Programming 1 (C++)

Object Oriented Programming Continued

REMOVING REDUNDANCY

- In C++, the inline keyword is a suggestion to the compiler to insert the complete body of the function wherever the function is called, rather than generating a typical function call.
- This can potentially reduce the overhead of function calls and improve performance, especially for small functions.

REMOVING REDUNDANCY

Base Class

```
class base_matrix {
public:
    base_matrix(size_t nr, size_t nc) : nr{nr}, nc{nc} {}
    size_t num_rows() const { return nr; }
    size_t num_cols() const { return nc; }

private:
    size_t nr, nc;
};
```

Derived Class

```
class dense_matrix : public base_matrix {
    // Additional functionality specific to dense_matrix
};

class compressed_matrix : public base_matrix {
    // Additional functionality specific to compressed_matrix
};

class banded_matrix : public base_matrix {
    // Additional functionality specific to banded_matrix
};
```

REMOVING REDUNDANCY

Free Function

```
inline size_t num_rows(const base_matrix &A) {
    return A.num_rows();
}

inline size_t num_cols(const base_matrix &A) {
    return A.num_cols();
}

inline size_t size(const base_matrix &A) {
    return A.num_rows() * A.num_cols();
}
```

REMOVING REDUNDANCY

Without Free Function

```
class base_matrix {
public:
    base_matrix(size_t nr, size_t nc) : nr{nr}, nc{nc} {}
    size_t num_rows() const { return nr; }
    size_t num_cols() const { return nc; }

private:
    size_t nr, nc;
};

class dense_matrix : public base_matrix {
    // Additional functionality specific to dense_matrix
};

class compressed_matrix : public base_matrix {
    // Additional functionality specific to compressed_matrix
};

class banded_matrix : public base_matrix {
    // Additional functionality specific to banded_matrix
};
```

REMOVING REDUNDANCY

Without Free Function

```
// Usage example
int main() {
    dense_matrix dm(3, 4);
    compressed_matrix cm(5, 6);
    banded_matrix bm(7, 8);

    // Direct calls to member functions
    size_t dm_rows = dm.num_rows();
    size_t dm_cols = dm.num_cols();
    size_t dm_size = dm.num_rows() * dm.num_cols();

    size_t cm_rows = cm.num_rows();
    size_t cm_cols = cm.num_cols();
    size_t cm_size = cm.num_rows() * cm.num_cols();

    size_t bm_rows = bm.num_rows();
    size_t bm_cols = bm.num_cols();
    size_t bm_size = bm.num_rows() * bm.num_cols();

    // Output results (for example purposes)
    std::cout << "Dense Matrix: " << dm_rows << " rows, " << dm_cols << " cols, size: " << dm_size << "\n";
    std::cout << "Compressed Matrix: " << cm_rows << " rows, " << cm_cols << " cols, size: " << cm_size << "\n";
    std::cout << "Banded Matrix: " << bm_rows << " rows, " << bm_cols << " cols, size: " << bm_size << "\n";

    return 0;
}
```

MULTIPLE INHERITANCE

- A class can be derived from multiple super-classes. For more figurative descriptions and for a less clumsy discussion of base classes' base classeswe occasionally use the terms parents and grandparents which areintuitively understood.
- With two parents, the class hierarchy looks like a V(and with many like a bouquet). The members of the sub-class are the unionof all super-class members. This bears the danger of ambiguities:

MULTIPLE INHERITANCE

```
class student {
    virtual void all_info() const {
        cout << "[student] My name is " << name << endl;
        cout << "     I passed the following grades: " << passed << endl;
    }
    // ...
};

class mathematician {
    virtual void all_info() const {
        cout << "[mathman] My name is " << name << endl;
        cout << " I proved: " << proved << endl;
    }
    // ...
};

class math_student : public student, public mathematician {
    // all_info not defined -> ambiguously inherited
};

int main() {
    math_student bob{"Robert Robson", "Algebra", "Fermat's Last Theorem"};
    bob.all_info();  // Error: ambiguity
    bob.student::all_info();  // Explicitly calling student::all_info
}
```

Ambiguous Method Call:

- Both student and mathematician have a method named all_info.
- math_student inherits from both classes, resulting in ambiguity when calling all_info on a math_student object.

MULTIPLE INHERITANCE

```
class student {
    virtual void all_info() const {
        cout << "[student] My name is " << name << endl;
        cout << "     I passed the following grades: " << passed << endl;
    }
    // ...
};

class mathematician {
    virtual void all_info() const {
        cout << "[mathman] My name is " << name << endl;
        cout << " I proved: " << proved << endl;
    }
    // ...
};

class math_student : public student, public mathematician {
    // all_info not defined -> ambiguously inherited
};

int main() {
    math_student bob{"Robert Robson", "Algebra", "Fermat's Last Theorem"};
    bob.all_info();  // Error: ambiguity
    bob.student::all_info();  // Explicitly calling student::all_info
}
```

Ambiguous Method Call:

- Both student and mathematician have a method named all_info.
- math_student inherits from both classes, resulting in ambiguity when calling all_info on a math_student object.

MULTIPLE INHERITANCE

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

using namespace std;

class person {
public:
    person(const string& name) : name{name} {}
    virtual void all_info() const {
        cout << "My name is " << name << endl;
    }
protected:
    string name;
};

class student {
public:
    student(const string& name, const string& passed)
        : name{name}, passed{passed} {}

    virtual void all_info() const {
        cout << "[student] My name is " << name << endl;
        cout << "      I passed the following grades: " << passed << endl;
    }
protected:
    string name;
    string passed;
```

MULTIPLE INHERITANCE

```
};

class mathematician : public person {
public:
    mathematician(const string& name, const string& proved)
        : person{name}, proved{proved} {}

    virtual void all_info() const override {
        person::all_info();
        cout << "    I proved: " << proved << endl;
    }
private:
    string proved;
};

class math_student : public student, public mathematician {
public:
    math_student(const string& name, const string& passed, const string& proved)
        : student{name, passed}, mathematician{name, proved} {}

    virtual void all_info() const override {
        student::all_info();
        mathematician::all_info();
    }
};

int main() {
    math_student bob{"Robert Robson", "Algebra", "Fermat's Last Theorem"};
    bob.all_info();
    return 0;
}
```

FACTORY DESIGN PATTERN

- The Factory Design Pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
- This pattern is useful when the client code doesn't need to know the exact class of the object it's creating. Instead, it delegates the responsibility of object creation to a factory method.
- Here's how the Factory Design Pattern works in C++:
  - Components of Factory Design Pattern:
  - Product: This is the interface or abstract class that defines the type of objects the factory method will create.
  - Concrete Product: These are the specific implementations of the product interface.
  - Factory: This is an interface or abstract class that declares a factory method for creating objects. It may also contain other methods for managing products.
  - Concrete Factory: These are the specific implementations of the factory interface, which override the factory method to create specific concrete products.

FACTORY DESIGN PATTERN

- The Factory Design Pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
- This pattern is useful when the client code doesn't need to know the exact class of the object it's creating. Instead, it delegates the responsibility of object creation to a factory method.
- Here's how the Factory Design Pattern works in C++:
  - Components of Factory Design Pattern:
  - Product: This is the interface or abstract class that defines the type of objects the factory method will create.
  - Concrete Product: These are the specific implementations of the product interface.
  - Factory: This is an interface or abstract class that declares a factory method for creating objects. It may also contain other methods for managing products.
  - Concrete Factory: These are the specific implementations of the factory interface, which override the factory method to create specific concrete products.

FACTORY DESIGN PATTERN

```
#include <iostream>

// Abstract Product
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {}
};

// Concrete Products
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Circle" << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Rectangle" << std::endl;
    }
};
```

FACTORY DESIGN PATTERN

```
// Abstract Factory
class ShapeFactory {
public:
    virtual Shape* createShape() = 0;
    virtual ~ShapeFactory() {}
};
// Concrete Factories
class CircleFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Circle();
    }
};

class RectangleFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Rectangle();
    }
};
int main() {
    // Client code
    ShapeFactory* factory = new CircleFactory();
    Shape* shape = factory->createShape();
    shape->draw(); // Drawing Circle

    delete shape;
    delete factory;
    return 0;
}
```

ADVANCE TECHNIQUE

- The Curiously Recurring Template Pattern (CRTP) is an advanced C++ programming technique that allows a derived class to access methods or types defined in its base class without using virtual functions. It involves a derived class inheriting from a template instantiation of its own base class.

ADVANCE TECHNIQUE

- Basic Idea: In CRTP, a derived class inherits from a template instantiation of its own base class, passing itself as a template argument to the base class.
- Static Polymorphism: Unlike traditional runtime polymorphism achieved through virtual functions, CRTP enables static polymorphism. The compiler resolves method calls at compile time, resulting in potentially more efficient code execution.
- Access to Base Class Interface: The derived class can access methods or types defined in its base class without virtual function overhead. This allows for performance optimizations and flexibility in design.

ADVANCE TECHNIQUE

```
#include <iostream>

// Base class using CRTP
template <typename Derived>
class Base {
public:
    void greet() {
        // Access derived class methods using 'static_cast' to the derived type
        std::cout << "Greetings from Base to " << static_cast<Derived*>(this)->getName() << std::endl;
    }
};

// Derived class inheriting from Base using CRTP
class DerivedClass : public Base<DerivedClass> {
public:
    // Method specific to DerivedClass
    std::string getName() {
        return "Derived";
    }
};

int main() {
    DerivedClass obj;
    obj.greet(); // Outputs: Greetings from Base to Derived
    return 0;
}
```

THANK YOU