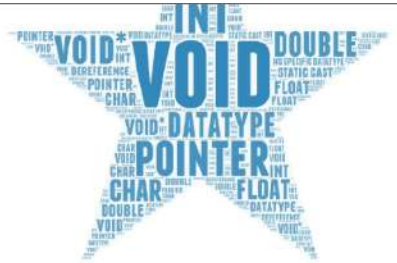


Void Pointers

iamneo

What is a Void Pointer?

- In C++, a void pointer is a pointer that is declared using the 'void' keyword (void*).
- It is different from regular pointers it is used to point to data of no specified data type.
- It can point to any type of data so it is also called a "Generic Pointer".

Syntax of Void Pointer:

```
void* ptr_name;
```

Syntax of Void Pointer: `void* ptr_name;`

iamneo

Declaring & Initializing Void Pointers

- Use **void*** to declare a void pointer.
- Initially when uninitialized, the pointer does not point to any specific type.

```
int a = 10;
void* int_ptr = &a;

float b = 3.14;
void* float_ptr = &b;

char c = 'd';
void* char_ptr = &c;
```

iamneo

Dereferencing Void Pointers

Void pointers must be typecast to the appropriate pointer type before dereferencing.

```
int a = 100; // Integer variable
void* void_ptr = &a; // Void pointer pointing to "a"
int* int_ptr = (int*)void_ptr; // Accessing the value from "void_ptr"
```

- **int a = 100;** - Declares an integer variable a and initializes it with the value 100.
- **void* void_ptr = &a;** - Initializes a void pointer void_ptr to point to the memory address of a.
- **int* int_ptr = (int*)void_ptr;** - Typecasts void_ptr to an int* and assigns it to int_ptr, allowing access to the integer value stored at that address.

iamneo

- **int a = 100;** - Declares an integer variable a and initializes it with the value 100.
- **void* void_ptr = &a;** - Initializes a void pointer void_ptr to point to the memory address of a.
- **int* int_ptr = (int*)void_ptr;** - Typecasts void_ptr to an int* and assigns it to int_ptr, allowing access to the integer value stored at that address.

Example code

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

int main() {
    int a = 100;
    void* void_ptr = &a;
    int* int_ptr = (int*)void_ptr;
    cout << "Integer value: " << *int_ptr << endl;

    float b = 3.14;
    void_ptr = &b;
    float* float_ptr = (float*)void_ptr;
    cout << "Float value: " << *float_ptr << endl;

    char c = 'd';
    void_ptr = &c;
    char* char_ptr = (char*)void_ptr;
    cout << "Character value: " << *char_ptr;

    return 0;
}
```

Output

```
Integer value: 100
Float value: 3.14
Character value: d
```

Pointer Arithmetic

iamneo

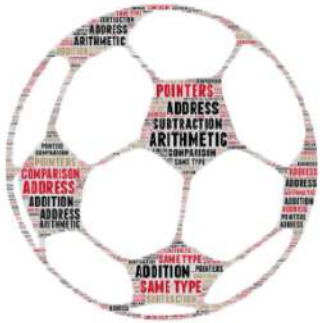
Pointer Arithmetic

- In C++, pointer variables are used to **store the addresses** of other variables, functions, structures, and even other pointers and we can use these pointers to access and manipulate the data stored at that address.
- Pointer arithmetic means performing **arithmetic operations** on pointers.
- It refers to the operations that are valid to perform on pointers.

Arithmetic operations

Following are the **arithmetic operations** valid on pointers in C++:

1. Incrementing and Decrementing Pointers
2. Addition of Constant to Pointers
3. Subtraction of Constant from Pointers
4. Subtraction of Two Pointers of the Same Type
5. Comparison of Pointers

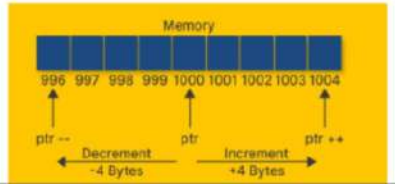


iamneo

1. Incrementing and Decrementing Pointer

Incrementing or decrementing a pointer will make it refer to the address of the next or previous data in the memory.

For example: If a pointer holds the address 1000 and we increment the pointer, then the pointer will be incremented by 4 or 8 bytes (size of the integer), and the pointer will now hold the address 1004.



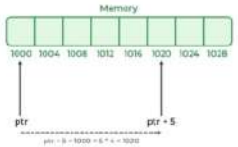
iamneo

2. Addition of Constant to Pointers

We can add integer values to Pointers and the pointer is adjusted based on the size of the data type it points to.

For example: If an integer pointer stores the address 1000 and we add the value 5 to the pointer, it will store the new address as: **1000 + (5 * 4(size of an integer)) = 1020**

Pointer Addition



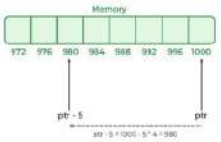
iamneo

3. Subtraction of Constant to Pointers

We can also subtract a constant from Pointers and it is the same as the addition of a constant to a pointer.

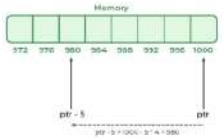
For example: If an integer pointer stores the address 1000 and we subtract the value 5 from the pointer, it will store the new address as: **1000 - (5 * 4(size of an integer)) = 980**

Pointer Subtraction



iamneo

4. Subtraction of Two Pointers of the Same Datatype



iamneo

4. Subtraction of Two Pointers of the Same Datatype

The Subtraction of two pointers can be done only when both pointers are of the **same data type.**

Example:

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

int main() {
    int num = 45;
    int* ptr1 = &num;
    int* ptr2 = ptr1 + 4;
    cout << "ptr1 address: " << ptr1 << endl;
    cout << "ptr2 address: " << ptr2 << endl;
    cout << "ptr2 - ptr1 = " << ptr2 - ptr1;
    return 0;
}
```

Output

```
ptr1 address: 0x7ffd28f1b134
ptr2 address: 0x7ffd28f1b144
ptr2 - ptr1 = 4
```

iamneo

5. Comparison of Pointers

In C++, we can perform a comparison between the two pointers using the relational operators(>, <, >=, <=, ==, !=).

Example: Comparing Pointer Variables

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

int main() {
    int num = 10;
    int* ptr1 = &num;
    int** ptr2 = &ptr1;
    int* ptr3 = *ptr2;

    if (ptr1 == ptr3) {
        cout << "Same memory location!";
    }
    else {
        cout << "ptr1 points to: " << ptr1 << endl;
        cout << "ptr3 points to: " << ptr3;
    }
    return 0;
}
```

Output

```
Same memory location!
```

iamneo

Pointer to pointer

iamneo

Pointers - Recall

- In C++ a Pointer is a variable that is used to store the **memory address** of other variables.
- It is a variable that points to a data type (like int or string) of the same type and is created with the * operator.
- **Syntax of a Pointer in C++:**

```
data_type_of_pointer *name_of_variable = & normal_variable;
```

iamneo

What is a Pointer to a Pointer or Double Pointer in C++?

- When we define a pointer to a pointer, the first pointer is used

```
data_type_of_pointer *name_of_variable = & normal_variable;
```

iamneo

What is a Pointer to a Pointer or Double Pointer in C++?

- When we define a pointer to a pointer, the first pointer is used to store the address of the variables, and the second pointer stores the **address of the first pointer**.
- For this very reason, this is known as a Double Pointer or Pointer to Pointer.

iamneo

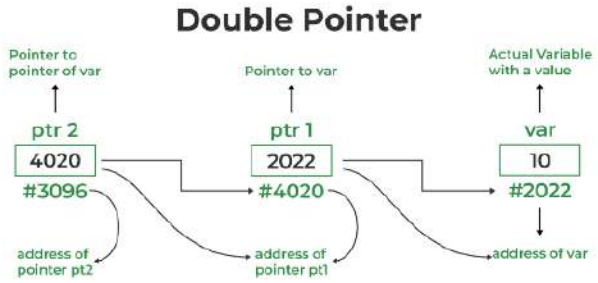
Declare a Pointer to a Pointer

- Declaring a Pointer to Pointer is similar to declaring a pointer in C++.
- The difference is we have to use an **additional * operator** before the name of a Pointer in C++.
- Syntax of a Pointer to Pointer(Double Pointer) in C++:

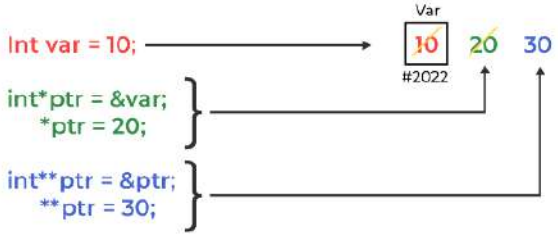
```
data_type_of_pointer **name_of_variable = & normal_pointer_variable;
```

iamneo

Concept of Double pointers



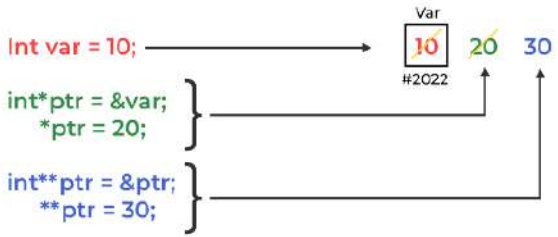
How double pointer works?



Example

iamneo

How double pointer works?



iamneo

Example

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

int main() {
    int V = 100;
    int* ptr1;
    int** ptr2;
    ptr1 = &V;
    ptr2 = &ptr1;
    cout << "V = " << V << endl;
    cout << "ptr1 = " << *ptr1 << endl;
    cout << "ptr2 = " << **ptr2;
    return 0;
}
```

Output

```
V = 100
ptr1 = 100
ptr2 = 100
```

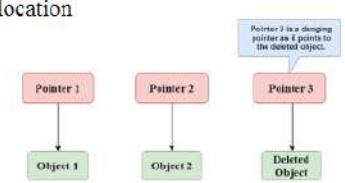
iamneo

Types of Pointers -
Dangling pointer, Wild pointer, Null pointer

iamneo

Dangling Pointer

- Dangling pointer is a pointer pointing to a memory location that has been **freed (or deleted).**
- It means that memory location has **no value to store.**
- We use the **free()** call to clear the allocated space (de-allocate) from the heap memory. Now pointer will point to a freed memory location



iamneo

Example code - Dangling Pointer

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

void createDanglingPointer() {
    int *ptr = new int(10); // dynamically allocate memory
    cout << "Value: " << *ptr;
    delete ptr; // deallocate memory
    // ptr is now a dangling pointer
}

int main() {
    createDanglingPointer();
    return 0;
}
```

Output

```
Value: 10
```


Wild Pointer

- Wild pointers are different from pointers i.e. they also store the memory addresses but point the **unallocated memory or data value which has been deallocated**. Such pointers are known as wild pointers.
- A pointer behaves like a wild pointer when it is **declared but not initialized.**

```cpp
int *ptr // Wild pointer
```


How can we avoid a Wild Pointer?

If a pointer points to a **known variable,** then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

How can we avoid a Wild Pointer?

If a pointer points to a **known variable,** then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```
int* p; // Wild pointer
int a = 10;
// p is not a wild pointer now
p = &a;
// This is fine. Value of a is changed
*p = 12;
```

If we want a pointer to a value (or set of values) without having a variable for the value, we should **explicitly allocate memory** and put the value in the allocated memory.

```
int* p = (int*)malloc(sizeof(int));
// This is fine (assuming malloc doesn't return NULL)
*p = 12;
```

iamneo

Null Pointer

- A NULL Pointer in C++ indicates the **absence of a valid memory address** in C++.
- It tells that the pointer is not pointing to any valid memory location.
- In other words, it has the value **"NULL"** (or "nullptr" since C++11).
- We can create a NULL pointer of any type by simply assigning the value NULL to the pointer as shown:

```
int* ptrName = NULL;  // before C++11
int* ptrName = nullptr // since C++11
int* ptrName = 0; // by assigning the value 0
```

iamneo

Example of NULL Pointer in C++

```
int main() {
```

- We can create a NULL pointer of any type by simply assigning the value NULL to the pointer as shown:

```
int* ptrName = NULL;  // before C++11
int* ptrName = nullptr // since C++11
int* ptrName = 0; // by assigning the value 0
```

iamneo

Example of NULL Pointer in C++

```
int main() {
    int* ptr = nullptr;

    if (ptr == nullptr) {
        cout << "NULL" << endl;
    }
    else {
        cout << "Not NULL." << endl;
    }


    int value = 5;
    ptr = &value;

    if (ptr == nullptr) {
        cout << "NULL";
    }
    else {
        cout << "Not NULL" << endl;
        cout << "Value = " << *ptr;
    }
```

Output

```
NULL
Not NULL
Value = 5
```

iamneo

Recall

Pointer Type	Definition	Cause	Safety Check	Example Declaration
Wild Pointer	Uninitialized pointer, points to an unknown memory location	Not initialized	Initialize before use	`int *wildPtr;`
Dangling Pointer	Pointer to a memory location that has been freed or deleted	Memory deallocated	Set to `nullptr` after delete	`int *danglingPtr = new int(5); delete danglingPtr;`
Null Pointer	Pointer explicitly set to point to nothing (null)	Explicitly assigned `nullptr` or `NULL`	Check if `nullptr` before use	`int *nullPtr = nullptr;`

iamneo

To exit full screen, press Esc

Pointer to objects

iamneo

Pointers to objects

- A pointer is a variable that stores the **memory address** of another variable (or object) as its value.
- A pointer aims to **point to a data type** which may be int, character, double, etc.
- Pointers to objects aim to make a pointer that can access the **object**, not the variables.
- Pointer to object in C++ refers to accessing an object.
- There are **two approaches** by which you can access an object.

iamneo

Approaches - Pointers to objects

Approaches: One is directly and the other is by using a pointer to an object in C++.

Approaches - Pointers to objects

Approaches: One is directly and the other is by using a pointer to an object in C++.

- A pointer to an object in C++ is used to store the address of an object.
- For creating a pointer to an object in C++, we use the following syntax:

```
class_name* pointer_to_object;
```

iamneo

Approaches - Pointers to objects

- For storing the address of an object into a pointer in C++, we use the following syntax:

```
pointer_to_object = &object_name;
```

- The above syntax can be used to store the **address** in the pointer to the object.
- After storing the address in the pointer to the object, the **member function** can be called using the pointer to the object with the help of an arrow operator.

iamneo

Example - Pointers to objects

```
class My_Class {
    int num;
public:
    void set_number(int value) {
```

In this example, a simple class

- The above syntax can be used to store the **address** in the pointer to the object.
- After storing the address in the pointer to the object, the **member function** can be called using the pointer to the object with the help of an arrow operator.

iamneo

Example - Pointers to objects

```
class My_Class {
    int num;
public:
    void set_number(int value) {
        num = value;
    }
    void show_number();
};

void My_Class::show_number() {
    cout << num << endl;
}
```

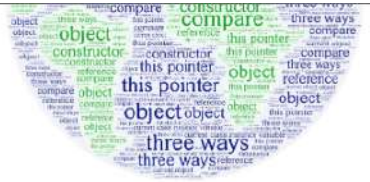
In this example, a simple class named **My_Class** is created. An object of the class is defined as named **object**. Here a pointer is also defined named **p**.

```
int main() {
    // an object is declared and a pointer to it
    My_Class object, *p;
    object.set_number(1);
    object.show_number();
    p = &object;
    // object is accessed using the pointer
    p->show_number();
    return 0;
}
```

Output

```
1
1
```

iamneo



this Pointer in C++

iamneo

this pointer

- In C++ programming, ***this*** is a keyword that refers to the **current instance** of the class.
- It's not necessary to explicitly define the *"this"* pointer as a function argument within the class, as the compiler handles it **automatically**.
- There can be 3 main usage of this keyword in C++.

```
class test {
int x=10;        ——→ this.x=10
void demo(){
int x=20;}       ——→ x=10
}
```

iamneo

1. Accessing data members

1. Accessing data members

- Inside a member function, you can use the "this" pointer to **access** the data members of the object that called the function.

```
class MyClass {
private:
    int value;
public:
    void setValue(int value) {
        this->value = value;
    }
};
```

- In this example, the ***'this'*** pointer is used to access the ***value*** data member of the object.

iamneo

2. Returning the object itself

- Inside a member function, you can use the "this" pointer to return a **reference** to the object that called the function.
- This can be useful for chaining multiple member function calls together.

```
class MyClass {
public:
    MyClass& doSomething() {
        // do something...
        return *this; // return a reference to the object itself
    }
};
```

- In this example, the ***doSomething*** member function returns a reference to the object itself by dereferencing the ***'this'*** pointer and returning it.

3. Comparing objects

- Inside a member function, you can use the "this" pointer to **compare** the object that called the function to another object.

```
class MyClass {
public:
    bool isSameAs(const MyClass& other) {
        return this == &other; // compare the object to "other"
    }
};
```

- In this example, the ***isSameAs*** member function compares the object that called the function to another object by comparing their addresses using the ***"this"*** pointer and the address-of operator "&".

iamneo

Example

```
class MyClass {
  private:
    int value;
  public:
    void setValue(int value) {
        // accessing class member using "this" pointer
        this->value = value;
    }
    void printValue() {
        // accessing class member using "this" pointer
        cout << "Value: " << this->value ;
    }
};

int main() {
    MyClass obj;
    obj.setValue(42);
    obj.printValue();
    return 0;
}
```

Output

```
Value: 42
```

this Pointer as a Constructor

- In C++, the "this" pointer can also be used as a constructor, a special member function that is called when an object of the class is created.
- When used as a constructor, the "this" pointer is used to **refer to the object that is being created.**

```
class MyClass {
private:
    int value;
public:
    MyClass(int value) {
        this->value = value;
    }
};
```

iamneo

Deleting this Pointer

- It is not recommended to delete the *"this"* pointer in C++.
- The *"this"* pointer **points to the object** that is currently executing a member function, and deleting it would cause **undefined behavior** and likely result in a program crash.
- The *"this"* pointer is an implicit pointer and is managed by the C++ **compiler**.
- Attempting to delete the *"this"* pointer is a common mistake and a potential source of bugs in C++ programs.

Class containing pointers

iamneo

Classes containing pointers

- Classes containing pointers refer to classes in C++ where one or more member variables are **pointers.**
- These pointers can point to dynamically allocated memory, other objects, or **primitive data types.**
- Requires careful handling of memory allocation and deallocation.

iamneo

Classes containing pointers

- When a class contains pointers, it often requires a custom **constructor and destructor.**
- The constructor initializes the pointer, often allocating memory, and the destructor ensures that any dynamically allocated memory is properly deallocated.
- For classes containing pointers, a **deep copy** is typically necessary. The default copy constructor and assignment operator perform shallow copies, which can lead to issues like double deletion.

Example - Classes containing pointers

```
class DynamicArray {
    int* arr;
    int size;
public:
    DynamicArray(int s) : size(s) {
        arr = new int[size];
    }
    ~DynamicArray() {
        delete[] arr;
    }
};
```

```
class DynamicArray {
    int* arr;
    int size;
public:
    DynamicArray(int s) : size(s) {
        arr = new int[size];
    }
    ~DynamicArray() {
        delete[] arr;
    }
};
```


Uses - Classes containing pointers

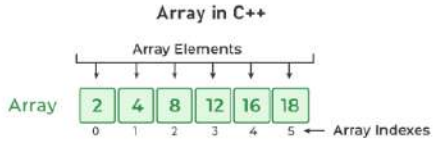
- In object-oriented programming, classes containing pointers enable the implementation of complex behaviors and interactions between objects.
- Such classes are useful in resource management scenarios where objects manage resources like file handles, network connections, or large blocks of memory.
- Classes containing pointers are often used to implement dynamic data structures like linked lists, trees, graphs, and hash tables.

1D Array in C++

iamneo

Arrays in C++

- An Array is a collection of data of the **same data type**, stored at a contiguous memory location.
- Indexing of an array **starts from 0**. It means the first element is stored at the 0th index, the second at 1st, and so on.
- Once an array is declared its size remains **constant** throughout the program.
- An array can have multiple dimensions.



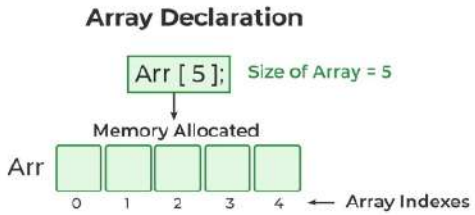
iamneo

Declaring an array

- We can declare an array by specifying its name, the type of its elements, and the size of its dimensions.
- When we declare an array, the compiler allocates the memory block of the specified size to the array name.
- **Syntax: data_type array_name [size];**

Example:

- int Arr[5]
- float Arr[5]
- Char Arr[5]



iamneo

Types of Arrays

Arrays in C++ are classified into three types:

- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

Declaring and Defining 1D Array

1D Array: A one-dimensional array is a collection of elements of the same data type that are stored in contiguous memory locations.

Syntax & Example:

```
data_type array_name[array_size];
int numbers[5];
```

Here, int numbers[5]; declares an integer array named "numbers" with a size of 5 elements.

iamneo

Processing inside main function

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

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int n = sizeof(numbers) / sizeof(numbers[0]);
    for(int i = 0; i < n; i++) {
        cout << numbers[i] << " ";
    }
    return 0;
}
```

Output

```cpp
using namespace std;

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int n = sizeof(numbers) / sizeof(numbers[0]);
    for(int i = 0; i < n; i++) {
        cout << numbers[i] << " ";
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

iamneo

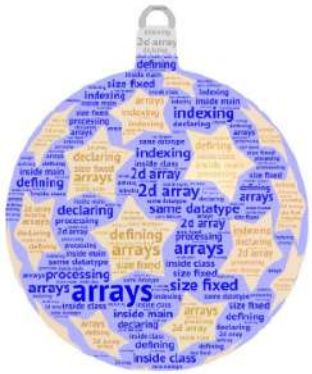
Processing inside class

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

class MyClass {
private:
    int numbers[5] = {1, 2, 3, 4, 5};

public:
    void displayNumbers() {
        int n = sizeof(numbers) / sizeof(numbers[0]);
        for(int i = 0; i < n; i++) {
            cout << numbers[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    MyClass obj;
    obj.displayNumbers();
    return 0;
}
```

Output

```
1 2 3 4 5
```

iamneo



2D Arrays
Multi-Dimensional
Arrays in C++

iamneo

Two-dimensional Array

- In C++, we can create an array of an array, known as a multidimensional array.
- **For example: int x[3][4];** Here, x is a two-dimensional array. It can hold a maximum of 12 elements.
- We can think of this array as a table with 3 rows and each row has 4 columns as shown below:

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

iamneo

Initialization of two-dimensional array

```
int  test[2][3] = { {2, 4, 5}, {9, 0, 19}};
```

The above array has 2 rows and 3 columns, which is why we have two rows of elements with 3 elements each.

	Col 1	Col 2	Col 3
Row 1	2	4	5
Row 2	9	0	19

iamneo

Processing inside main

The below array has 2 rows and 3 columns, which is why we have two rows of elements with 3 elements each.

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

cout << "Original Matrix:" << endl;
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}

cout << "\nMatrix After Processing:" << endl;
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
        matrix[i][j] *= 2;
        cout << matrix[i][j] << " ";
    }
    cout << endl;
```

Output

```
Original Matrix:
1 2 3
4 5 6
7 8 9

Matrix After Processing:
2 4 6
8 10 12
14 16 18
```

Processing inside class

```cpp
class Matrix {
public:
    int matrix[2][3];
    void display() {
        cout << "Matrix:" << endl;
        for(int i = 0; i < 2; i++) {
            for(int j = 0; j < 3; j++) {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    Matrix p;
    int count = 1;
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            p.matrix[i][j] = count++;
        }
    }
    p.display();
    return 0;
}
```

Output

```
Matrix:
1 2 3
4 5 6
```

This C++ code defines a class named **Matrix** with a member function **display()** to show the elements of a 2x3 matrix initialized inside the main function.

iamneo

Multi-dimensional Array

- A multidimensional array is an array with **more than one dimension.**
- It is the homogeneous collection of items where each element is accessed using multiple indices.

Multidimensional Array Declaration:

```
datatype arrayName[size1][size2]...[sizeN];
```

Where;

- **datatype:** Type of data to be stored in the array.

dimension.

- It is the homogeneous collection of items where each element is accessed using multiple indices.

Multidimensional Array Declaration:

```
datatype arrayName[size1][size2]...[sizeN];
```

Where;

- **datatype:** Type of data to be stored in the array.
- **arrayName:** Name of the array.
- **size1, size2,…, sizeN:** Size of each dimension.

iamneo

Example code

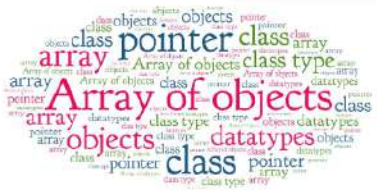
```
#include <iostream>
using namespace std;

int main() {
    int arr1[2][4];
    int arr2[2][4][8];
    cout << "arr1 size " << sizeof(arr1) << " bytes";
    cout << "\narr2 size " << sizeof(arr2) << " bytes";
    return 0;
}
```

Output

```
arr1 size 32 bytes
arr2 size 256 bytes
```

- The array int **arr1[2][4]** can store total **(2*4) = 8** elements.
- In C++ int data type takes 4 bytes and we have 8 elements in the array **'arr1'** of the int type.
- Total size = 4*8 = **32 bytes**.
- Array int **arr2[2][4][8]** can store total (2*4*8) = 64 elements.
- The Total size of '**arr2**' = 64*4 = **256 bytes**.

iamneo

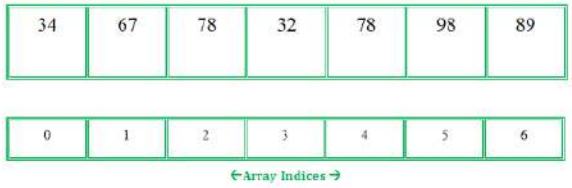


Array of objects

iamneo

Array - Recall

- An array in C/C++ or any programming language is a **collection of similar data items** stored at contiguous memory locations, allowing random access via indices.
- Arrays can store primitive data types like int, float, double, char, etc., as well as derived data types like structures and pointers.
- Below is a pictorial representation of an array.



iamneo

Array of objects

- When a class is defined, only the specification for the object is defined; **no memory or storage is allocated.**
- To use the data and access functions defined in the class, you need to create objects.

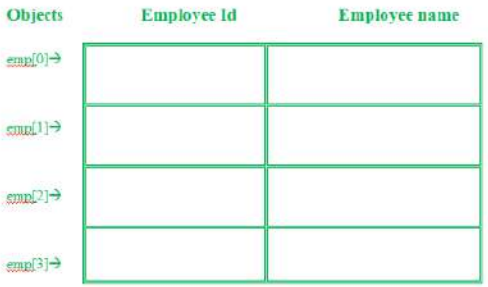
Syntax: ***ClassName ObjectName[number of objects];***

- The Array of Objects stores objects.
- An array of a **class type** is also known as an array of objects.

iamneo

Example - Array of objects

- Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].



Example code - Array of objects

An array of objects can be used if there is a need to store data of more than one employee.

```
class Employee {
    int id;
    char name[30];
public:
    void getdata();
    void putdata();
};

    void Employee::getdata() {
    cin >> id;
    cin >> name;
}

void Employee::putdata() {
    cout << id << " ";
    cout << name << " ";
    cout << endl;
}
```

iamneo

Example code - Array of objects

```
int main() {
    Employee emp[30];
    int n, i;
    cin >> n;

    for(i = 0; i < n; i++)
        emp[i].getdata();

    cout << "Employee Data - " << endl;

    for(i = 0; i < n; i++)
        emp[i].putdata();
        return 0;
}
```

Input:
2
10203
Sita
10205
Ram

Output

```
Employee Data -
```

Member functions

iamneo

Member functions

- A class member function is a function that, like any other variable, is defined or prototyped **within the class** declaration.
- It has access to all the members of the class and can operate on any object of that class.
- Let us use a member function to access the members of a previously created class instead of directly accessing them.

```
class Dice {
public:
    double L; // a dice's length
    double B; // a dice's breadth
    double H; // a dice's height
    double getVolume(void); // Returns dice volume
};
```

iamneo

Member functions

- Member functions can be defined either within the class definition or separately with the **scope resolution operator,:.**
- Even if the inline specifier is not used, specifying a member function within the class declaration declares the function inline.
- So, you may either define the Volume() function as shown below.

```
class Dice {
public:
    double L; // a dice's length
    double B; // a dice's breadth
    double H; // a dice's height
    double getVolume(void) {
        return L * B * H;
    }
};
```

iamneo

Member functions

- If we want, we may define the identical function outside of the class using the scope resolution operator (::), as seen below.

```
double Dice::getVolume(void) {
   return L * B * H;
}
```

- The main thing to remember here is that we must use the class name exactly before the :: operator.
- The dot operator (.) will be used to perform a member function on an object and will only manipulate data relevant to that object as follows:

```
Dice myDice; // Generate an object
```

- If we want, we may define the identical function outside of the class using the scope resolution operator (::), as seen below.

```
double Dice::getVolume(void) {
   return L * B * H;
}
```

- The main thing to remember here is that we must use the class name exactly before the :: operator.
- The dot operator (.) will be used to perform a member function on an object and will only manipulate data relevant to that object as follows:

```
Dice myDice; // Generate an object
myDice.getVolume();  // Call the object's member function
```

iamneo

Example - Member functions inside class

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

class MyClass {
public:
    int data = 5;
    void printData() {  // inside class
        cout << "Data " << data;
    }
};

int main() {
    MyClass obj;
    obj.printData();
    return 0;
}
```

Output

```
Data 5
```

```cpp
using namespace std;

class MyClass {
public:
    int data = 5;
    void printData() {  // inside class
        cout << "Data " << data;
    }
};

int main() {
    MyClass obj;
    obj.printData();
    return 0;
}
```

Output

```
Data 5
```

iamneo

Example - Member functions outside class

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

class MyClass {
public:
    int data = 5;
    void printData();
};

// outside class
void MyClass::printData() {
    cout << "Data " << data;
}


int main() {
    MyClass obj;
    obj.printData();
    return 0;
}
```

Output

```
Data 5
```

iamneo

Modifiers of string class

iamneo

Strings

- C++ has in its definition a way to represent a **sequence of characters** as an object of the class.
- This class is called **std:: string**.
- The string class stores the characters as a sequence of bytes with the functionality of allowing access to the single-byte character.
- We can access the various string class functions by including the **<string>** header in our file.

```
#include <string>
```

iamneo

1. Input Functions

- **getline()** - This function is used to store a stream of characters as entered by the user in the object memory.
- **push_back()** - This function is used to input a character at the end of the string.
- **pop_back()** - Introduced from C++11(for strings), this function is used to delete the last character from the string.

iamneo

Example

```
int main() {
    string str;
    getline(cin, str);
    cout << "Initial string - ";
    cout << str << endl;
    str.push_back('s');
    cout << "After push_back - ";
    cout << str << endl;
    str.pop_back();
    cout << "After pop_back - ";
    cout << str;
    return 0;
}
```

Input

Output

```
Initial string - ABCD 123
After push_back - ABCD 123s
```

```
    getline(cin, str);
    cout << "Initial string - ";
    cout << str << endl;
    str.push_back('s');
    cout << "After push_back - ";
    cout << str << endl;
    str.pop_back();
    cout << "After pop_back - ";
    cout << str;
    return 0;
}
```

Input

ABCD 123

Output

```
Initial string - ABCD 123
After push_back - ABCD 123s
After pop_back - ABCD 123
```

iamneo

2. Capacity Functions

- **capacity()** - This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
- **resize()** - This function changes the size of the string, the size can be increased or decreased.
- **length()** - This function finds the length of the string.
- **shrink_to_fit()** - This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory.

iamneo

Example

```
int main() {
    string str = "ABCD EFH 123";
    cout << "Initial string - ";
    cout << str << endl;
    str.resize(10);
    cout << "After resize - ";
    cout << str << endl;
    cout << "Capacity - ";
    cout << str.capacity() << endl;
    cout << "Length - " << str.length();
    str.shrink_to_fit();
    cout << "\nNew capacity - ";
    cout << str.capacity() << endl;
    return 0;
}
```

Output

```
Initial string - ABCD EFH 123
After resize - ABCD EFH 1
Capacity - 15
Length - 10
New capacity - 15
```

iamneo

3. Iterator Functions

- **begin()** - This function returns an iterator to the beginning of the string.
- **end()** - This function returns an iterator to the next to the end of the string.
- **rbegin()** - This function returns a reverse iterator pointing at the end of the string.
- **rend()** - This function returns a reverse iterator pointing to the previous of beginning of the string.

4. Manipulating Functions

- **copy("char array", len, pos)** - This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying.
- **swap()** - This function swaps one string with another.

Example 1:

```
int main() {
    string str1 = "iamneo";
    string str2 = "edutech";
    char ch[80];
    str1.copy(ch, 5, 1);
    cout << "Copied character array - ";
    cout << ch;
    return 0;
}
```

Output

```
Copied character array - amneo
```

iamneo

Example 2

```
int main() {
    string str1 = "iamneo";
    string str2 = "edutech";

    cout << "str1 before swapping - ";
    cout << str1 << endl;
    cout << "str2 before swapping is - ";
    cout << str2 << endl;
    str1.swap(str2);

    cout << "str1 after swapping - ";
    cout << str1 << endl;
    cout << "str2 after swapping - ";
    cout << str2 << endl;
    return 0;
}
```

Output

```
str1 before swapping - iamneo
str2 before swapping is - edutech
str1 after swapping - edutech
str2 after swapping - iamneo
```