

Object Oriented Programming C++

Chapter 6: Polymorphism

Coding Problems & Solutions — Set 2

Problems 10–17: Further practice on overloading, overriding, virtual functions, binding, abstract classes and virtual destructors

Coding Problems & Solutions — Set 2

Problem 10: Media Player — Overriding play() Across Media Types

Write a program with a base class MediaFile having a virtual play() function, overridden by AudioFile and VideoFile, called polymorphically from an array of base pointers.

```
#include <iostream>
using namespace std;
class MediaFile {
public:
    virtual void play() { cout << "Playing generic media file" << endl; }
    virtual ~MediaFile() {}
};
class AudioFile : public MediaFile {
public:
    void play() override { cout << "Playing audio track" << endl; }
};
class VideoFile : public MediaFile {
public:
    void play() override { cout << "Playing video clip" << endl; }
};
int main() {
    MediaFile *playlist[2];
    playlist[0] = new AudioFile();
    playlist[1] = new VideoFile();
    for (int i = 0; i < 2; i++)
        playlist[i]->play();
    delete playlist[0]; delete playlist[1];
}
```

Output:

```
Playing audio track
Playing video clip
```

Explanation: Each element of the playlist array is a MediaFile* but actually points to a different derived object. Calling play() through the base pointer triggers dynamic binding, so the correct overridden version runs for each media type in the loop.

Problem 11: Function Overloading + Overriding Together — Notification System

Write a program where a base class Notifier has an overloaded send() (for string and int message codes), and a derived class EmailNotifier overrides only the string version.

```
#include <iostream>
using namespace std;
class Notifier {
public:
    virtual void send(string msg) { cout << "Notifier: " << msg << endl; }
    virtual void send(int code) { cout << "Notifier code: " << code << endl; }
};
class EmailNotifier : public Notifier {
```

```

public:
    using Notifier::send;
    void send(string msg) override { cout << "Email: " << msg << endl; }
};
int main() {
    Notifier *n = new EmailNotifier();
    n->send(string("Server down"));    // Email: Server down
    n->send(404);                     // Notifier code: 404 (inherited, unhidden)
    delete n;
}

```

Output:

Email: Server down
Notifier code: 404

Explanation: This combines overloading (two send() versions in the base class) with overriding (EmailNotifier redefines only the string version). The using-declaration prevents the int overload from being hidden, so both calls resolve correctly.

Problem 12: Static Binding Pitfall — Calling a Virtual Function Inside a Constructor

Write a program that demonstrates why calling a virtual function inside a base class constructor does NOT achieve dynamic binding, even though the function is virtual.

```

#include <iostream>
using namespace std;
class Base {
public:
    Base() {
        cout << "Base constructor calling identify(): ";
        identify();    // Calls Base::identify(), NOT Derived::identify()
    }
    virtual void identify() { cout << "I am Base" << endl; }
};
class Derived : public Base {
public:
    void identify() override { cout << "I am Derived" << endl; }
};
int main() {
    Derived d;
}

```

Output:

Base constructor calling identify(): I am Base

Explanation: While Base's constructor is executing, the Derived part of the object has not yet been constructed and the object's vptr still points to Base's vtable. So even though identify() is virtual, the call inside the base constructor always binds to Base::identify() — a classic exception every C++ student should know: virtual dispatch does not work as expected during construction (and similarly during destruction).

Problem 13: Abstract Class with Multiple Pure Virtual Functions — Library System

Write a program with an abstract class LibraryItem declaring two pure virtual functions, getType() and getLateFee(), implemented differently by Book and DVD.

```

#include <iostream>
using namespace std;
class LibraryItem {
public:
    virtual string getType() = 0;
    virtual double getLateFee(int daysLate) = 0;
    virtual ~LibraryItem() {}
};
class Book : public LibraryItem {
public:
    string getType() override { return "Book"; }
    double getLateFee(int daysLate) override { return daysLate * 2.0; }
};
class DVD : public LibraryItem {
public:
    string getType() override { return "DVD"; }
    double getLateFee(int daysLate) override { return daysLate * 5.0; }
};
int main() {
    LibraryItem *items[2] = { new Book(), new DVD() };
    for (auto *item : items)
        cout << item->getType() << " late fee (3 days): " << item->getLateFee(3) <<
endl;
    for (auto *item : items) delete item;
}

```

Output:

```

Book late fee (3 days): 6
DVD late fee (3 days): 15

```

Explanation: LibraryItem is abstract because it leaves BOTH getType() and getLateFee() undefined. A derived class must override every pure virtual function of its base, or it too remains abstract — here, both Book and DVD fully implement the interface, so they can be instantiated.

Problem 14: Virtual Constructor Idiom — Game Character Prototype System

Write a program where game characters are duplicated (“spawned”) using a clone() virtual constructor, without the spawning code knowing the exact character type.

```

#include <iostream>
using namespace std;
class Character {
public:
    virtual Character* clone() = 0;
    virtual void stats() = 0;
    virtual ~Character() {}
};
class Warrior : public Character {
public:
    Character* clone() override { return new Warrior(*this); }
    void stats() override { cout << "Warrior: HP 150, ATK 30" << endl; }
};
class Mage : public Character {
public:
    Character* clone() override { return new Mage(*this); }
    void stats() override { cout << "Mage: HP 90, MAG 45" << endl; }
};

```

```

Character* spawnCopy(Character *prototype) {
    return prototype->clone();    // virtual constructor in action
}
int main() {
    Character *original = new Warrior();
    Character *spawned = spawnCopy(original);
    spawned->stats();
    delete original; delete spawned;
}

```

Output:

Warrior: HP 150, ATK 30

Explanation: spawnCopy() only ever handles a Character*, yet correctly creates a new Warrior object because clone() is overridden to “new” the exact derived type internally. This lets new character types be added later without changing spawnCopy() at all.

Problem 15: Virtual Destructor with a Container of Base Pointers

Write a program that stores different derived Shape objects in a vector of base pointers and safely deletes all of them, relying on a virtual destructor.

```

#include <iostream>
#include <vector>
using namespace std;
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() { cout << "Shape destroyed" << endl; }
};
class Triangle : public Shape {
public:
    void draw() override { cout << "Drawing Triangle" << endl; }
    ~Triangle() { cout << "Triangle destroyed" << endl; }
};
class Hexagon : public Shape {
public:
    void draw() override { cout << "Drawing Hexagon" << endl; }
    ~Hexagon() { cout << "Hexagon destroyed" << endl; }
};
int main() {
    vector<Shape*> shapes;
    shapes.push_back(new Triangle());
    shapes.push_back(new Hexagon());
    for (auto *s : shapes) s->draw();
    for (auto *s : shapes) delete s;    // safe only because ~Shape() is virtual
}

```

Output:

Drawing Triangle
 Drawing Hexagon
 Triangle destroyed
 Shape destroyed
 Hexagon destroyed
 Shape destroyed

Explanation: Because Shape's destructor is virtual, delete s correctly triggers the full destructor chain (derived first, then base) for whichever object each pointer actually refers to. If ~Shape() were not virtual, only "Shape destroyed" would print for each element, and any derived class cleanup would be silently skipped.

Problem 16: Overloaded Operators + Overridden Virtual Function — Employee Bonus

Write a program that overloads the '+' operator to merge two Employee bonus amounts, and separately uses a virtual function to compute a role-based bonus percentage.

```
#include <iostream>
using namespace std;
class Employee {
protected:
    double baseSalary;
public:
    Employee(double s) : baseSalary(s) {}
    virtual double bonusPercent() { return 0.05; }
    double operator+(const Employee &e) {
        return (baseSalary * bonusPercent()) + (e.baseSalary * e.bonusPercent());
    }
};
class TeamLead : public Employee {
public:
    TeamLead(double s) : Employee(s) {}
    double bonusPercent() override { return 0.12; }
};
int main() {
    Employee e1(30000);
    TeamLead t1(50000);
    cout << "Combined bonus: " << (e1 + t1);
}
```

Output:

Combined bonus: 7500

Explanation: bonusPercent() is virtual, so even though operator+ is defined once in the base class, calling e.bonusPercent() inside it dispatches dynamically — $30000 \times 0.05 = 1500$ for the plain Employee and $50000 \times 0.12 = 6000$ for the TeamLead, giving 7500 total. This shows overloaded operators and virtual functions working together.

Problem 17: Pure Virtual Function with a Default Implementation Body

Write a program showing that in C++ a pure virtual function CAN still have a body, which derived classes may optionally call using base class scope resolution.

```
#include <iostream>
using namespace std;
class Logger {
public:
    virtual void log(string msg) = 0; // pure virtual, but defined below
};
void Logger::log(string msg) { // default/shared implementation
    cout << "[LOG] " << msg << endl;
}
```

```
class FileLogger : public Logger {
public:
    void log(string msg) override {
        Logger::log(msg);           // explicitly reuse base implementation
        cout << "(also written to file)" << endl;
    }
};

int main() {
    Logger *l = new FileLogger();
    l->log("Startup complete");
    delete l;
}
```

Output:

```
[LOG] Startup complete
(also written to file)
```

Explanation: Logger remains abstract (it cannot be instantiated) purely because log() is declared with = 0, regardless of whether a body is supplied elsewhere. FileLogger is still required to override log(), but its override can choose to call Logger::log() explicitly for shared behaviour — a lesser-known but exam-relevant detail about pure virtual functions.