C++ OOP Workshop: Debugging, Encapsulation, and Memory Management

El Amine Bechorfa

Introduction

This workshop will cover essential concepts in C++ related to encapsulation, constructors, destructors, and memory management. We will analyze existing code, identify problems, and refactor it using proper object-oriented programming (OOP) principles.

Part 1: Debugging & Encapsulation

Problem 1: Bank Transaction Bug

Scenario: A bank's transaction system is malfunctioning. Canceled transactions still debit money from the sender's account, leading to overdrafts. This issue arises because the code allows direct modification of the `balance` attribute of the `BankAccount` class.

Questions Before Coding

- **What is encapsulation?** Encapsulation is an OOP technique that restricts direct access to an object's internal data. It ensures data integrity by allowing access only through defined methods.
- **Why should we use encapsulation here?** To prevent unauthorized or accidental modification of the account balance, thus ensuring the integrity of financial transactions.

Given Code

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

class BankAccount {
public:
    double balance;
    BankAccount(double balance) : balance(balance) {}
```

```
};

class Transaction {
private:
    BankAccount* from;
    BankAccount* to;
    double amount;
public:
    Transaction(BankAccount* from, BankAccount* to, double
        amount)
        : from(from), to(to), amount(amount) {}

    bool execute() {
        from->balance -= amount;
        if (from->balance > 0) {
            to->balance += amount;
            return true;
        }
        return false;
    }
};

int main() {
    BankAccount* account1 = new BankAccount(1000.0);
    BankAccount* account2 = new BankAccount(500.0);

    Transaction* transaction1 = new Transaction(account1,
        account2, 500.0);
    cout << (transaction1->execute() ? "Transaction
        successful" : "Transaction failed") << endl;

    Transaction* transaction2 = new Transaction(account1,
        account2, 2000.0);
    cout << (transaction2->execute() ? "Transaction
        successful" : "Transaction failed") << endl;

    cout << "Account 1 Balance: " << account1->balance <<
        endl;
    cout << "Account 2 Balance: " << account2->balance <<
        endl;

    delete account1;
    delete account2;
    delete transaction1;
    delete transaction2;
    return 0;
}
```

Issues:

- The `Transaction` class directly modifies the `balance` attribute, which violates encapsulation.
- If a transaction fails, the money is still deducted.

Solution: Applying Encapsulation

Refactored Code

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

// BankAccount class with encapsulation
class BankAccount {
private:
    double balance;
public:
    BankAccount(double balance) : balance(balance) {}

    double getBalance() const { return balance; }

    bool withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    void deposit(double amount) {
        balance += amount;
    }
};

class Transaction {
private:
    BankAccount* from;
    BankAccount* to;
    double amount;
public:
    Transaction(BankAccount* from, BankAccount* to, double
        amount)
        : from(from), to(to), amount(amount) {}

    bool execute() {
        if (from->withdraw(amount)) {
            to->deposit(amount);
            return true;
        }
        return false;
```

```
    }
};

int main() {
    BankAccount* account1 = new BankAccount(1000.0);
    BankAccount* account2 = new BankAccount(500.0);

    Transaction* transaction1 = new Transaction(account1,
        account2, 500.0);
    cout << (transaction1->execute() ? "Transaction
        successful" : "Transaction failed") << endl;

    Transaction* transaction2 = new Transaction(account1,
        account2, 2000.0);
    cout << (transaction2->execute() ? "Transaction
        successful" : "Transaction failed") << endl;

    cout << "Account 1 Balance: " << account1->getBalance()
        << endl;
    cout << "Account 2 Balance: " << account2->getBalance()
        << endl;

    delete account1;
    delete account2;
    delete transaction1;
    delete transaction2;
    return 0;
}
```

Explanation:

- `balance` is now private, ensuring that it can only be modified through methods.
- The methods `withdraw()` and `deposit()` safely manage the balance, maintaining data integrity.

—

Part 2: Constructors, Destructors, and Copy Semantics

Problem 2: Understanding Object Lifecycle

Questions Before Coding

- **What is a constructor?** A constructor is a special method that initializes an object when it is created.

- **What is a destructor?** A destructor is called when an object is destroyed. It is used to release resources.
- **What is a copy constructor?** A copy constructor creates a new object as a copy of an existing one.

Given Code

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

class Creature {
public:
    Creature() { cout << "Default Constructor" << endl; }
    Creature(const Creature &c) { cout << "Copy Constructor"
        << endl; }
    ~Creature() { cout << "Destructor" << endl; }
};

void foo(Creature c) { cout << "Inside foo" << endl; }

int main() {
    Creature ogre;
    Creature shrek = ogre;
    foo(shrek);
    return 0;
}
```

Explanation:

- The default constructor is called when `ogre` is created.
- The copy constructor is called when `shrek` is created and when `shrek` is passed to `foo()`.
- The destructor is called when objects go out of scope.

Expected Output:

```
Default Constructor
Copy Constructor
Copy Constructor
Inside foo
Destructor
Destructor
Destructor
```