C++ Problem Set Worksheet 1.D CMPE 261 Large Scale Programming

Osman Emre Tutay

October 2025

Problem 1: Polymorphism with Virtual Functions

Create a base class `Employee` and two derived classes, `SalariedEmployee` and `HourlyEmployee`. Use virtual functions to allow for polymorphic behavior when calling a function to calculate their weekly pay.

Instructions:

- Define a base class `Employee` with a virtual function named `calculateWeeklyPay()` that prints "Calculating pay for a generic employee." to the console.
- Create a `SalariedEmployee` class that inherits publicly from `Employee` and overrides the `calculateWeeklyPay()` function to print "Calculating fixed weekly salary.".
- Create an `HourlyEmployee` class that inherits publicly from `Employee` and overrides the `calculateWeeklyPay()` function to print "Calculating pay based on hours worked.".
- In your `main` function, demonstrate polymorphism by using an `Employee*` pointer to call the `calculateWeeklyPay()` method on instances of `SalariedEmployee` and `HourlyEmployee`.

Expected Output:

```
Calculating fixed weekly salary.
Calculating pay based on hours worked.
```

Problem 2: Abstract Class for Weapon Types

Define an abstract base class `Weapon` that declares a common interface for an attack action. Implement this interface in two different concrete classes.

Instructions:

- Create an abstract class `Weapon` containing a pure virtual function `attack() const`.
- A class with a pure virtual function cannot be instantiated.
- Define a `Sword` class that inherits from `Weapon`. It should implement `attack` to print "Swing! The sword cuts the air.".
- Define a `Bow` class that also inherits from `Weapon` and implements `attack` to print "Fwoosh! The arrow is loosed.".

Sample Usage:

```
int main() {
    // Sample Usage
    Sword longsword;
    Bow yewBow;

    performAttack(longsword);
    performAttack(yewBow);

    return 0;
}
```

Expected Output:

```
Swing! The sword cuts the air.
Fwoosh! The arrow is loosed.
```

Problem 3: Logger Interface

In $C++$, an interface is typically implemented as an abstract class where all member functions are pure virtual. Create an `ILoggable` interface and a class that implements it for logging messages to the console.

Instructions:

- Define an interface class named `ILoggable`.
- The interface should have two pure virtual functions: `logInfo(const std::string& message)` and `logError(const std::string& message)`.
- Create a `ConsoleLogger` class that inherits from `ILoggable`.
- Implement the `logInfo` method to print the message to the console, prefixed with "INFO: ".

- Implement the `logError` method to print the message to the console, prefixed with "ERROR: ".

Sample Usage:

```
ConsoleLogger logger;
logger.logInfo("System startup complete.");
logger.logError("Failed to connect to database.");
```

Expected Output:

```
INFO: System startup complete.
ERROR: Failed to connect to database.
```

Problem 4: Object Lifecycle Tracking

Create a `Widget` class that uses static members, a constructor, and a destructor to track both the total number of objects ever created and the number of objects that are currently active (i.e., have not been destroyed).

Instructions:

- Define a class `Widget`.
- Add a `static int totalWidgets` to count every object created. Initialize it to 0.
- Add a `static int activeWidgets` to count objects currently in memory. Initialize it to 0.
- In the constructor, increment both `totalWidgets` and `activeWidgets`.
- In the destructor, decrement only `activeWidgets`.
- Implement two `static` functions, `getTotalWidgets()` and `getActiveWidgets()`, to return the values of the counters.

Sample Usage:

```
#include <iostream>

// ... Widget class definition ...

int main() {
    std::cout << "--- Program Start ---\n";
    std::cout << "Total: " << Widget::getTotalWidgets()
              << ", Active: " << Widget::getActiveWidgets() << std::endl;
```

```
    Widget w1;
    Widget w2;
    std::cout << "--- After creating w1, w2 ---\n";
    std::cout << "Total: " << Widget::getTotalWidgets()
              << ", Active: " << Widget::getActiveWidgets() << std::endl;

    {
        Widget w3;
        std::cout << "--- Inside inner scope with w3 ---\n";
        std::cout << "Total: " << Widget::getTotalWidgets()
                  << ", Active: " << Widget::getActiveWidgets() << std::endl;
    }

    std::cout << "--- Exited inner scope ---\n";
    std::cout << "Total: " << Widget::getTotalWidgets()
              << ", Active: " << Widget::getActiveWidgets() << std::endl;

    return 0;
}
```

Expected Output:

```
--- Program Start ---
Total: 0, Active: 0
--- After creating w1, w2 ---
Total: 2, Active: 2
--- Inside inner scope with w3 ---
Total: 3, Active: 3
--- Exited inner scope ---
Total: 3, Active: 2
```

Problem 5: Code Organization with Header Files

Separate a `Calculator` class into three distinct files: a header for the declaration (`.h`), a source file for the implementation (`.cpp`), and a main file (`main.cpp`) to use the class. This practice is fundamental for organizing larger $C++$ projects.

Instructions:

- `Calculator.h`:
  - Declare a `Calculator` class.
  - Include header guards (e.g., `#ifndef CALCULATOR_H`) to prevent multiple inclusions.

  - Declare public member functions: `int add(int a, int b);` and `int subtract(int a, int b);`.

- `Calculator.cpp`:
  - Include the `Calculator.h` header.
  - Provide the implementation for the `add` and `subtract` member functions.

- `main.cpp`:
  - Include `iostream` and `Calculator.h`.
  - Create an instance of the `Calculator` class.
  - Use the object to call the `add` and `subtract` methods and print the results.

Sample Output:

```
8 + 5 = 13
8 - 5 = 3
```

Problem 6: Generic Pair Class Template

Write a $C++$ class template named `Pair` that can store two values of the same type. The class should also provide functions to retrieve the minimum and maximum of the two stored values.

Instructions:

- Define a class template named `Pair` that accepts a single template parameter `T`.
- The class should have two private member variables, `first` and `second`, of type `T`.
- Implement a constructor that accepts two arguments of type `T` to initialize the private members.
- Implement a public member function `getMin()` that returns the smaller of the two stored values.
- Implement a public member function `getMax()` that returns the larger of the two stored values.
- The template should work with any data type that supports `<` and `>` operators, such as `int`, `double`, and `std::string`.

Sample Usage:

```
#include <iostream>
#include <string>

// ... (Pair class template definition)

int main() {
    Pair<int> intPair(15, 7);
    std::cout << "Int Pair - Min: " << intPair.getMin()
              << ", Max: " << intPair.getMax() << std::endl;

    Pair<std::string> stringPair("apple", "orange");
    std::cout << "String Pair - Min: " << stringPair.getMin()
              << ", Max: " << stringPair.getMax() << std::endl;

    return 0;
}
```

Expected Output:

```
Int Pair - Min: 7, Max: 15
String Pair - Min: apple, Max: orange
```