

# Termin Egzaminu (Język C):

>> **ŚRODA, 29 STYCZNIA 2025** <<

**GODZ. 11.00**

[ **POPRAWKA: ŚRODA, 19 LUTEGO** ]

**!!!**

**Forma zaliczenia kursu:** Egzamin pisemny – test wyboru \*) \*\*)

\*) Warunkiem przystąpienia do egzaminu jest **zaliczenie ćwiczeń**  
(w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

\*\*) **Ocena 5.0 (bdb)** z ćwiczeń *zwalnia z pisemnej części egzaminu*

[ **OCENA KOŃCOWA:** *0.5\*ocena z ćwiczeń + 0.5\*wynik egzaminu* ]

# Poprzedni wykład [ 19.11.2024 ] :

- Wskaźniki i tablice; wskaźniki jako argumenty funkcji
- Arytmetyka adresów
- Funkcje operujące na *wskaźnikach znakowych*
- Wskaźniki plikowe ( FILE \* ), funkcje fopen i fclose
- ***Funkcje o zmiennej liczbie argumentów***

# Wskaźniki i tablice (esencja ...)

Po deklaracjach:

```
double x[100], *px; int i=7;
```

i przypisaniu: `px = &x[0];` [ *lub równoważnie: `px = x;`* ]

**poprawne** ( *i równoważne!* ) będą wyrażenia:

```
x[i]      *(px+i)      *(x+i)      px[i]
```

oraz instrukcje: `px++; px += i;` itp. [ *arytmetyka wskaźników...* ]

**Niepoprawne** będą: `x++; x+=i;` itp. [ *... nie jest dla tablic!* ]

[ *Po przekazaniu jako argument funkcji, np.: `my_fun(char *s);`  
parametr działa (wewnątrz funkcji) już jak **zwyčajny wskaźnik**,  
również po wywołaniu dla stałej: `my_fun("Stały Napis");` ]*

# Operacje plikowe (*przypomnienie podstaw*)

Czytamy liczbę całkowitą z *pliku tekstowego*:

```
FILE *dane;  
dane = fopen("../mojplik.txt", "r");  
fscanf(dane, "%d", &N);  
fclose(dane);
```

`fopen(nazwa, tryb)` — wiąże *wskaźnik plikowy* z plikiem dyskowym;  
**tryby otwarcia**: "r" — odczyt, "w" — zapis, "a" — dopisywanie;  
*ponadto*: "r+", "w+", "a+" — jednoczesne czytanie i pisanie po tym samym pliku.

Dla **plików binarnych** — dodajemy literę `b` w *trybie otwarcia*:  
"rb", "wb", "r+b", "a+b", itp.

[ *Więcej*: [https://www.tutorialspoint.com/cprogramming/c\\_file\\_io.htm](https://www.tutorialspoint.com/cprogramming/c_file_io.htm) ]

Obok funkcji *formatowanego wejścia/wyjścia* (`fprintf`, `fscanf`), w przypadku **plików tekstowych** mamy do dyspozycji także funkcje tzw. wierszowego wejścia/wyjścia:

```
fgets(napis, rozmiar, plik)  
fputs(napis, plik)
```

W obu funkcjach, *pierwszym argumentem* jest tablica znakowa (lub wskaźnik `char *`) zaś ostatnim wskaźnik plikowy.

Drugi argument `fgets` (typu `int`) to *maksymalna l.znaków, które mogą zostać wczytane* **powiększona o 1** [ **na ogół podajemy tutaj rozmiar tablicy `napis`, aby uniknąć błędu naruszenia ochrony pamięci** ].

Dla **plików binarnych** działa wejście/wyjście znakowe (makra: `getc`, `putc`), istnieje także możliwość „hurtowego” transferu danych np. z pliku do tablicy:

```
FILE *plik=fopen("myfile.dat","rb");
double X[100]; int N;

N=fread(X, sizeof(double), 100, plik); /* <== */
fclose(plik);
```

Pierwszy z parametrów `fread` to wskaźnik (`void *`), drugim jest **rozmiar komórki pamięci**, trzecim — *l.elementów* do wczytania, a ostatnim — *wskaźnik plikowy*. [ **Wartość zwracana — to liczba faktycznie wczytanych elementów.** ]

Identyczny zestaw parametrów ma *komplementarna* (do `fread`) funkcja `fwrite`.

Obliczając *rozmiar komórki pamięci* użyliśmy **operatora `sizeof`**, który (w standardzie ANSI C) może pojawić się w dwóch formach:

```
sizeof obiekt      lub      sizeof( nazwa-typu )
```

gdzie ***obiekt*** to ***zmienna skalarna, tablica lub struktura*** ( => ***rozmiar obiektu musi być ustalony w momencie kompilacji !*** ).

Przykładowo, stała `100` podana jako trzeci argument `fwrite` (*l.elementów do wczytania*) może być zastąpiona przez wyrażenie:

```
sizeof X / sizeof(double)
sizeof X / sizeof X[0]          /* <=== */
```

[ Operator `sizeof` zwraca liczbę całkowitą typu `size_t`, który jest zdefiniowany w nagłówku `<stddef.h>`. ]

Jeśli używamy **niskopoziomowych** funkcji `fread`, `fwrite` (np. czytając/pisząc do *pliku tymczasowego* — o trybie dostępu `"wb+"` — otwartego funkcją `tmpfile()` z biblioteki standardowej) zwykle będziemy również używać funkcji

```
rewind( plik );
```

która „przewija” plik do początku, lub — bardziej uniwersalnej:

```
fseek( plik, przesunięcie, punkt—odniesienia );
```

gdzie *przesunięcie* podajemy w bajtach [ *tutaj często przydaje się operator `sizeof`...* ], zaś *punkt—odniesienia* można zdefiniować wybierając **jedną z trzech wartości**:

`SEEK_SET` — *początek pliku*

`SEEK_CUR` — *bieżąca pozycja w pliku*

`SEEK_END` — *koniec pliku*



Przykładowo: `fseek(f, 0, SEEK_SET);`

będzie *równoważne* wywołaniu `rewind(f);` a z kolei:

`fseek(f, 0, SEEK_END);`

— oznacza przesunięcie na koniec pliku.

[ *Więcej:* <http://man7.org/linux/man-pages/man3/fseek.3.html> ]

Z operacjami wejścia/wyjścia często wiąże się potrzeba **wywoływania poleceń systemowych**.

W systemach UNIX/LINUX służy do tego celu funkcja:

`system( polecenie )`

zdefiniowana w nagłówku `<stdlib.h>`

Przykładowo, instrukcja: `system("date");`

wyświetli aktualną datę i czas na standardowe wyjście.

# Dynamiczny przydział pamięci

W języku ANSI C rozmiary tablic muszą być **zadane jako stałe** lub **wyrażenia stałe**, np.:

```
int tab[100];    double X[3*3+7];
```

nieco innym (*ale tylko pozornie...*) sposobem jest *inicjacja tablicy* z użyciem obiektu o rozmiarze znanym w chwili kompilacji:

```
char s[] = "Napis";    int numbers[] = {1,2,3,4,5};
```

W przypadku, kiedy rozmiar tablicy jest **wynikiem obliczeń** niemożliwych (lub *niewygodnych*) do wykonania przed uruchomieniem programu (a nie chcemy *marnować pamięci* tworząc *na zapas* bardzo duże tablice ... )

==> stosujemy tzw. **dynamiczny przydział pamięci**.

Przykładowa sekwencja poleceń:

```
int N;
double *tab;
scanf("%d", &N);
tab = (double*)malloc(N * sizeof(double));
if (NULL == tab) {
    printf("Nie udało sie ...\n"); return 1;
}
```

tworzy tablicę `N` liczb typu `double` (gdzie `N` jest czytane ze standardowego wejścia) i wiąże ją ze wskaźnikiem `tab`.

Zamiast funkcji `malloc` można też użyć:

```
tab = (double*)calloc(N, sizeof(double));
```

— funkcja `calloc` dodatkowo **zeruje** tworzoną tablicę.

Kiedy zachodzi potrzeba **zwolnienia pamięci** używamy:

```
free( tab );
```

**Zmiana wielkości** już przydzielonej tablicy ( bez zerowania! ):

```
tab = realloc( tab, nowy-rozmiar );
```

Należy pamiętać, że po dynamicznym przydziale pamięci `tab` **pozostaje wskaźnikiem** ( $\Leftrightarrow$  *nie staje się tablicą!*),

a zatem np. działać będzie operator zwiększania: `tab++`  
[ *ze wszystkimi tego konsekwencjami ...* ]

Z kolei operator: `sizeof tab` obliczy po prostu **rozmiar wskaźnika** (na ogół: 4 bajty ...).

*Zilustrowane powyżej pewne rozmycie różnicy pomiędzy tablicą a wskaźnikiem stanowi przesłankę, aby dynamicznej alokacji nie nadużywać, kiedy nie jest ona konieczna.*

Przykład – **tablica trójkątna** do przechowywania współczynników dwumianu Newtona:

```
#define N_MAX 1000;
int j, n, *newt[N_MAX+1]; /* tablica wskaźników */

printf("Podaj n: \n");
scanf("%d", &n);
for (j=0; j<=n; j++) {
    newt[j] = (int*)calloc(j+1, sizeof(int));
    if ( NULL == newt[j] ) return 1;
}
/* ... Obliczenia ==> ĆWICZENIE! */
```

# Tablice wskaźników

Wskaźniki są także zmiennymi — mogą być przechowywane w tablicach. *W ostatnim przykładzie — tablica wskaźników pojawiła się jako **realizacja tablicy dwuwymiarowej**. [ ==> **c.d.n.** ]*

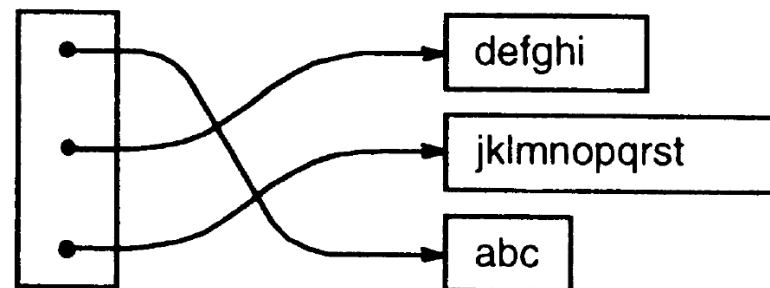
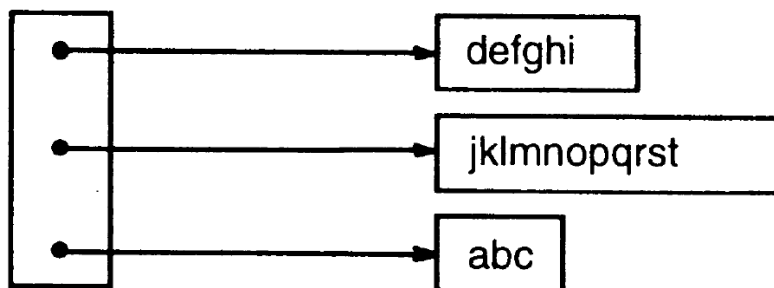
Znacznie częściej, tablice wskaźników przydają się w sytuacji, gdy chcemy **wprowadzić porządek** w zbiorze pewnych obiektów (= posortować ten zbiór), bez faktycznego przemieszczania ich w pamięci komputera:

**==> wystarczy wówczas jedynie uporządkować (według przyjętej reguły) odpowiednie wskaźniki w tablicy (!)**

[ *Kilka tablic: pozwoli wprowadzić kilka różnych uporządkowań na raz ...* ]

Rozważymy teraz przykład **porządkowania wierszy tekstu**.

*Możliwość przemieszczania samych wskaźników **eliminuje problemy:**  
(i) zarządzania pamięcią oraz (ii) kosztów przesuwania wierszy tekstu.*



**Proces porządkowania można podzielić na 3 etapy:**

- *przeczytaj wiersze z wejścia*
- *uporządkuj je (np. alfabetycznie)*
- *wypisz uporządkowane wiersze*

Program do **porządkowania wierszy** dzielimy na funkcje, spośród których 3 — *dostępne dla funkcji main* — będą realizować:

1. Wczytywanie (oraz *liczenie!*) kolejnych wierszy; z jednoczesnym *budowaniem tablicy wskaźników*
2. Porządkowanie *alfabetyczne napisów* (za pośrednictwem utworzonej tablicy wskaźników)
3. Wypisywanie ustalonej wcześniej liczby wierszy wskazywanych przez *kolejne wskaźniki w tablicy*.

**Funkcja wejściowa** (por. pkt.1) w praktyce będzie pracować z *ograniczoną maks. liczbą wierszy*; może zatem zwrócić np. -1 aby zasygnalizować *zbyt dużą ilość danych*.



```
/* Porządkowanie wierszy: Wersja 1 – "bez malloc" */
/* wg Kernighan-Ritchie, 1994 */

#include <stdio.h>
#include <string.h>

#define ALLOCSIZE 500000L /* potrzebna pamięć */
#define MAXLINES 10000 /* maks. liczba wierszy */

char *lineptr[MAXLINES];

int readlines(char *lineptr[], int maxlines);
void writelines(char *lineptr[], int nlines);

void qsortlines(char *lineptr[], int left, int right);
```

```
int main()
{
    int nlines;

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsortlines(lineptr, 0, nlines-1);

        fprintf(stderr, "==> PO SORTOWANIU: <==\n");
        writelines(lineptr, nlines);
        return 0;
    }
    else { /* stderr: pisze ZAWSZE na ekran! */
        fprintf(stderr, "ERROR: INPUT TOO BIG TO SORT\n");
        return 1;
    }
}
```

Funkcja **wypisująca wiersze** adresowane przez kolejne wskaźniki w tablicy (`lineptr[0], ..., lineptr[nlines-1]`):

```
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
    /* ==> Po każdej linii – dopisujemy '\n'! */
}
```

[ W 2. argumencie printf mamy z synonim: *nazwa tablicy wskaźników* to jednocześnie wskaźnik do wskaźnika do 1. znaku 1.wiersza! ]

***Dla porównania:*** elementarna wersja tej samej funkcji:

```
void writelines(char *lineptr[], int nlines)
{
    int j;

    for (j=0; j<nlines; j++)
        printf("%s\n", lineptr[j]);
    /* ==> Po każdej linii – dopisujemy '\n'! */
}
```

Dalej — ***funkcja czytająca wiersze:***

```

int readlines(char *lineptr[], int maxlines)
{
    static char text[ALLOCSIZE]; /* tablica statyczna! */
    int c, nlines=0;
    long j=0;

    lineptr[0] = &text[0];
    while (EOF != (c=getchar())) {
        if (j >= ALLOCSIZE || nlines >= maxlines)
            return -1;
        else if (c != '\n')
            text[j++] = c;
        else {
            text[j++] = '\0';    nlines++;
            lineptr[nlines] = &text[j]; /* 1. ZA tablica-OK! */
        }
    }
    return nlines;
}

```

## Maksymalny rozmiar obiektów *static*

Ponieważ obiekty (*zmienne, tablice*) poprzedzone kwalifikatorem ***static*** istnieją przez cały czas działania programu, wiele systemów przewiduje pewne ograniczenie dla ich rozmiaru (typowo, dopuszczalny rozmiar jest większy niż dla „zwykłych” tablic).

W systemach *UNIX / linux / MacOS* — dopuszczalny rozmiar definiują zmienne środowiskowe; odpowiednie wartości wyświetlamy pisząc polecenie:

```
$ ulimit -a
```

[ *Dalsze ograniczenia — wynikają z architektury komputera; w systemach 32 bitowych blok danych na ogół  $\leq 2$  GB. ]*

```
% ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)          unlimited
-d: data seg size (kbytes)      unlimited
-s: stack size (kbytes)         8192
-c: core file size (blocks)     0
-v: address space (kbytes)      unlimited
-l: locked-in-memory size (kbytes) unlimited
-u: processes                    2784
-n: file descriptors            256
```

Wartość `ulimit -s` to rozmiar stosu, który określa (w szczególności) rozmiar zmiennej automatycznej (*tutaj* — 8MB).

Wielkość tablic klasy `static` określa rozmiar segmentu danych `ulimit -d` tutaj — bez ograniczeń.

**Zmiana wartości:** `ulimit -d NEW_VALUE`

Funkcja **porządkująca alfabetycznie** używa omawianego już kilkakrotnie algorytmu **quicksort**.

*Drobne różnice* — w porównaniu z podanymi wcześniej implementacjami dla różnych *tablic liczbowych* — sprowadzają się do: (1) zastąpienia wyrażenia porównania:  $(v[i] < v[j])$  wywołaniem funkcji `strcmp(v[i], v[j])` (zwraca wartość ujemną, jeśli  $v[i]$  jest *leksykalnie mniejsze* od  $v[j]$ ); oraz (2) oczywistych *zmian typów* zmiennych.

Efektywność porządkowania wynika w dużej mierze z faktu, że — niezależnie od tego, jak odległe w pamięci maszyny są porządkowane napisy — wskaźniki zebrane w tablicy zawsze zajmować będą *ciągły (i stosunkowo mały)* blok pamięci; zatem ich przemieszczanie będzie odbywać się szybko.



```

void qsortlines(char *v[], int left, int right)
{
    int i, last;
    void swap2lines(char *v[], int i, int j);

    if (left >= right) return;    /* koniec! */
    swap2lines(v, left, (left + right)/2); /* wybr. el.*/
    last = left;    /* ostatni el. < wybrany el. */
    for (i=left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap2lines(v, ++last, i);

    swap2lines(v, left, last); /* wybrany elem. wraca */
    qsortlines(v, left, last-1);
    qsortlines(v, last+1, right);
}

```

Funkcja ***zamieniające 2 elementy*** wymaga drobnej modyfikacji w porównaniu z wersjami z poprzednich wykładów:

```
void swap2lines(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

=> Każdy element tablicy `v[ ]` (czyli: `lineptr[ ]`) to *wskaźnik znakowy*, zmienna `temp` również musi mieć taki typ.

Funkcja czytająca wiersze może także zostać napisana nieco inaczej, z wykorzystaniem **dynamicznego przydziału pamięci**. (a dokładniej — funkcji `malloc`).

W tym celu, musimy włączyć w naszym programie dodatkowo nagłówek standardowy:

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

Definiujemy także maksymalną *długość pojedynczego wiersza*:

```
#define MAXLEN 1000
```

( Definicja **ALLOCSIZE** przestaje wówczas być potrzebna! )

```
int readlines(char *lineptr[], int maxlines)
{
    char s[MAXLEN+1];
    int len, nlines=0;

    while (fgets(s, MAXLEN+1, stdin)) {
        len = strlen(s);
        s[len-1] = '\0';    /* kasujemy '\n' */
        lineptr[nlines] = (char*)malloc(len*sizeof(char));
        if (nlines>=maxlines || NULL==lineptr[nlines])
            return -1;
        strcpy(lineptr[nlines++], s);
    }
    return nlines;
}
```

Do **czytania pojedynczego wiersza** użyliśmy tym razem funkcji:

```
fgets( ... )
```

której środkowym argumentem jest zawsze: maksymalna liczba znaków do przeczytania powiększona o 1.

Taka konwencja ma związek z potrzebą przechowywania znaku końca napisu ( `'\0'` ); podobnie — lokalna (teraz: *automatyczna*) tablica znakowa, służąca do przechowania ostatnio wczytanego wiersza, jest zadeklarowana jako:

```
char s[MAXLEN+1];
```

Po obliczeniu długości wczytanego wiersza `len` (funkcja `strlen`), znak końca linii ( `'\n'` ) jest zastępowany przez `'\0'`; w ten sposób skracamy każdy wiersz o jeden znak (dlatego wystarczy przydzielić pamięć na `len` znaków, a nie `len+1` !).

## Przykładowy *wynik uruchomienia programu*:

```
$ ./napisy1.out
do
re
mi
fa
sol
do so la sol
.
==> PO SORTOWANIU: <==
.
do
do so la sol
fa
mi
re
sol
```

Pomiary **czasów wykonania** 2 wersji programu dla dużego pliku:

```
$ time ./napisy1.out < AR_dokto.tex >output1.txt
==> PO SORTOWANIU: <==
.
real 0m0.020s
user 0m0.015s
sys 0m0.003s
$ time ./napisy2.out < AR_dokto.tex >output2.txt
==> PO SORTOWANIU: <==
.
real 0m0.009s
user 0m0.005s
sys 0m0.003s
```

**Identyczność plików** po sortowaniu sprawdzamy poleceniem:

```
diff output1.txt output2.txt
```

**Rozmiary plików** (źródłowego i wynikowych) użytych w przykładzie:

\$ wc	AR_dokto.tex	output*	
5025	30153	234448	AR_dokto.tex
5025	30153	234448	output1.txt
5025	30153	234448	output2.txt
15075	90459	703344	total

=> Czas wykonania wersji 2. (z dynamicznym przydziałem pamięci), na komputerze z procesorem Intel Core i7 2.3GHz, wyniósł 0.005s i okazuje się trzykrotnie krótszy od czasu wykonania wersji 1. (który wyniósł 0.015s).

Przyczyną są *narzuty czasowe* związane z obsługą dużej statycznej tablicy znakowej oraz (częściowo) czytanie danych *znak-po-znaku*.

[ *Dalsze przyspieszenie możemy uzyskać np. korzystając z funkcji bibliotecznej **qsort** zamiast jej własnej wersji => **ĆWICZENIE!** ]*



## Wskaźniki na wskaźniki

Jak pamiętamy, *nazwa tablicy* w języku C jest synonimem wskaźnika do pierwszego (=zerowego) elementu.

Zasada ta przenosi się w prosty sposób na tablice wskaźników: nazwa takiej tablicy także będzie *synonimem wskaźnika* do jej pierwszego elementu, a zatem — ***wskaźnika do wskaźnika*** do komórki pamięci zawierającej daną ustalonego typu.

Przykładowo, jeśli po deklaracjach:

```
char *tabptr[N];  
char **ptr;
```

wczytamy wiersze do `tabptr`, a następnie dokonamy przypisania

```
ptr = tabptr;
```

8. znak 4. wiersza to: `tabptr[3][7]` oraz `*(*(ptr+3)+7)`

Dla **wskaźników na wskaźniki** dopuszczalne są wszystkie operacje w ramach arytmetyki wskaźników, np.

`**ptr++`      lub *inna*:      `*(*ptr)++`

W pierwszym przypadku: wyłuskujemy 1. znak z 1. wiersza i przesuwamy `ptr` do 1. znaku 2. wiersza;

w drugim przypadku: przesuwamy `ptr[0]` do 2. znaku 1. wiersza, bez modyfikowania `ptr[1] .. ptr[N-1]`

Dalej, w analogii do zwykłych *tablic* — operacje zwiększania/zmniejszania **będą niedozwolone** dla `tabptr`, ale dla jej elementów składowych ( `tabptr[0]`, `tabptr[1]`, ... ) już tak.

[ Naturalnie, po przekazaniu do funkcji (takiej jak `readlines`) **tablica wskaźników zachowuje się jak wskaźnik na wskaźnik.** ]

# Tablice wielowymiarowe

W języku C dostępne są także tablice **dwu-** (*i więcej*) **wymiarowe**, chociaż używane są zdecydowanie rzadziej niż tablice wskaźników.

Przykładowo, w programach „*kalendarzowych*” przydać się mogą dwie listy długości miesięcy (dla lat *zwykłych* i *przestępnych*):

```
int daytab[2][13] = {  
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},  
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}  
};
```

Wówczas, po obliczeniu:

```
leap = year%4==0 && year%100!=0 || year%400==0;
```

odwołanie: `daytab[leap][j]` odczyta liczbę dni w *j*-tym miesiącu danego roku (`year`). [ **Miesiące numerujemy od 1!** ]

Kiedy **tablica dwuwymiarowa** jest przekazywana do funkcji, konieczne jest podanie długości wiersza (= liczby kolumn) w deklaracji parametrów; w naszym przykładzie — funkcja `f` operująca na tablicy `daytab` może wyglądać tak:

```
f(int daytab[2][13]) { ... };
```

lub też tak:

```
f(int daytab[][13]) { ... };
```

albo tak:

```
f(int (*daytab)[13]) { ... };
```

W ostatnim przypadku — deklaracja mówi, że parametr funkcji to **wskaźnik do tablicy** 13 liczb całkowitych.

[ *Ogólnie - przekazując tablicę wielowymiarową — możemy pominąć tylko pierwszy wymiar, pozostałe — muszą być podane jawnie.* ]

## Inicjowanie tablic wskaźników

Poniższa funkcja zwraca napis - *nazwę miesiąca* o numerze n:

```
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month", "January", "February",
        "March", "April", "May", "June", "July",
        "August", "October", "November", "December"
    };
    return (n<1 || n>12) ? name[0] : name[n];
}
```

[ Podobnie jak przy inicjowaniu tablic stałymi — podawanie rozmiarów jest niepotrzebne, kompilator sam je oblicza. ]

# Tablice wskaźników a tablice wielowymiarowe

Obie poniższe deklaracje:

```
double a[10][20];      double *b[10];
```

umożliwiają działania na dwuwymiarowej tablicy liczbowej (*macierzy!*) zawierającej 10 wierszy.

Zmienna `a` jest jednak **prawdziwą tablicą**:  $10 \times 20 = 200$  liczb typu `double` zajmujących ciągły obszar pamięci (pozycja elementu `[wiersz][kolumna]` to po prostu  **$20 \times \text{wiersz} + \text{kolumna}$**  ).

[ => *Konieczność ustalania dł. wiersza przy przekazywaniu t. do funkcji!* ]

W drugim przypadku: zmienna `b` to **tablica 10 wskaźników**, które muszą zostać powiązane z *tablicami jednowymiarowymi* gdzieś dalej w programie ( => *wiersze mogą być różnej długości!* )