C++ Cheat Sheet: Structs, Classes, Access Modifiers, and Templates

1. Structs in C++

A `struct` (structure) in C++ is a user-defined data type that allows you to group different data types under a single name. It's a way to create a composite data type that can hold related pieces of information. Structs are primarily used for grouping data, and their members are `public` by default.

Key Characteristics of Structs:

- **Data Grouping:** Structs are ideal for representing a record or a collection of related data items that may be of different types.
- **Default Access Specifier:** All members (variables and functions) declared within a `struct` are `public` by default. This means they can be accessed directly from outside the struct.
- **Value Type:** In C++, structs are typically value types. When you pass a struct to a function or assign it to another struct, a copy of the entire struct is made.
- **Can have Member Functions:** Although primarily for data, structs can also contain member functions, constructors, and destructors, just like classes.

Example of a C++ Struct:

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

// Define a struct to represent a Car
struct Car {
    std::string brand;
    std::string model;
    int year;

    // Member function to display car information
    void displayInfo() {
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: "
<< year << std::endl;
    }
};

int main() {
    // Create an instance of the Car struct
    Car myCar;

    // Access and assign values to members
    myCar.brand = "Toyota";
    myCar.model = "Camry";
    myCar.year = 2022;

    // Call the member function
    myCar.displayInfo(); // Output: Brand: Toyota, Model: Camry, Year: 2022

    // Access members directly (public by default)
    std::cout << "My car's brand is: " << myCar.brand << std::endl;

    return 0;
}
```

2. Classes in C++

A `class` in C++ is also a user-defined data type, but it is a blueprint for creating objects. Classes encapsulate data (member variables) and functions (member methods) that operate on that data into a single unit. Classes are fundamental to Object-Oriented Programming (OOP) and provide features like encapsulation, inheritance, and polymorphism. The members of a `class` are `private` by default.

Key Characteristics of Classes:

- **Object-Oriented Programming (OOP):** Classes are the cornerstone of OOP, enabling the creation of objects that combine data and behavior.

- **Encapsulation:** Classes promote encapsulation by bundling data and the methods that operate on that data within a single unit. This helps in data hiding and protecting data from external, unauthorized access.
- **Default Access Specifier:** All members declared within a `class` are `private` by default. This means they can only be accessed from within the class itself, promoting data hiding.
- **Reference Type (often):** While C++ doesn't strictly enforce reference vs. value types for classes in the same way some other languages do, objects of classes are often manipulated via pointers or references, giving them a

reference-like behavior when passed by reference or pointer.

Example of a C++ Class:

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

// Define a class to represent a Dog
class Dog {
private:
    std::string name;
    std::string breed;
    int age;

public:
    // Constructor
    Dog(std::string n, std::string b, int a) : name(n), breed(b), age(a) {}

    // Member function to display dog information
    void displayInfo() {
        std::cout << "Name: " << name << ", Breed: " << breed << ", Age: " <<
age << std::endl;
    }

    // Getter for name
    std::string getName() {
        return name;
    }

    // Setter for age
    void setAge(int a) {
        if (a > 0) {
            age = a;
        }
    }
};

int main() {
    // Create an object of the Dog class
    Dog myDog("Buddy", "Golden Retriever", 3);

    // Call the public member function
    myDog.displayInfo(); // Output: Name: Buddy, Breed: Golden Retriever, Age:
3

    // Access public member function
    std::cout << "My dog's name is: " << myDog.getName() << std::endl;

    // Try to access private member directly (will cause a compile-time error)
    // std::cout << myDog.age;

    myDog.setAge(4);
    myDog.displayInfo(); // Output: Name: Buddy, Breed: Golden Retriever, Age:
4

    return 0;
}
```

3. Access Modifiers in C++

Access modifiers (also known as access specifiers) in C++ are keywords that set the accessibility of class members (attributes and methods). They control which parts of the program can access the members of a class. C++ provides three types of access modifiers: `public`, `private`, and `protected`.

Understanding Access Modifiers:

- **`public`:** Members declared as `public` are accessible from anywhere, both from within the class and from outside the class. They form the interface of the class.
- **`private`:** Members declared as `private` are accessible only from within the same class. They cannot be accessed directly from outside the class or by derived classes. This is crucial for data hiding and encapsulation.
- **`protected`:** Members declared as `protected` are accessible from within the same class and by derived classes. They are not accessible from outside the class directly.

Here's a table summarizing the accessibility:

Access Modifier	Accessible Within Class	Accessible by Derived Class	Accessible from Outside Class
`public`	Yes	Yes	Yes
`private`	Yes	No	No
`protected`	Yes	Yes	No

4. Templates in C++

Templates are a powerful feature in C++ that allow you to write generic programs. They enable you to define functions and classes that operate with generic types, rather than specific data types. This means you can write a single function or class definition that works with any data type, avoiding code duplication and making your code more flexible and reusable. Templates are a form of compile-time polymorphism.

Types of Templates:

1. **Function Templates:** These are used to write generic functions that can handle different data types.
2. **Class Templates:** These are used to write generic classes that can work with different data types.

Function Templates Example:

Let's say you want to write a function that finds the maximum of two values. Without templates, you would need to write separate functions for `int`, `double`, `float`, etc.

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

// Function template to find the maximum of two values
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    // Using the function template with integers
    std::cout << "Max of 5 and 10: " << maximum(5, 10) << std::endl; // Output:
10

    // Using the function template with doubles
    std::cout << "Max of 3.14 and 2.71: " << maximum(3.14, 2.71) << std::endl;
// Output: 3.14

    // Using the function template with characters
    std::cout << "Max of 'a' and 'z': " << maximum('a', 'z') << std::endl; //
Output: z

    // Using the function template with strings
    std::string s1 = "hello";
    std::string s2 = "world";
    std::cout << "Max of \"hello\" and \"world\": " << maximum(s1, s2) <<
std::endl; // Output: world

    return 0;
}
```

Explanation:

The `template <typename T>` syntax declares `T` as a template type parameter. When `maximum(5, 10)` is called, the compiler deduces `T` to be `int` and generates a version of `maximum` for integers. Similarly, for `maximum(3.14, 2.71)`, `T` becomes `double`.

Class Templates Example:

Class templates are useful when a class needs to operate on data of various types, such as a generic container like a stack, queue, or a pair.

```cpp
#include <iostream>

// Class template for a simple Pair
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;

public:
    Pair(T1 f, T2 s) : first(f), second(s) {}

    void display() {
        std::cout << "First: " << first << ", Second: " << second << std::endl;
    }

    T1 getFirst() {
        return first;
    }

    T2 getSecond() {
        return second;
    }
};

int main() {
    // Create a Pair of integers
    Pair<int, int> p1(10, 20);
    p1.display(); // Output: First: 10, Second: 20

    // Create a Pair of a string and a double
    Pair<std::string, double> p2("Hello", 3.14);
    p2.display(); // Output: First: Hello, Second: 3.14

    // Accessing members
    std::cout << "First element of p1: " << p1.getFirst() << std::endl; //
Output: 10
    std::cout << "Second element of p2: " << p2.getSecond() << std::endl; //
Output: 3.14

    return 0;
}
```

Explanation:

The `template <typename T1, typename T2>` syntax allows the `Pair` class to be parameterized with two different types, `T1` and `T2`. When `Pair<int, int> p1(10, 20);` is declared, the compiler generates a `Pair` class where both `first` and `second` are `int`. Similarly, for `Pair<std::string, double> p2("Hello", 3.14);`, the compiler generates a `Pair` class with `first` as `std::string` and `second` as `double`.

Conclusion

Structs, Classes, Access Modifiers, and Templates are fundamental concepts in C++ that empower developers to write organized, efficient, and reusable code. Structs are primarily for data aggregation with public members by default, while classes are the foundation of OOP, providing encapsulation and data hiding with private members by default. Access modifiers (`public`, `private`, `protected`) are crucial for controlling visibility and enforcing encapsulation. Templates provide a powerful mechanism for generic programming, allowing you to write code that works independently of data types, thereby reducing redundancy and enhancing flexibility. Mastering these concepts is essential for effective C++ programming.

References

[1] W3Schools. "C++ Structures (struct)". Available at: https://www.w3schools.com/cpp/cpp_structs.asp [2] GeeksforGeeks. "Difference Between Structure and Class in C++". Available at: https://www.geeksforgeeks.org/cpp/structure-vs-class-in-cpp/ [3] W3Schools. "C++ Classes and Objects". Available at: https://www.w3schools.com/cpp/cpp_classes.asp [4] GeeksforGeeks. "Access Modifiers in C++". Available at: https://www.geeksforgeeks.org/cpp/access-modifiers-in-c/ [5] W3Schools. "C++ Templates". Available at: https://www.w3schools.com/cpp/cpp_templates.asp [6] GeeksforGeeks. "Templates in C++". Available at: https://www.geeksforgeeks.org/cpp/templates-cpp/

Optional Section: The 4 Pillars of Object-Oriented Programming (OOP) in C++

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. C++ is a multi-paradigm language that strongly supports OOP principles. The four fundamental pillars of OOP are Encapsulation, Abstraction, Inheritance, and Polymorphism.

1. Encapsulation

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of data. This is achieved using access modifiers (`public`, `private`, `protected`).

Key Idea: Data hiding and bundling data with methods.

Example (from Class section):

```
 class Dog {
private:
    std::string name; // private data
    // ...
public:
    void displayInfo(); // public method to access private data
    // ...
};
```

2. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary and essential features of an object. It focuses on what the object does rather than how it does it. In C++, abstraction can be achieved using abstract classes and interfaces (pure virtual functions).

Key Idea: Showing only essential features, hiding complexity.

Example (Abstract Class):

```cpp
#include <iostream>

class Shape { // Abstract class
public:
    // Pure virtual function (makes Shape an abstract class)
    virtual double area() = 0;
    virtual void draw() = 0;

    void displayMessage() {
        std::cout << "This is a shape." << std::endl;
    }
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override {
        return 3.14159 * radius * radius;
    }
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class
    Circle c(5);
    c.draw();
    std::cout << "Area of circle: " << c.area() << std::endl;
    return 0;
}
```

3. Inheritance

Inheritance is a mechanism that allows a new class (derived or child class) to inherit properties and behaviors (member variables and methods) from an existing class (base or parent class). This promotes code reusability and establishes an "is-a" relationship between classes.

Key Idea: Code reusability and hierarchical relationships.

Example:

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

class Animal {
public:
    std::string name;
    Animal(std::string n) : name(n) {}
    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

class Dog : public Animal { // Dog inherits from Animal
public:
    std::string breed;
    Dog(std::string n, std::string b) : Animal(n), breed(b) {}
    void bark() {
        std::cout << name << " is barking." << std::endl;
    }
};

int main() {
    Dog myDog("Buddy", "Golden Retriever");
    myDog.eat();  // Inherited method
    myDog.bark(); // Dog's own method
    return 0;
}
```

4. Polymorphism

Polymorphism means "many forms." It allows objects of different classes to be treated as objects of a common base class. This enables a single interface to represent different underlying forms (data types). In C++, polymorphism is primarily achieved through virtual functions and function overloading/overriding.

Key Idea: One interface, multiple forms.

Types of Polymorphism:

- **Compile-time Polymorphism (Static Polymorphism):** Achieved through function overloading and operator overloading. The decision of which function to call is made at compile time.
- **Run-time Polymorphism (Dynamic Polymorphism):** Achieved through virtual functions and pointers/references to base classes. The decision of which function to call is made at runtime.

Example (Run-time Polymorphism with Virtual Functions):

```cpp
#include <iostream>

class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal makes a sound." << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog barks!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat meows!" << std::endl;
    }
};

int main() {
    Animal* myAnimal;

    Dog myDog;
    myAnimal = &myDog;
    myAnimal->makeSound(); // Output: Dog barks! (Runtime decision)

    Cat myCat;
    myAnimal = &myCat;
    myAnimal->makeSound(); // Output: Cat meows! (Runtime decision)

    return 0;
}
```

In this example, `makeSound()` is a virtual function. When `myAnimal->makeSound()` is called, the actual method executed depends on the type of object `myAnimal` points to at runtime, demonstrating polymorphism.

References

[1] W3Schools. "C++ Structures (struct)". Available at: https://www.w3schools.com/cpp/cpp_structs.asp [2] GeeksforGeeks. "Difference Between Structure and Class in C++". Available at: https://www.geeksforgeeks.org/cpp/structure-vs-class-in-cpp/ [3] W3Schools. "C++ Classes and Objects". Available at: https://www.w3schools.com/cpp/cpp_classes.asp [4] GeeksforGeeks. "Access Modifiers in C++". Available at: https://www.geeksforgeeks.org/cpp/access-modifiers-in-c/ [5] W3Schools. "C++

Templates". Available at: https://www.w3schools.com/cpp/cpp_templates.asp [6] GeeksforGeeks. "Templates in C++". Available at: https://www.geeksforgeeks.org/cpp/templates-cpp/ [7] GeeksforGeeks. "Object Oriented Programming (OOP) in C++". Available at: https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-cpp/