Okay, I've reviewed the notes and examples thoroughly. Here's the final version, incorporating modifications and clarifications. I've marked the sections with significant changes with " **Modified:** " to help you identify them easily.

C++ Study Notes (Final Version)

1. Static Variables and Functions

- **Static Variables:**
  - Should be initialized outside the class definition.
  - Accessed using the scope resolution operator (::).
  - Can be modified within the main() function using the class name or an object of the class.
  - Accessible within the class's member functions.
  - Example:

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

class Human {
public:
    static int count;

    Human() {
```

```
        count++; // Increment count when a Human object is created
    }

    ~Human() {
        count--; // Decrement count when a Human object is destroyed
    }
};

int Human::count = 0; // Initialization

int main() {
    cout << "Initial Human count: " << Human::count << endl; // Accessing static variable

    Human human1;
    Human human2;

    cout << "Human count after creating two objects: " << Human::count << endl;

    return 0;
}
```

- **Static Functions:**
  - Can only utilize static variables.
  - Static variables can be modified and accessed within class member functions.
  - Example:

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

class MyClass {
private:
    static int staticVar;

public:
    static void staticFunction() {
```

```
        cout << "Static variable value: " << staticVar << endl;
    }

    MyClass() {
        staticVar++;
    }
};

int MyClass::staticVar = 0; // Initialize static variable

int main() {
    MyClass::staticFunction(); // Call static function without object
    MyClass obj1;
    MyClass obj2;
    MyClass::staticFunction();
    return 0;
}
```

2. Default Values for Function Parameters

- Functions can be defined with default parameter values.
  - Example: void fun(int a, int b = 20);
  - C++

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

void fun(int a, int b = 20) {
    cout << "a: " << a << ", b: " << b << endl;
}

int main() {
    fun(10);    // Output: a: 10, b: 20 (uses default value for b)
    fun(10, 30); // Output: a: 10, b: 30
    return 0;
}
```

3. Encapsulation

- Data hiding mechanism.
- Restricting access to variables and member functions from outside the class.
- Only member functions of the same class can access private members.
- The private access specifier is used for data hiding.
- Notes explanation: Encapsulation restricts access to members, ensuring only member functions of a class can access its private members.
  - Example:

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

class MyClass {
private:
    int secretValue;

public:
    void setValue(int value) {
        secretValue = value;
    }

    int getValue() const {
        return secretValue;
```

```
    }
};

int main() {
    MyClass obj;
    obj.setValue(42);
    cout << "The secret value is: " << obj.getValue() << endl;
    return 0;
}
```

4. Friend Functions and Friend Classes

- friend declarations grant external functions or classes access to the private members of a class.
  - Example: friend void fun();
  - C++

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

class MyClass {
private:
    int privateVar;

public:
    MyClass(int value) : privateVar(value) {}
    friend void friendFunction(MyClass obj);
};

void friendFunction(MyClass obj) {
    cout << "Accessing private variable: " << obj.privateVar << endl;
}

int main() {
    MyClass obj(10);
```

```
    friendFunction(obj);
    return 0;
}
```

5. Constructors

- Special member functions with the same name as the class.
- Do not have a return type.
- Automatically invoked when an object of the class is created.
- A default constructor is provided by the compiler if no constructor is explicitly defined.
- Used to initialize the data members of an object.
- Types:
  - Constructors with no statements (empty constructors).
  - Constructors with parameters (parameterized constructors).
  - Example:

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

class Human {
public:
    string name;
    int age;
```

```
    Human() { // Constructor with no statements (Default Constructor)
        name = "Unknown";
        age = 0;
        cout << "Default constructor called" << endl;
    }

    Human(string iname) { // Parameterized constructor
        name = iname;
        age = 25;
        cout << "Parameterized constructor called for " << name << endl;
    }

    Human(string iname, int iage) : name(iname), age(iage) { // Another
Parameterized constructor
        cout << "Parameterized constructor called for " << name << " with age " << age <<
endl;
    }
};

int main() {
    Human defaultHuman;
    Human jagHuman("Jag");
    Human person1("Alice", 30);
    return 0;
}
```

6. Destructors

- Member functions used to deallocate memory and perform cleanup when an object is destroyed.
- Automatically called when an object goes out of scope or is explicitly deleted.
- Crucial for releasing resources acquired by the

object.

- If an object is created using the new keyword, delete must be used to invoke the destructor and free the allocated memory.
- Notes emphasize:
  - Destructors are called when objects go out of scope.
  - They release memory allocated to object members.
  - delete is necessary to destroy objects created with new.
  - Example:

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

class MyClass {
public:
    MyClass() {
        cout << "Constructor called" << endl;
        // Allocate memory (e.g., using new) if needed
    }

    ~MyClass() {
        cout << "Destructor called" << endl;
        // Deallocate memory (e.g., using delete) if allocated in the constructor
    }
};

int main() {
    MyClass obj1; // Destructor called when obj1 goes out of scope
```

```
    MyClass *obj2 = new MyClass();
    delete obj2; // Destructor called when obj2 is deleted
    return 0;
}
```

7. Function Overloading

- Defining multiple functions with the same name but with different signatures (different number, type, or order of parameters).
- Enables using the same function name for different operations based on the arguments provided.
- Notes state: Function overloading involves functions with the same name but different parameters.
    - Example:

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

class Calculator {
public:
    int add(int a, int b) {
        cout << "Add(int, int) called" << endl;
        return a + b;
    }

    double add(double a, double b) {
        cout << "Add(double, double) called" << endl;
        return a + b;
    }
```

```
    int add(int a, int b, int c) {
        cout << "Add(int, int, int) called" << endl;
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(1, 2) << endl;
    cout << calc.add(1.5, 2.5) << endl;
    cout << calc.add(1, 2, 3) << endl;
    return 0;
}
```

8. Operator Overloading

- Redefining the behavior of operators for user-defined types.
- Allows operators to work with objects of classes.
- Example: Overloading the + operator to add two objects.
  - n1.operator+(n2);
  - n2 + n3
  - n1.operator+(n2) + n3
- Binary operator overloading:

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

class Number {
public:
```

```
    int num;

    Number(int num) : num(num) {}

    Number operator+(Number obj) {
        Number temp;
        temp.num = this->num + obj.num;
        return temp;
    }
};

int main() {
    Number n1(10), n2(5), n3(20);
    Number sum = n1 + n2; // Operator overloading
    cout << "Sum: " << sum.num << endl;      // Output: 15

    Number total = n1 + n2 + n3;
    cout << "Total: " << total.num << endl; // Output: 35
    return 0;
}
```

- Prefix and postfix increment operator overloading (notes include a table-like representation).

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

class Counter {
private:
    int count;

public:
    Counter() : count(0) {}

    // Prefix increment (++obj)
    Counter& operator++() {
```

```
        ++count;
        return *this;
    }

    // Postfix increment (obj++)
    Counter operator++(int) {
        Counter temp = *this;
        ++count;
        return temp;
    }

    int getCount() const {
        return count;
    }
};

int main() {
    Counter c1, c2;

    cout << "Initial count: c1 = " << c1.getCount() << ", c2 = " << c2.getCount() << endl; // 0, 0

    ++c1;
    c2++;

    cout << "After increment: c1 = " << c1.getCount() << ", c2 = " << c2.getCount() << endl; // 1, 1

    Counter c3 = c2++;
    cout << "c3 = c2++ : c3 = " << c3.getCount() << ", c2 = " << c2.getCount() << endl; // c3 = 1, c2 = 2
    return 0;
}
```

9. Access Specifiers

- Keywords that define the accessibility of class

members.

- public: Members are accessible from anywhere.
- protected: Members are accessible within the class and derived classes.
- private: Members are only accessible within the class itself.
- Notes include a table summarizing access levels.
    - C++

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

class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};

class Derived : public Base {
public:
    void accessProtected() {
        protectedVar = 10; // OK
        publicVar = 5;    // OK
        // privateVar = 5; // Error: Cannot access private members
    }
};

int main() {
    Base b;
    b.publicVar = 5; // OK
    // b.protectedVar = 5; // Error: Cannot access protected members from outside
    // b.privateVar = 5;   // Error: Cannot access private members from outside
```

```
    Derived d;
    d.publicVar = 10; // OK
    d.accessProtected();
    return 0;
}
```

10. Inheritance

- A mechanism for creating new classes (derived classes) from existing classes (base classes).
- Derived classes inherit the members of the base class.
- Types of Inheritance:
  - **Public Inheritance:**
    - Public members of the base class remain public in the derived class.
    - Protected members of the base class remain protected in the derived class.
    - Private members of the base class are not inherited.
    - C++

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

class Base {
public:
    int publicVar;
protected:
    int protectedVar;
```

```
private:
    int privateVar;
};

class Derived : public Base {
public:
    void access() {
        publicVar = 1;   // OK
        protectedVar = 2; // OK
        // privateVar = 3;  // Error
    }
};

int main() {
    Derived d;
    d.publicVar = 10; // OK
    return 0;
}
```

- **Protected Inheritance:**
  - Public and protected members of the base class become protected members of the derived class.
  - Private members are not inherited.
  - C++

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

class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
```

```
};

class Derived : protected Base {
public:
    void access() {
        publicVar = 1;   // OK, but publicVar is protected in Derived
        protectedVar = 2; // OK
        // privateVar = 3;  // Error
    }
};

int main() {
    Derived d;
    // d.publicVar = 10; // Error: publicVar is protected in Derived
    return 0;
}
```

- **Private Inheritance:**
  - Public and protected members of the base class become private members of the derived class.
  - Private members are not inherited.
  - C++

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

class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};
```

```
class Derived : private Base {
public:
    void access() {
        publicVar = 1;   // OK, but publicVar is private in Derived
        protectedVar = 2; // OK
        // privateVar = 3;  // Error
    }
};

int main() {
    Derived d;
    // d.publicVar = 10; // Error: publicVar is private in Derived
    return 0;
}
```

- Upcasting can be used to initialize data in further derived classes.
- Access levels of inherited members can be changed in the derived class.
  - Example:

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

class Person {
protected:
   string name;

public:
   void setName(string iname) {
      name = iname;
   }
};

class Student : public Person {
public:
```

```
    Person::name;    // Now 'name' is public in Student
    Person::setName; //  setName is public in Student
};

int main() {
    Student s;
    s.name = "Alice";    // OK
    s.setName("Bob"); // OK
    return 0;
}
```

- Multiple Inheritance: A class can inherit from multiple base classes.
  - Constructor and destructor call order: Base constructor(s), Derived constructor, Derived destructor, Base destructor(s).
  - C++

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

class Base1 {
public:
    Base1() {
        cout << "Base1 Constructor" << endl;
    }

    ~Base1() {
        cout << "Base1 Destructor" << endl;
    }
};

class Base2 {
public:
    Base2() {
        cout << "Base2 Constructor" << endl;
```

```
    }

    ~Base2() {
        cout << "Base2 Destructor" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    Derived() {
        cout << "Derived Constructor" << endl;
    }

    ~Derived() {
        cout << "Derived Destructor" << endl;
    }
};

int main() {
    Derived d;
    return 0;
}
```

- Passing values to base class constructors from the derived class:
  - C++

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

// 10
```

Sources

1. https://github.com/ranjit9876/CPP
2. https://github.com/TenexDesigns/TypeScript
3. https://github.com/AsAdityaSonu/C_PlusPlus_Assignments
4. https://thejat.in/learn/2-inheritance
5.

https://www.upgrad.com/tutorials/software-engineering/cpp-tutorial/hierarchical-inheritance-in-cpp/
6. https://slideplayer.com/slide/16851545/
7. https://fliphtml5.com/ebybf/fniu/programmingwithc%2B%2B/137/
8. https://quizlet.com/837651803/cs341-exam-butler-university-dr-ryan-flash-cards/
9. https://github.com/AMARTYA2020/Object-oriented-programming-code-CPPs
10. https://github.com/Anshul20052000/Cpp-Programs