Object-Oriented Programming

CL-1004

Lab 09

Abstraction

National University of Computer and Emerging Sciences–NUCES – Karachi

Contents

1. What is Abstraction in OOP?

Abstraction is the process of identifying the essential characteristics of an object and ignoring the irrelevant details. In C++, abstraction helps to:

- Simplify complex systems.
- Provide a clear interface for the user.
- Improve code reusability and maintenance.
- Enhance security by hiding internal implementation.

2. Abstraction vs Encapsulation

While both concepts hide details, they differ in focus:

- **Abstraction** focuses on exposing only the necessary features (the "**what**") and hiding implementation (the "**how**").
- **Encapsulation** bundles the data and methods into a single unit (a class) and restricts direct access to some of the object's components.

For example, when using an ATM, you see options like withdrawal and deposit (abstraction) while the internal processing of your transaction is hidden (encapsulation).

3. Ways to Achieve Abstraction

Abstraction can be implemented in several ways.

3.1. Data Abstraction with Classes and Objects

Using classes, you can hide internal data (declared as private) and expose public methods to interact with that data.

Example:

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

class Person {
private:
    string name;
    int age;
public:

    Person(string n, int a) : name(n), age(a) { }


    string getName(){ return name; }

    int getAge(){ return age; }
```

```
    void setName(string n) { name = n; }

    void setAge(int a) {
        if(a >= 0)
            age = a;
    }


    void displayInfo(){
        cout << "Name: " << name << endl << "Age: " << age << endl;
    }
};

int main() {
    Person person("Mr. Waseem", 25);
    person.displayInfo();

    // update details without exposing internal data structure
    person.setName("Mr. Abdullah");
    person.setAge(24);
    person.displayInfo();

    return 0;
}
```

Output	Name: Mr. Waseem Age: 25 Name: Mr. Abdullah Age: 24

Explanation: The **Person** class encapsulates the data members and exposes only the necessary methods to interact with them.

3.2. Abstract Classes and Pure Virtual Functions

3.2.1. Abstract Classes

An abstract class in C++ is a class that is specifically designed to serve as a base for other classes. It cannot be instantiated on its own because it is meant to represent a generic concept rather than a complete object. In C++, a class becomes abstract when it contains at least one **pure virtual function**.

Essential Properties:

- **Inability to Instantiate:**

  Since abstract classes are incomplete (they don't provide a full implementation for all of their member functions), you cannot create an object directly from an abstract class. Instead, they are used as a base class from which derived classes inherit and provide concrete implementations for the abstract (pure virtual) functions.

- **Interface Enforcement:**

  Abstract classes define an interface (a set of functions) that all derived classes must implement. This enforces a contract; any concrete subclass derived from an abstract class must override every pure virtual function. This ensures consistency and supports polymorphism in a program.

- **Usage in Hierarchies:**

  Abstract classes are often used in designing class hierarchies. For example, a class named Shape can be made abstract if it contains a pure virtual function like **calculateArea()**. Derived classes such as **Circle**, **Rectangle**, etc., then override this function with their specific implementations.

Syntax:

```
class Shape {
    public:
        // pure virtual function makes this class abstract
        virtual double calculateArea() = 0;

        // you can also have non-virtual functions and member variables.
        void displayInfo() {
            // common code, available to all derived classes
        }
    };
```

3.2.2. Pure Virtual Functions

A pure virtual function is a function declared in a class that does not provide an implementation in that class. Its declaration ends with the pure specifier = 0. This signals that the function must be overridden in any concrete (non-abstract) derived class.

Purpose:

- **Enforcing Implementation**

  Pure virtual functions act as placeholders in an abstract class. They specify the function signature and establish a contract that all derived classes are required to fulfill by providing their own implementations. This ensures that the derived classes implement behavior that is specific to their context.

Syntax:

```
virtual ReturnType FunctionName(Parameters) = 0;
```

- **Option to Provide a Default Implementation:**

  Although pure virtual functions are intended to be overridden, C++ allows (in some cases) the programmer to provide a default implementation even for a pure virtual function. However, even if a definition is provided, the function remains pure virtual and the class remains abstract. This can be useful for common fallback behavior that derived classes can call explicitly if needed.

- **Impact on Class Design:**

  By using pure virtual functions, you force the design of a class hierarchy where base classes only define the interface (the "what" the class should do) and derived classes are responsible for the implementation (the "**how**" it does it). This separation of interface and implementation enhances flexibility and makes the code easier to maintain.

Example:

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

// abstract class representing a TV remote

class RemoteControl {

    public:

    virtual void changeChannel(int channel) = 0;
    virtual void adjustVolume(int volume) = 0;
};
```

```cpp
// concrete implementation of RemoteControl
class ConcreteRemoteControl : public RemoteControl {

    private:

    int currentChannel;
    int currentVolume;

    public:

    ConcreteRemoteControl() : currentChannel(1), currentVolume(50) { }

    void changeChannel(int channel){
        currentChannel = channel;
        cout << "Changed to channel " << currentChannel << endl;
    }

    void adjustVolume(int volume){
        currentVolume = volume;
        cout << "Adjusted volume to " << currentVolume << endl;
    }
};

int main() {

    ConcreteRemoteControl remote;
    remote.changeChannel(5);
    remote.adjustVolume(70);

    return 0;
}
```

Output	Changed to channel 5 Adjusted volume to 70

3.3 Abstraction via Header Files (Interface Separation)

Another approach is to separate the interface (declarations in header files) from the implementation (source files). This not only hides implementation details but also organizes the code better.

Example: `BankAccount` (Header and Source Separation)

`BankAccount.h` (Header file)

```
#ifndef BANKACCOUNT
#define BANKACCOUNT

class BankAccount {

    private:

    double balance;

    public:

    BankAccount(double initial_balance);

    void deposit(double amount);
    bool withdraw(double amount);
    double getBalance() const;
};

#endif
```

`BankAccount.cpp` (Implementation file)

```
#include "BankAccount.h"
#include <iostream>
using namespace std;

BankAccount::BankAccount(double initial_balance) :
balance(initial_balance) { }

void BankAccount::deposit(double amount) {
    if(amount > 0)
        balance += amount;
}

bool BankAccount::withdraw(double amount) {
    if(amount <= balance) {
        balance -= amount;
        return true;
    }
```

```
    else {
        cout << "Insufficient funds" << endl;
        return false;
    }
}

double BankAccount::getBalance(){
    return balance;
}
```

main.cpp

```
#include <iostream>
#include "BankAccount.h"
using namespace std;

int main() {
    BankAccount account(1000.0);
    account.deposit(500.0);
    account.withdraw(200.0);
    cout << "Current balance: $" << account.getBalance() << endl;
    return 0;
}
```

To compile and run the files:

```
g++ -o myprogram main.cpp BankAccount.cpp
./myprogram.exe
```

The header file exposes the public interface (the function prototypes) while hiding the private data member balance and the implementation details. This is a common way to achieve data abstraction in large-scale projects.