



Politechnika
Wroclawska

Bardzo szybkie podsumowanie: wykład 3

wer. 10 z **drobnymi modyfikacjami!**

Wojciech Myszka

2023-04-03 06:51:01 +0200



HR EXCELLENCE IN RESEARCH

Uwagi

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

Tablice (jedno i wielowymiarowe), łańcuchy znaków

wer. 8 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2022-03-25 07:28:03 +0100



HR EXCELLENCE IN RESEARCH



Politechnika Wroclawska

Zmienne

Przypomnienie/podsumowanie

1. Wszystkie zmienne muszą być zadeklarowane.
2. Nazwa zmiennej składa się z liter i cyfr, a rozpoczyna się literą; znak podkreślenia zalicza się do liter.
3. Nazwy zmiennych nie powinny się zaczynać od znaku podkreślenia (tak nazywają się zmienne systemowe).
4. Deklaracja obowiązuje wewnątrz bloku (i we wszystkich blokach znajdujących się „niżej”).
5. W C występują zmienne globalne (zewnętrzne) i lokalne.
6. Deklaracja lokalna przysłania deklarację globalną (jeżeli nawa zmiennej jest taka sama).



W szczególności...

...poniższa konstrukcja

```
for (int i = 0; i < 10; i++)  
{  
    printf("i = %d\n", i);  
}
```

jest poprawna.

...gdyz zmienna i jest zmienną lokalną tylko dla pętli!

Ale poniższa

```
for (int i = 0; i < 10; i++)  
{  
    printf("i = %d\n", i);  
}  
printf("i = %d\n", i);
```

już nie...



Zmienne zewnętrzne i wewnętrzne

```
#include <stdio.h>
int a; // <— Zmienna zewnetrzna
int main(void)
{
    int b; // <— Zmienna wewnetrzna
    ...
}
```

Zmienne zewnętrzne nazywane bywają zmiennymi „globalnymi” (czyli dostępnymi dla każdej funkcji programu).



Zmienne statyczne i automatyczne

Trochę zamętu

1. Dodatkowo można zażądać od zmiennej żeby była „statyczna” (co deklaruje się dodając słowo kluczowe **static** przed nazwą typu).

```
static int x;
```

2. Zmienna statyczna zewnętrzna pozostaje zdefiniowana **tylko** dla funkcji zdefiniowanych w jednym pliku źródłowym (ukryta jest dla funkcji z innych plików źródłowych).
3. Zmienna statyczna wewnętrzna zachowuje swoją wartość pomiędzy kolejnymi wywołaniami funkcji.
4. Zmienne, które nie są statyczne **nie muszą** zachowywać wartości między wejściami do funkcji (ale mogą) — ale nie można na to liczyć!
5. Dodatkowo zmienne statyczne wewnętrzne inicjowane są na wartość zero (jeżeli programista nie zażąda żeby było inaczej).



Zmienne statyczne i automatyczne

```
#include <stdio.h>
void f(void)
{
    static int x ; /* zmienna statyczna */
    int y = 0;     /* zmienna automatyczna */
    x++;
    y++;
    printf("X=%d, Y=%d\n", x, y);
}
int main()
{
    f();
    f();
    f();
    return 0;
}
```



Tablice

1. Gdy potrzebujemy przechować kilka zmiennych tego samego typu (i jakoś powiązanych ze sobą) stosujemy **tablicę**.
2. Tablica to ciąg zmiennych o tej samej nazwie; dostęp do poszczególnych elementów odbywa się przez podanie numeru zmiennej (indeksu/ów).

0 1 2 3 4 5 6 7 8 9

--	--	--	--	--	--	--	--	--	--

3. Elementy numerowane są **począwszy od zera**.
4. Deklaracja wygląda tak:
typ nazwa_tablicy[rozmiar];



Tablice

1. Tablica jest zmienną złożoną (strukturą pewnego rodzaju).
2. Służy do przechowywania danych tego samego typu.
3. Jeżeli chcemy nadać elementom tablicy wartości początkowe
`int tablica[3] = {1,2,3};`
4. To jest również poprawna deklaracja:
`int tablica[20] = {1,};`
(pierwszy element tablicy ma wartość 1, pozostałe mają wartość 0)
5. Nie zawsze trzeba podawać rozmiar tablicy — czasami kompilator może się domyślić sam:
`int tablica[] = {1, 2, 3, 4, 5};`
zostanie zadeklarowana tablica o pięciu elementach.



Wielkość tablic

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int t[] = {1, 2, 3, 4, };
5     int i;
6     for (i = -1; i < 7; i++)
7         printf("t[%d] = %d\n", i, t[i]);
8     return 0;
9 }
```



Wielkość tablic

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int t[] = {1, 2, 3, 4, };
5     int i;
6     for (i = -1; i < 7; i++)
7         printf("t[%d] = %d\n", i, t[i]);
8     return 0;
9 }
```

Ile elementów ma tablica t?



Wykonany po raz pierwszy

t[-1] = 11131

t[0] = 1

t[1] = 2

t[2] = 3

t[3] = 4

t[4] = -1296194160

t[5] = 32767

t[6] = 0



Wykonany po raz drugi

t[-1] = 10955

t[0] = 1

t[1] = 2

t[2] = 3

t[3] = 4

t[4] = -868000288

t[5] = 32767

t[6] = 0



Wykonany po raz trzeci

t[-1] = 11015

t[0] = 1

t[1] = 2

t[2] = 3

t[3] = 4

t[4] = -143761264

t[5] = 32767

t[6] = 0



Inicjowanie I

1. W deklaracji obiektu można zawrzeć wartość początkową deklarowanego identyfikatora.
2. Inicjator, który poprzedza się operatorem = jest albo wyrażeniem, albo listą inicjatorów zawartą w nawiasach klamrowych.
3. Lista może kończyć się przecinkiem.
4. Dla obiektów i tablic statycznych wszystkie wyrażenia w inicjatorach muszą być wyrażeniami stałymi.
5. Nie inicjowany jawnie obiekt statyczny jest inicjowany tak, jakby jemu, (lub jego składowym) przypisano wartość zero.
6. Początkowa wartość nie zainicjowanego jawnie obiektu automatycznego jest niezdefiniowana.
7. Inicjatorem dla obiektu arytmetycznego jest pojedyncze wyrażenie (być może ujęte w nawiasy klamrowe).



Inicjowanie II

8. Inicjatorem dla struktury jest albo wyrażenie tego samego typu albo ujęta w nawiasy klamrowe lista inicjatorów dla jej kolejnych składowych.
9. Inicjatorem dla tablicy jest ujęta w klamry lista inicjatorów dla jej kolejnych elementów.
10. Jeśli nie jest znany rozmiar tablicy — to rozmiar ten wylicza się na podstawie liczby inicjatorów.
11. Jeśli tablica ma ustalony rozmiar — liczba inicjatorów nie może przekroczyć liczby elementów tablicy; jeśli lista jest krótsza uzupełniana jest zerami.
12. Specjalnym przypadkiem jest tablica znakowa, która może być inicjowana napisem (kolejne znaki napisu inicjują kolejne elementy tablicy).



Inicjowanie III

13. Jeżeli nie jest znany rozmiar tablicy znakowej jest on wyliczany na podstawie liczby znaków w napisie (włączając w to końcowy znak zerowy).



Tablice wielowymiarowe

Przykład

```
#include <stdio.h>
int main(void)
{
    int a[4][3] = {
        {1, 3, 5},
        {2, 4, 6},
        {3, 5, 7},
    };
    int i, j;
    for (i = 0; i < 4; i++){
        for (j = 0; j < 3; j++)
            printf(" %d |", a[i][j]);
        printf("\n");
    }
    return 0;
}
```

Tablice wielowymiarowe

Wynik działania programu

1		3		5	
2		4		6	
3		5		7	
0		0		0	



Tablice wielowymiarowe

Zadanie domowe

Zmodyfikować tak przykładowy program, żeby drukował wyniki w następującej postaci:

```
| 1 | 3 | 5 |  
| 2 | 4 | 6 |  
| 3 | 5 | 7 |  
| 0 | 0 | 0 |
```



Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {  
    1, 3, 5, 2, 4, 6, 3, 5, 7  
};
```



Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {  
    1, 3, 5, 2, 4, 6, 3, 5, 7  
};
```

Niestety, powyższe nie do końca jest poprawne: pojawią się ostrzeżenia podczas kompilacji!



Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {  
    1, 3, 5, 2, 4, 6, 3, 5, 7  
};
```

Niestety, powyższe nie do końca jest poprawne: pojawią się ostrzeżenia podczas kompilacji!

1		3		5	
2		4		6	
3		5		7	
0		0		0	



Tablice wielowymiarowe

Inicjowanie — warianty

```
int a[4][3] = {  
    { 1 }, { 2 }, { 3 }, { 4 }  
};
```



Napisy

1. Stała znakowa (złożona z jednego znaku) zapisywana jest tak 'c' („c” to dowolny znak lub specjalna stała złożona ze znaku „backslash” (\) i specjalnego symbolu.
2. Stała tekstowa zapisywana jest w cudzysłowach (podwójne apostrofy) "Ala ma kota"
3. Sąsiadujące ze sobą napisy łączone są w jeden napis ("Ala" "ma" "kota" tworzy napis "Alamakota"; "Ala " "ma" " kota" tworzy "Ala ma kota").
4. Na końcu napisu umieszczany jest znak o kodzie ASCII równym zero ('\x00') pozwalający rozpoznać koniec tekstu.
5. W napisach można używać wszystkich symboli specjalnych dostępnych w statych znakowych.
6. Typem do przechowywania znaków jest **char**.
7. Napisy trzeba przechowywać w tablicach typu **char**.
8. Polskie znaki — na razie proponuję o tym zapomnieć!



Tablice znakowe

To jest poprawna deklaracja. Tablica będzie miała rozmiar 14 (13 znaków napisu i znak null kończący napis). Można to sprawdzić za pomocą funkcji `sizeof(tekscik)`.

```
char tekscik[] = "Ala ma kotała";
```

Poniżej również poprawna deklaracja tablicy (o łącznym rozmiarze 21 znaków). Liczba wierszy wyliczana jest automatycznie podczas kompilacji.

```
char teksty[][7] = {  
    {"Ala"},  
    {"ma"},  
    {"kotała"}  
};
```

To niepoprawna forma deklaracji:

```
char teksty[3][] = {  
    {"Ala"},  
    {"ma"},  
    {"kotała"}  
};
```



Tablice jako argumenty funkcji

1. Trzeba bardzo uważać i myśleć zanim się coś zrobi!
2. Rzecz nie jest prosta (choć może nie aż tak skomplikowana).



Przykład

- ▶ Chcemy napisać funkcję, która zeruje tablicę (wypełnia wartością zero wszystkie elementy tablicy).
- ▶ Sam algorytm jest bardzo prosty

```
int N = 10;  
double A[N];  
for (int i = 0; i < N; i++)  
    A[i] = 0.;
```

- ▶ Musimy go tylko obudować w funkcję
 - ▶ nie zwraca parametrów
 - ▶ ma dwa parametry:
 1. tablicę
 2. jej rozmiar



Przykład c.d.

```
void zeruj(int N, double A[])
{
    for(int i = 0; i < N; i++)
        A[i] = 0.;
}
```

A używać jej będziemy tak:

```
int main()
{
    double X[1000];
    int M = 1000;
    // ...
    zeruj(M, X);
    // ...
}
```

Wskaźniki. Pamięć dynamiczna

wer. 11 z drobnymi modyfikacjami!

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2022-04-19 14:57:37 +0200



HR EXCELLENCE IN RESEARCH



Politechnika Wroclawska

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 bez nich pewne operacje.
 - ▶ Pozwalają na tworzenie zwartego i efektywnego kodu.
 - ▶ Są bardzo efektywnym narzędziem.
- ▶ Korzystając z:
 - ▶ tablic,
 - ▶ struktur (będzie później),
 - ▶ funkcjikorzystamy 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 RAM) innej zmiennej.



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 deklaruowania 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!
5. Należy natomiast pamiętać, że polecenie:



Zmienne a pamięć operacyjna II

```
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 wskaźnikiem do obiektu typu int */  
double *fp;  
    /* fp jest wskaźnikiem do obiektu typu double */  
char* cp // wskaźnik do typu char  
float*Fp // wskaźnik 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 całkowita 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 wskaźnikiem do obiektu typu int */  
  
ip = &x;    /* teraz ip wskazuje na x */  
y = *ip;    /* y ma teraz wartość 1 */  
*ip = 0;    /* x ma teraz wartość 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;
```

jest równoważne poleceniu



Wskaźniki i tablice II

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

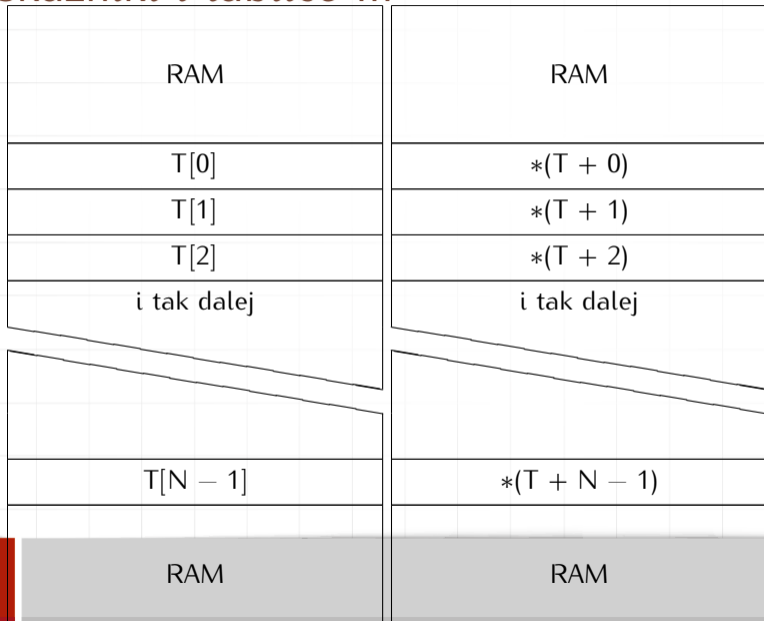
Natomiast zwiększenie wskaźnika o 1 da nam dostęp do następnej komórki w tablicy **a**; zapisujemy to tak:

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

gdyż jednoargumentowy operator ***** ma wyższy priorytet niż dodawanie!



Wskaźniki i tablice III



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;
```

5. Natomiast odwołanie do $a[i]$ można zapisać jako $*(a+i)$. Nazwa tablicy pełni rolę wskaźnika do jej pierwszego elementu.



Wskaźniki i tablice II

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. Następnie zostaje wykonana funkcja.
5. Ewentualny wynik przekazywany jest poleceniem **return**.
6. Jakiegokolwiek zmiany wartości zmiennych lokalnych lub parametrów **wewnątrz** funkcji nie są przekazywane na zewnątrz.



Parametry

Jednym z ważniejszych obowiązków programisty jest zadbanie aby:

1. liczba parametrów w wywołaniu funkcji była identyczna z liczbą parametrów w definicji
2. zgadzał się typ parametrów.

O ile brak zgodności w pierwszym punkcie kończy się niepowodzeniem kompilacji, to w drugim przypadku zazwyczaj powoduje ostrzeżenia. W większości przypadków dokonywane jest rzutowanie, które może powodować utratę informacji...

Uwaga

Szczególnie niebezpieczne nieprzemyślane rzutowanie wskaźników.



Wartość zwracana przez funkcje

1. Może być tylko **jedna**!
2. Typ funkcji **musi** być zgodny z typem wyrażenia pojawiającego się jako argument polecenia `return`.

```
typ f()  
{  
    ...;  
    return cos  
}
```

Czyli typy zmiennej `cos` lub wyrażenia arytmetycznego `cos` powinny być identyczne. W pewnym zakresie można liczyć na **automatyczne** rzutowanie.



Gdy nie określimy typów funkcji ani parametrów

```
f(x)
{
    x = x * 2;
    return x;
}
```

system **automatycznie** przyjmie, że typ zmiennej x oraz typ wartości zwracanej przez funkcję $f(x)$ jest `int`.



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 = funkcja(&a);
```



Prosty program

```
#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;
}
```

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);
```

- ▶ 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

- ▶ Tablice dwuwymiarowe mogą sprawić 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);`



Wskaźnik jako wynik funkcji

W szczególności wynikiem funkcji może być wskaźnik. Funkcja taka będzie zadeklarowana jakoś tak:

```
int * test (...)
{
    int * a;
    ...
    ...
    return a;
}
```

Argumentem polecenia **return** musi być stała lub zmienna typu wskaźnikowego.



Przykład I

```
int * test (...)
{
    int a[10]={1, 2, 3, 4};
    ...
    ...
    return a;
}
```

- ▶ Powyższy przykład z formalnego punktu widzenia jest OK.
- ▶ Zmienna a (tablica) jest automatyczna i w związku z tym po wyjściu z funkcji przestaje istnieć, zatem zwracany wskaźnik wskazuje na coś czego już nie ma!



Przykład II

- ▶ Program można zmodyfikować tak żeby było lepiej, zastępując tablicę a tablicą statyczną:

```
static int a[10] = {...};
```



Argumenty wywołania programu I

- ▶ Możemy teraz (szybko) wrócić do przekazywania parametrów do funkcji main czyli programu.
- ▶ W chwili uruchomienia programu System Operacyjny tworzy strukturę danych zawierającą liczbę parametrów (integer, zazwyczaj nazywany `argc` — *argument count*) oraz tablicę zawierającą parametry (`argv` — *argument vector*).



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 dostać 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 (pamiętamy o `#include<stdlib.h>`)
2. Sposób użycia funkcji („prototyp” 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.
3. Żeby z funkcji korzystać trzeba użyć dyrektywy `#include <stdlib.h>` gdzieś na początku programu.



Pamięć dynamiczna II

malloc, calloc

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>
#include <stdlib.h>
int main() {
    int number;
    int *ptr;
    int i;
    printf( "How many ints would you like store? " );
    scanf( "%d", &number );
    /* allocate memory */
    ptr = malloc( number * sizeof( int ) );
    if ( ptr != NULL )
    {
        for ( i = 0; i < number; i++ )
```

Pamięć dynamiczna II

Przykład

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



Pamięć dynamiczna III

Przykład

```
}  
}
```



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

```
#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 */
    ptr = malloc( number * sizeof( int ) );
    if ( ptr != NULL )
    {
```


Dynamiczny przydział pamięci II

Pożytek ze wskaźników

```
    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]);
        /* free allocated memory */
    free(ptr);
    return 0;
}
else
```



Dynamiczny przydział pamięci III

Pożytek ze wskaźników

```
{  
    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] );
```



Tablice wielowymiarowe...

- ▶ ...również mogą być deklarowane w sposób dynamiczny,
- ▶ ...ale jest to bardziej skomplikowane

Tablica o N wierszach i M kolumnach:

```
int ** U;  
...  
U = malloc(N * sizeof(int *))  
for(int i = 0; i < N; i++)  
    U[i] = malloc(M * sizeof(int));
```



Co już wiemy? (Co powinniśmy wiedzieć) I

1. Zmienne **automatyczne** to zmienne deklarowane w sposób następujący:

```
int a, A[10], B[N][M];
```

- ▶ są one deklarowane w pamięci stosu (o ograniczonym rozmiarze);
- ▶ deklaracja obowiązuje jedynie w ramach bloku (`{...}`), w którym została wykonana (i bloków niższych);
- ▶ A to stała typu „wskaźnik na int” (czyli `int *`) zawierająca adres tablicy A;
- ▶ adres zmiennej a trzeba wyliczyć za pomocą operatora `&`;
- ▶ tablice (automatyczne) dwu (i więcej wymiarowe) to konstrukcje *skomplikowane...*
- ▶ zajmują ciągły obszar pamięci (o rozmiarze `N * M * sizeof(typ)`)



Co już wiemy? (Co powinniśmy wiedzieć) II

- ▶ mogą być traktowane jako „tablica tablic” — w przypadku tablicy dwuwymiarowej B jest to tablica o N wierszach, każdy wiersz tej tablicy to tablica o M kolumnach, zatem...
- ▶ ... B to również **stała** typu wskaźnikowego, ale typ jest *dziwaczny*: `int (*) [M]` — tablica o M elementach — (zwiększenie B o jeden w arytmetyce wskaźników zmienia adres tak, by wskazywał na kolejny wiersz tablicy)

2. Zmienne dynamiczne tworzone są z wykorzystaniem funkcji

- ▶ `void *malloc(size_t size)`
- ▶ `void *calloc(size_t nmemb, size_t size)`

w powyższym `void *` to wskaźnik **bez określonego typu**,
`size_t` to specjalny typ (zbliżony do `int`, ale 64-bitowy, bez znaku),
`size` rozmiar pamięci (w bajtach),
`nmemb` — liczba elementów tablicy,
`size` rozmiar (w bajtach) jednego elementu tablicy;



Co już wiemy? (Co powinniśmy wiedzieć) III

- ▶ funkcja `calloc()` zeruje przydzieloną pamięć;
- ▶ obie funkcje zwracają adres początku tablicy.

3. Tworzenie tablicy jednowymiarowej T o N elementach jest proste:

```
typ *T;  
T = malloc(N * sizeof(typ));
```

lub

```
typ *T;  
T = calloc(N, sizeof(typ));
```

zawsze jednak trzeba sprawdzić, czy wartość zwracana przez funkcję nie jest równa zero — w tym przypadku pamięć **nie została** przydzielona.

Podczas operacji podstawienia (`T =`) dokonywana jest konwersja (rzutowanie) wskaźnika `void` na wskaźnik odpowiedni dla typu zmiennej T.



Co już wiemy? (Co powinniśmy wiedzieć) IV

4. Tworzenie tablicy dwuwymiarowej U o N wierszach i M kolumnach jest bardziej skomplikowane

- ▶ zaczynamy od zadeklarowania zmiennej U typu

```
typ ** U;
```

(jednowymiarowa tablica, której każdy element jest wskaźnikiem typu typ)

- ▶ przydzielamy jej pamięć używając funkcji `malloc()`:

```
U = malloc(N * sizeof(typ));
```

(i sprawdzamy, czy zwrócona wartość jest różna od zera)
tworzy się struktura, która będzie użyta do przechowywania adresów początkowych każdego wiersza,

- ▶ następnie przydzielamy pamięć dla każdego wiersza tablicy:

```
for (int i = 0; i < N; i++)
```

```
    U[i] = malloc(M * sizeof(typ));
```



Co już wiemy? (Co powinniśmy wiedzieć) V

a uzyskane adresy wpisujemy do kolejnych komórek tablicy U ;
(przypominam o konieczności każdorazowego sprawdzenia czy wartość zwracana przez `malloc()` jest różna od zera).

5. W każdym przypadku $B[i]$ oraz $U[i]$ definiują adres i -tego wiersza w pamięci operacyjnej; ale w każdym przypadku wartość ta jest inaczej **wyliczana**:

- ▶ w przypadku tablicy B jest to adres B zwiększony (w arytmetyce wskaźników) o i ,
- ▶ w przypadku tablicy U , jest to zawartość i -tej komórki tablicy U ;

kolejna operacja służąca „dobrani” się do j -tego elementu wiersza tablicy $(B[i])[j]$ czy $(U[i])[j]$ wykonywana jest tak samo: adres zwiększany jest korzystając z arytmetyki wskaźników.



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;

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



Drobne podsumowanie II

Ciąg dalszy

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 $u + 1$,

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



Drobne podsumowanie IV

Ciąg dalszy

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

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

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

```
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).

