Agenda

- Dynamic memory allocation
- Static (Data Member & Member Functions),
- simple and dynamic Array(1D and 2D)

Dynamic Memory Allocation

- If we want to allocate memory dynamically then we should use new operator and to deallocate that memory we should use delete operator.
- If pointer contains, address of deallocated memory then such pointer is called dangling pointer.
- When we allocate space in memory, and if we loose pointer to reach to that memory then such wastage of memory is called memory leakage.
- If new operator fails to allocate memory then it throws bad_alloc exception.
- If malloc/calloc/realloc function fails to allocate memory then it returns NULL.
- If new operator fails to allocate memory then it throws bad_alloc exception.
- If we create dynamic object using malloc then constructor do not call. But if we create dynamic object using new operator then constructor gets called.

```
int main()
{
    int *ptr = new int;
    *ptr = 20;
    cout << "Address of dynamic Memory - " << ptr << endl;
    cout << "Value on dynamic memory - " << *ptr << endl;
    delete ptr;
    ptr = NULL;
    return 0;
}
```

Difference between malloc() vs new and free() vs delete

malloc() vs. new:

- Type:
  - malloc(): malloc() is a function. Declared in <stdlib.h>.
  - new: new is an operator. No separate header file needed.
- Initialization:
  - malloc(): Memory allocation only. Doesn't call constructors for objects.
  - new: Memory allocation and initialization. Calls constructors for objects.
- Usage with Arrays:
  - malloc(): No special handling for arrays.
  - new: Supports array allocation. new[] is used for arrays, which can later be deallocated with delete[].
- Return Type:
  - malloc(): Returns a void* pointer.

    - new: Returns a pointer to the type of object being allocated.
- Type Safety:
    - malloc(): Not type-safe. Requires explicit casting.
    - new: Type-safe. No need for explicit casting.
- Overloading:
    - new: Supports overloading to customize memory allocation behavior.
    - malloc(): malloc() is not meant to be overloaded.
- Usage in C++:
    - malloc(): Can be used in C++ but not recommended due to lack of support for constructors.
    - new: Preferred in C++ for dynamic memory allocation because it supports constructors.

free() vs. delete:

- Type:
    - free(): free() is a function. Declared in <stdlib.h>.
    - delete: delete is an operator. No separate header file needed.
- Type of Memory:
    - free(): Used to deallocate memory allocated with malloc() or calloc().
    - delete: Used to deallocate memory allocated with new.
- De-Initialization:
    - free(): Only deallocates memory. Doesn't call destructors for objects.
    - delete: Deallocates memory and calls destructors for objects.
- Usage with Arrays:
    - free(): No special handling for arrays.
    - delete: Used with delete[] to deallocate memory allocated for arrays.
- Overloading:
    - delete: Supports overloading to customize memory deallocation behavior.
    - free(): free() is not meant to be overloaded.
- Usage in C++:
    - free(): Not used in C++. Deallocating memory allocated with malloc() or calloc() using free() in C++ can lead to undefined behavior if
    - the object has non-trivial constructors or destructors.
    - delete: Preferred in C++ for deallocating memory allocated with new because it properly calls destructors for objects.

Static

- All the static and global variables get space only once during program loading
- Static variable is also called as shared variable.
- If we declare function static then local variables are not considered as static.
- If we dont want to access any global function inside different file then we should declare global function static.
- In C/C++, we can not declare main function static.
- In C++ we can declare
    1. Data member as static
    2. Member function as static

Static Data Member

- If we want to share value of the data member in all the objects of same class then we should declare datamember static.
- Static data member do not get space inside object rather all the objects of same class share single copy of it. Hence size of object is depends on size of all the non static data members declared inside class.
- If class contains all static data members then size of object will be 1 byte.
- Data member of a class, which get space inside object is called instance variable. In short, non static data member is also called as instance variable.
- Instance variable gets space once per object. Hence to access it we must use object, pointer or reference.
- Data member of the class, which do not get space inside object is called class level variable. In other words, static data member is also called as class level variable.
- Class level variable get space once per class. Hence to access it we should use class name and scope resolution operator.
- If we want to declare data member static then we must provide global definition for it otherwise linker generates error.
- Instance variable get space inside instance hence we should initialize it using constructor.
- Class level variable do not get space inside instance hence we should not initialize it inside constructor. We must initialize it in global definition.
- We can declare constant data member static.

Static Member Function

- We can not declare global function constant but we can declare member function constant.
- Except main function, we can declare global function as well as member function static.
- To access non static members of the class, we should declare member function non static and to access static members of the class we should declare member function static.
- Member function of a class which is designed to call on object is called instance method. In short non static member function is also called as instance method.
- To access instance method either we should use object, pointer or reference to object.
- Member function of a class which is designed to call on class name is called class level method In short static member function is also called as class level method.
- To access class level method we should use classname and :: operator.
- Since static member functions are not designed to call on object it doesnt get this pointer.
- this pointer is considered as link between non static data member and non static member function.
- Since static member function do not get this pointer, we can not access non static members inside static member function directly.
- Inside non static member function, we can access static as well as non static members.
- Using object, we can access non static members inside static member function.
- We can declare static data member constant but we can not declare static member function constant.
- We can not declare static member function constant, volatile and virtual.

Array

- Array is a data structure that is used to store the elements of same type in contagious memory locations.

- the elements stored in the array can be accessed using their index number;
- Types of array
    1. Single Dimension Array
    2. Multi Dimension Array
- we can create array for fundamental data types as well as derived data types

Single Dimension Array

```
// single dimension array
int main()
{
    // int arr[5] = {10, 20, 30, 40, 50};
    int arr[] = {10, 20, 30, 40, 50};
    // int arr[5];
    //  arr[0] = 10;
    //  ...

    for (int i = 0; i < 5; i++)
        cout << arr[i] << ",";
    cout << endl;
    return 0;
}

// single dimension array of ptrs (Dynamic memory allocation)
int main()
{
    int *arr[5];
    for (int i = 0; i < 5; i++)
        arr[i] = new int(10 * (i + 1));

    for (int i = 0; i < 5; i++)
        cout << *arr[i] << ",";
    cout << endl;

    for (int i = 0; i < 5; i++)
        delete arr[i];
    return 0;
}

// single dimension array with Dynamic memory allocation
int main()
{
    int *arr = new int[5]{10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++)
        cout << arr[i] << ",";
    cout << endl;
    delete[] arr;
    return 0;
}
```

Multi Dimension Array

// multi dimension array with Dynamic memory allocation int main2() {

```
int **arr = new int *[2];
arr[0] = new int[3]{10, 20, 30};
arr[1] = new int[3]{40, 50, 60};

for (int i = 0; i < 2; i++)
    for (int j = 0; j < 3; j++)
        cout << arr[i][j] << ",";
cout << endl;
delete[] arr[0];
delete[] arr[1];
delete[] arr;
return 0;
```

}