynamiczna typu **double** w, zadeklarowana jakoś tak:

Drobne podsumowanie V

Ciąg dalszy

```
double ** w;
int m, n, i;
w = (double **) malloc(n * sizeof( double *));
for(i = 0; i < n; i++)
w[i] = (double *) malloc(m * sizeof(double));
```

o n wierszach i m kolumnach.
(Uwaga: Wszędzie powinno być dodane sprawdzenie, czy funkcja malloc nie zwróciła wartości zero (NULL).

Dla dociekliwych

Indeks tablicy i wskaźnik do elementu

```c
/* Program 2.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];      /* point our pointer to the first
                                element of the array */
    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
      printf("my_array[%d] = %d   ", i, my_array[i]);    /*<-- A */
      printf("ptr + %d = %d\n", i, *(ptr + i));          /*<-- B */
    }
    return 0;
}
```