Example:

```
delete ptr; // Deallocates memory allocated for the integer pointed by ptr
```

In the example above, the delete operator is used to deallocate the memory allocated for the integer pointed to by ptr. After deletion, the memory is returned to the system for reuse.

Example:

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

int main() {
    // Dynamic memory allocation
    int *ptr = new int; // Allocates memory for an integer dynamically
    *ptr = 10; // Assigns a value to the dynamically allocated memory

    // Accessing and printing the dynamically allocated value
    cout << "Dynamically allocated value: " << *ptr << endl;

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In this example:

- Memory for an integer is allocated dynamically using the new operator.
- The value 10 is assigned to the dynamically allocated memory.
- The value stored in the dynamically allocated memory is printed.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

Notes:

- Dynamically allocated memory remains allocated until explicitly deallocated using the delete operator.
- Failure to deallocate dynamically allocated memory can lead to memory leaks, causing the program to consume more memory than necessary.
- It's essential to deallocate dynamically allocated memory when it's no longer needed to ensure efficient memory usage.

2.6 Pointer to an Object

In C++, pointers can also be used to point to objects of classes. This allows for dynamic allocation of objects and provide flexibility in memory management.

1. Creating Pointers to Objects:

Pointers to objects are declared similarly to pointers to primitive data types.

Syntax:

```
ClassName *pointer_name = new ClassName;
```

Example:

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

// Class definition
class MyClass {
public:
    void display() {
        cout << "Inside MyClass" << endl;
    }
};

int main() {
    // Creating an object of MyClass dynamically
    MyClass *ptr = new MyClass;

    // Accessing member function using pointer
    ptr->display(); // Output: Inside MyClass

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In the example above:

- Memory for an object of MyClass is allocated dynamically using the new operator.
- A pointer ptr of type MyClass is used to point to the dynamically allocated object.
- The display() member function of the MyClass object is called using the pointer.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

2. Accessing Object Members through Pointers:

Once a pointer is pointing to an object, its member functions and variables can be accessed using the arrow (->) operator.

Syntax:

```
pointer_name->member_function();
pointer_name->member_variable;
```

Example:

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

```
// Class definition
class MyClass {
public:
    int data;

    void display() {
        cout << "Data: " << data << endl;
    }
};

int main() {
    // Creating an object of MyClass dynamically
    MyClass *ptr = new MyClass;

    // Accessing and modifying member variable using pointer
    ptr->data = 10;

    // Accessing member function using pointer
    ptr->display(); // Output: Data: 10

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In this example:

- A pointer ptr of type MyClass points to the dynamically allocated object of MyClass.
- The member variable data is accessed and modified using the pointer.
- The member function display() is called using the pointer to display the value of the member variable.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

2.7 'this' Pointer

In C++, the this pointer is a special pointer that points to the current instance of the class. It is available as a keyword within non-static member functions of a class. It is used to distinguish member variable from parameter.

Key Characteristics of this Pointer:

1. **Points to the Current Object:** The this pointer points to the address of the object invoking the member function.
2. **Distinguishing Member Variables:** It helps distinguish between member variables and parameters with the same name.
3. **Returning the Current Object:** It can be used to return the current object from member functions.

Example of Using this Pointer:

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

class MyClass {
private:
    int value;

public:
    // Constructor with a parameter
    MyClass(int value) {
        // Using 'this' pointer to distinguish member variable from
parameter
        this->value = value;
    }

    // Method to display the value
    void display() {
        cout << "Value: " << this->value << endl;
    }
};

int main() {
    // Creating an object of MyClass
    MyClass obj(10);

    // Displaying the value using the object
    obj.display();  // Output: Value: 10

    return 0;
}
```

2.8 Pointer-Related Problems

Pointers are powerful features in C++ that provide flexibility and control over memory management. However, improper use of pointers can lead to several issues, including dangling pointers, wild pointers, null pointer assignments, memory leaks, and allocation failures.

1. Dangling Pointers

- A dangling pointer is a pointer that references a memory location that has been deallocated (freed). Accessing or modifying the memory location through a dangling pointer leads to undefined behavior.

Example:

```
int* ptr = new int(10); // Allocate memory
delete ptr; // Deallocate memory
// ptr is now a dangling pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- After deallocating memory, set the pointer to nullptr.
  ```
  delete ptr;
  ptr = nullptr;
  ```

2. Wild Pointers

- Wild pointers are uninitialized pointers that point to arbitrary memory locations. Accessing or modifying memory through wild pointers can lead to unpredictable behavior and program crashes.

Example:

```
int* ptr; // Uninitialized pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- Initialize pointers when they are declared.
  ```
  int* ptr = nullptr;
  ```

3. Null Pointer Assignment

- A null pointer is a pointer that points to nothing (typically initialized to nullptr). Dereferencing a null pointer results in undefined behavior and typically crashes the program.

Example:

```
int* ptr = nullptr; // Null pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- Always check if a pointer is nullptr before dereferencing it.
  ```
  if (ptr != nullptr) {
      cout << *ptr << endl;
  }
  ```

4. Memory Leak

- A memory leak occurs when memory is allocated but not deallocated properly, leading to a gradual increase in memory usage. This can eventually exhaust available memory, causing the program to crash or slow down.

Example:

```
for (int i = 0; i < 1000000; ++i) {
    int* ptr = new int(10); // Allocate memory
    // No delete operation, memory leak occurs
}
```

Prevention:

- Ensure that every new operation has a corresponding delete operation.
  `delete ptr;`

5. Allocation Failures

- Memory allocation failures occur when the system cannot allocate the requested memory, typically due to insufficient available memory.

Example:

```
int* ptr = new int[1000000000]; // Large allocation request
```

Prevention:

- Always check the return value of memory allocation functions and handle exceptions for new.

```
try {
    int* ptr = new int[1000000000];
} catch (bad_alloc& ex) {
    cout << "Memory allocation failed: " << ex.what() << endl;
}
```