1. Create a simple class with an overloaded **operator++**. Try calling this operator in both pre- and postfix form and see what kind of compiler warning you get.

2. Create a simple class containing an **int** and overload the **operator+** as a member function. Also provide a **print ( )** member function that takes an **ostream&** as an argument and prints to that **ostream&**. Test your class to show that it works correctly.

3. Add a binary **operator-** to Exercise 2 as a member function. Demonstrate that you can use your objects in complex expressions like **a + b – c**.

4. Add an **operator++** and **operator--** to Exercise 2, both the prefix and the postfix versions, such that they return the incremented or decremented object. Make sure that the postfix versions return the correct value.

5. Modify the increment and decrement operators in Exercise 4 so that the prefix versions return a non-**const** reference and the postfix versions return a **const** object. Show that they work correctly and explain why this would be done in practice.

6. Change the **print ( )** function in Exercise 2 so that it is the overloaded **operator<<** as in **Iostream Operator Overloading.cpp**

7. Modify Exercise 3 so that the **operator+** and **operator-** are non-member functions. Demonstrate that they still work correctly.

8. Add the unary **operator-** to Exercise 2 and demonstrate that it works correctly.

9. Create a class that contains a single **private char**. Overload the iostream operators **<<** and **>>** (as in **Iostream Operator Overloading.cpp** ) and test them. You can test them with **fstreams**, **string stream**s, and **cin** and **cout**.

10. Determine the dummy constant value that your compiler passes for postfix **operator++** and **operator--**.

11. Write a **Number** class that holds a **double**, and add overloaded operators for **+, –, \*, /**, and assignment. Choose the return values for these functions so that expressions can be chained together, and for efficiency. Write an automatic type conversion **operator int( )**.

12. Modify Exercise 11 so that the *return value optimization* is used, if you have not already done so.

13. Create a class that contains a pointer, and demonstrate that if you allow the compiler to synthesize the **operator=** the result of using that operator will be pointers that are aliased to the same storage. Now fix the problem by defining your own **operator=** and demonstrate that it corrects the aliasing. Make sure you check for self-assignment and handle that case properly.

14. Write a class called **Bird** that contains a **string** member and a **static int**. In the default constructor, use the **int** to automatically generate an identifier that you build in the **string**, along with the name of the class (**Bird #1**, **Bird #2**, etc.). Add an **operator<<**for **ostream**s to print out the **Bird** objects. Write an assignment **operator=** and a copyconstructor. In **main( )**, verify that everything works correctly.

15. Write a class called **BirdHouse** that contains an object, a pointer and a reference for class **Bird** from Exercise 14. The constructor should take the three **Bird**s as arguments. Add an **operator<<** for **ostream**s for **BirdHouse**. Disallow the assignment **operator=** and copy-constructor. In **main( )**, verify that everything works correctly. Make sure that you can chain assignments for **BirdHouse** objects and build expressions involving multiple operators.

16. Add an **int** data member to both **Bird** and **BirdHouse** in Exercise 15. Add member operators **+**, **-**, **\***, and **/** that use the **int** members to perform the operations on the respective members. Verify that these work.

17. Repeat Exercise 16 using non-member operators.

18. Add an **operator--**to **SmartPointer.cpp** and **NestedSmartPointer.cpp**.

19. Modify **CopyingVsInitialization.cpp** so that all of the constructors print a message that tells you what’s going on. Now verify that the two forms of calls to the copyconstructor (the assignment form and the parenthesized form) are equivalent.

20. Attempt to create a non-member **operator=** for a class and see what kind of compiler message you get.

21. Create a class with an assignment operator that has a second argument, a **string** that has a default value that says “op= call.” Create a function that assigns an object of your class to another one and show that your assignment operator is called correctly.

22. In **CopyingWithPointers.cpp**, remove the **operator=** in **DogHouse** and show that the compiler-synthesized **operator=** correctly copies the **string** but simply aliases the **Dog** pointer.

23. In **ReferenceCounting.cpp**, add a **static int** and an ordinary **int** as data members to both **Dog** and **DogHouse**. In all constructors for both classes, increment the **static int** and assign the result to the ordinary **int** to keep track of the number of objects that have been created. Make the necessary modifications so that all the printing statements will say the **int** identifiers of the objects involved.

24. Create a class containing a **string** as a data member. Initialize the **string** in the constructor, but do not create a copy-constructor or **operator=**. Make a second class that has a member object of your first class; do not create a copy-constructor or **operator=** for this class either. Demonstrate that the copy-constructor and **operator=** are properly synthesized by the compiler.

25. Combine the classes in **OverloadingUnaryOperators.cpp** and **Integer.cpp**.

26. Modify **PointerToMemberOperator.cpp** by adding two new member functions to **Dog** that take no arguments and return **void**. Create and test an overloaded **operator->\*** that works with your two new functions.

27. Add an **operator->\***to **NestedSmartPointer.cpp**.

28. Create two classes, **Apple** and **Orange**. In **Apple**, create a constructor that takes an **Orange** as an argument. Create a function that takes an **Apple** and call that function with an **Orange** to show that it works. Now make the **Apple** constructor **explicit** to demonstrate that the automatic type conversion is thus prevented. Modify the call to your function so that the conversion is made explicitly and thus succeeds.

29. Add a global **operator\*** to **ReflexivityInOverloading.cpp** and demonstrate that it is reflexive.

30. Create two classes and create an **operator+** and the conversion functions such that addition is reflexive for the two classes.

31. Fix **TypeConversionFanout.cpp** by creating an explicit function to call to perform the type conversion, instead of one of the automatic conversion operators.

32. Write simple code that uses the **+**, **-**, **\***, and **/** operators for **double**s. Figure out how your compiler generates assembly code and look at the assembly language that’s generated to discover and explain what’s going on under the hood.