Advanced Object-Oriented Concepts

A `new` and `delete` operators

In C++, the `new` operator is used to allocate memory for an object or an array of objects dynamically. It returns a pointer to the allocated memory. The `delete` operator is used to deallocate memory that was previously allocated with `new`.

Example: `new` and `delete` operators

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

class MyClass {
public:
    MyClass() {
        cout << "Constructor called!" << endl;
    }
    ~MyClass() {
        cout << "Destructor called!" << endl;
    }
};
int main() {
    MyClass* obj = new MyClass(); // dynamically allocate memory
       for MyClass object
    delete obj; // deallocate memory
    return 0;
}
```

In this example, we define a class `MyClass` with a constructor and a destructor. In the `main` function, we dynamically allocate an object of `MyClass` using `new` and then deallocate it using `delete`. The constructor and destructor messages will be printed to the console.

B Dynamic Memory Management

Dynamic memory management in C++ allows you to allocate and deallocate memory at runtime. This is particularly useful when the size of data structures cannot be determined at compile time.

Example: Dynamic Memory Management

```
#include <iostream>
using namespace std;
int main() {
    int* arr = new int[5]; // dynamically allocate an array of 5
       integers
    for (int i = 0; i < 5; ++i) {
        arr[i] = i * 10; // initialize the array
    }

    for (int i = 0; i < 5; ++i) {
        cout << arr[i] << " "; // print the array elements
    }
    cout << endl;

    delete[] arr; // deallocate the array memory
    return 0;
}
```

In this example, we dynamically allocate an array of integers using `new` and initialize it. After using the array, we deallocate the memory using `delete[]`. It is important to use `delete[]` for arrays to ensure proper memory management.

C Arrays of Objects

You can create arrays of objects either statically or dynamically. When you create an array of objects, the constructor is called for each object in the array, and when the array goes out of scope or is deleted, the destructor is called for each object.

Example: Arrays of Objects

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

class MyClass {
public:
    MyClass() {
        cout << "Constructor called!" << endl;
    }
    ~MyClass() {
        cout << "Destructor called!" << endl;
    }
};

int main() {
    // Static array of objects
    MyClass arr1[3];

    // Dynamic array of objects
    MyClass* arr2 = new MyClass[3];

    delete[] arr2; // deallocate dynamic array
    return 0;
}
```

In this example, both static and dynamic arrays of `MyClass` objects are created. The constructor and destructor are called for each object in the arrays. For dynamic arrays, always use `delete[]` to deallocate memory.

D Copy Constructor and Assignment Operator

A copy constructor is a special constructor that initializes a new object as a copy of an existing object. The assignment operator allows you to assign the values of one object to another existing object. This is particularly important when dealing with dynamic memory to avoid shallow copies.

Example: Copy Constructor and Assignment Operator

```
#include <iostream>
using namespace std;
class MyClass {
private:
    int* data;
public:
    MyClass(int value) {
        data = new int(value); // dynamically allocate memory
        cout << "Constructor called with value: " << *data << endl;
    }

    // Copy constructor
    MyClass(const MyClass& other) {
        data = new int(*other.data); // deep copy
        cout << "Copy constructor called!" << endl;
    }

    // Assignment operator
    MyClass& operator=(const MyClass& other) {
        if (this != &other) { // self-assignment check
            delete data; // free existing resource
            data = new int(*other.data); // deep copy
            cout << "Assignment operator called!" << endl;
        }
        return *this;
    }

    ~MyClass() {
        delete data; // deallocate memory
        cout << "Destructor called!" << endl;
    }
};
int main() {
    MyClass obj1(10); // Constructor called
    MyClass obj2 = obj1; // Copy constructor called
    MyClass obj3(20);
    obj3 = obj1; // Assignment operator called

    return 0; // Destructors called for obj1, obj2, and obj3
}
```

In this example, we define a class `MyClass` with a constructor, a copy constructor, and an assignment operator. The copy constructor performs a deep copy of the dynamically allocated memory to avoid issues with shallow copies. The assignment operator also checks for self-assignment and properly manages memory.

Note that when creating a dynamic array of objects (e.g., using `new MyClass[n]`), the default constructor is called for each object in the array. You cannot directly specify constructor arguments for each element in the array using this syntax. If your class does not have a default constructor, or if you need to initialize each object differently, you must allocate an array of pointers and construct each object individually using `new` with the desired arguments.

E Friend Functions and Classes

Friend functions and classes in C++ allow you to grant access to private and protected members of a class to specific functions or other classes. This is useful when you want to allow certain functions or classes to access the internals of another class without exposing those members publicly.

Example: Friend Functions

```
#include <iostream>
using namespace std;
class MyClass {
private:
    int data;
public:
    MyClass(int value) : data(value) {}

    // Declare a friend function
    friend void showData(const MyClass& obj);
};
// Friend function definition
void showData(const MyClass& obj) {
    cout << "Data: " << obj.data << endl; // Accessing private
       member
}
int main() {
    MyClass obj(42);
    showData(obj); // Call the friend function
    return 0;
}
```

In this example, we define a class `MyClass` with a private member `data`. We declare the function `showData` as a friend of `MyClass`, allowing it to access the private member `data`. When we call `showData(obj)`, it can access and print the value of `data`.

In a similar way, you can declare an entire class as a friend of another class, allowing all member functions of the friend class to access the private and protected members of the other class.

Practice Exercise:

Write a class `Point` that represents a point in 2D space with `x` and `y` coordinates. Implement the following: 1. A constructor that initializes the coordinates. 2. A method to display the coordinates. 3. A friend function that takes two `Point` objects and calculates the distance between them using the formula:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

F Template Classes

Template classes in C++ allow you to create generic classes that can operate with any data type. This is useful for creating data structures and algorithms that are type-independent.

Example: Template Class

```
#include <iostream>
using namespace std;
template <typename T>
class MyArray {
private:
    T* arr;
    int size;
public:
    MyArray(int s) : size(s) {
        arr = new T[size]; // dynamically allocate memory for the
           array
    }

    ~MyArray() {
        delete[] arr; // deallocate memory
    }

    void set(int index, T value) {
        if (index >= 0 && index < size) {
            arr[index] = value; // set value at index
        }
    }

    T get(int index) const {
        if (index >= 0 && index < size) {
            return arr[index]; // get value at index
        }
        return T(); // return default value if index is out of
           bounds
    }
};
int main() {
    MyArray<int> intArray(5); // create an array of integers
    for (int i = 0; i < 5; ++i) {
        intArray.set(i, i * 10); // set values
    }

    for (int i = 0; i < 5; ++i) {
        cout << intArray.get(i) << "␣"; // print values
    }
    cout << endl;

    MyArray<double> doubleArray(3); // create an array of doubles
    doubleArray.set(0, 1.1);
    doubleArray.set(1, 2.2);
    doubleArray.set(2, 3.3);

    for (int i = 0; i < 3; ++i) {
        cout << doubleArray.get(i) << "␣"; // print values
    }
    cout << endl;

    return 0;
}
```

In this example, we define a template class `MyArray` that can hold an array of any data type. The class has methods to set and get values at specific indices. We create instances of `MyArray` for both `int` and `double` types, demonstrating the flexibility of template classes.

G Exceptions in Object-Oriented Programming

Exceptions are special objects or mechanisms used in C++ to signal and handle error conditions or unexpected situations that occur during program execution. When an exception is thrown, the normal flow of the program is interrupted, and control is transferred to a handler that can deal with the problem. This allows programs to separate error-handling code from regular logic, making code more robust and easier to maintain.

Exception handling in C++ allows you to manage errors and exceptional conditions in a structured way. You can define custom exception classes that inherit from the standard `std::exception` class.

Example: Exception Handling

```
#include <iostream>
#include <stdexcept>
using namespace std;
class MyException
: public runtime_error {
public:
    MyException(const string& message)
        : runtime_error(message) {}
};
void riskyFunction() {
    throw MyException("Something␣went␣wrong␣in␣riskyFunction!");
}
int main() {
    try {
        riskyFunction(); // Call a function that may throw an
           exception
    } catch (const MyException& e) {
        cout << "Caught␣MyException:␣" << e.what() << endl; //
           Handle the exception
    } catch (const exception& e) {
        cout << "Caught␣a␣general␣exception:␣" << e.what() << endl;
    }
    return 0;
}
```

In this example, we define a custom exception class `MyException` that inherits from `std::runtime_error`. The function `riskyFunction` throws an instance of `MyException`. In the `main` function, we use a `try-catch` block to handle the exception. If an exception is thrown, control is transferred to the catch block, where we can handle the error gracefully.

Practice Exercise:

Write a class `BankAccount` that represents a bank account with the following features: 1. A constructor that initializes the account with a balance. 2. A method to deposit money into the account. 3. A method to withdraw money from the account, which throws an exception if there are insufficient funds. 4. A method to display the current balance.

H Design Patterns in Object-Oriented Programming

Design patterns are reusable solutions to common problems in software design. They provide a way to structure code and promote best practices in object-oriented programming. Some common design patterns include:

- **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it.
- **Factory Pattern**: Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- **Observer Pattern**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

H.1 Singleton Pattern

The Singleton pattern restricts the instantiation of a class to one single instance and provides a global access point to that instance. This is useful when exactly one object is needed to coordinate actions across the system.

Example: Singleton Pattern

```
#include <iostream>
using namespace std;
class Singleton {
private:
    static Singleton* instance; // static instance pointer
    Singleton() {} // private constructor to prevent instantiation
public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton(); // create instance if it
               doesn't exist
        }
        return instance;
    }

    void showMessage() {
        cout << "Hello from Singleton!" << endl;
    }
};
Singleton* Singleton::instance = nullptr; // initialize static
   instance pointer
int main() {
    Singleton* singleton = Singleton::getInstance(); // get the
       singleton instance
    singleton->showMessage(); // call a method on the singleton
       instance
    return 0;
}
```

In this example, we implement the Singleton design pattern. The `Singleton` class has a private static instance pointer and a private constructor to prevent direct instantiation. The `getInstance` method checks if the instance already exists; if not, it creates a new instance. This ensures that only one instance of the class can exist throughout the program.

H.2 Factory Pattern

The Factory pattern provides a way to create objects without specifying the exact class of object that will be created. It defines an interface for creating an object, but allows subclasses to alter the type of objects that will be created.

Example: Factory Pattern

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() = 0; // pure virtual function
};
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing␣a␣Circle!" << endl;
    }
};
class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing␣a␣Square!" << endl;
    }
};

class ShapeFactory {
public:
    static Shape* createShape(const string& shapeType) {
        if (shapeType == "Circle") {
            return new Circle();
        } else if (shapeType == "Square") {
            return new Square();
        }
        return nullptr; // return null if shape type is unknown
    }
};
int main() {
    Shape* shape1 = ShapeFactory::createShape("Circle"); // create
       a Circle
    shape1->draw(); // call draw method on Circle

    Shape* shape2 = ShapeFactory::createShape("Square"); // create
       a Square
    shape2->draw(); // call draw method on Square

    delete shape1; // clean up memory
    delete shape2; // clean up memory
    return 0;
}
```

In this example, we define a base class `Shape` with a pure virtual function `draw`. We create two derived classes, `Circle` and `Square`, that implement the `draw` method. The `ShapeFactory` class has a static method `createShape` that takes a string parameter and returns a pointer to the appropriate shape object based on the input. This allows us to create different shapes

without knowing their exact types at compile time.

H.3 Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This is useful for implementing event-driven systems.

Example: Observer Pattern

```
#include <iostream>
#include <vector>
using namespace std;
class Observer {
public:
    virtual void update(int value) = 0; // pure virtual function
};
class ConcreteObserver : public Observer {
private:
    string name;
public:
    ConcreteObserver(const string& name) : name(name) {}
    void update(int value) override {
        cout << "Observer " << name << " received update: " <<
           value << endl;
    }
};
class Subject {
private:
    vector<Observer*> observers; // list of observers
    int state; // state of the subject
public:
    void attach(Observer* observer) {
        observers.push_back(observer); // add observer to the list
    }

    void setState(int value) {
        state = value; // update state
        notify(); // notify all observers
    }

    void notify() {
        for (Observer* observer : observers) {
            observer->update(state); // call update on each
               observer
        }
    }
};
int main() {
    Subject subject; // create a subject
    ConcreteObserver observer1("Observer1"); // first observer
    ConcreteObserver observer2("Observer2"); // second observer

    subject.attach(&observer1); // attach observers to the subject
    subject.attach(&observer2);

    subject.setState(10); // change state and notify observers
    subject.setState(20); // change state again and notify
       observers

    return 0;
}
```

In this example, we define an `Observer` interface with a pure virtual function `update`. The `ConcreteObserver` class implements this interface and provides a specific implementation of the `update` method. The `Subject` class maintains a list of observers and notifies them whenever its state changes. When we change the state of the subject, all attached observers receive updates.