By understanding the implications of object slicing and employing appropriate solutions, you can ensure correct behavior and prevent loss of information when working with inheritance and polymorphism in C++.

1.7 Overriding Member Functions

1. Concept of Function Overriding:

- Function overriding in C++ occurs when a derived class provides a specific implementation for a function that is already defined in its base class. The function in the derived class should have the same name, return type, and parameters as the function in the base class.
- Function overriding is a way to achieve runtime polymorphism. When you call an overridden function using a base class pointer or reference, the derived class's version of the function is called, provided that the base class function is declared virtual.

Rules for Overriding

1. Same Signature: The function in the derived class must have the same name, return type, and parameter list as the function in the base class.
2. Virtual Keyword: The base class function should be marked with the virtual keyword to enable overriding.
3. Override Keyword: It is good practice to use the override keyword in the derived class to indicate that the function is intended to override a base class function. This helps the compiler catch errors if the function does not actually override any base class function.

2. Syntax and Examples:

Syntax:

```
class Base {
public:
    virtual void functionName() {
        // Base class function implementation
    }
};

class Derived : public Base {
public:
    void functionName() override {
        // Derived class function implementation (override)
    }
};
```

Example:

```
#include <iostream>
```

```
using namespace std;

// Base class
class Base {
public:
    // Virtual function
    virtual void display() {
        cout << "Display method of Base class" << endl;
    }
};

// Derived class
class Derived : public Base {
public:
    // Overriding function
    void display() override {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Base* basePtr;
    Derived derivedObj;

    basePtr = &derivedObj;

    // Call overridden method
    basePtr->display(); // Output: Display method of Derived class

    return 0;
}
```

In this example:

- Base class has a virtual function display().
- Derived class overrides the display() function.
- When calling display() through a Base class pointer that points to a Derived class object, the Derived class's display() function is invoked.

1.8 Object Composition and Delegation

1. Object Composition

Object Composition is a design principle where complex objects are created by combining simpler objects. This is often referred to as a "has-a" relationship. In composition, an object is composed of one or more objects from other classes, making it a more flexible and modular approach to building complex systems.

Example

Here's a simple example of object composition:

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

// Engine class
class Engine {
public:
    void start() {
        cout << "Engine starts" << endl;
    }
};

// Car class composed of Engine
class Car {
private:
    Engine engine;  // Car "has-a" Engine
public:
    void start() {
        engine.start();  // Delegate the call to the Engine object
        cout << "Car starts" << endl;
    }
};

int main() {
    Car myCar;
    myCar.start();
    // Output:
    // Engine starts
    // Car starts

    return 0;
}
```

In this example:

- The Car class has an Engine object.
- The Car class delegates the starting functionality to the Engine object by calling its start() method.

2. Delegation

Delegation is a design pattern where an object handles a request by passing it to a second "delegate" object. The delegate object provides a means to share functionality between objects. This allows for more flexible and reusable code by promoting composition over inheritance.

Example

Here's an example of delegation:

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

// Printer class
class Printer {
public:
    void print(const string &text) {
        cout << text << endl;
    }
};

// Document class using Printer for printing
class Document {
private:
    Printer printer;  // Delegating printing responsibility to Printer
public:
    void print(const string &text) {
        printer.print(text);  // Delegation
    }
};

int main() {
    Document myDocument;
    myDocument.print("Hello World!");
    // Output:
    // Hello World!

    return 0;
}
```

In this example:

- The Document class delegates the printing functionality to the Printer class.
- This separation allows the Printer class to handle the printing logic, making the Document class simpler and more focused on its core responsibilities.

Differences Between Composition and Inheritance

1. **Composition (Has-a Relationship):**
    - Combines objects to build complex systems.
    - More flexible and allows for changing behavior at runtime.
    - Promotes code reuse and encapsulation.
    - Example: A Car has an Engine.
2. **Inheritance (Is-a Relationship):**
    - Creates a new class based on an existing class.

- Establishes a parent-child relationship.
- Allows for extending and modifying behavior of the base class.
- Example: A Dog is an Animal.

Advantages of Composition and Delegation

- Flexibility: Objects can be composed and recomposed easily.
- Encapsulation: Internal details of composed objects are hidden.
- Reuse: Common functionality can be reused through composition.
- Modularity: Systems can be built in a modular fashion, promoting maintainability and scalability.
- Avoids Fragile Base Class Problem: Changes in the base class in inheritance can affect all derived classes, while composition mitigates this risk.

1.9 Order of Execution of Constructors and Destructors

In C++, the order of execution of constructors and destructors depends on the inheritance hierarchy and the construction/destruction sequence of objects.

Order of Execution:

- Constructors execute in the order of inheritance (base to derived).
- Destructors execute in the reverse order of inheritance (derived to base).

Constructors:

1. *Base Class Constructors:* Constructors of base classes are executed first, starting from the topmost base class and proceeding down the inheritance hierarchy.
2. *Member Object Constructors:* Constructors for member objects of the class are then executed, in the order they are declared within the class definition.
3. *Derived Class Constructor Body:* Finally, the constructor body of the derived class is executed.

Destructors:

The order of destruction is essentially the reverse of construction:

1. *Derived Class Destructor Body:* The destructor body of the derived class is executed first.
2. *Member Object Destructors:* Destructors for member objects are then called, in the reverse order of their construction.
3. *Base Class Destructors:* Destructors for base classes are called last, starting from the most derived class and proceeding up the inheritance hierarchy.

Example:

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

class Base {
public:
    Base() { cout << "Base Constructor" << endl; }
```

```
    ~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
```