**CS 37 Notes Part 4**

Overloaded Operators

Using overloaded operators makes the main easier to write and more consistent.

Without overloaded operators, the main programmer needs to know the syntax for calling each method.

|  |  |
| --- | --- |
| Without overloaded operators the main could look like: | With overloaded operators, the main could look like the following: |
| s.loadData();  s.printData();  sum.add(a,b);> | cin >> s;  cout << s;  sum = a + b; |

Any operator can be overloaded for an object.

Before getting to the overloaded operators programs, there is some background information needed: the this pointer and friend functions

**this1.cpp**

This program’s only purpose is to show you that the **this** pointer does exist. Of course, you would not use the this pointer just to print out data members. Other object-oriented languages also have an equivalent to the this pointer, for example, in Visual Basic, it is called me instead of this. The *this* pointer is the starting address of the object. The this pointer can reference both data members and methods of an object. We will be using the statement “return \*this”; later to return a modified object from an operator overloading method.

For example, for students, the ++ operator adds one point to each test score. The ++ method (preincrement) must first add one to each test score and then return the modified object to the calling function. The this pointer also allows cascading method (member function) calls. Cascading means calling more than one method in one line of code.

Eg s.setData(10, 20, 30).printData();

The setData method is called first and must return \*this as the object is modified and these modified values need to be printed in the printData method. If you do not want to allow cascading by the main programmer, then you do not need the this pointer. However, it is much better to allow the programmer the opportunity to cascade.

Looking at the this1.cpp program, the only change from previous programs is in the printData() method. Instead of just saying cout << t1 << t2 << t3; you can instead say:

cout << this ->t1 << this ->t2 << this ->t3;

|  |
| --- |
| #include<iostream>  using **namespace** std;  **class** Student{  **public:**      Student();*//default constructor*  **void** getData(**int**, **int**, **int**);  **void** printData();  **private:**  **int** test1, test2, test3;  **float** avg;  };  Student::Student()      { test1 = test2 = test3 = 1;        avg = (test1 + test2 + test3) / (**float**)3; }  **void** Student::getData(**int** a, **int** b, **int** c)  {      test1 = a;      test2 = b;      test3 = c;      avg = (test1 + test2 + test3) / (**float**)3;  }  **void** Student::printData()  {      cout << "test 1 is " << this->test1 << endl;      cout << "test 2 is " << this->test2 << endl;      cout << "test 3 is " << this->test3 << endl;      cout << "the average is " << this->avg << endl;  }  *// Every object has access to its own address through the this pointer.*  *// can reference both the data members and member functions of an object.*  *// Will be used later to prevent an object from being assigned to itself*  *// in operator overloading.  Also, this enables cascading member function calls.*  *// eg two.setData(50,60,70).printData();*  **int** main()  {      Student one, two;      cout << "student one data is" << endl;      one.printData();        two.getData(10, 15, 30);      cout <<endl << "student two data is"  <<endl;      two.printData();        return 0;  }  */\**  *student one data is*  *test 1 is 1*  *test 2 is 1*  *test 3 is 1*  *the average is 1*  *student two data is*  *test 1 is 10*  *test 2 is 15*  *test 3 is 30*  *the average is 18.3333*  *Press any key to continue*  *\*/* |

**this2.cpp**

This program uses **set** and **get** functions with the **this** pointer to allow cascading in the main. For example, in the main, you can do the following:

s.setTest1(a).setTest2(b).setTest3(c).printData(); to call all four methods in one line of code.

    two.setData(50,60,70).printData();

Please note the &’s in the prototypes and methods where return \*this; appears. You only need the &’s when a method does return \*this.

    Student **&**setData(**int**, **int**, **int**);

    Student **&**setTest1(**int**);*// set functions*

    Student **&**setTest2(**int**);

    Student **&**setTest3(**int**);

Thus:

    return \*this;*// enables cascading*

Also, note the const on the get functions. When you place a const on the end of the prototype, the const means that the current object cannot change values. That is, \*this stays the same values in the method.

**int** getTest1() **const**;*//get functions*

**int** getTest2() **const**;*// const means object cannot be modified*

**int** getTest3() **const**;

    **void** printData() **const**;

THEREFORE Because the return \*this; is being used in the set functions, you can do the following in the main:

s.setTest1(a).setTest2(b).setTest3(c).printData();

*OUTLINE OF MAIN*

The setTest1 method is called first and the modified object is returned and sent to the setTest2 method and this modified object is sent to the setTest3 method.

    one.setTest1(10).setTest2(20).setTest3(30);*// cascade member function calls*

Finally, the modified object is sent to the printData() method and printed.

    one.printData();

*The other background information needed is friends. There are friend functions which we will need for operator overloading and friend classes.*

**friend1.cpp**

Friend functions (NOT METHODS) are functions defined outside of (the private and public parts of) a class which CAN STILL have access to private arts of a class. *We will need friend functions for overloading stream insertion and stream extraction later.*

These operator overloaded functions must have access to the private parts of the class. It would be ridiculous if we could not cin or cout any private data members of a class.

*OUTLINE OF MAIN*

To declare a friend function, the first line of the class should have a friend declaration which is the keyword friend followed by the prototype of the friend function to be.

**class** Student{

**friend** **void** change( Student **&**, **int**,**int**,**int**);*// friend declaration*

There is a function **change** written by the main programmer to give extra credit to each of three exam scores. Because function **change** will be a friend function, the main programmer can access the private data members directly. For example, s.t1 += add1; where t1 is a private data member.

*// friend functions (NOT METHODS)*

**void** change(Student **&**pupil, **int** add1, **int** add2, **int** add3)

{

*// add extra credit using friend function*

    pupil.test1 += add1;

    pupil.test2 += add2;

    pupil.test3 += add3;

}

Eg friend void change(Student &, int, int, int); where the three int parameters are the amount of extra credit to give to each of the three tests.

friend1 has the same get and set functions from previous programs.

*// Declaration functions*

**void** Student::setTest1(**int** a) { test1 = a;}

**void** Student::setTest2(**int** b) { test2 = b;}

**void** Student::setTest3(**int** c) { test3 = c;}

*// Accessing functions*

**int** Student::getTest1() { return test1; }

**int** Student::getTest2() { return test2; }

**int** Student::getTest3() { return test3; }

|  |
| --- |
| #include<iostream>  #include<iomanip>  using **namespace** std;  **class** Student{  **friend** **void** change( Student **&**, **int**,**int**,**int**);*// friend declaration*  *// a friend function of a class is defined outside of the PRIVATE AND PUBLIC*  *// but it's still a part of a class ~~class~~.*  *// but has access to private members of the class.*  **public:**        Student(**int** = 0, **int** = 0, **int** = 0);*//default arguments with constructors*      ~Student();*// destructor*  **void** setData(**int**, **int**, **int**);  **void** setTest1(**int**);*// set functions*  **void** setTest2(**int**);  **void** setTest3(**int**);    **int** getTest1();*//get functions*  **int** getTest2();  **int** getTest3();  **void** printData();  **private:**  **int** test1, test2, test3;  };  Student::Student(**int** a, **int** b, **int** c) { setData(a,b,c); }  Student::~Student()*// destructor - called when object is destroyed*  {      cout << setw(10) <<test1 << setw(10) << test2;      cout << setw(10) << test3 << " destroyed" <<endl;  }  **void** Student::setData(**int** a, **int** b, **int** c)  {      setTest1(a);      setTest2(b);      setTest3(c);  }  *// Declaration functions*  **void** Student::setTest1(**int** a) { test1 = a;}  **void** Student::setTest2(**int** b) { test2 = b;}  **void** Student::setTest3(**int** c) { test3 = c;}    *// Accessing functions*  **int** Student::getTest1() { return test1; }  **int** Student::getTest2() { return test2; }  **int** Student::getTest3() { return test3; }  *// Print*  **void** Student::printData()  {      cout << "test 1, 2, 3 are " << setw(10) << test1;      cout << setw(10) << test2 << setw(10) << test3 << endl;  }  *// friend functions (NOT METHODS)*  **void** change(Student **&**pupil, **int** add1, **int** add2, **int** add3)  {  *// add extra credit using friend function*      pupil.test1 += add1;      pupil.test2 += add2;      pupil.test3 += add3;  }  **int** main()  {      Student two;  **int** qz1, qz2, qz3;      two.setTest1(10);      two.setTest2(20);      two.setTest3(30);      cout <<endl << "student two data is"  <<endl;      two.printData();      cout <<endl;      qz1 = two.getTest1();      qz2 = two.getTest2();      qz3 = two.getTest3();      cout << "using the get functions" << endl;      cout << setw(10) << qz1 << setw(10) << qz2;      cout << setw(10) << qz3 << endl << endl;      change(two, 10, 20, 30);      cout << "after the change function "  << endl;      two.printData();      cout << endl;      return 0;  }  */\**  *student two data is*  *test 1, 2, 3 are         10        20        30*  *using the get functions*  *10        20        30*  *after the change function*  *test 1, 2, 3 are         20        40        60*  *20        40        60 destroyed*  *Press any key to continue*  *\*/* |

**friend2.cpp** (optional program)

This program presents a friend class. There is a student class declared in the program with two methods to return the largest and smallest test scores.

Another programmer needs to maintain stock values. The second class declared is for the stocks. This second programmer also needs methods to return the largest and smallest values for the stock values. The stock programmer should be able to use the same methods in the student class to find the stock’s largest and smallest values. It is always good to use reusable code if possible.

Operator Overloading Programs

overld1.cpp only overloads the stream insertion and stream extraction operators.

C++ requires the overloaded stream insertion and stream extraction operator code to be friend functions. All the other overloaded operators will be class methods.

overld2a and overld2b are identical with different mains only. += ++ preincrement and ++ postincrement along with stream insertion and stream extraction are overloaded. This program’s principles can be used to overload -= \*= /= %= -- predecrement and – postdecrement.

overld3.cpp overloads + and > along with stream insertion. This program’s principles can be used to overload all the other mathematical operators ( - \* / %) and all the other relational operators (== != >= < <=).

overld4.cpp is an optional program that shows a small string library of overloaded operators. For example, overloading == means the main programmer can do the following:

if(string1 == string2) which is nice as no strcmp would then be needed by the main programmer.