

OOPS ASSIGNMENT 2

Q1) What is the difference between structure in C++ and class in C++?

In C++, a structure (struct) and a class are both used to define user-defined data types, but they have some key differences:

1. Access Control:

- In a class, members (variables and functions) have default access control set to private, meaning they are not accessible outside the class unless explicitly specified.
- In a struct, members have default access control set to public, making them accessible from outside the struct without restrictions.

2. Default Constructor:

- A class provides a default constructor (a constructor with no parameters) if you don't define one explicitly.
- A struct also provides a default constructor if not defined, but it behaves as if you defined it with public access.

3. Inheritance:

- Classes support access specifiers like private, protected, and public for inheritance, allowing finer control over member visibility.
- Structs support only public inheritance by default, meaning derived classes have public access to base class members.

4. Use Cases:

- Classes are often used when you need encapsulation, data hiding, and more control over member access.
- Structs are typically used when you have a collection of data with minimal encapsulation, and you want all the data members to be easily accessible.

Q2) Explain the utility of 'this' pointer in detail with example:

The **this** pointer in C++ is a special pointer that points to the current instance of the class or object. It is often used within member functions to access the members (variables and functions) of the current object. Here's a detailed explanation and an example to illustrate its utility:

1. Accessing Object Members:

- When you have a member function in a class, and you want to access or modify a member variable of that class, you can use the **this** pointer to distinguish between the local variables and the member variables that might have the same name.

```

class MyClass {
private:
    int myVar;

public:
    void setVar(int myVar) {
        // Using 'this' to access the member variable
        this->myVar = myVar;
    }
};

```

In this example, `this->myVar` refers to the member variable `myVar`, whereas `myVar` without `this` would refer to the local parameter.

Returning the Current Object:

- You can use `this` to return the current object from a member function. This can be helpful for method chaining.

```

class MyClass {
public:
    MyClass& increment() {
        // Increment the member variable and return the current object
        this->value++;
        return *this;
    }

private:
    int value = 0;
};

```

This allows you to chain multiple calls together like this:

```
cpp
```

```

MyClass obj;
obj.increment().increment();

```

3. Passing Objects as Arguments:

- Sometimes, you might want to pass the current object itself to another function. You can do this using the `this` pointer.

```
cpp Copy code  
  
class MyClass {  
public:  
    void callAnotherFunction() {  
        anotherFunction(this);  
    }  
  
    void anotherFunction(MyClass* objPtr) {  
        // Access or modify 'objPtr' here  
    }  
};
```

This allows you to work with the current object in another function.

4. Preventing Ambiguity:

- In cases where there's ambiguity between a parameter and a member variable, using `this` helps to clarify your intention.

```
cpp Copy code  
  
class MyClass {  
private:  
    int value;  
  
public:  
    void setValue(int value) {  
        // Using 'this' to specify the member variable  
        this->value = value;  
    }  
};
```

In summary, the `this` pointer in C++ is a valuable tool for working with member functions within a class, allowing you to access member variables, return the current object, and pass objects to other functions while avoiding naming conflicts and ambiguity.

Q3) What is a reference variable? Explain with use of reference variable with example:

In C++, a reference variable is an alias or nickname for an existing variable. It provides an alternative name to access the same memory location as the original variable. Reference variables are often used to make function calls more efficient and to enable functions to modify their arguments directly without making copies. Here's an explanation and an example to illustrate the concept of reference variables:

Declaring a Reference Variable:

In C++, you declare a reference variable using the `&` symbol after the data type. For example:

```
cpp Copy code  
  
int originalVar = 42;  
int &refVar = originalVar; // 'refVar' is a reference to 'originalVar'
```


In this example, `refVar` is a reference to `originalVar`, meaning both `refVar` and `originalVar` refer to the same memory location, and changes made to one will affect the other.

Use of Reference Variables:

1. Passing by Reference:

One of the most common uses of reference variables is in function parameters to pass arguments by reference. This allows functions to modify the original values directly, rather than working with copies of the values.

cpp

 Copy code


```
void modifyValue(int &x) {  
    x = x * 2;  
}  
  
int main() {  
    int num = 5;  
    modifyValue(num);  
    cout << num; // Outputs 10  
    return 0;  
}
```

In this example, `modifyValue` takes an integer reference as a parameter and doubles the value of `num` directly within the function.

2. Avoiding Copies:

When you pass objects or complex data structures like classes by reference, you can avoid making unnecessary copies of the data, which can be more efficient in terms of memory and performance.

cpp

 Copy code

```
void modifyVector(vector<int> &vec) {  
    // Modify 'vec' directly  
    vec.push_back(42);  
}  
  
int main() {  
    vector<int> myVector = {1, 2, 3};  
    modifyVector(myVector);  
    for (int num : myVector) {  
        cout << num << " "; // Outputs 1 2 3 42  
    }  
    return 0;  
}
```

In this example, `modifyVector` takes a reference to a vector and appends a value to it without copying the entire vector.

3. Alias for a Variable:

Reference variables can also be used as aliases to make code more readable or to work with complex data structures more easily.

```
cpp Copy code  
  
int &aliasVar = originalVar; // 'aliasVar' is an alias for 'originalVar'  
aliasVar = 55; // Modifies 'originalVar' to 55
```

Here, `aliasVar` acts as an alternative name for `originalVar`.

In summary, reference variables in C++ provide a way to create aliases for existing variables, enabling more efficient function parameter passing, avoiding unnecessary copies, and simplifying code by creating alternative names for variables. They are a powerful feature that enhances both code readability and performance.

Q4) What are the advantages of using new operators compared to function malloc()?

In C++, the `new` operator and the `malloc()` function serve similar purposes, which is to allocate memory dynamically during program execution. However, there are several advantages to using the `new` operator over `malloc()` in modern C++ code:

1. Type Safety:

- `new` operator is type-safe. It allocates memory for a specific data type and returns a pointer of that type. This helps prevent type-related bugs.
- `malloc()` returns a `void*`, so you need to cast it to the appropriate type, which can lead to type-related errors if not done correctly.

```
cpp Copy code  
  
int* intPtr = new int; // Type-safe allocation  
int* intPtr2 = (int*)malloc(sizeof(int)); // Requires casting
```

2. Constructor Invocation:

- When using `new` for objects of user-defined classes, it not only allocates memory but also calls the constructor, initializing the object correctly.
- `malloc()` does not call constructors for objects, so you must manually initialize objects created with `malloc()`.

```
cpp Copy code  
  
class MyClass {  
public:  
    MyClass() {  
        // Constructor logic  
    }  
};  
  
MyClass* obj = new MyClass; // Calls constructor  
MyClass* obj2 = (MyClass*)malloc(sizeof(MyClass)); // No constructor called
```

3. Operator Overloading:

- The `new` operator can be overloaded for custom memory allocation strategies in user-defined classes.
- `malloc()` cannot be overloaded because it is a standard C library function.

4. Automatic Memory Management (RAII):

- When using `new`, you can pair it with smart pointers like `std::unique_ptr` and `std::shared_ptr` to ensure automatic memory management (RAII - Resource Acquisition Is Initialization).
- These smart pointers automatically release the memory when it is no longer needed, preventing memory leaks.
- `malloc()` does not integrate with RAII directly, so manual memory management is required.

```
cpp Copy code  
  
std::unique_ptr<int> intPtr = std::make_unique<int>(); // Automatic memory  
int* intPtr2 = (int*)malloc(sizeof(int)); // Manual memory management
```

In modern C++ programming, the use of the `new` operator, along with smart pointers and RAII principles, is generally favored over `malloc()` because it provides safer, more convenient, and more idiomatic memory management. However, if you're working with C code or need to interface with C libraries, `malloc()` may still be necessary in those specific cases.

Q5) Why do we need static data member and member function?

Static data members and member functions in C++ are associated with the class itself rather than with instances (objects) of the class. They serve several important purposes:

1. Class-Wide Data Sharing:

- Static data members allow you to share data among all instances (objects) of a class. This shared data is common to all objects of the class and retains its value across different instances.

```
class MyClass {
public:
    static int count; // Static data member
    MyClass() {
        count++;
    }
};

int MyClass::count = 0; // Initialize the static data member


int main() {
    MyClass obj1;
    MyClass obj2;
    cout << MyClass::count; // Outputs 2
    return 0;
}
```

In this example, the `count` static data member is shared by all instances of the `MyClass` class, allowing you to keep track of the total number of instances.

2. Utility Functions:

- Static member functions are not associated with any specific instance of the class. They can be called using the class name itself and are often used for utility functions or operations that don't depend on instance-specific data.

cpp

 Copy code

```
class MathUtils {
public:
    static int add(int a, int b) {
        return a + b;
    }
};

int result = MathUtils::add(5, 3); // Call the static member function
```

In this example, the `add` static member function of the `MathUtils` class can be used without creating an instance of the class, making it convenient for utility functions.

In this example, the `add` static member function of the `MathUtils` class can be used without creating an instance of the class, making it convenient for utility functions.

3. Resource Management:

- Static data members can be used for resource management, such as managing a global configuration or a shared resource (e.g., a database connection) across all instances of a class.

4. Constants:

- Static data members can be used to define class-wide constants that are shared among all instances of the class.

cpp

```
class Constants {  
public:  
    static const double PI;  
};  
  
const double Constants::PI = 3.141592653589793;  
  
// Access the constant using the class name  
double circleArea = Constants::PI * radius * radius;
```

5. Memory Efficiency:

- Static data members are stored in a single memory location shared among all instances, which can be more memory-efficient than having each instance store its own copy of the data.

6. Singleton Pattern:

- Static data members and member functions are often used in implementing the Singleton design pattern, ensuring that only one instance of a class exists.

In summary, static data members and member functions provide class-level behavior and data sharing, which is essential for various programming scenarios such as managing shared resources, defining constants, and implementing utility functions. They complement instance-specific data and member functions in C++ classes, allowing for more versatile and efficient class designs.

Q6) Write a program to print no. of objects created and destroyed using constructor and destructor only.

We can create a C++ program to print the number of objects created and destroyed using constructors and destructors. To do this, you can increment a static counter in the constructor and decrement it in the destructor of the class. Here's an example:

```
cpp#include <iostream>
```

```
class MyClass {
```

```
public:
```

```
    static int count; // Static counter to keep track of objects
```

```

MyClass() {

    count++; // Increment the counter when an object is created

}

~MyClass() {

    count--; // Decrement the counter when an object is destroyed

}

};

// Initialize the static counter
int MyClass::count = 0;

int main() {

    MyClass obj1;

    MyClass obj2;

    MyClass obj3;

    std::cout << "Number of objects created:" << MyClass::count << std::endl;

    // When objects go out of scope or are explicitly destroyed, the destructor is called, decrementing the count.

    // Here, obj1 and obj3 go out of scope, so they are destroyed.

    // The count is decremented accordingly.

    return 0;

}

```

In this program, we define a class **MyClass** with a static counter variable **count**. The constructor increments the counter whenever an object is created, and the destructor decrements the counter when an object is destroyed.

When you run this program, it will print the number of objects created at the end of the **main()** function. As objects go out of scope or are explicitly destroyed, the destructor is called for each, and the count is decremented accordingly.

Q7) Explain the brief concept of friend in c++. Write a program to design a class complex to represent complex numbers. The complex class should use an external function to add two complex numbers. The function should return an object of type complex representing sum of two complex numbers.

In C++, the `friend` keyword is used to grant access to non-member functions or other classes to the private and protected members of a class. When a function or class is declared as a friend of another class, it can access the private and protected members of that class as if they were its own members. This is often used for specific scenarios where you need external functions or classes to have privileged access to the internals of a class, but it should be used sparingly to maintain encapsulation.

Here's a basic example of how the `friend` keyword can be used:

```
class MyClass {  
private:  
    int privateData;  
  
public:  
    MyClass(int data) : privateData(data) {}  
  
    // Declare the external function as a friend  
    friend void externalFunction(MyClass obj);  
};  
  
void externalFunction(MyClass obj) {  
    std::cout << "Accessing privateData from externalFunction: " << obj.privateData << std::endl;  
}  
  
int main() {  
    MyClass myObject(42);  
    externalFunction(myObject); // This will work because externalFunction is a friend of MyClass  
    return 0;  
}
```

In this example, `externalFunction` is declared as a friend of the `MyClass` class, so it can access the private member `privateData` of `MyClass` without any issues.

Now, let's design a `Complex` class to represent complex numbers and use an external function to add two complex numbers:

```
#include <iostream>
```

```
class Complex {
```

```
private:
```

```
    double real;
```

```
    double imag;
```

```
public:
```

```
    Complex(double r, double i) : real(r), imag(i) {}
```

```
    // Declare the external function as a friend
```

```
    friend Complex addComplex(const Complex& c1, const Complex& c2);
```

```
    void display() {
```

```
        std::cout << real << " + " << imag << "i" << std::endl;
```

```
    }
```

```
};
```

```
Complex addComplex(const Complex& c1, const Complex& c2) {
```

```
    // Add two complex numbers and return the result
```

```
    double realSum = c1.real + c2.real;
```

```
    double imagSum = c1.imag + c2.imag;
```

```
    return Complex(realSum, imagSum);
```

```
}
```

```
int main() {
```

```
    Complex complex1(3.0, 4.0);
```

```
Complex complex2(1.5, 2.5);

Complex sum = addComplex(complex1, complex2);

std::cout << "Sum of complex numbers: ";

sum.display();

return 0;

}
```

In this program, we've created a `Complex` class to represent complex numbers with real and imaginary parts. We've declared the `addComplex` function as a friend of the `Complex` class. This function takes two `Complex` objects as parameters, adds them, and returns a new `Complex` object representing the sum.

The `main()` function demonstrates how to use this `Complex` class and the `addComplex` function to add two complex numbers.