```
    ~Base() { cout << "Base Destructor" << endl; }
};

class Member {
public:
    Member() { cout << "Member Constructor" << endl; }
    ~Member() { cout << "Member Destructor" << endl; }
};

class Derived : public Base {
private:
    Member member;

public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Derived obj; // Creating object of derived class
    return 0;
}
```

Output:

```
Base Constructor
Member Constructor
Derived Constructor
Derived Destructor
Member Destructor
Base Destructor
```

In the example above:

- Constructors are called in the order: Base, Member, Derived.
- Destructors are called in the order: Derived, Member, Base.

Section 2: Pointers and Dynamic Memory Management

2.1 Declaring and Initializing Pointers

1. Pointer Syntax and Declaration:

- Pointers are variables that store memory addresses of another variable.
- Syntax: `data_type *pointer_name;`
- Example:

```
int *ptr; // Declares a pointer to an integer
```

```
float *floatPtr; // Declares a pointer to a float
char *charPtr; // Declares a pointer to a character
```

2. Initialization Methods:

- Pointers can be initialized in several ways:

Direct Initialization:

- Assign the address of a variable to a pointer during declaration.
- Syntax: `data_type *pointer_name = &variable_name;`
- Example:

```
int num = 10;
int *ptr = &num; // Initializes ptr with the address of num
```

Assignment after Declaration:

- Declare a pointer first and then assign it a value (address) later.
- Syntax:

```
data_type *pointer_name;
pointer_name = &variable_name;
```

- Example:

```
int *ptr; // Declare pointer
int num = 20;
ptr = &num; // Assign address of num to ptr
```

Using new Operator:

- Dynamically allocate memory for a variable and assign its address to a pointer.
- Syntax: data_type *pointer_name = new data_type;
- Example:

```
int *ptr = new int; // Allocates memory for an integer dynamically
```

Example of Pointer Declaration and Initialization:

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

int main() {
    // Direct Initialization
    int num = 5;
    int *ptr1 = &num;

    // Assignment after Declaration
    float val = 3.14;
    float *ptr2;
    ptr2 = &val;

    // Using new Operator
    char *charPtr = new char;
```

```
    *charPtr = 'A'; // Assign value to dynamically allocated memory

    // Output
    cout << "Value of num: " << num << endl;
    cout << "Address of num: " << ptr1 << endl;
    cout << "Value of val: " << val << endl;
    cout << "Address of val: " << ptr2 << endl;
    cout << "Value of charPtr: " << *charPtr << endl;

    // Deallocating dynamically allocated memory
    delete charPtr;

    return 0;
}
```

Output:

```
Value of num: 5
Address of num: 0x7ffc7e4f81f4
Value of val: 3.14
Address of val: 0x7ffc7e4f81f8
Value of charPtr: A
```

2.2 Accessing Data through Pointers

1. Dereferencing Pointers:

Dereferencing a pointer means accessing the value stored at the memory address pointed to by the pointer.
Syntax: `*pointer_name;`

Example:

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

int main() {
    int num = 10;
    int *ptr = &num; // Pointer points to the address of num

    cout << "Value of num: " << *ptr << endl; // Dereferencing ptr to access the value of num

    return 0;
}
```

Output:

```
Value of num: 10
```

2. Pointer Notation for Accessing Elements:

For arrays, pointer notation can be used to access array elements using pointers.

Syntax:

```
*(ptr + i)  // Equivalent to ptr[i]
```

Example:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    // Accessing elements using pointer notation
    for (int i = 0; i < 5; ++i) {
        cout << *(ptr + i) << " "; // Equivalent to ptr[i]
    }
    cout << endl;

    return 0;
}
```

Output:

```
1 2 3 4 5
```

In this example, *(ptr + i) and ptr[i] are equivalent and both are used to access array elements using pointers.

2.3 Pointer Arithmetic

1. Increment and Decrement Operations:

Pointers can be incremented and decremented to move to the next or previous memory location.

Increment Operation:

```
ptr++; // Moves the pointer to the next memory location
```

Decrement Operation:

```
ptr--; // Moves the pointer to the previous memory location
```

Example:

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

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    cout << "Original pointer value: " << ptr << endl;

    ptr++; // Incrementing pointer
    cout << "After incrementing, pointer value: " << ptr << endl;

    ptr--; // Decrementing pointer
    cout << "After decrementing, pointer value: " << ptr << endl;

    return 0;
}
```

Output:

```
Original pointer value: 0x7ffe36a552a0
After incrementing, pointer value: 0x7ffe36a552a4
After decrementing, pointer value: 0x7ffe36a552a0
```

2. Arithmetic Operations on Pointers:

Arithmetic operations such as addition and subtraction can be performed on pointers.

Addition Operation:

```
ptr += n; // Moves the pointer n positions forward
```

Subtraction Operation:

```
ptr -= n; // Moves the pointer n positions backward
```

Example:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    cout << "Original pointer value: " << ptr << endl;

    ptr += 2; // Moving pointer 2 positions forward
    cout << "After addition, pointer value: " << ptr << endl;

    ptr -= 1; // Moving pointer 1 position backward
    cout << "After subtraction, pointer value: " << ptr << endl;
```

```
    return 0;
}
```

Output:

```
Original pointer value: 0x7ffe88bfe9d0
After addition, pointer value: 0x7ffe88bfe9d8
After subtraction, pointer value: 0x7ffe88bfe9d4
```

In this example, ptr += 2 moves the pointer two positions forward, and ptr -= 1 moves the pointer one position backward.

2.4 Memory Allocation (Static and Dynamic)

Memory allocation refers to the process of reserving memory space for variables or objects during program execution. In C++, memory allocation can be categorized into two types: static memory allocation and dynamic memory allocation.

1. Static Memory Allocation:

In static memory allocation, memory is allocated at compile-time, and the size of memory is fixed throughout the program execution. Compile time refers to the period when the programming code (such as C++) is converted to the machine code (i.e. binary code).

Example:

```
int arr[5]; // Static allocation of an array of 5 integers
```

- Memory for arr is allocated on the stack.
- Memory size is fixed and determined during compile-time.
- Memory is automatically deallocated when the scope containing the variable ends.

2. Dynamic Memory Allocation:

In dynamic memory allocation, memory is allocated at runtime, and the size of memory can be determined during program execution. Runtime is the period of time when a program is actually running and occurs after compile time.

Dynamic Memory Allocation using new operator:

Single Object Allocation:

```
int *ptr = new int; // Allocates memory for a single integer dynamically
```

- Memory for ptr is allocated on the heap.
- Memory size can be determined during runtime.
- Memory needs to be explicitly deallocated using the delete operator to prevent memory leaks.

Array Allocation:

```
int *arr = new int[5]; // Allocates memory for an array of 5 integers dynamically
```

- Memory for arr is allocated on the heap.
- Size of the array is determined during runtime.
- Individual elements of the array can be accessed using pointer notation.