

ПОХІДНІ ТИПИ ДАНИХ У МОВІ С

Лекція 04
Основи інформатики, програмування та алгоритмічні
мови



План лекції

- Масиви та вказівники.
- Робота з рядками в мові програмування C.
- Структури та інші похідні типи.



МАСИВИ ТА ВКАЗІВНИКИ

Питання 4.1.

Похідний тип: масив

- Масив утворює певна послідовність елементів одного типу даних.
 - Оголошення масиву повідомляє компілятору, скільки елементів містить масив і який тип мають його елементи.
 - `float candy[365];` `/* масив з 365 значень типу float */`
`char code[12];` `/* масив з 12 значень типу char */`
`int states [50];` `/* масив з 50 значень типу int */`
- Для доступу до елементів масиву вказується їх номер (*індекс*).
 - Нумерація елементів масиву починається з 0.
 - `candy [0]` - це перший елемент масиву `candy`,
 - `candy [364]` - 365-й, останній, елемент масиву.

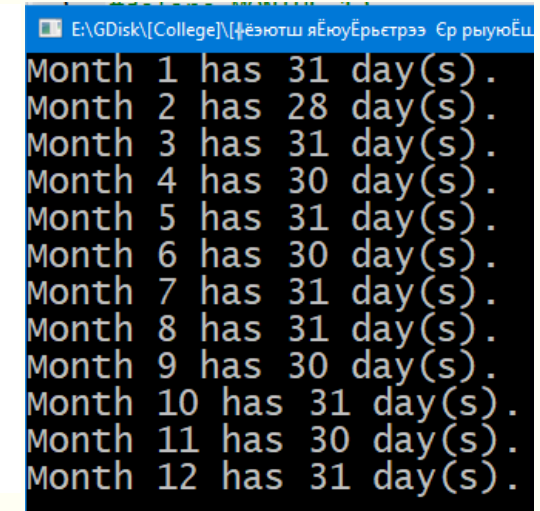
First Element						
score[0]	score[1]	score[2]	score[3]	score[4]	score[5]	score[6]
5	2	8	0	1	9	4
1000	1002	1004	1006	1008	1010	1012
Base Address						

Коли забув, що індексація масиву починається з 0



Ініціалізація масивів

```
1 #include <stdio.h>
2 #define MONTHS 12
3
4 int main (void)
5 {
6     int days [MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
7     int index;
8     for (index = 0; index < MONTHS; index++)
9         printf ( "Month %d has %2d day(s).\n", index + 1, days[index]);
10    return 0;
11 }
```



```
Month 1 has 31 day(s).
Month 2 has 28 day(s).
Month 3 has 31 day(s).
Month 4 has 30 day(s).
Month 5 has 31 day(s).
Month 6 has 30 day(s).
Month 7 has 31 day(s).
Month 8 has 31 day(s).
Month 9 has 30 day(s).
Month 10 has 31 day(s).
Month 11 has 30 day(s).
Month 12 has 31 day(s).
```

- Іноді доводиться використовувати масив, що призначений тільки для зчитування.

- `const int days [MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`

- Ініціалізація масиву без вказування його розміру

- `const int days [] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`

```
for (index = 0; index < sizeof days / sizeof days[0]; index++)
    printf("Месяц %2d имеет %d дней (день).\n", index +1,
           days[index]);
```



monthArray.c

Неініціалізовані масиви

```
1  #include <stdio.h>
2  #include <conio.h>
3  #define SIZE 4
4
5  int main (void)
6  {
7      int no_data[SIZE];
8      int i;
9      printf ("%2s%14s\n", " i " , "no_data[i]");
10
11     for (i = 0; i < SIZE; i++)
12         printf ("%2d%14d\n", i, no_data[i]);
13
14     getch();
15     return 0;
16 }
```



no_data.c

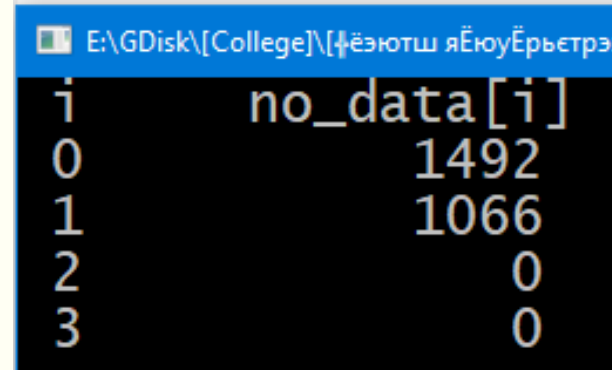
C:\Users\User\Documents\no_data.exe

i	no_data[i]
0	0
1	0
2	268501009
3	0

- Проте поведінка залежить від **класу пам'яті**, до якого належить масив.
 - За умовчанням – автоматичний клас пам'яті.
 - Для інших класів пам'яті неініціалізовані значення можуть занулятись.

Частково ініціалізовані масиви

```
1 #include <stdio.h>
2 #include <conio.h>
3 #define SIZE 4
4
5 int main (void)
6 {
7     int no_data[SIZE] = {1492, 1066};
8     int i;
9     printf ("%2s%14s\n", " i ", "no_data[i]");
10
11     for (i = 0; i < SIZE; i++)
12         printf ("%2d%14d\n", i, no_data[i]);
13
14     getch();
15     return 0;
16 }
```



i	no_data[i]
0	1492
1	1066
2	0
3	0

- Як тільки значення в списку закінчаться, решта елементів ініціалізується нулями.
 - Компілятори не допускають ініціалізації більшою кількістю елементів, ніж передбачено розміром масиву.



part_data.c

Виділені ініціалізатори (додано в стандарті C99)

- Дозволяє обирати, які елементи повинні бути ініціалізовані.

```
#include <stdio.h>
#define MONTHS 12
int main(void)
{
    int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
    int i;
    for (i = 0; i < MONTHS; i++)
        printf("%2d %d\n", i + 1, days[i]);
    return 0;
}
```

1	31
2	29
3	0
4	0
5	31
6	30
7	31
8	0
9	0
10	0
11	0
12	0

Присвоювання значень масивам

- Використовується *індекс* елементу масива.

- Приклад 1. Всі парні числа від 1 до 100

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
    int counter, evens[SIZE];
    for (counter = 0; counter < SIZE; counter++)
        evens[counter] = 2 * counter;
    ...
}
```

- Недопустимі методи присвоювання (SIZE = 5):

```
int oxen[SIZE] = {5,3,2,8};    /* здесь все в порядке */
int yaks[SIZE];

yaks = oxen;                    /* недопустимый оператор */
yaks[SIZE] = oxen[SIZE];        /* неправильно */
yaks[SIZE] = {5,3,2,8};         /* этот метод не работает */
```

Вказування розміру масиву

- До появи стандарту C99 розмір виражався лише за допомогою *константного цілочисельного виразу, значення якого більше нуля*.
 - Стандарт C99 увів масиви змінної довжини.
 - На них накладаються певні обмеження, наприклад, неможливість ініціалізації в оголошенні.

```
int n = 5;
int m = 8;
float a1[5];           // да
float a2[5*2 + 1];     // да
float a3[sizeof(int) + 1]; // да
float a4[-4];          // нет, размер должен быть > 0
float a5[0];           // нет, размер должен быть > 0
float a6[2.5];         // нет, размер должен быть целым числом
float a7[(int)2.5];    // да, преобразование типа из float int constant
float a8[n];           // не разрешалось до появления стандарта C99
float a9[m];           // не разрешалось до появления стандарта C99
```

Масиви змінної довжини значно повільніше

```
void call_me(char *stuff, int step)
{
    char buf[10];

    strcpy(buf, stuff, sizeof(buf));
    printf("%d:[%s]\n", step, buf);
}
```

```
void call_me(char *stuff, int step)
{
    char buf[step];

    strcpy(buf, stuff, sizeof(buf));
    printf("%d:[%s]\n", step, buf);
}
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %rdi,-0x18(%rbp)
mov     %esi,-0x1c(%rbp)
mov     -0x18(%rbp),%rcx
lea     -0xa(%rbp),%rax
mov     $0xa,%edx
mov     %rcx,%rsi
mov     %rax,%rdi
callq   5d0 <strcpy@plt>
lea     -0xa(%rbp),%rdx
mov     -0x1c(%rbp),%eax
mov     %eax,%esi
lea     0xd3(%rip),%rdi
mov     $0x0,%eax
callq   5c0 <printf@plt>
nop
leaveq
retq
```

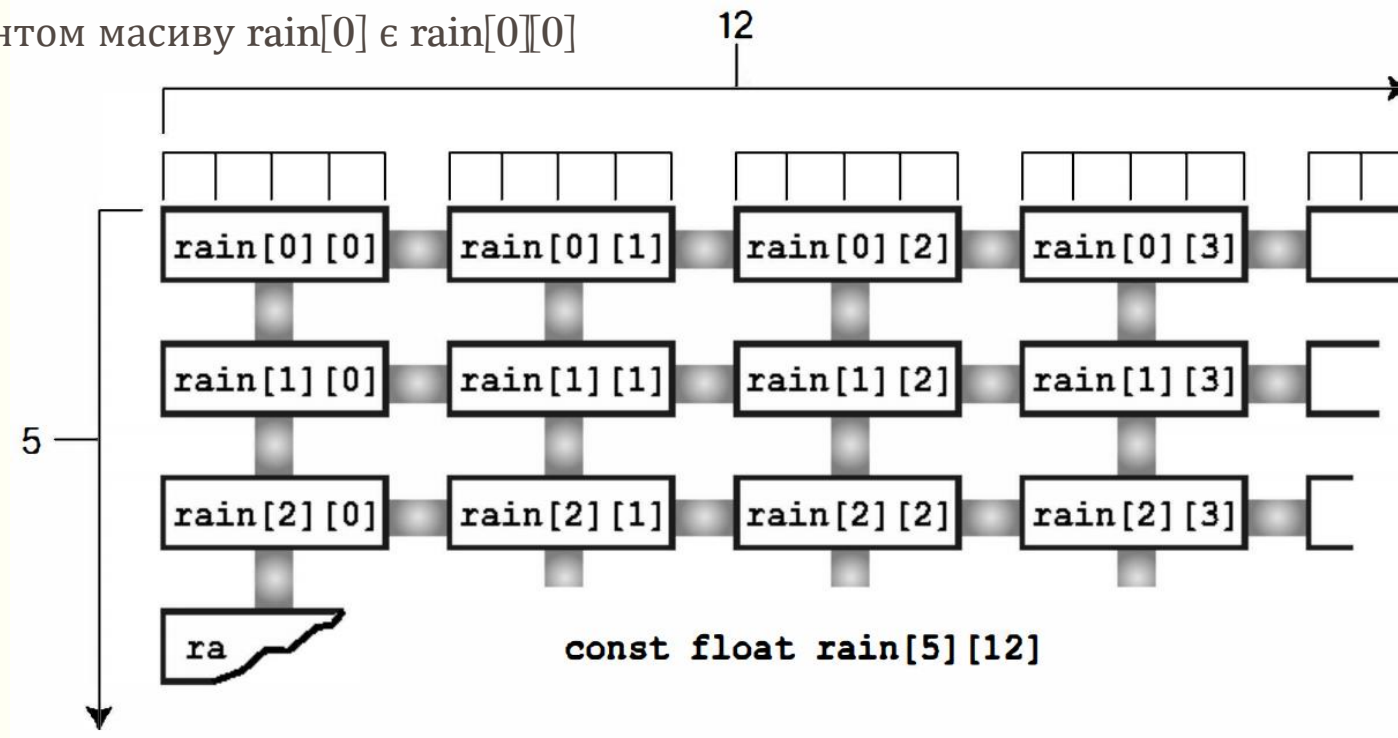
fixed-size array

variable length array

```
push    %rbp
mov     %rsp,%rbp
push    %rbx
sub     $0x28,%rsp
mov     %rdi,-0x28(%rbp)
mov     %esi,-0x2c(%rbp)
mov     %rsp,%rax
mov     %rax,%rbx
mov     -0x2c(%rbp),%ecx
movslq  %ecx,%rax
sub     $0x1,%rax
mov     %rax,-0x18(%rbp)
movslq  %ecx,%rax
mov     %rax,%r10
mov     $0x0,%r11d
movslq  %ecx,%rax
mov     %rax,%r8
mov     $0x0,%r9d
movslq  %ecx,%rax
mov     $0x10,%edx
sub     $0x1,%rdx
add     %rdx,%rax
mov     $0x10,%esi
mov     $0x0,%edx
div     %rsi
imul    $0x10,%rax,%rax
sub     %rax,%rsp
mov     %rsp,%rax
add     $0x0,%rax
mov     %rax,-0x20(%rbp)
movslq  %ecx,%rdx
mov     -0x20(%rbp),%rax
mov     -0x28(%rbp),%rcx
mov     %rcx,%rsi
mov     %rax,%rdi
callq   5d0 <strcpy@plt>
mov     -0x20(%rbp),%rdx
mov     -0x2c(%rbp),%eax
```

Багатовимірні масиви

- Нехай існує набір даних (60 значень) щодо середньомісячної кількості опадів за 5 років.
 - Зручно сформувати масив масивів (5 масивів по 12 елементів): `float rain[5][12]`
 - `rain[0]` – перший елемент масиву `rain`, є масивом на 12 значень за перший рік спостережень.
 - Першим елементом масиву `rain[0]` є `rain[0][0]`



Багатовимірні масиви

- Двовимірне представлення – лише зручний спосіб перегляду.
 - У пам'яті комп'ютера такий масив зберігається послідовно.
- Ініціалізація двовимірного масиву:

```
1 #include <stdio.h>
2 #define MONTHS 12
3 #define YEARS 5
4 int main(void) {
5     const float rain[YEARS][MONTHS] = {
6         {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
7         {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
8         {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
9         {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
10        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
11    }; // initializing rainfall data for 2010 - 2014
12    int year, month;
13    float subtot, total;
```

Можна опустити внутрішні фігурні дужки та залишити 2 зовнішні.

Якщо кількість елементів коректна, результат буде тим же.

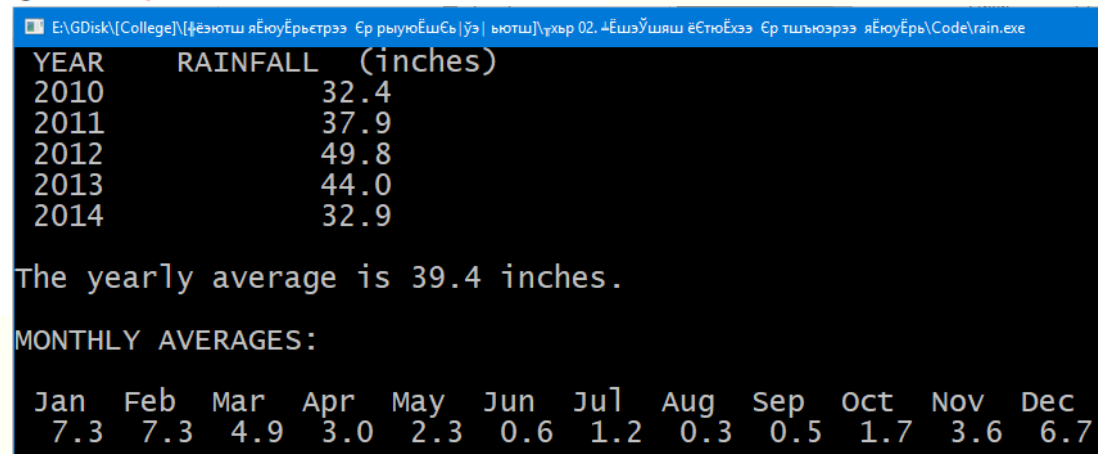


rain.c

Продовження програми

```
15 printf(" YEAR      RAINFALL  (inches)\n");
16 for (year = 0, total = 0; year < YEARS; year++) {
17     for (month = 0, subtot = 0; month < MONTHS; month++)
18         subtot += rain[year][month];
19     printf("%5d %15.1f\n", 2010 + year, subtot);
20     total += subtot; // total for all years
21 }
22 printf("\nThe yearly average is %.1f inches.\n\n", total/YEARS);
23 printf("MONTHLY AVERAGES:\n\n");
24 printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec\n ");
25
26 for (month = 0; month < MONTHS; month++) {
27     for (year = 0, subtot = 0; year < YEARS; year++)
28         subtot += rain[year][month];
29     printf("%4.1f ", subtot/YEARS);
30 }
31 printf("\n");
32
33 return 0;
34 }
```

- Для обчислення ітогової суми за рік значення year залишається незмінним, а month пробігає весь діапазон значень.
- Це внутрішній цикл for з першої частини програми.



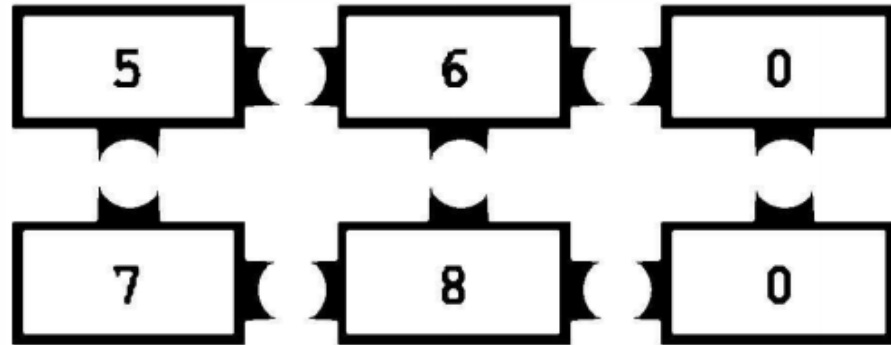
```
E:\GDisk\College\фёзюш яёюёрьстрээ Ер рыюёшсь|ўэ| ьютш|хър 02.4ёшзўшыш ёётюёхээ Ер тшьюэрээ яёюёрь\Code\rain.exe
YEAR      RAINFALL  (inches)
2010      32.4
2011      37.9
2012      49.8
2013      44.0
2014      32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

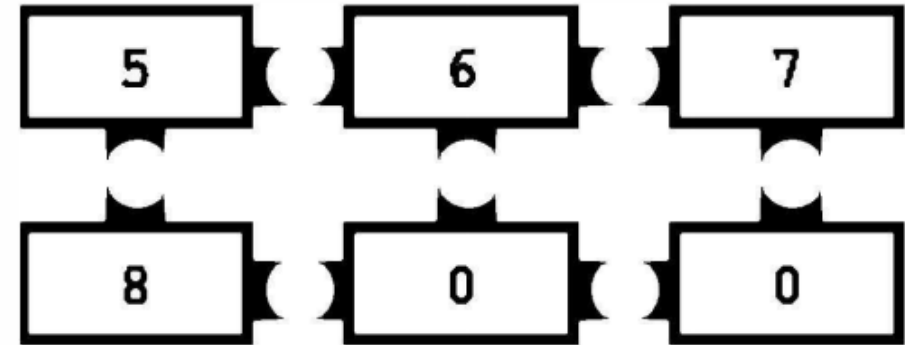
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
 7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

Два методи ініціалізації багатовимірних масивів



New:

```
int sq[2][3] = {  
    {5, 6},  
    {7, 8}  
};
```

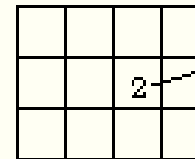


```
int sq[2][3]={5,6,7, 8};
```

Масиви розмірністю більше 2

- Растрове зображення можна представити як тривимірний масив

- `int image[64][64][3];`
- Зазвичай для обробки тривимірних масивів використовується 3 вкладених цикли

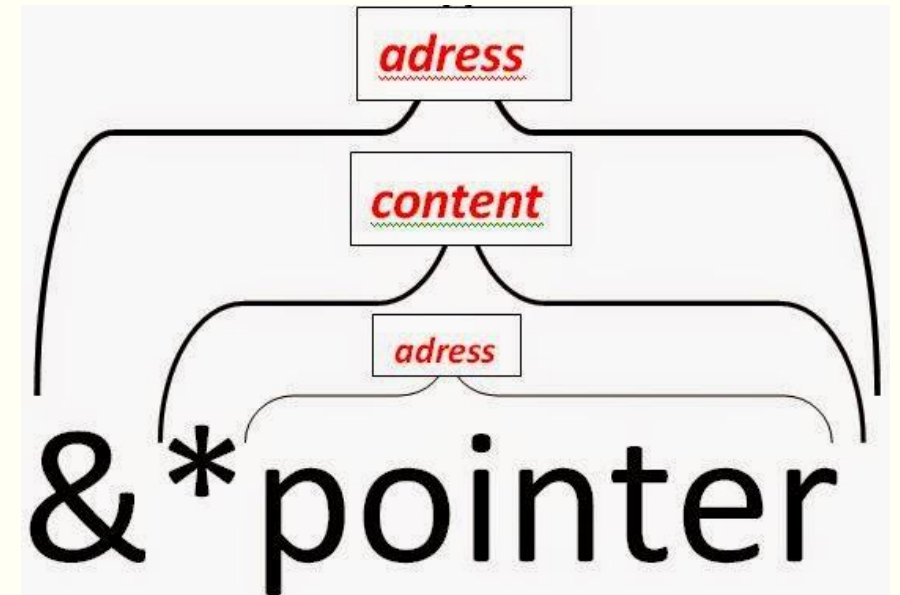


Index	Red	Green	Blue
0			
1			
2	0-255	0-255	0-255
3			
4			
14			
15			



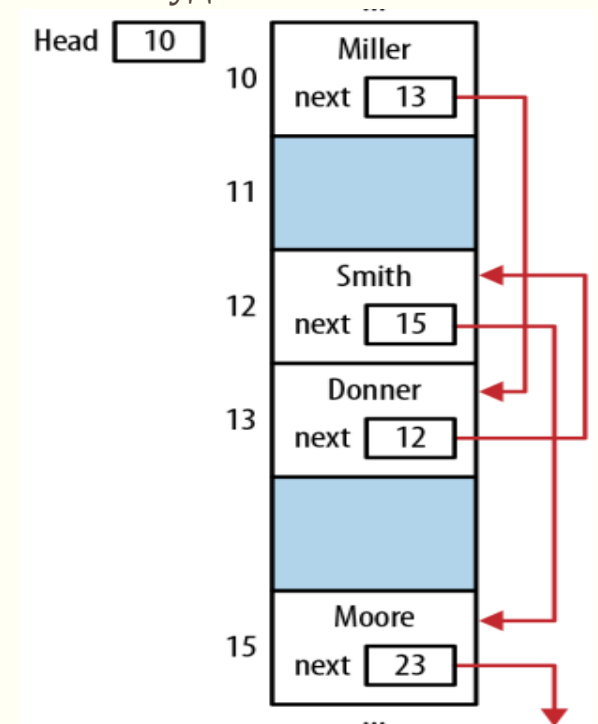
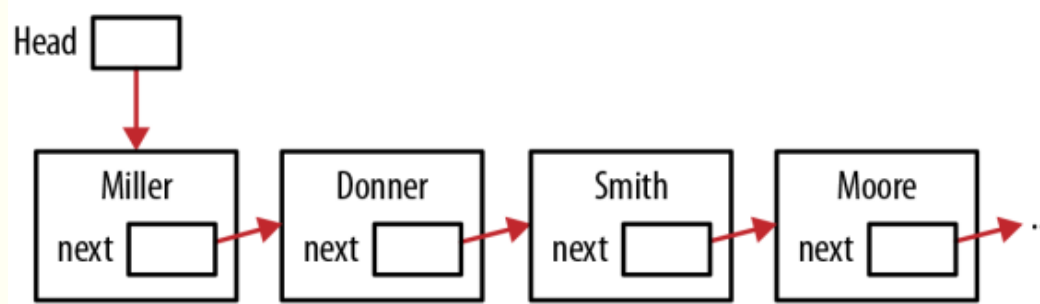
Похідний тип: вказівник

- Вказівники застосовуються в основному для маніпулювання даними в пам'яті.
 - Змінна-вказівник містить адресу в пам'яті *інших* змінної, об'єкту чи функції.
 - Зазвичай вказівник оголошується з конкретним типом, наприклад, вказівник на char.
- Використання вказівників:
 - Створення швидкого та ефективного коду
 - Підтримка динамічного виділення пам'яті
 - Більша компактність виразів
 - Можливість передачі структур даних за вказівником без даремних витрат пам'яті
 - Захист даних, переданих у якості параметра функції



Навіщо використовувати вказівники?

- Код реалізується ефективніше, оскільки вказівники ближчі до «заліза».
 - Компілятору простіше виконувати перетворення в машинний код.
 - There is not as much overhead associated with pointers as might be present with other operators.
- Багато структур даних простіше реалізувати за допомогою вказівників
 - Змінна head вказує, що у зв'язному списку на базі масиву індекс першого елемента буде 10.
 - Кожен елемент масиву містить структуру, яка представляє робітника.
 - Поле next структури містить індекс, який вказує на наступного робітника.
 - Затінені елементи представляють невикористані елементи масиву.



Оголошення вказівників

- Змінні-вказівники використовують оператор розіменування * для взяття значення за адресою (dereference):

- `int* pi;`

- `int * pi;`

- `int *pi;`

- `int*pi;`

- Вказівники на неініціалізовану (uninitialized) пам'ять можуть призвести до проблем.
 - Якщо вказівник розіменовано, його вміст, ймовірно, не представлятиме коректну адресу, а якщо все ж представлятиме, то може не містити коректних даних.
 - Некоректна адреса (invalid address) – одна з речей, до якої в програми немає доступу.
 - Програма перерве своє виконання на більшості платформ, що може призвести до цілого ряду проблем.
- Правильно читаємо: `const int *pci;`
 - Вказівник (pointer variable) на цілочисельну константу.

Оператор взяття адреси (Address of Operator) & повертає адресу свого операнду

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int num = 0;
6     int *pi = &num;
7
8     printf("Address of num: %d Value: %d\n",&num, num);
9     printf("Address of pi: %d Value: %d\n",&pi, pi);
10
11     return 0;
12 }
```

```
Address of num: 6487628 Value: 0
Address of pi: 6487616 Value: 6487628
```

- Адресами можна управляти ще на етапі оголошення змінних:

- `int num;`
`int *pi = #`



printPointer.c

- Якщо забути оператор взяття адреси, при присвоєнні може виникнути помилка:
 - error: invalid conversion from 'int' to 'int*'
- Вказівники та цілі числа можуть мати однаковий байтовий розмір, проте не є однаковими.
 - Проте можливо звести ціле число до вказівника на ціле число: `pi = (int *)num;`
 - Якщо адреса нульова, її розіменування може призвести до переривання роботи програми.

Розіменування вказівника за допомогою оператора * (Indirection Operator)

- Оператор * повертає значення, на яке вказує змінна-вказівник.

- `int num = 5;`
`int *pi = #`
 - `printf("%p\n", *pi);` *// Виведе 5*

Специфікатор	Значення
%x	Відображає значення як шістнадцяткове число
%o	Відображає значення як вісімкове число
%p	Відображає значення залежно від компілятора; зазвичай у шістнадцятковій формі

- Вказівник може вказувати на функцію:
 - Наприклад, `void (*foo)();`

Поняття про NULL

- Часто маємо справу з декількома схожими поняттями:
 - `null` як концепція
 - `null` як константний вказівник (`pointer constant`)
 - макрос `NULL`
 - ASCII-значення `NUL`
 - `null` як рядок
 - `null` як інструкція (`statement`)
- Коли `NULL` присвоюється вказівнику, вказівник ні на що не вказує (на жодну область пам'яті).
 - Два `null`-вказівника завжди рівні.
- Концепція `null` – це абстракція, що підтримується `null pointer constant`.
 - Ця константа: `0` або `(void *)0`.
 - Для програміста внутрішнє представлення `null` не важливе.

Поняття про NULL

- Макрос `NULL` – цілочисельна константа `0`, зведена до вказівника на `void`.
 - У багатьох бібліотеках: `#define NULL ((void *)0)`, проте внутрішня реалізація залежить від компілятора.
 - Оголошення може знаходитись в різних заголовкових файлах: *stddef.h*, *stdlib.h* або *stdio.h*.
- У таблиці ASCII `NUL` – це байт з усіма нулями, тобто не `null`-вказівник.
 - `Null`-рядок – це порожній рядок, він не містить символів.
 - `null statement` складається з інструкції з одним оператором ;.
 - Часто еквівалентний символу `'\0'`
- Якщо задача – присвоїти `null`-значення для `pi`, використовуємо `NULL type`: `pi = NULL;`
 - `Null`-вказівник та неініціалізований вказівник відрізняються.
 - Неініціалізований вказівник може містити будь-яке значення, а `NULL`-вказівник не посилається на жодне місце в пам'яті.
- Для `null`-вказівника ніколи не слід брати адреси, оскільки він її не має.
 - Програма припинить роботу.


Використовувати NULL чи 0 при роботі із вказівниками?

- Обидва варіанти доречні.
 - Деякі розробники віддають перевагу NULL, оскільки це нагадує про роботу із вказівниками.
- Проте NULL не слід використовувати в С-кодi, де немає вказівників.
 - Коректна робота не гарантована, а символ не визначений у стандартному С.
- Значення 0 залежить від контексту:
 - ```
int num;
int *pi = 0; // 0 – це null-вказівник, NULL
pi = #
*pi = 0; // 0 – ціле число
```
  - Нульове значення перевантажується (overloaded).

# Вказівник на void

---

- Вказівник загального призначення, що використовується для зберігання посилань (references) на будь-який тип даних (крім function pointers).
  - `void *pv;`
  - У пам'яті представляється так же, як і вказівник на `char`.
  - Ніколи не дорівнює іншому вказівнику, проте два `void`-вказівника, яким присвоєно `NULL` – еквівалентні.
- Вказівнику на `void` можна присвоїти будь-який вказівник.
  - Потім його можна звести назад до початкового типу вказівника.
  - Тоді значення буде рівним початковому значенню вказівника.
- ```
int num;  
int *pi = &num;  
printf("Value of pi: %p\n", pi);  
void* pv = pi;  
pi = (int*) pv;  
printf("Value of pi: %p\n", pi);
```

 C:\Users\User\Documents\voidPointer.exe

```
Value of pi: 000000000006AFD6C  
Value of pi: 000000000006AFD6C
```


Null-вказівник vs Void-вказівник

Null Pointer	Void Pointer
Спеціально зарезервоване <i>значення</i> вказівника	Конкретний <i>тип</i> вказівника
Доречний для всіх типів даних	Void сам є типом з розміром 1
Використовується для присвоєння 0 змінній-вказівнику довільного типу.	Використовується для зберігання адреси іншої змінної незалежно від її типу даних

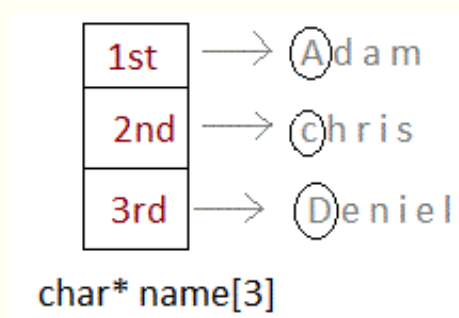
- Можливі проблеми із вказівниками:
 - Доступ до масивів чи інших структур даних за межами виділеної їм пам'яті.
 - Посилання на автоматичні змінні після того, як вони припинили своє існування.
 - Посилання на частину пам'яті, виділену в кучі, після її вивільнення (release).
 - Розіменування вказівника до того, як під нього буде виділено пам'ять.

Масиви та вказівники

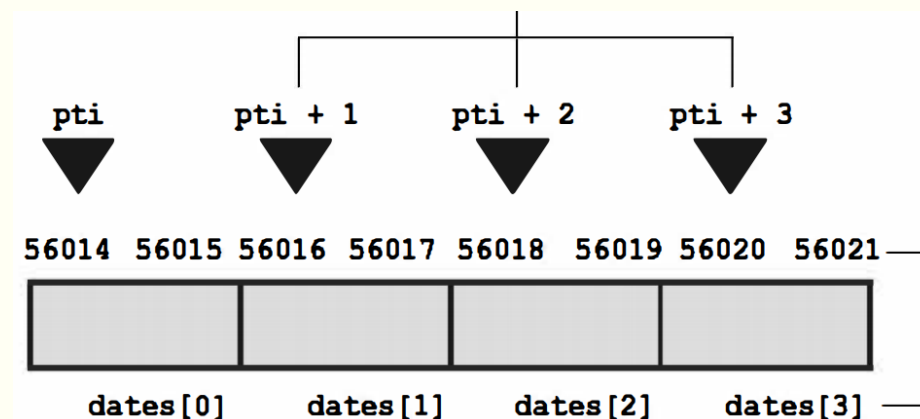
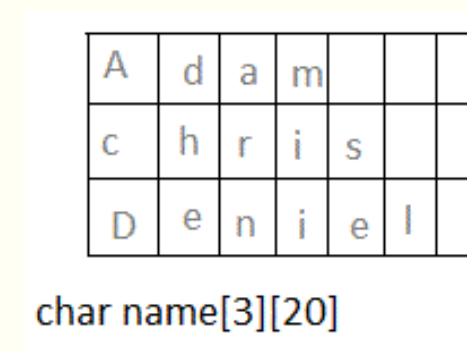
- Вказівники дозволяють ефективно працювати з масивами.

- Ім'я масиву також є адресою першого елементу масиву: `flizny == &flizny[0]; // true`
- Стандарт мови C описує масиви через вказівники: `ar[n]` означає `*(ar+n)`

- Форми запису: використовуючи вказівник:



- Не використовуючи вказівник:



Машинна
адреса

Елементи
масиву

Масиви та додавання вказівників

```
1 #include <stdio.h>
2 #define SIZE 4
3 int main(void)
4 {
5     short dates [SIZE];
6     short * pti;
7     short index;
8     double bills[SIZE];
9     double * ptf;
10
11     pti = dates;    // assign address of array to pointer
12     ptf = bills;
13     printf("%23s %15s\n", "short", "double");
14     for (index = 0; index < SIZE; index++)
15         printf("pointers + %d: %10p %10p\n", index, pti + index, ptf + index);
16
17     return 0;
18 }
```

C:\Users\spuasson\AppData\Local\Temp\Rar\$DIa0.694\pnt_add.exe

	short	double
pointers + 0:	000000000062FE30	000000000062FE10
pointers + 1:	000000000062FE32	000000000062FE18
pointers + 2:	000000000062FE34	000000000062FE20
pointers + 3:	000000000062FE36	000000000062FE28



pnt_add.c

- При додаванні 1 до вказівника мова С додає одну *одиницю зберігання*.
- `dates + 2 == &date[2]` /* та ж адреса */
- `* (dates + 2) == dates [2]` /* те ж значення */

Операції із вказівниками

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int urn[5] = {100,200,300,400,500};
5     int * ptr1, * ptr2, *ptr3;
6
7     ptr1 = urn;           // присвоєння вказівнику адреси
8     ptr2 = &urn[2];       // другий екземпляр
9     // розіменування вказівника та взяття адреси вказівника
10    printf("pointer value, dereferenced pointer, pointer address:\n");
11    printf("ptr1 = %p, *ptr1 = %d, &ptr1 = %p\n",
12           ptr1, *ptr1, &ptr1);
13    // додавання вказівників
14    ptr3 = ptr1 + 4;
15    printf("\nadding an int to a pointer:\n");
16    printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n", ptr1 + 4, *(ptr1 + 3));
17    ptr1++;               // збільшення значення вказівника на 1
18    printf("\nvalues after ptr1++:\n");
19    printf("ptr1 = %p, *ptr1 = %d, &ptr1 = %p\n", ptr1, *ptr1, &ptr1);
20    ptr2--;               // зменшення значення вказівника на 1
21    printf("\nvalues after --ptr2:\n");
22    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n", ptr2, *ptr2, &ptr2);
23    --ptr1;               // відновлення початкового значення
24    ++ptr2;               // відновлення початкового значення
25    printf("\nPointers reset to original values:\n");
26    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
27    // віднімання одного вказівника від іншого
28    printf("\nsubtracting one pointer from another:\n");
29    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n", ptr2, ptr1, ptr2 - ptr1);
30    // віднімання цілого значення від вказівника
31    printf("\nsubtracting an int from a pointer:\n");
32    printf("ptr3 = %p, ptr3 - 2 = %p\n", ptr3, ptr3 - 2);
33
34    return 0;
35 }
```

16.10.2020

C:\Users\spuasson\AppData\Local\Temp\Rar\$Dla0.065\ptr_ops.exe

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 00000000062FE30, *ptr1 =100, &ptr1 = 00000000062FE28

adding an int to a pointer:
ptr1 + 4 = 00000000062FE40, *(ptr4 + 3) = 400

values after ptr1++:
ptr1 = 00000000062FE34, *ptr1 =200, &ptr1 = 00000000062FE28

values after --ptr2:
ptr2 = 00000000062FE34, *ptr2 = 200, &ptr2 = 00000000062FE20

Pointers reset to original values:
ptr1 = 00000000062FE30, ptr2 = 00000000062FE38

subtracting one pointer from another:
ptr2 = 00000000062FE38, ptr1 = 00000000062FE30, ptr2 - ptr1 = td

subtracting an int from a pointer:
ptr3 = 00000000062FE40, ptr3 - 2 = 00000000062FE38
```



ptr_ops.c

Базові операції із вказівниками

▪ Присвоювання вказівників.

- Вказівнику можна присвоїти адресу.
- Зазвичай це робиться з використанням назви масиву або операції адресації (&).
- У прикладі змінній ptr1 присвоюється адреса початку масива urn (000000000062FE30).
- Адреса повинна бути сумісною з типом вказівника.

▪ Визначення значення (розіменування).

- Операція * повертає значення, що зберігається в комірці, на яку посилається вказівник.
- Тому початковим значенням *ptr1 є 100, воно зберігається в комірці 000000000062FE30.

▪ Взяття адреси вказівника.

- Операція & визначає, де зберігається сам вказівник.
- У прикладі ptr1 зберігається в комірці 000000000062FE28.
- Вміст цієї комірки – 000000000062FE30 – адреса масиву urn.

Базові операції із вказівниками

▪ Додавання / віднімання цілочисельного значення до вказівника.

- За допомогою операції + можна додати ціле число до вказівника або навпаки.
- Ціле число множиться на кількість байтів, що представляє тип даних, на який посилається вказівник.
- Після цього результат додається до початкової адреси.
- Результат операції `ptr1 + 4` той же, що і результат операції `&urn[4]`.
- Результат додавання не визначено, якщо він виходить за межі масиву, на який посилається початковий вказівник, за виключенням посилання на адресу, яка йде за кінцем масиву.

▪ Інкремент / декремент значення вказівника

▪ Різниця вказівників

- Зазвичай обчислюється для двох вказівників на елементи, які належать одному масиву – визначає відстань між ними.
- Гарантовано допустима операція, якщо обидва вказівники посилаються на один масив.
- Застосування до вказівників з різних масивів може дати результат, але й може призвести до помилки.

▪ Порівняння вказівників

- Використовуються оператори порівняння за умови того, що вказівники мають один тип.

Базові операції із вказівниками

- Зверніть увагу на існування 2 форм віднімання.
 - Можна відняти вказівник від вказівника та отримати ціле число.
 - Можна відняти ціле число від вказівника та отримати вказівник.
- При виконанні операцій інкремента та декремента вказівника слід бути обережним.
 - Комп'ютер не відстежує, чи посилається результуючий вказівник на елемент масиву.
 - Результат інкременту або декремента вказівника, який виходить за межі, не визначено.
 - Також можна розіменовувати вказівник будь-якого елемента масиву.
 - Допустимість вказівника на наступний за кінцем масиву елемент не гарантує можливості його розіменування.

Масиви (рядки) проти вказівників

- Запис у формі масиву (`m3[]`) призводить до того, що масив з 42 елементів (по одному на кожний символ +1 елемент для завершального символу `'\0'`), розміщується в пам'яті комп'ютера.
 - Кожен елемент ініціалізується відповідним символом.
 - Рядок в лапках знаходиться в статичній пам'яті – завантажується разом з програмою.
 - Проте пам'ять буде розподілена після того, як програма почне виконуватись.
 - У цей момент рядок копіюється в масив, далі компілятор розглядає назву масива `m3` як синонім адреси першого елементу масива, у даному випадку, `&m3[0]`.
- При використанні запису в формі масиву `m3` – це адресна константа.
 - Змінити значення `m3` не можна, оскільки це означає зміну місця зберігання (адреси) масиву.
 - Можна використовувати операції, подібні `m3+1`, проте операція `++m3` буде недопустимою.
 - Операція інкременту може застосовуватись тільки до імен змінних, але не констант.

Відмінності між масивами та вказівниками

- Розглянемо 2 оголошення:

- `char heart [] = "Я люблю Тилли!";`
- `char * head = " Я люблю Милли!";`
- Назва масиву `heart` є константою, а `head` - змінна.

- В обох випадках можна

- Використовувати запис у вигляді масиву

```
for (i = 0; i < 7; i++)  
    putchar(heart[i]);  
putchar('\n');  
for (i = 0; i < 7; i++)  
    putchar(head[i]);  
putchar('\n');
```

- Використовувати операцію додавання із вказівником

```
for (i = 0; i < 7; i++)  
    putchar(*(heart + i));  
putchar('\n');  
for (i = 0; i < 7; i++)  
    putchar(*(head + i));  
putchar('\n');
```

Відмінності між масивами та вказівниками

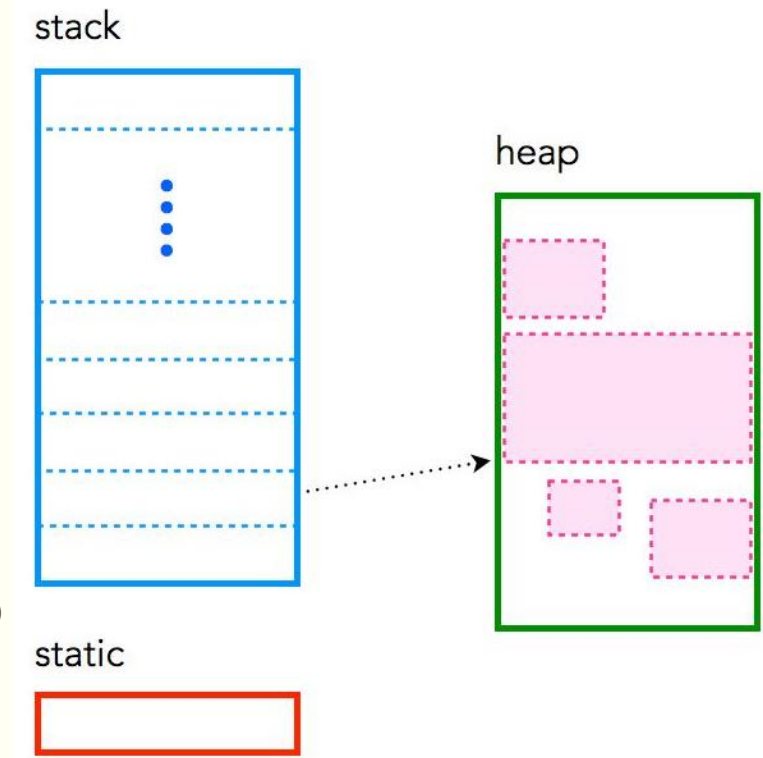
- Тільки в записі в формі вказівника може застосовуватись операція інкремента:

```
while (* (head) != '\0')    /* остановиться в кінці строки */  
    putchar (* (head++) );    /* вивести символ, переместить указатель */
```

- Нехай потрібно, щоб head і heart співпали.
 - head = heart;
 - не призводить до затирання рядка про Миллі; а тільки змінює адресу, збережену в head.
 - Проте не зберігши адресу рядка "Я люблю Милли !", не зможете отримати до нього доступ після того, як head стане вказувати на іншу комірку пам'яті.
 - Проте не можна так: heart = head;
- *Елементи масиву є змінними (якщо масив не оголошений як const), проте назва масиву змінною не є.*
 - Чи можна використати вказівник для зміни рядка?
 - За умовами стандарту C99 наслідки таких дій непрогнозовані.
 - Приклад: Прата (5-те видання, ст. 456-457).

Пам'ять та вказівники

- При компіляції робота відбувається з 3 типами пам'яті:
 - *Static/Global – у стековій пам'яті*
 - Статично оголошені змінні оголошуються в цьому виді пам'яті
 - Глобальні змінні також використовують цю область пам'яті, розміщуються в ній при запуску програми та залишаються до її завершення.
 - Доступ до глобальних змінних мають усі функції, а область видимості статичних змінних обмежується функцією, в якій вони оголошені.
 - *Automatic – у стековій пам'яті*
 - Такі змінні оголошуються всередині функції та створюються при виклику функції.
 - Їх область видимості обмежується функцією, а тривалість життя (lifetime) – періодом виконання функції.
 - *Dynamic – у кучі (heap)*
 - Пам'ять виділяється з кучі та може вивільнитись (release) за необхідності.
 - Вказівник указує на виділену (allocated) пам'ять.
 - Область видимості обмежена видимістю відповідних вказівників.
 - Тривалість життя обмежена періодом від алокації до вивільнення пам'яті, на яку вказує відповідний вказівник.



Режими виділення пам'яті в мові С: статичний, автоматичний, динамічний

- **Статична аллокація передбачає, що пам'ять для таких змінних буде виділена при запуску програми**
 - Розмір змінних фіксований.
 - Застосовується до глобальних змінних,
- **Переваги статичного виділення пам'яті**
 - Простота використання.
 - Висока швидкодія процесу виділення.
 - Відсутність потреби аллокації/реаллокації/деаллокації пам'яті.
 - Змінні залишаються перманентно аллокованими.
- **Недоліки статичної аллокації пам'яті**
 - Даремні витрати пам'яті.
 - Неможливість вивільнення пам'яті після завершення її використання.
 - Статично виділена пам'ять очищується на основі області видимості (scope) автоматично.

Динамічне виділення пам'яті (бібліотека `stdlib.h`)

Функція	Що виконує
malloc	Виділяє (allocate) пам'ять заданого розміру та повертає вказівник на перший байт аллокованого простору
calloc	Виділяє пам'ять для елементів масиву. Ініціалізує елементи нулями та повертає вказівник на масив у пам'яті
realloc	Використовується для зміни розміру попередньо виділеного обсягу пам'яті
free	Очищає (вивільняє) попередньо аллоковану пам'ять

- Функція `malloc()` повертає значення типу `void *`.
 - `int *sieve = (int *) malloc(sizeof(int) * length);`
 - Раніше зведення типу було обов'язковим, тепер код без нього вже безпечний:
`int *sieve = malloc(sizeof(int) * length);`
 - Ще один варіант запису:
`int *sieve = malloc(length * sizeof *sieve);`

Функція free()

- При динамічній аллокації виконувати очищення пам'яті потрібно явно.
 - Інакше можуть трапитись помилки роботи з пам'яттю.
 - Функція free() викликається, щоб очистити (release/deallocate) пам'яті.

```
#include <stdio.h>

int main() {
    int* ptr = malloc(10 * sizeof(*ptr));
    if (ptr != NULL) {
        *(ptr + 2) = 50;
        printf("Value of the 2nd integer is %d",*(ptr + 2));
    }
    free(ptr);
}
```

- Відсутність free() призведе до витоку пам'яті у зв'язку з недосяжними аллокаціями (Unreachable allocations)

Функція calloc()

- Скорочена назва від «contiguous allocation» – неперервний розподіл пам'яті.
 - Функція використовується для виділення **кількох** (на відміну від malloc) блоків пам'яті.
 - Використовується для аллокації пам'яті для складних структур даних: масивів, структур тощо.
 - Кожен виділений блок матиме однаковий розмір.
- Синтаксис:
 - `ptr = (cast_type *) calloc (n, size);`
 - Виділяється n блоків пам'яті з розміром кожного рівним size.
 - Після аллокації всі байти зануляються.
 - Повертається вказівник на перший байт виділеної пам'яті.
 - При помилці аллокації повертається null-вказівник.

Приклад використання функції calloc()

```
1 #include <stdio.h>
2 int main() {
3     int i, * ptr, sum = 0;
4     ptr = calloc(10, sizeof(int));
5     if (ptr == NULL) {
6         printf("Error! memory not allocated.");
7         exit(0);
8     }
9     printf("Building and calculating the sequence sum of the first 10 terms \n ");
10
11     for (i = 0; i < 10; ++i) { * (ptr + i) = i;
12         sum += * (ptr + i);
13     }
14     printf("Sum = %d", sum);
15
16     free(ptr);
17
18     return 0;
19 }
```

E:\GDisk\College\фёютш яёюёрьстрэ ёр рыуюёшсь|ўэ|ютш\ухьр 03. 4юї|фэ| ёшяш фрэші є ют|т\Code\test_calloc.exe

building and calculating the sequence sum of the first 10 terms n Sum = 45

Process exited after 0.04416 seconds with return value 0

для продолжения нажмите любую клавишу . . .

Функція realloc()

- Використовується для розширення або урізання вже виділеної пам'яті.
- Синтаксис
 - `ptr = realloc (ptr, newsize);`
 - Виділяється новий простір пам'яті заданого розміру `newsize`.
 - Після виконання функції повертається вказівник на перший байт блоку пам'яті.
 - Немає гарантії, що новий виділений блок буде за тою ж адресою, що й раніше.
 - Функція скопіює всі попередні дані в нову область пам'яті та переконається, що вони в безпеці.

```
1  #include <stdio.h>
2  int main () {
3      char *ptr;
4      ptr = (char *) malloc(10);
5      strcpy(ptr, "Programming");
6      printf(" %s, Address = %u\n", ptr, ptr);
7
8      ptr = (char *) realloc(ptr, 20); //ptr is reallocated with new size
9      strcat(ptr, " In 'C'");
10     printf(" %s, Address = %u\n", ptr, ptr);
11     free(ptr);
12     return 0;
13 }
```

Статичне та динамічне виділення пам'яті

Статична аллокація пам'яті	Динамічна аллокація пам'яті
Пам'ять виділяється ДО початку виконання програми	Пам'ять виділяється ПІД ЧАС виконання програми
Дій з аллокації або деаллокації під час виконання програми не відбувається	Зв'язки в пам'яті (memory bindings) встановлюються та руйнуються під час виконання програми
Змінні залишаються перманентно аллокованими	Пам'ять аллокована, поки активний відповідний програмний модуль
Реалізується через стеки та кучі	Реалізується через сегменти даних
Вища швидкодія	Нижча швидкодія
Вимагає більше пам'яті	Потребує менше пам'яті

Самостійне опрацювання

- Прата, 6те видання, ст. 397-409



ДЯКУЮ ЗА УВАГУ!

Вказівники та багатовимірні масиви

- Проаналізуємо в термінах вказівників запис `int zippo[4][2];`
 - `zippo` – назва масиву та адреса першого елемента в цьому масиві: `zippo` дорівнює `&zippo[0]`;
 - тут `zippo` сам є масивом з двох значень типу `int`: `zippo[0]` дорівнює `&zippo[0][0]`;
 - `zippo[0]` — это адрес объекта с размером значения `int`, а `zippo` — адрес объекта с размером двух значений `int`.
- Добавление 1 к указателю или адресу дает значение, которое больше исходного на размер указываемого объекта.
 - В этом отношении `zippo` и `zippo[0]` отличаются друг от друга, потому что `zippo` ссылается на объект с размером в два значения `int`, а `zippo[0]` — на объект с размером в одно значение `int`.
 - Таким образом, `zippo + 1` имеет значение, не совпадающее с `zippo[0] + 1`
- Разыменование указателя или адреса (применение операции `*` или операции `[]` с индексом) дает значение, представленное объектом, на который производится ссылка.
 - `*(zippo[0])` представляет значение, хранящееся в `zippo[0][0]`, т.е. значение `int`.
 - Аналогично, `* zippo` представляет значение своего первого элемента (`zippo[0]`), но `zippo[0]` сам по себе — адрес значения `int`. Это адрес `&zippo[0][0]`, так что `*zippo` является `&zippo[0][0]`.
 - При м е н е н и е операции разыменования к обоим выражениям предполагает, что `**zippo` равно `*&zippo[0][0]`, что сокращается до `zippo[0][0]`, т.е. Значения типа `int`.
 - Короче говоря, `zippo` — это адрес адреса, и для получения обычного значения потребуется двукратное разыменование.
 - Адрес адреса или указатель на указатель представляют собой п р и м е р ы двойной косвенности.

Тема доповіді



Тема доповіді: проблеми безпеки та некоректне використання вказівників



Тема доповіді: Рядкові функції в мові програмування С. Бібліотека strings.h