Лекція 6.
Пам’ять. Вказівники. Адреси.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Що таке пам’ять?
2. Що таке адреса?
3. Що таке вказівник?
4. Що таке посилання?
5. Динамічне виділення пам’яті.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Що таке пам’ять?

Що таке пам’ять?

Пам’ять - це ресурс комп'ютера, який використовується для зберігання даних під час виконання програми.

Види пам’яті

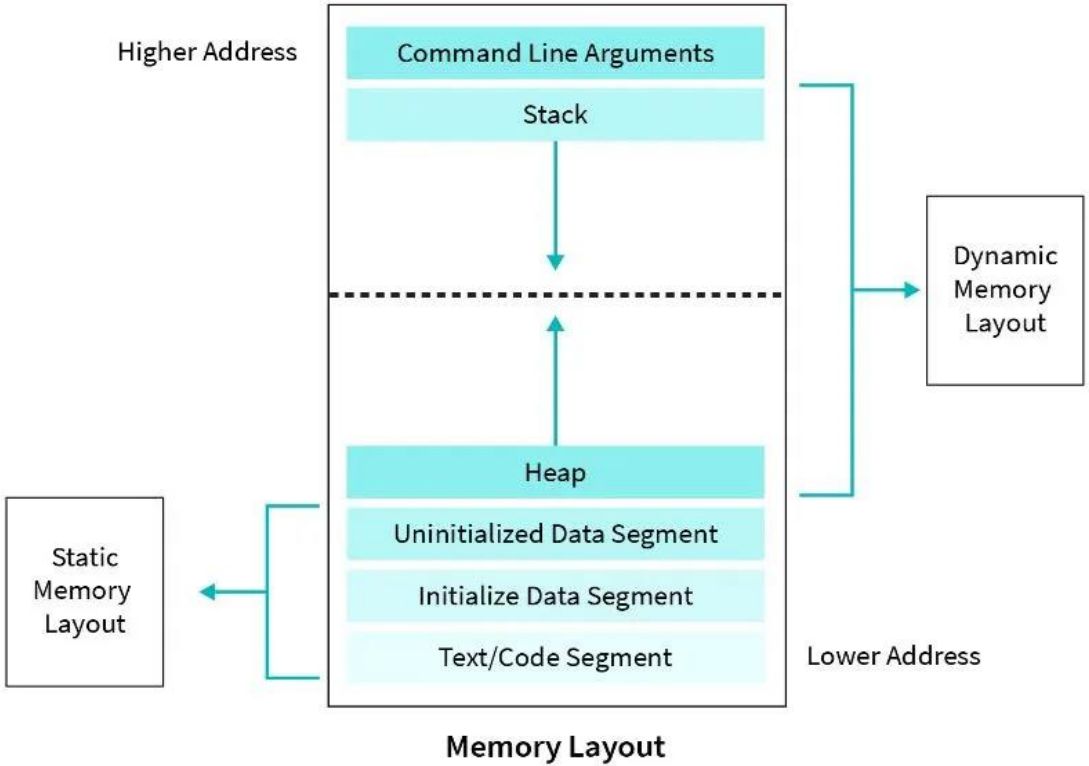
Стек (Stack)

Стек забезпечує швидкий доступ до пам'яті, яка виділяється і звільняється автоматично, використовується для локальних змінних і параметрів функцій, але має обмежений розмір.

Купа (Heap)

Купа підходить для великих або змінних об'єктів, що змінюються під час виконання програми. Купа, дозволяє динамічно виділяти пам'ять за допомогою new, але потребує ручного звільнення через delete.

Структура пам’яті



Структура пам’яті

- **Text Segment**: зберігає скомпільований машинний код (тільки для читання).
- **Initialized Data Segment**: зберігає явно ініціалізовані глобальні та статичні змінні.
- **Uninitialized Data Segment (BSS)**: зберігає неініціалізовані глобальні та статичні змінні, нуль за замовчуванням.
- **Heap**: використовується для динамічного розподілу пам’яті, керується програмістом вручну.
- **Stack**: зберігає локальні змінні, параметри функції та адреси повернення, керовані автоматично за допомогою викликів функцій і повернення.

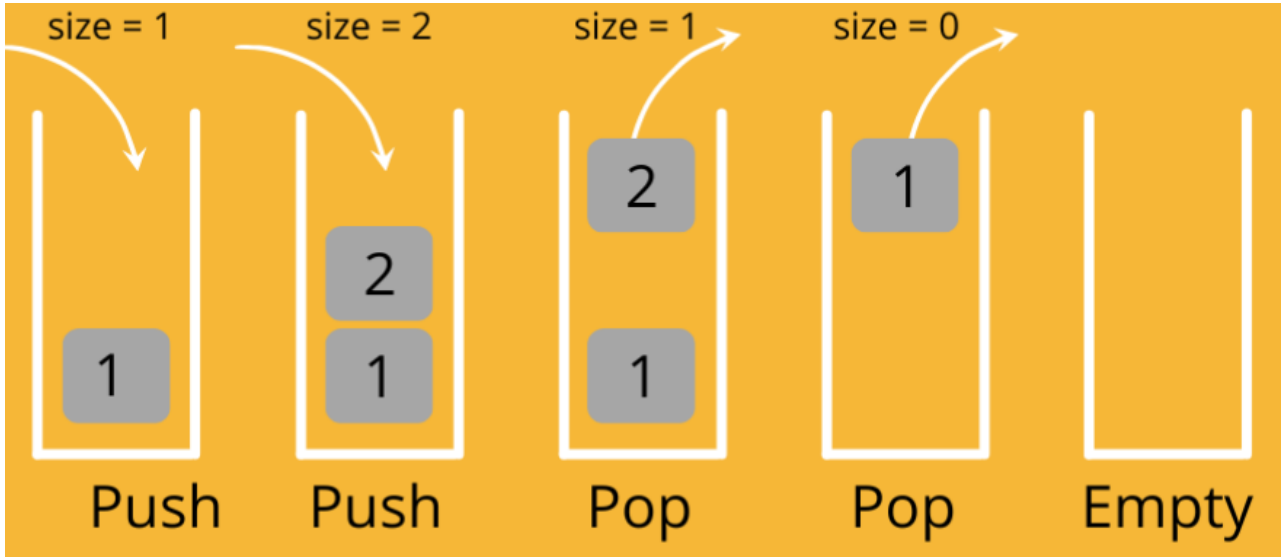
Структура пам’яті

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

int initialized_global = 42; // Initialized Data Segment
int uninitialized_global;    // Uninitialized Data Segment (BSS)

int main() {
    static int initialized_static = 100; // Initialized Data Segment
    static int uninitialized_static;     // Uninitialized Data Segment (BSS)

    const size_t size = 1000000;         // Stored in stack
    int8_t nStack[size];                 // Stored in stack

    int* dynamic_array = (int*)malloc(5 * sizeof(int)); // Memory allocation in heap
    if (dynamic_array == nullptr) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }
    for (int i = 0; i < 5; i++) {
        dynamic_array[i] = i * 10;
        cout << dynamic_array[i] << ' ';
    }
    free(dynamic_array);                 // Memory free in heap

    cout << "\nOkay!";

    return 0;
}
```

Call Stack - LIFO (LAST IN FIRST OUT)

Розмір **call stack** на **linux** можна визначити командою **ulimit -s**

Windows stack is 1MB by default, Linux and Mac 8MB.



Приклад Stack Overflow

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int8_t nStack[1028558];  //1028557
6      //cout<<"Okay!";
7      return 0;
8  }
9
10
11
12
13
14
15
16
17
18
19
```

Exception Unhandled

Unhandled exception at 0x00007FF6CE782787 in ConsoleApplicationDynArrays.exe: 0xC00000FD: Stack overflow (parameters: 0x0000000000000001, 0x000000FF39673000).

Show Call Stack | Copy Details | Start Live Share session...

▷ Exception Settings

Що таке адреса?

Що таке адреса?

Адреса в C++ — це місце розташування змінної в пам'яті комп'ютера. Кожна змінна, яку ми оголошуємо в програмі, має свою унікальну адресу в пам'яті, яка визначає, де зберігаються її значення.

Якщо у нас є змінна var у програмі, вираз &var повертає її адресу в пам'яті.

Розмір адреси

Основна відмінність між **64-розрядною** системою та **32-розрядною** системою полягає в розмірі адрес пам'яті. **64-розрядна** система використовує **64-розрядні** адреси пам’яті, що дозволяє отримати набагато більший адресний простір пам’яті порівняно з обмеженням у 4 гігабайти (ГБ) для **32-розрядної** системи.

```cpp
#include <iostream>
using namespace std;
int main()
{
    // declare variables
    int var1 = 3;
    short int var2 = 5;
    double var3 = 17.5;

    // print addresses
    cout << "Address var1: " << &var1 << ", adress size: " << sizeof(&var1)
         << ", var1 size: "<< sizeof(var1) << endl;
    cout << "Address var2: " << &var2 << ", adress size: " << sizeof(&var1)
         << ", var2 size: " << sizeof(var2) << endl;
    cout << "Address var3: " << &var3 << ", adress size: " << sizeof(&var1)
         << ", var3 size: " << sizeof(var3) << endl;
}
```

```
Address var1: 00000084869DF924, adress size: 8, var1 size: 4
Address var2: 00000084869DF944, adress size: 8, var2 size: 2
Address var3: 00000084869DF968, adress size: 8, var3 size: 8
```

Що таке вказівник?

Що таке вказівник?

У C++ вказівник — це змінні, які зберігають адреси пам'яті інших змінних.

Вказівник оголошується за допомогою символу *.

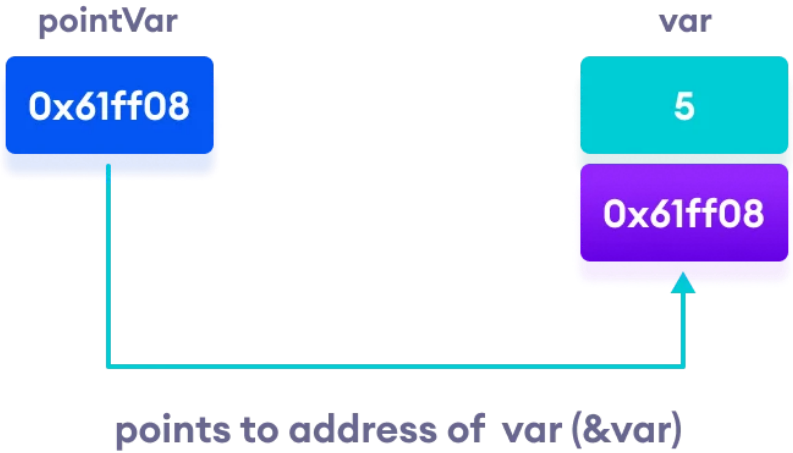
Тут ми оголосили змінну point_var, яка вказує на int.

```
int* point_var;
```

Присвоєння адрес вказівникам

Тут змінній var присвоюється значення 5. А адреса змінної var присвоюється вказівнику point_var за допомогою коду point_var = &var.

```
int var = 5;
int* point_var = &var;
```



Отримання значення за адресою за допомогою вказівників

Щоб отримати значення, на яке вказує вказівник, ми використовуємо оператор *.

Коли * використовується з вказівниками, його називають оператором розіменування. Він повертає значення, на яке вказує адреса, збережена у вказівнику.

```
int var = 5;
int* point_var = &var;
cout << *point_var;
```

Зміна значення, на яке вказують вказівники

Якщо point_var вказує на адресу змінної var, ми можемо змінити значення var, використовуючи *point_var.

```
int var = 5;
int* point_var = &var;

// change value at address
point_var
*point_var = 1;

cout << var << endl; // Output: 1
```

Ініціалізація вказівника

- **Вказівнику** можна присвоїти адресу змінної відповідного типу або **nullptr**
- **Вказівнику** можна присвоїти адресу об'єкта тільки вказаного типу (або похідного від нього)

```
int iValue = 7;
double dValue = 9.0;

int* iPtr = &iValue; // ok
double* dPtr = &dValue; // ok
short* sPtr = nullptr;

iPtr = &dValue; // error
dPtr = &iValue; // error
```

Універсальний вказівник void*

- Може приймати значення адреси об'єкта будь-якого типу, тобто йому можна присвоїти значення будь-якого іншого вказівника
- При розіменуванні обов'язкове приведення до конкретного типу

```
int n = 10;
float f = 25.25;
char c = '$';
void* ptr;
ptr = &n;          // pointing to int
ptr = &f;          // pointing to
float
ptr = &c;          // pointing to
char
```

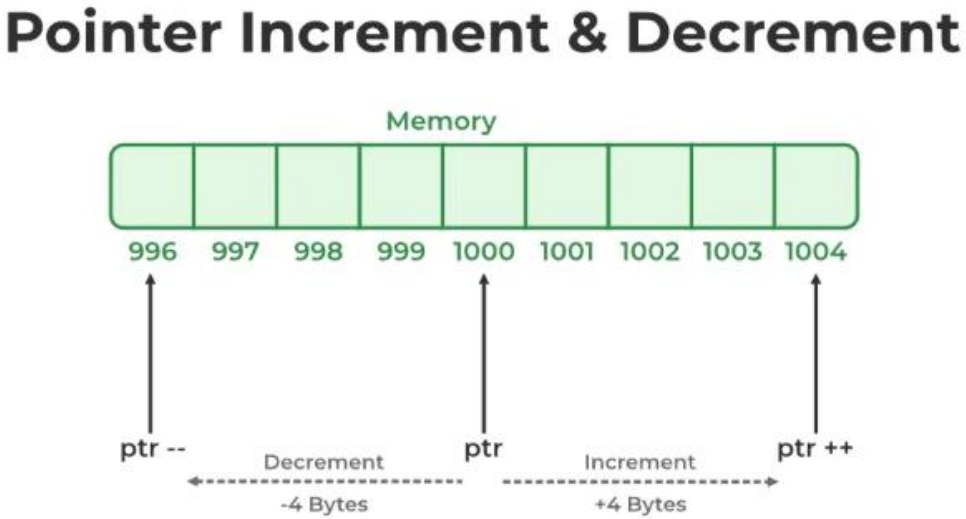
Операції над вказівниками.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

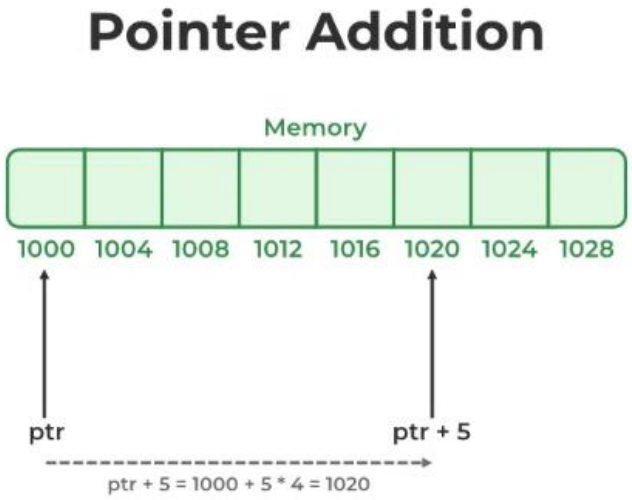
Операції над вказівниками

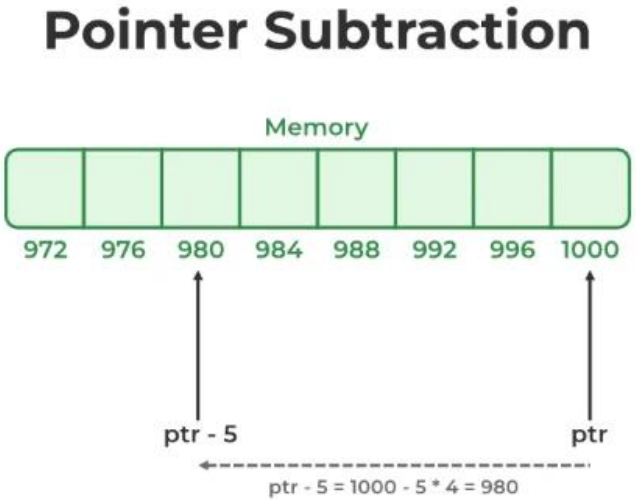
`T* pointer;`	визначення вказівника на тип T
`*pointer`	розіменування – отримання значення змінної, адресу якої містить вказівник
`pointer + n`	зміщення на **sizeof (T)*n** адрес вправо
`pointer - n`	зміщення на **sizeof (T)*n** адрес вліво
`pointer++`	зміщення на **sizeof (T)** вправо
`++pointer`	
`pointer--`	зміщення на **sizeof (T)*n** вліво
`--pointer`	
`pointer = pointer1`	присвоєння
`pointer == pointer1`	порівняння

Операції над вказівниками



Операції над вказівниками





Порівняння вказівників

```cpp
#include <iostream>
using namespace std;
int main()
{
    // declaring some pointers
    int num = 10;
    int* ptr1 = &num;
    int** ptr2 = &ptr1;
    int* ptr3 = *ptr2;

    // comparing equality
    if (ptr1 == ptr3) {
        cout << "Both point to same memory location";
    }
    else {
        cout << "ptr1 points to: " << ptr1 << endl;
        cout << "ptr3 points to: " << ptr3 << endl;
    }

    cout << endl;
    return 0;
}
```

```
Both point to same memory location
```

Константний вказівник

```cpp
#include <iostream>
using namespace std;
int main(void)
{
    {
        int var1 = 0;
        int var2 = 10;
        const int* ptr = &var1;
        ptr = &var2; // OK
        //*ptr = 1; // ERROR: ptr is a pointer to constant int
        cout << "*ptr = " << *ptr << endl;
    }
    {
        int var1 = 0;
        int var2 = 10;
        int* const ptr = &var1;
        //ptr = &var2; // ERROR: ptr is a constant pointer to int
        *ptr = 1; // OK
        cout << "*ptr = " << *ptr << endl;
    }
    {
        int var1 = 0;
        int var2 = 10;
        const int* const ptr = &var1;
        //ptr = &var2; // ERROR: ptr is a constant pointer to int
        //*ptr = 1; // ERROR: ptr is a pointer to constant int
        cout << "*ptr = " << *ptr << endl;
    }

    return 0;
}
```

```
*ptr = 10
*ptr = 1
*ptr = 0
```

Вказівники і масиви.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Вказівники і масиви

Вказівник може зберігати не лише адресу окремої змінної, але й адреси елементів масиву.

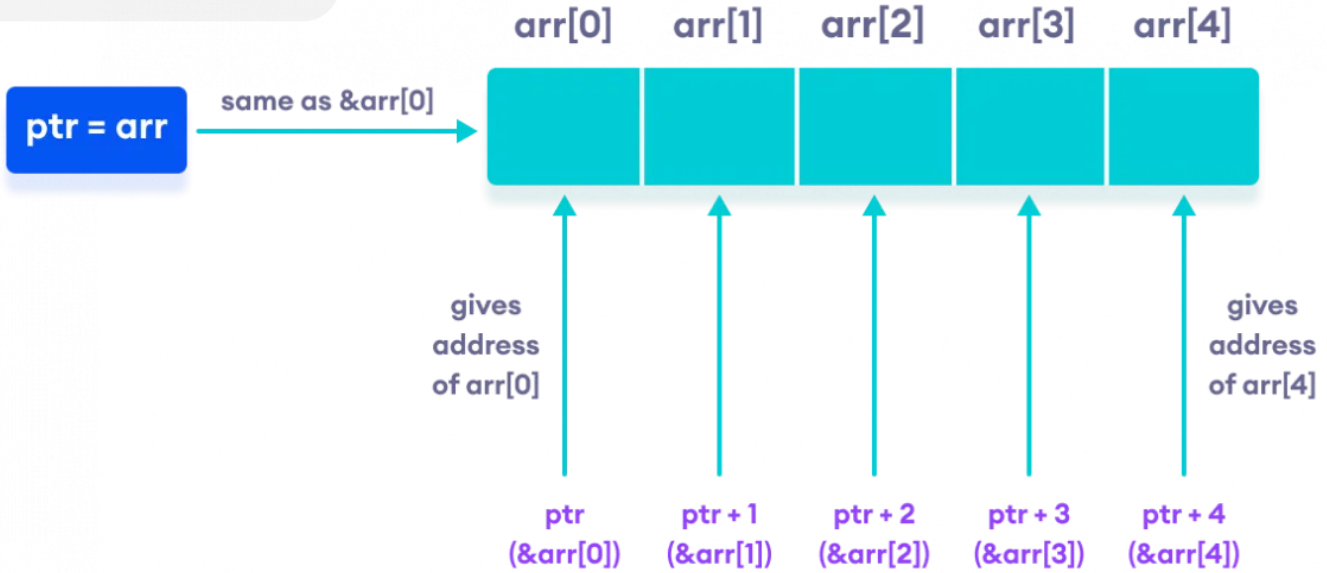
Тут ptr є змінною-вказівником, а arr — масивом типу int. Код ptr = arr; зберігає адресу першого елемента масиву в змінній ptr.

Зверніть увагу, що ми використовуємо arr замість &arr[0], оскільки це одне й те саме.

```
int *ptr;
int arr[5];

// store the address of the
first
// element of arr in ptr
ptr = arr;
// save as
//ptr = &arr[0];
```

Вказівники і масиви

Якщо ptr вказує на перший елемент у масиві, тоді ptr + 3 буде вказувати на четвертий елемент.



Вказівники і масиви

Щоб працювати з двовимірними масивами в C++ за допомогою вказівників, можна використовувати адресу першого елемента масиву. Наприклад, якщо маємо масив arr[3][3], то ptr = &arr[0][0] присвоює вказівнику адресу першого елемента. Вказівник ptr + 1 буде вказувати на наступний елемент у пам'яті, тобто arr[0][1].

Аналогічно, щоб перейти до наступного рядка масиву, можна скористатися виразом ptr + 3, де 3 — це кількість елементів у рядку, що дасть адресу arr[1][0]. Таким чином, вказівники дозволяють ітерувати як по рядках, так і по стовпцях двовимірного масиву.

Що таке посилання?

Що таке посилання?

У C++ ми використовуємо посилання (&), щоб створити псевдонім для змінної. Ми можемо використовувати змінну-посилання для доступу до змінної або її зміни.

- Визначає **константний вказівник**, який автоматично розіменовується
- На відміну від **вказівника посилання** завжди ініціалізується при визначенні

string — тип даних змінної
& — позначає, що ми створюємо посилання
ref_city — ім'я змінної-посилання
city — змінна, для якої створено посилання

```
string& ref_city = city;
```

Зміна змінних за допомогою посилань

Можна змінити значення змінної, просто присвоївши нове значення змінній-посиланню.

```
string& ref_city = city;
ref_city = "New York";
```

Костантне посилання

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

int main()
{
    {
        string val1 = "Test";
        string& val2 = val1;
        val2 = "Hello";          // OK
        cout << val2 << endl;
    }
    {
        string val1 = "Test";
        const string& val2 = val1;
        //val2 = "Hello";          // ERROR: const string can't be changed
        cout << val2 << endl;
    }
    {
        //string& val1 = "Test"; // ERROR: The literal "Hello" value is a rvalue
                                 //(temporary value that expires after the line ends).

        const string& val2 = "Hello"; // OK: const reference prolongates life time of temp value
        cout << val2 << endl;
    }

    return 0;
}
```

- **Посилання може** вказувати лише на значення яке вже зберігається в пам’яті **(lvalue)**
- **Константне посилання** продовжує життя **тимчасового об’єкту (rvalue)**

Динамічне виділення пам’яті.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Динамічне виділення пам’яті

C++ дозволяє виділяти пам'ять динамічно під час виконання програми, що називається динамічним виділенням пам'яті.

У C++ ми повинні вручну звільняти динамічно виділену пам'ять, коли вона більше не потрібна.

Ми можемо виділяти і звільняти пам'ять динамічно за допомогою операторів new і delete відповідно.

Оператор new

Синтакс: data_type* pointer_variable = new data_type{value};

Тут ми виділяємо пам'ять для змінної типу int за допомогою оператора new.

Ми використовували вказівник point_var для динамічного виділення пам'яті, оскільки оператор new повертає адресу виділеної області пам'яті.

```
// declare an int pointer
int* point_var;

// dynamically allocate memory
// using the new keyword
point_var = new int;

// assign value to allocated memory
*point_var = 45;
```

Оператор delete

Синтакс: delete pointer_variable;

Коли змінна, яку ми оголосили динамічно, більше не потрібна, ми можемо звільнити пам'ять, яку вона займає. Для цього можна використовувати оператор delete, який повертає пам'ять операційній системі. Це називається звільненням пам'яті.

```
// dynamically allocate memory
// and assign int variable
int* point_var = new int{45};

// print the value stored in memory
cout << *point_var; // Output: 45

// deallocate the memory
delete point_var;

// set pointer to nullptr
point_var = nullptr;
```

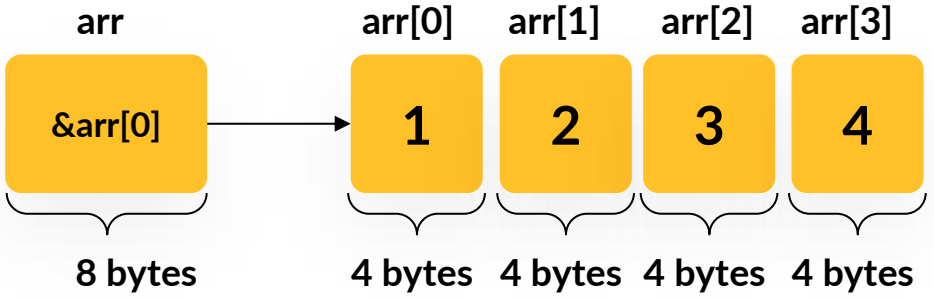
Робота з об'єктами в динамічній пам’яті

#	Назва оператора	Формат оператора
3	утворити об'єкт у купі	**new** *type*
	утворити об'єкт у купі і проініціалізувати	**new** *type* **(** expr-list **)**
	утворити масив об'єктів у купі і проініціалізувати	**new** *type* **[** *expr* **]**
	знищити об'єкт у купі (звільнити пам'ять)	**delete** pointer
	знищити масив об'єктів у купі (звільнити пам'ять)	**delete[]** pointer

Одновимірні динамічні масиви

```
size_t n = 4;
// Memory allocation and initialization
int *arr = new int[n]{1, 2, 3, 4};
for (size_t i = 0; i < n; ++i) {

        cout << *(arr + i) << " "; // arr[i]
}
// Memory deallocation
delete[] arr;
```

```
1 2 3 4
```



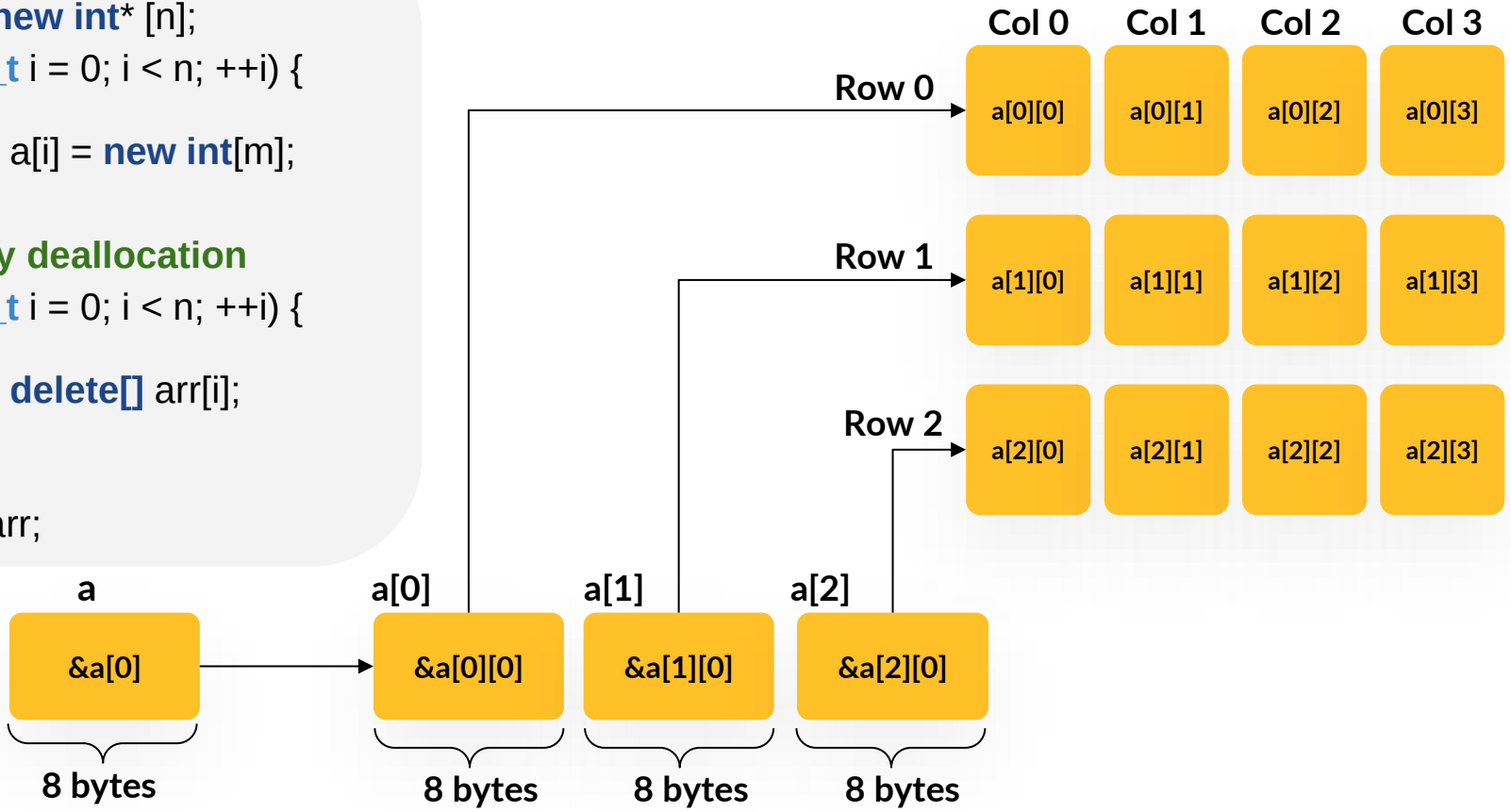
Одновимірні динамічні масиви

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


int main() {
    size_t n;
    cout << "Enter the number of items:" << "\n";
    cin >> n;
    int* arr = new int[n];

    cout << "Enter " << n << " items" << endl;
    for (size_t i = 0; i < n; ++i) {
        cout << "arr[" << i << "]=";
        cin >> *(arr+i); // arr[i]
    }

    cout << "You entered: ";
    for (size_t i = 0; i < n; ++i) {
        cout << *(arr+i) << " "; // arr[i]
    }
    delete[] arr;

    return 0;
}
```

```
Enter the number of items:
4
Enter 4 items
arr[0]=1
arr[1]=2
arr[2]=3
arr[3]=4
You entered: 1 2 3 4
```

Двовимірні динамічні масиви

```
size_t n = 3, m=4;
Int **a = new int* [n];
for (size_t i = 0; i < n; ++i) {

        a[i] = new int[m];
}
// Memory deallocation
for (size_t i = 0; i < n; ++i) {

        delete[] arr[i];

}
delete[] arr;
```



Двовимірні динамічні масиви

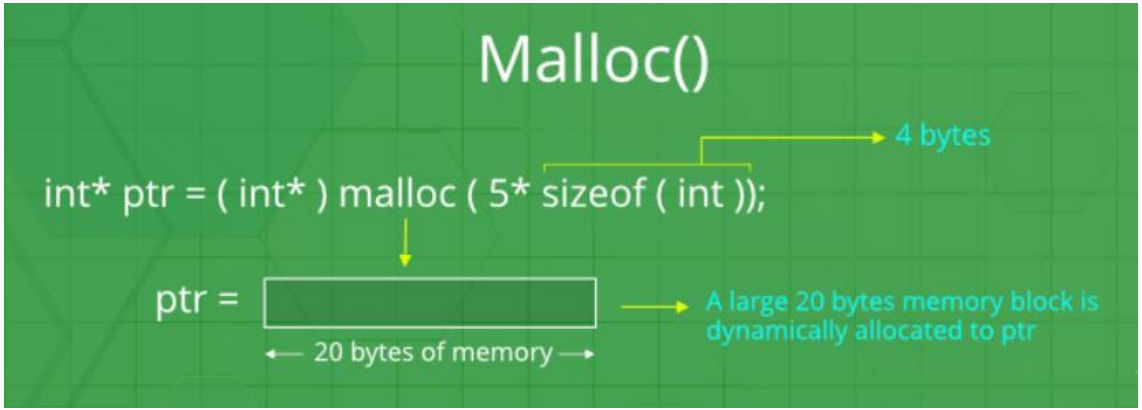
```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      size_t n = 3, m = 4;
5      // Memory allocation
6      int *arr = new int[n*m];
7
8      // Matrix initialization
9      for (size_t i = 0; i < n; ++i) {
10         for (size_t j = 0; j < m; ++j) {
11
12             *(arr + m*i + j) = j + i + 1; // arr[m*i+j];
13
14             cout << arr[m*i+j] << " ";
15         }
16         cout << endl;
17     }
18     // Memory deallocation
19     delete[] arr;
20 }
```

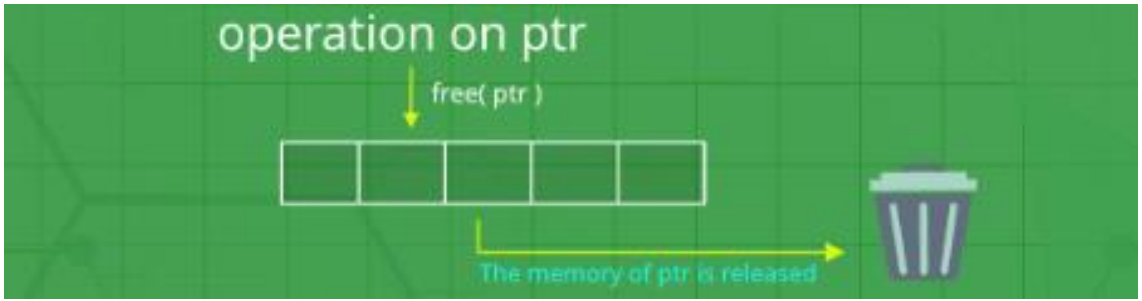
Двовимірні динамічні масиви

```
3  int main() {
4      size_t n;
5      cout << "Input n: ";
6      cin >> n;
7      // Memory allocation
8      int** arr = new int* [n];
9      for (size_t i = 0; i < n; ++i) {
10         arr[i] = new int[n];
11     }
12     // Matrix initialization
13     for (size_t i = 0; i < n; ++i) {
14         for (size_t j = 0; j < n; ++j) {
15             if (j < n - i) {
16                 *(*(arr + i) + j) = j + i + 1; // arr[i][j] = j + i + 1;
17             }
18             else {
19                 *(arr[i] + j) = 0; // arr[i][j] = 0;
20             }
21             cout << arr[i][j] << " ";
22         }
23         cout << endl;
24     }
25     // Memory deallocation
26     for (size_t i = 0; i < n; ++i) {
27         delete[] arr[i];
28     }
29     delete[] arr;
30 }
```

```
Input n: 10
1 2 3 4 5 6 7 8 9 10
2 3 4 5 6 7 8 9 10 0
3 4 5 6 7 8 9 10 0 0
4 5 6 7 8 9 10 0 0 0
5 6 7 8 9 10 0 0 0 0
6 7 8 9 10 0 0 0 0 0
7 8 9 10 0 0 0 0 0 0
8 9 10 0 0 0 0 0 0 0
9 10 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0
```

Методи C malloc/free





Методи malloc/free vs new/delete

- **new** є оператором, тоді як **malloc()** є функцією.
- **new** викликає конструктори об’єктів, тоді як **malloc()** ні.
- **new** повертає точний тип даних, тоді як malloc() повертає **void***
- **new** ніколи не повертає **NULL** (викидає виняток **std::bad_alloc** в разі помилки), тоді як malloc() повертає **NULL**
- Різниця між **free** та **delete** полягає в тому, що **delete** викличе деструктор вашого об’єкта на додаток до звільнення пам’яті.

Ніколи не використовуйте malloc/free якщо ви маєте справу з об’єктами C++

Копіювання масивів - memcpy

```
void *memcpy(void *to, const void *from, size_t numBytes);
```

```
5   int main()
6   {
7       const size_t n = 7;
8       int source[n] = { 1,2,3,4,5 };
9
10      int dest[n];
11      cout << "dest=";
12      for (size_t i = 0; i < n; ++i) {
13          dest[i] = source[i];
14          cout << dest[i] << ' ';
15      }
16      cout << endl;
17
18      int dest2[n];
19      cout << "dest2=";
20      memcpy(dest2, source, sizeof(source));
21      for (size_t i = 0; i < n; ++i) {
22          cout << dest2[i] << ' ';
23      }
24      cout << endl;
25
26      return 0;
27  }
```

```
dest=1 2 3 4 5 0 0
dest2=1 2 3 4 5 0 0
```

Дякую!

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY